Transaktionale replizierte Objekte für verteilte und parallele Anwendungen

Inaugural-Dissertation

zur Erlangung des Doktorgrades der
Mathematisch-Naturwissenschaftlichen Fakultät
der Heinrich-Heine-Universität Düsseldorf

vorgelegt von

Marc-Florian Müller
aus Hannover

Düsseldorf, April 2011

aus dem Institut für Informatik
der Heinrich-Heine-Universität Düsseldorf

Gedruckt mit der Genehmigung der
Mathematisch-Naturwissenschaftlichen Fakultät der
Heinrich-Heine-Universität Düsseldorf

Referent: Prof. Dr. Michael Schöttner
Koreferent: Prof. Dr. Stefan Conrad

Tag der mündlichen Prüfung: 23.05.2011

Kurzfassung

Die Taktraten moderner Prozessoren wachsen immer langsamer, bedingt durch physikalische Grenzen. Deshalb wird die parallele und verteilte Ausführung von Anwendungen immer wichtiger. In Anlehnung an das Konzept eines transaktionalen Speichers für Mehrkern- und Multiprozessorsysteme schlägt diese Arbeit einen neuartigen verteilten transaktionalen Speicher vor, der Speicherinhalte einzelner Rechner für verteilte Anwendungen transparent mithilfe von Transaktionen synchronisiert. Der datenzentrierte Ansatz trennt Programmlogik und Netzwerkkommunikation strikt voneinander und ermöglicht die Entwicklung verteilter Anwendungen, die direkt über Speicherzugriffe kommunizieren können. Aufgrund der verwendeten optimistischen Synchronisierung können bei der Ausführung von Transaktionen keine Verklemmungen entstehen, wodurch sich die verteilte und parallele Programmierung vereinfacht.

Der gemeinsame verteilte transaktionale Speicher unterstützt unbeschränkte Transaktionen für Linux-Betriebssysteme in Verbindung mit einer transparenten Zugriffserkennung, wobei die Rücksetzung von Transaktionen im Konfliktfall ebenso transparent erfolgt. Dies vereinfacht die Entwicklung verteilter Anwendungen, da eine manuelle Registrierung von zugegriffenen Transaktionsobjekten entfallen kann und Anwendungen selbst keine Transaktionskonflikte behandeln müssen.

Der in dieser Dissertation entwickelte gemeinsame verteilte transaktionale Speicher baut auf einem strukturierten Overlay-Netzwerk auf, um auch bei vielen Rechnern skalierbar zu bleiben. Die neu entwickelten peer-to-peer- und koordinatorbasierten Commit-Protokolle kooperieren miteinander, um eine mehrstufige und effiziente Verarbeitung von Transaktionen zu gewährleisten. Eine wesentliche Neuerung ist die overlay-gestützte Steuerung der Netzwerkkommunikation, um die Netzwerkbandbreite effizient zu nutzen und Verzögerungen durch die Netzwerkkommunikation zu minimieren. Ein Monitor analysiert die Kommunikation im Hintergrund und begrenzt diese automatisch regional oder vermeidet sie vollständig. Hierfür kommen knoten- und gruppenlokale Commits zum Einsatz, die sich nahtlos in das Overlay-Netzwerk integrieren und dynamisch die Netzwerkkommunikation steuern. Über den Monitor erkennt das Overlay-Netzwerk ferner veränderte Netzwerkparameter und Zugriffsmuster und kann sich durch eine dynamische Restrukturierung adaptiv anpassen.

Ein neu entwickeltes peer-to-peer-basiertes Tokenverfahren für das Peer-to-Peer-Commit-Protokoll synchronisiert die Transaktionen im Netzwerk. Es integriert eine interne Warteschlange von Tokenanfragen für eine effiziente Weitergabe des Tokens zwischen Rechnern. Weitere Aspekte sind die intelligente Tokenvorhersage, welche die Netzwerkkommunikation für Tokenanfragen minimiert. Ein weiteres koordiniertes Tokenverfahren vermeidet die Suche des Tokens im Netzwerk, ist aber aufgrund der Kombination mit einem peer-to-peer-basierten Austausch sehr effizient.

Im Umfeld der verteilten transaktionalen Speicher wurde erstmalig das Konzept kaskadierter Transaktionen realisiert, welches auch in Netzwerken mit hohen Netzwerklatenzen einen hohen Transaktionsdurchsatz erlaubt. Das transaktionale Speichersystem schließt Transaktionen hierzu transparent nebenläufig ab. Für Anwendungen entstehen während eines Transaktionsabschlusses deshalb keine Wartezeiten durch das Commit-Protokoll, da sie nebenläufig nachfolgende Transaktionen ausführen können.

Die Experimente belegen, daß der gemeinsame verteilte transaktionale Speicher in Kombination mit dem hierarchisch strukturierten Overlay-Netzwerk und den weiteren Optimierungstechniken besser als die direkte Synchronisierung von Transaktionen ohne das strukturierte Overlay-Netzwerk skaliert. Die zusammen mit dem Overlay-Netzwerk entwickelten Optimierungen machen den gemeinsamen verteilten transaktionalen Speicher für viele Rechner skalierbar und erlauben dessen Verwendung auch in Weitverkehrsnetzen, wie beispielsweise auch in Grid-Systemen.

Abstract

Because of physical constraints CPU clock rates are no longer growing as fast as in the past. As a consequence there is a paradigm shift in computer architecture moving to multi and many core CPUs. Only concurrent (multi-threaded) programs can exploit the potential of such CPUs pushing parallel programming into mainstream. This thesis adopts the basic idea for a distributed transactional memory, which transparently synchronizes in-memory data of distributed and parallel applications using transactions. The data-centric approach separates program logic and network communication and allows distributed applications to communicate through memory. By using a transparent optimistic synchronization scheme deadlocks can be avoided while at the same time simplifying development of distributed and parallel applications.

The proposed distributed transactional memory supports unbounded transactions for Linux-based operating systems combined with a transparent memory access detection and transparent transaction rollbacks in case of conflicts. This eases the development of distributed applications by relieving programmers from using specific functions to access transactional objects and doing manual rollbacks.

For scalability reasons the distributed transactional memory proposed in this thesis relies on a structured overlay network. For this purpose new peer-to-peer- and coordinator-based commit protocols have been designed. These protocols cooperate at multiple levels allowing efficient transaction processing. Another contribution is the integration of overlay-based network communication within a distributed transactional memory. This allows minimizing communication-related delays and to use network bandwidth efficiently. The overlay network monitors communication patterns and automatically limits transaction commits regionally, or completely avoids them whenever possible. Therefor, the overlay network uses peer and group local commit techniques. The overlay network monitoring also tracks changing network parameters and application-related memory-access patterns used to dynamically adapt the overlay structure.

A new commit token with a built-in request queue has been developed extending traditional peer-to-peer commit protocols. This improves token passing efficiency and is further improved by a smart token prediction scheme. Another coordinated token approach avoids searching the token among many nodes in the network. The latter also integrates the peer-to-peer-based token passing mechanism.

The concept of cascading transactions has been introduced for the first time in a distributed transactional memory system improving transaction throughput in networks with high network latencies. It allows applications to execute the next transaction concurrently while another transaction is in its commit phase. This avoids waiting for the outcome of commits taking the risk of cascading aborts. Nevertheless, we prefer a speculative execution of transactions over waiting.

The experiments presented in this thesis show that the proposed distributed transactional memory scales better if used with the hierarchical structured overlay network and optimizations, in contrast to the transaction synchronization without the overlay network. The optimized commit protocols improve the scalability of the distributed transactional memory and also allow transaction processing in wide area networks like for example grid environments.

Danksagung

Meinen herzlichsten Dank möchte ich Herrn Prof. Dr. Michael Schöttner aussprechen, der mir die Möglichkeit zu dieser Dissertation eröffnet hat. Ich bedanke mich für die großzügige und konsequente Unterstützung und über die vielen Jahre dieser Arbeit produktiven Diskussionen sowie Denkanstöße, die entscheidend zu dieser Arbeit beigetragen haben. Außerdem möchte ich mich vielmals bei Prof. Dr. Stefan Conrad für die Begutachtung meiner Arbeit bedanken.

Weiterhin danke ich meinen Kollegen Dr. Michael Sonnenfroh, Kim-Thomas Rehmann, Dr. John Mehnert-Spahn und Dr. Michael Braitmeier. Der fachliche und persönliche Austausch haben mich während meiner Dissertation und der Arbeit am Institut sehr bereichert. Ich bedanke mich zudem bei unserem Hilfswissenschaftler Matthias Janson, der mich bei der Implementierung des Overlay-Netzwerks und den damit verbundenen Diskussionen sehr unterstützt hat. Mein weiterer Dank gilt ebenso unseren Hilfswissenschaftlern Oliver Verlinden und Kevin Beineke für die Weiterentwicklung von Testanwendungen. Ebenso möchte ich Mario Kilies für den Beitrag seiner Masterarbeit danken.

Ein ganz besonderer Dank gilt meinen Eltern Annegret und Reiner Müller, die mir zu jeder Zeit hilfreich mit Rat und Tat zur Seite stehen und mich während meiner Ausbildung und der Dissertation immer konsequent unterstützt haben.

Inhaltsverzeichnis

1 Einleitung

Verteilte Anwendungen treten in unterschiedlichen Formen auf, haben aber allgemein das gemeinsame Ziel, verteilte Ressourcen miteinander zu verknüpfen. Weit verbreitete Anwendungen liegen beispielsweise im Bereich des Hochleistungsrechnens (*engl. High Performance Computing (HPC)*), in dem Rechen- und Speicherkapazitäten gebündelt werden, um große Rechenaufgaben in kurzer Zeit zu lösen, was einzelne Rechner nicht leisten können. Genauso existieren auch über längere Zeiträume laufende Anwendungen, die auch auf eine hohe Rechenleistung angewiesen sind. Diese sind dem *High Throughput Computing (HTC)* zuzuordnen. Der Fokus liegt hier aber mehr auf effizienter Ausnutzung der Rechenkapazitäten, beispielsweise durch Stapelverarbeitung, und Zuverlässigkeit über längere Zeit. Beim HPC und HTC sind Rechner zumeist eng miteinander gekoppelt.

Eine weitere Form des verteilten Rechnens mit lose gekoppelten Rechnern bezeichnet das *Grid Computing*. Rechner können dem Grid dynamisch beitreten (und auch wieder verlassen) und Ressourcen konsumieren, aber ebenso auch Ressourcen anderen Grid-Teilnehmern anbieten. Im Mittelpunkt stehen hier unter anderem Heterogenitäts- und Sicherheitsaspekte. Ein prominentes Projekt auf dem Gebiet des Grid Computings ist das von der EU im sechsten Forschungsrahmenprogramm (FP6-033576) geförderte Projekt *XtreemOS* [106], in dessen Rahmen auch diese Dissertation entstanden ist. Ziel von XtreemOS ist es, ein Grid-Betriebssystem zu entwickeln, welches auf dem Linux-Betriebssystem aufsetzt und mit der Grid-Funktionalität vereint. Ein Teilprojekt von XtreemOS besteht in der Entwicklung eines verteilten transaktionalen Speichers [73] für verteilte Cluster und Grid-Anwendungen, welches diese Arbeit konzeptuell diskutiert.

Weitere bekannte Grid-Systeme sind Unicore [87] und Globus [37], die allerdings als Grid-Middleware fungieren und sich in bestehende Infrastrukturen einfügen. XtreemOS ist im Gegensatz dazu keine Middleware, da es die elementaren Grid-Funktionen direkt in das Betriebssystem integriert und weitere Grid-Dienste anbietet. Weitere Grid-Systeme, die allerdings auf Clustersysteme mit schnellen Netzwerkverbindungen abzielen, sind auch unter dem Namen *In-Memory Data Grids* bekannt. Beispiele sind hierfür Oracle Coherence [77], Terracotta [99] und Velocity [70].

Bei der Entwicklung verteilter Anwendungen finden oftmals weit verbreitete Kommunikationstechniken der Interprozesskommunikation (IPC) wie beispielsweise *CORBA (Common Object Request Broker Architecture)* oder *Java RMI (Remote Method Invocation)* Anwendung. Auf unterster Ebene erfolgt die Kommunikation verteilter Anwendungen über den direkten Austausch von Netzwerknachrichten (*engl. Message Passing (MP)*), wobei verteilte Anwendungen Message-Passing-Kommunikation auch selbst implementieren oder entsprechende Schnittstellen wie zum Beispiel *OpenMP* verwenden können. Message Passing hat den Nachteil, daß die Kommunikation sehr stark mit der Programmlogik verwoben ist, was die Anwendungsprogrammierung schwierig und fehleranfällig macht (siehe Kapitel 2.11). CORBA und RMI abstrahieren gegenüber Message Passing stärker von der Kommunikationsinfrastruktur. Jedoch erfolgt die Verteilung von Daten hier nur explizit mithilfe entfernter Methodenaufrufe. Dies verkompliziert die Entwicklung verteilter Anwendungen, bei denen mehrere Rechner konkurrierend auf gemeinsame Daten zugreifen können. Der gemeinsame verteilte transaktionale Speicher (engl. Distributed Transactional Memory (DTM)) in dieser Arbeit führt eine

strikte Trennung von Programmlogik und Kommunikation durch, indem Anwendungen auf gemeinsamen Speicherbereichen arbeiten und hierüber kommunizieren.

1.1 Transaktionen

1.1.1 Datenbanksysteme

Transaktionen (TA) sind Konstrukte, die einzelne Operationen zu einer Einheit bündeln, um diese atomar unter Einhaltung bestimmter Transaktionseigenschaften auszuführen. Der Begriff *Transaktion* in der Informatik kommt ursprünglich aus dem Datenbankbereich, um dort den konkurrierenden Zugriff auf eine Datenbank (DB) zu regulieren. Die Zugriffe werden dabei von einem Datenbankmanagementsystem (DBMS) ausgeführt. Die Kombination aus DBMS und DB bezeichnet man als Datenbanksystem (DBS). Gegenwärtig wird der Transaktionsbegriff allgemein für Transaktionen mit vier zentralen Eigenschaften, den ACID-Eigenschaften, verwendet. ACID-Transaktionen wurden erstmals von Haerder et al. [44] definiert. Die vier ACID-Eigenschaften sind im einzelnen

- *Atomarität (engl. Atomicity)* – bedeutet, daß eine Transaktion entweder alle oder keine ihrer Einzeloperationen auf einer Datenbank ausführt.
- *Konsistenz (engl. Consistency)* – definiert bei einer erfolgreichen Ausführung einer Transaktion die Überführung einer Datenbank von einem konsistenten in einen neuen konsistenten Zustand.
- *Isolation (engl. Isolation)* – verhindert die gegenseitige Beeinflussung mehrerer überlappender Transaktionen.
- *Dauerhaftigkeit (engl. Durability)* – legt fest, daß Änderungen einer Transaktion nach einer erfolgreichen Validierung auf einem persistenten Datenspeicher festgeschrieben werden.

Transaktionen erhalten die Konsistenz einer Datenbank unter nebenläufigen Lese- und Schreibzugriffen auf Datenbankobjekte. Sie müssen serialisierbar sein, um die ACID-Eigenschaften einzuhalten. Zugriffe innerhalb von Transaktionen, welche die ACID-Eigenschaften verletzen, spiegeln sich in Konflikten zwischen den betroffenen Transaktionen wider. Das Datenbanksystem erkennt in Konflikt stehende Transaktionen (siehe Kapitel 2.5) und behandelt diese so, daß die Konsistenz der Datenbank gewahrt bleibt. Transaktionskonflikte lassen sich auflösen, indem das DBMS eine minimale Anzahl der beteiligten Transaktionen abbricht und gegebenenfalls erneut ausführt.

1.1.2 Abgrenzung zu Datenbanksystemen

Die Transaktionen aus dem Datenbankbereich regulieren in dieser Arbeit analog nebenläufige Zugriffe auf gemeinsame Speicherbereiche von verteilten Anwendungen. Wobei die ACID-Eigenschaften bezüglich der in dieser Arbeit präsentierten Konzepte eines gemeinsamen verteilten transaktionalen Speichers leicht modifiziert sind. Die Dauerhaftigkeitseigenschaft ist in transaktionalen Speichersystemen in der Regel nicht vorhanden, da diese auf den Hauptspeichern (engl. Random Access Memory (RAM)) von Computern aufsetzen, welche selbst flüchtig sind.

1.2 Transaktionaler Speicher

In heutigen Betriebssystemen und Systemarchitekturen erfolgt die Programmausführung nebenläufig in mehreren Prozessen und Threads. Deshalb können Speicherzugriffe von unterschiedlichen Prozessen und Threads verschachtelt auftreten. Solange die Zugriffe nur unterschiedliche Speicherbereiche beziehungsweise nur Lesezugriffe betreffen, spielt die Synchronisierung der nebenläufigen Zugriffe keine Rolle. Erfolgen allerdings überlappende Speicherzugriffe verschachtelt mittels Lese- und Schreibzugriffen oder nur durch Schreibzugriffe, sind diese ohne weitere Maßnahmen nicht synchronisiert. Sprich die Reihenfolge der Zugriffe ist nicht vorherbestimmbar. Nun ist die Reihenfolge aber mitunter wichtig für ein korrektes Laufverhalten von Programmen, da ansonsten ungültige Programmzustände auftreten können.

Neben einer falschen Zugriffsreihenfolge können Variablen im Speicher auch ungültige Werte annehmen, die keine der schreibenden Ausführungseinheiten jemals geschrieben hat. Dies trifft dann zu, wenn die unterliegende Hardware (Prozessor) die Schreibzugriffe nicht atomar behandeln kann. In der Regel unterstützen Prozessoren atomare Zugriffe auf Speicherinhalte nur auf primitive Datentypen (zum Beispiel Speicherzugriffe entsprechend der prozessoreigenen Datenbus- oder Registerbreite). So kann es vor allem bei zusammengesetzten Datentypen wie Datenstrukturen zu ungültigen Inhalten kommen.

Betriebssysteme bieten für einen nebenläufigen Zugriff von mehreren Threads und Prozessen Synchronisierungskonstrukte, die eine fehlerfreien Speicherzugriff garantieren. Solche Konstrukte, als *wechselseitiger Ausschluß*, *Sperre* oder *Semaphor* bezeichnet, regeln den Zugriff, indem sie parallele Zugriffe blockieren und soweit verzögern, daß eine serialisierte Zugriffsreihenfolge besteht. Mittels dieser Konstrukte ist es Programmen ebenfalls möglich, komplexe Datenstrukturen fehlerfrei zu synchronisieren. Unabhängig von der Art der verwendeten Sperren sind diese Konstrukte komplex und fehleranfällig, dies gilt gerade für hochgradig parallele Zugriffe und für viele feingranulare statt weniger grobe Sperren. Fehlerhafte Sperren können entweder zu unsynchronisierten Speicherzugriffen oder zur Verklemmung (engl. Dead- oder Livelock) führen. Eine unkompliziertere und weniger fehleranfällige Methode für die Synchronisierung bieten Transaktionen, bekannt aus dem Datenbankbereich. Dieses Konzept führt zum *transaktionalen Speicher* und wurde erstmals von Herlihy et al. mittels eines modifizierten Cache-Kohärenzprotokolls vorgestellt [49]. Speicherzugriffe erfolgen demnach über Einkapselung in Transaktionen. Vorarbeiten hierzu auf Basis funktionaler Programmiersprachen hat bereits T. Knight durch Behandlung kurzer funktionaler Codeabschnitte als Transaktionen geleistet [56].

Der transaktionale Speicher gruppiert mehrere Anweisungen zu einer Einheit und folgt größtenteils dem gleichen Schema wie transaktionale Datenbankzugriffe. Folglich muß er die Daten so vorhalten, daß er im Falle eines Transaktionsabbruchs die urspünglich geänderten Daten im Speicher wiederherstellen kann. Der Unterschied zu Datenbanksystemen besteht jedoch darin, daß dieses Änderungen an einer Datenbank in der Regel auf einem persistenten Speicher ablegt, so daß einmal festgeschriebene Transaktionen auch bei Fehlern wie Systemabstürze nicht verloren gehen und die Datenbank ebenso in einem konsistenten Zustand bleibt. Bei dem transaktionalen Speicher erfolgt vorwiegend kein Festschreiben auf einem zusätzlichen Festwertspeicher. Persistenz ist bei einem flüchtigen Speicher wie den Hauptspeichern von Computern daher nicht gegeben. Der Fokus dieser Arbeit liegt auf der Synchronisierung von flüchtigen Computerhauptspeichern mittels Transaktionen, weshalb eventuelle Persistenzanforderungen gesondert zu behandeln sind. Persistenz kann der transaktionale Speicher beispielsweise durch die Verwendung von *Checkpointing* [34] erreichen. Der transaktionale Speicher arbeitet somit ähnlich wie ACID-Transaktionen auf Datenbanken, nur daß der transaktionale Speicher nicht dessen Persistenzeigenschaft *(Durability)* unterliegt.

Ein transaktionaler Speicher ist auf unterschiedliche Arten implementierbar. Er kann zum einen

vollständig in Hardware oder auch in Software implementiert werden. Einen Implementierung in Hardware bezeichnet man als *Hardware Transactional Memory (HTM)* (siehe Kapitel 1.2.1), einen softwarebasierenden dagegen als *Software Transactional Memory (STM)* (siehe Kapitel 1.2.2). Weiterhin existieren auch hybride Ansätze. Bei diesen handelt es sich um einen STM mit Hardwareunterstützung für die Transaktionen.

1.2.1 Hardwarebasierter transaktionaler Speicher

Bei einem HTM übernimmt der Prozessor weitestgehend die Verarbeitung von Transaktionen, weswegen diese gegenüber Prozessoren, die keine Transaktionen unterstützen, Erweiterungen für die Transaktionsverarbeitung besitzen. Hierzu gehören unter anderen die Verwaltung von Transaktionen, Aufzeichnung von Speicherzugriffen, spekulative Konfliktauflösung [58, 100] und Konstrukte zur Festlegung von Transaktionsgrenzen, um atomare Ausführungseinheiten, bestehend aus mehreren Einzelinstruktionen (unter anderem Lese- und Schreibinstruktionen für Speicherzugriffe) definieren zu können. Je nach Ausführung eines HTM kann dieser noch weitere Optimierungen hinsichtlich Caches und paralleler Transaktionsausführung besitzen. Hardwaretransaktionale Speicher können aber auch nur einen Teil der Transaktionserweiterungen in Hardware implementieren und zusätzlich auf die Unterstützung eines STM aufbauen. Solche Systeme werden auch als hybride transaktionale Speicher bezeichnet, wobei die Grenzen zwischen hardwarebasierende und hybride transaktionale Speicher nicht eindeutig sind.

Im allgemeinen sind HTM schneller als STM, da dessen Hardware auf die Ausführung von Transaktionen optimiert ist, wohingegen bei letzterem der Standardbefehlssatz des Prozessors (engl. Central Processing Unit (CPU)) für die Ausführung und Verwaltung von Transaktionen in Software dient. Anwendungszweck für HTM ist die Synchronisierung von Speicherzugriffen über mehrere Threads und Prozesse entweder auf einer CPU, wobei hier keine echte parallele Programmausführung stattfindet, oder auch mit mehreren CPU-Kernen oder Prozessoren, wie sie heutzutage in Mehrkern- und Multiprozessorsystemen anzutreffen sind. In den letzteren beiden findet dagegen eine echte parallele Programmausführung statt. Verschachtelte Speicherzugriffe treffen auf alle drei Systemtypen zu, da auch bei nicht echt paralleler Programmausführung geschachtelte Speicherzugriffe von mehreren Threads und Prozessen auftreten können. Dies liegt in den präemptiven Schedulingverfahren, der heutzutage eingesetzten Betriebssysteme begründet, da bei diesen ein Task- oder Threadwechsel asynchron zur Programmausführung erfolgt.

Das Konzept des transaktionalen Speichers haben Hammond et al. in *Transactional Memory Coherence and Consistency (TCC)* weiterentwickelt. TCC bietet spekulative Transaktionen für die parallele Ausführung von Programmcode auf Mehrkern- und Multiprozessorsystemen, die über Hochgeschwindigkeitsbusse gekoppelt sind. Synchronisierung und Konflikterkennung erfolgen durch Aufzeichnen von Lese- und Schreibzugriffen einzelner Transaktionen und eines modifizierten Cache-Kohärenzprotokolls. Beim Commit sendet TCC die Schreibmenge (Menge aufgezeichneter Schreibzugriffe) atomar mittels Broadcast an das System. Hierdurch können andere parallele Ausführungseinheiten Konflikte aufgrund veralteter gelesener Daten erkennen [46, 45].

Die Verwaltung von Transaktionen in Hardware ist günstig bezüglich Geschwindigkeit, unterliegt jedoch eventuell einigen Einschränkungen. So haben Pufferspeicher innerhalb der Hardware eine beschränkte Größe, so daß Transaktionen hinsichtlich ihrer Lese- und Schreibmenge ebenfalls dieser Größe unterliegen. Weiterhin können Einschränkungen auch bei Kontextwechseln unterschiedlicher Tasks bestehen, sofern die Hardware diese nicht explizit unterstützt. Ungebundene transaktionale Speicher umgehen diese Einschränkungen, indem sie für die Transaktionsverwaltung den physischen oder virtuellen Speicher miteinbeziehen [7, 25].

Der Fokus dieser Arbeit liegt auf softwaretransaktionale Speicher für verteilte Anwendungen, aus diesem Grund finden HTM-Systeme in dieser Arbeit keine weitere Betrachtung mehr.

1.2.2 Softwarebasierter transaktionaler Speicher

Bei einem STM ist die gesamte Transaktionslogik in Software realisiert. Daher muß die Hardware Transaktionen nicht explizit unterstützen. Vorteile gegenüber einem HTM sind, daß STM-Systeme flexibel auf vielen unterschiedlichen Hardwarearchitekturen einsetzbar sind, wenngleich sie hinsichtlich Geschwindigkeit in der Regel langsamer sind, da sie die Transaktionsunterstützung der Hardware in Software nachbilden müssen. Weitere Vorteile sind, daß Transaktionen wie auch bei ungebundenen HTM oftmals nicht in ihrer Größe beschränkt sind und sich zudem gut auf Betriebssysteme anpassen lassen, zum Beispiel durch Beeinflussung des Prozeß- beziehungsweise Threadschedulers.

Basierend auf den hardwaretransaktionalen Speicher von Herlihy et al. haben Shavit et al. den ersten STM für Multiprozessorarchitekturen entwickelt [93]. Der Fokus liegt hierbei auf statischen Transaktionen, also Transaktionen mit zuvor festgelegten Speicheradressen, auf welche die Transaktion im Laufe ihrer Ausführung zugreift. Etwaige Zugriffe auf zuvor nicht festgelegte Speicheradressen finden bei der Validierung keine Berücksichtigung und unterliegen daher auch keine Berücksichtigung bei der Rückabwicklung von Änderungen im Konfliktfall.

Statische Transaktionen sind hinsichtlich der Programmierung nicht sehr flexibel, da dem Programmierer vorab bekannt sein muß, auf welche Speicheradressen die Transaktion möglicherweise zugreift. Sind Lese- und Schreibmenge der Transaktion für die Validierung größer als die tatsächlichen Speicherzugriffe, führt dies zu einer erhöhten Konfliktwahrscheinlichkeit. Sind die Mengen dagegen kleiner, kann die Transaktion keine Datenkonsistenz garantieren, da nicht alle Daten der Transaktionsvalidierung unterliegen. Diese Einschränkung lösen Herlihy et al. erstmals mit einem dynamischen STM [48] durch Beobachtung von Zugriffen auf Objektreferenzen.

1.2.3 Hybrider transaktionaler Speicher

Der hybride Ansatz verbindet die Vorzüge des HTM mit denen eines STM. Demnach bezeichnet er einen STM, welcher hardwaretransaktionale Funktionen des Prozessors als Unterstützung verwendet. Somit kann dies beispielsweise einem STM ermöglichen, ebenso wie einem vollständigen HTM, Objektzugriffe mit Hardware auf Basis von Cache-Lines durchzuführen. Da ein HTM gegenüber einem STM allgemein schneller ist, kann ein hybrides System die Skalierbarkeit eines reinen STM verbessern, welcher weiterhin die Flexibilität und Heterogenität eines gewöhnlichen STM bietet.

Das erste hybride transaktionale Speichersystem von Damron et al. [30] zeigt einen STM, der zur Leistungssteigerung HTM-Systeme mitverwendet. Das System ist jedoch nicht nur auf HTM-Systeme beschränkt, sondern kann auch als reines STM-System laufen. Der hybride transaktionale Speicher greift nur dann auf die Hardwareunterstützung für Transaktionen zurück, sofern diese vorhanden ist. Anderenfalls arbeitet es als STM. Ein ähnlichen Ansatz eines STM mit Unterstützung der Hardware verfolgen Saha et al. in [90].

1.3 Verteilter transaktionaler Speicher

Verteilte transaktionale Speicher erweitern das Paradigma des transaktionalen Speichers auf verteilte Systeme. Zugrunde liegen hier klassische verteilte Speicherkonzepte, die bereits seit

Bezeichnung	System	Anwendungszweck
Plurix	SSI-Betriebssystem[2]	Cluster
DiSTM	Programmiergerüst	Cluster
ClusterSTM	Programmiergerüst	Cluster
Sinfonia	Programmiergerüst	Cluster

Tabelle 1.1: Übersicht über verteilte transaktionale Speichersysteme.

den 1980er Jahren, bekannt als *Distributed Shared Memory (DSM)*, bestehen. DSM-Systeme können sowohl in Hardware als auch in Software implementiert sein. DSM-Systeme spiegeln Anwendungen vor, einen gemeinsamen globalen Adreßraum zu besitzen, obwohl sich dieser über mehrere physisch voneinander unbhängige Speicher auf unterschiedlichen Rechnern erstreckt. Der verteilte transaktionale Speicher in dieser Arbeit basiert auf der Synchronisierung unterschiedlicher physischer Speicher auf mehreren Rechnern, weshalb DSM-Systeme hierfür die Grundlage bilden. Den ersten verteilten gemeinsamen Speicher hat Kai Li mit *Ivy* [62] entwickelt, welcher mittels Message Passing die lokalen Speicher einzelner Rechner über Netzwerke miteinander synchronisiert und so einen virtuellen globalen Adreßraum bildet. Ein weiteres bekanntes System auf diesem Gebiet ist Treadmarks [6].

Einen anderen Ansatz verfolgt David Gelernter mit *Tupel-Spaces* [43] in der parallelen Programmiersprache Linda. Die Kommunikation zwischen Rechnern erfolgt über Tupel in einem gemeinsamen Tupelraum. Rechner können Tupel in den Tupelraum schreiben, die andere Rechner wiederum lesen, schreiben oder löschen können. Auf Basis der Tupel können Anwendungen eigene Protokolle entwickeln. Mit Java-Spaces [40] ist eine Implementierung eines Tuple-Space-Systems in der Programmiersprache Java verfügbar. Der verteilte transaktionale Speicher in dieser Arbeit und Ivy haben mehrere Gemeinsamkeiten, da beide einen virtuellen globalen Adreßraum (Replizierte Daten im physischen Speicher einzelner Rechner) mittels Message Passing synchronisieren. Auch findet die Zugriffserkennung bei beiden mittels der *Speicherverwaltungseinheit (Memory Management Unit, MMU)* statt (siehe Kapitel 2.4).

Verteilte transaktionale Speichersysteme wurden bisher noch nicht intensiv untersucht. Es gibt aber bereits einige Arbeiten hierzu, diese sind in Tabelle 1.1 aufgeführt. Zunächst benötigen *Transaktionen* im verteilten Kontext eine genaue Definition, da Transaktionen je nach Anwendungsgebiet eine unterschiedliche Bedeutung zukommt. Transaktionen können prinzipiell sowohl im lokalen als auch verteiltem Anwendungskontext operieren. Dies bedeutet in Bezug auf den gemeinsamen verteilten Speicher, daß der Geltungsbereich von Transaktionen entweder auf einen einzelnen Knoten[1] des transaktionalen Speichersystems beschränkt ist oder sich verteilt über mehrere physische Rechner erstreckt, wie dies beispielsweise bei verteilten Datenbanksystemen (DBS) [61] der Fall sein kann. Da alle Knoten des transaktionalen Speichersystems einer Illusion eines gemeinsamen Speichers unterliegen, obwohl jeder Rechner nur auf zwischengespeicherte Speicherinhalte in seinem physischen Speicher zugreifen kann, sind für verteilte gemeinsame transaktionale Speicher keine verteilten Transaktionen notwendig. Der Gültigkeitsbereich einer Transaktion beschränkt sich demnach nur auf den Prozeß beziehungsweise Thread, welcher die Transaktion ausführt, und den physischen Speichers des Rechners.

Somit dienen Transaktionen im verteilten Speicher der geordneten Synchronisierung komplexer replizierter Datenstrukturen, die von mehreren Rechnern nebenläufig zugegriffen werden. Die Vorgehensweise der Synchronisierung hängt dabei vom vereinbarten Konsistenzmodell ab (siehe Kapitel 2.2).

[1] Bezeichnung für einen Rechner in einem Netzwerk
[2] Single-System-Image

1.4 Zielsetzung dieser Arbeit

Diese Arbeit stellt Konzepte und Commit-Protokolle für die Synchronisierung eines verteilten transaktionalen Speichers für verteilte Anwendungen auf lose gekoppelten Rechnern vor. Dieser dient Programmierern, die Entwicklung verteilter Anwendungen zu vereinfachen und von allen Aspekten der Kommunikation zwischen verteilten Anwendungseinheiten zu entlasten. Hierzu abstrahiert das Modell von der Netzwerkkommunikation und dessen Infrastruktur und stellt mit dem gemeinsamen verteilten transaktionalen Speicher einen datenzentrierten Ansatz zur Verfügung. Transaktionen unterstützen Porgrammierer bei der Synchronisierung nebenläufiger Ausführungseinheiten einer Anwendung, die ansonsten traditionell über komplexe und fehleranfällige Sperren erfolgt.

Da alle Rechner autonom auf ihrem lokalen Speicher arbeiten, muß das System transaktionale Speicherinhalte über Netzwerke replizieren und synchronisieren. Die Latenz im Netzwerk ist um ein vielfaches höher als die des lokalen Speicherbusses eines Rechners, so daß bei der Synchronisierung des verteilten transaktionalen Speichers Wartezeiten entstehen, welche die Leistungsfähigkeit des Systems mindern. Dies trifft vor allem dann zu, wenn Rechner über Weitverkehrsnetze mit hohen Latenzen miteinander gekoppelt sind. Primäres Ziel dieser Arbeit ist es, die Synchronisierung verteilter transaktionaler Speicherinhalte effizient über Netzwerke zu koordinieren. Ferner gehört hierzu ebenso die Validierung von Transaktionen als auch deren Serialisierung über das Netzwerk. Dabei muß das System auch bei zunehmender Anzahl von Rechnern skalierbar bleiben. Hinzu kommt, daß bei lose gekoppelten Rechnern in Verbindung mit der Synchronisierung über Netzwerke, die Wahrscheinlichkeit für Fehler (zum Beispiel Ausfall einzelner Rechner oder Netzwerkfehler) höher als auf einem einzelnen Rechner ist.

1.5 Struktur der Arbeit

Das folgende Kapitel 2 stellt zunächst die grundlegenden Konzepte der Synchronisierung replizierter Speicher vor und dessen Unterschiede zu lokalen Systemen. Die Verwendung von Transaktionen und die Erkennung von Speicherzugriffen sind für den verteilten Speicher von zentraler Bedeutung, da diese Voraussetzung für die Synchronisierung sind. Synchronisierungsstrategien sowie die Speicherallozierung und Besonderheiten bei unterschiedlichen Daten im Speicher geben einen tieferen Einblick in die Thematik. Kapitel 3 behandelt die transparente Rücksetzung beziehungsweise den transparenten Abbruch von Transaktionen[3] und die damit einhergehenden Besonderheiten, die dabei in Zusammenhang mit kritischen Programmabschnitten wie Systemaufrufen auftreten können. Daran schließt sich in Kapitel 4 eine Diskussion über ein Commit-Protokoll für die Konsistenzerhaltung der Speicherinhalte und Regulierung von transaktionalen Konflikten an. Dabei findet eine tiefgehende Analyse von Skalierbarkeit und Netzkommunikation statt und ebenfalls, wie Transaktionen effizient serialisiert abschließen können. Im weiteren Verlauf stellt das folgende 5. Kapitel ein weiteres Commit-Verfahren und mit einem Overlay-Netzwerk Strategien vor, die dem System zu einer guten Skalierbarkeit verhelfen. Die hierzu notwendigen Optimierungen bezüglich der Commit-Protokolle und Transaktionsverwaltung diskutiert Kapitel 6. Das nachfolgende Kapitel 7 schließt die Arbeit mit Messungen und deren Analyse anhand eines implementierten Prototyps *Object Sharing Service (OSS)* ab, der sich als Programmbibliothek in unterschiedliche Programmiersprachen und Laufzeitumgebungen integrieren läßt.

[3] Bezeichnung für die Rückabwicklung sämtlicher in einer Transaktion ausgeführten Änderungen im transaktionalen Speicher, ohne zu definieren, ob die Transaktion anschließend erneut startet oder endgültig endet.

2 Programmiermodell für einen verteilten transaktionalen Speicher

Dieses Kapitel diskutiert ein Programmiermodell für die Programmiersprache *C*. Viele Komponenten des Linux-Betriebssystems sind in der Programmiersprache C geschrieben [105]. Weiterhin bietet C eine Schnittstelle für andere Programmiersprachen. Anwendungen, die in einer anderen Sprache geschrieben sind, können so in der Programmiersprache C geschriebene Funktionen aufrufen. Die Programmierschnittstelle legt die Verwendung des verteilten transaktionalen Speichers für Anwendungsentwickler fest. Diese sollte möglichst einfach und kompakt gehalten sein, damit sie die Entwicklung verteilter Anwendungen vereinfacht. Wichtige Aspekte beim Programmiermodell sind das Speichermodell und die Definition von Transaktionen, da hierüber die speicherbasierte Kommunikation erfolgt. Die Definition von Transaktionen im Programmkontext legt fest, welche Codebestandteile der Transaktionsverwaltung unterliegen, wobei die Definition eng mit dem Speichermodell zusammenhängt. Daneben definiert das Modell, wie die Erkennung und Verwaltung von Transaktionsobjekten (siehe Kapitel 2.3) erfolgt, damit das System den transaktionalen Speicher synchronisieren und Konflikte bei der Validierung abzuschließender Transaktionen erkennen kann. Hierbei soll die Zugriffserkennung automatisch und für Programmierer transparent erfolgen. Zuletzt definiert das Modell noch Besonderheiten bei der Programmausführung und Datenstrukturen im transaktionalen Speicher sowie Optimierungen bei Speicherzugriffen außerhalb von Transaktionen.

2.1 Speicherarchitekturen

Der wesentliche Unterschied zwischen lokalen transaktionalen Speichern, wie er beispielsweise oftmals bei Mehrkern- und Multiprozessorsystemen vorliegt, und gemeinsamen verteilten transaktionalen Speichern, wie in dieser Arbeit diskutiert, ist die unterliegende Speicherarchitektur.

Transaktionaler Speicher auf einem einzigen physischen Rechner unterliegt einer *UMA*-Speicherarchitektur *(Uniform Memory Access)*. Gleiches gilt oftmals auch für Mehrkern- und Multiprozessorsysteme. Alle Prozesse, die über die Kerne oder Prozessoren des Rechners verteilt sind, greifen auf einen gemeinsamen physischen Speicher zu. Daher benötigt diese Speicherarchitektur keine Synchronisierung von replizierten Speicherinhalten. Lokale Caches der Prozessorkerne beziehungsweise Prozessoren muß die Hardware konsistent halten, dies kann beispielsweise über das MESI-Protokoll erfolgen [28]. Transaktionen können unter dieser Speicherarchitektur schnell ablaufen. Manche Multiprozessorsysteme besitzen dagegen eine *NUMA*-Speicherarchitektur *(Non Uniform Memory Access)*. Hier besitzt jeder Prozessor seinen eigenen physischen Speicher in einem gemeinsamen Adreßraum, kann aber anderen Prozessoren über den Bus hierauf Zugriff gewähren. Da bei dieser Architektur die Cache-Kohärenz nicht gewährleistet ist, ist hier bei prozessorübergreifenden Zugriffen eine komplexe Cachesynchronisierung notwendig. Deshalb sind diese Systeme oftmals cache-kohärent ausgelegt. Die Cache-Kohärenz wird hier von der Hardware übernommen, so daß hier ebenfalls keine explizite Synchronisierung von Speicherinhalten erforderlich ist. Diese Speicherarchitektur nennt sich *ccNUMA (Cache Coherent NUMA)* [71].

Der in dieser Arbeit behandelte verteilte gemeinsame transaktionale Speicher unterliegt wie viele DSM-Systeme ebenfalls einer *NoRMA*-Speicherarchitektur *(No Remote Memory Access)*. Genauso wie bei NUMA besitzt jeder Rechner seinen eigenen physischen Speicher, kann anderen Rechnern jedoch nicht direkten Zugriff darauf gewähren, da die Rechner nicht über einen gemeinsamen Bus gekoppelt sind. Daher erfordert diese Speicherarchitektur bei einem rechnerübergreifenden Zugriff einen expliziten Austausch von Speicherinhalten über Kommunikationsnetzwerke. Ein Rechner muß angeforderte Daten eines anderen Rechners vor dem Zugriff zunächst in seinem eigenen Speicher ablegen. So entstehen Replikate, die das transaktionale Speichersystem synchronisieren muß (siehe Abbildung 2.1).

2.2 Konsistenzsicherung

Bei replizierten Daten ist es notwendig, deren Konsistenz zu sichern. Ändert ein Rechner replizierte Daten, sind die Änderungen für andere Rechner, die das gleiche Replikat besitzen, nicht sichtbar. Rechner müssen geänderte Replikate untereinander synchronisieren. Hierfür existieren Modelle, die festlegen, zu welchem Zeitpunkt Schreibzugriffe auf Daten bei Lesezugriffen auf anderen Rechnern sichtbar werden. Die Modelle bezeichnet man allgemein als *Konsistenzmodelle* [72, 2]. Strenge Konsistenzmodelle bestimmen eine zügige Sichtbarkeit geschriebener Daten. Hierzu gehören beispielsweise *strikte* sowie *sequentielle Konsistenz*, wobei erstere eine unmittelbare Sichtbarkeit entsprechend der Schreibreihenfolge und letztere nur eine einheitliche Lesereihenfolge unabhängig von der Reihenfolge der Schreibzugriffe festlegt. Schwächere Konsistenzmodelle wie beispielsweise *Eventual Consistency* [102] haben dagegen weniger strenge Vorgaben hinsichtlich Sichtbarkeit geschriebener Daten. Das einzusetzende Konsistenzmodell hängt von den Anforderungen der ausführenden Anwendung ab.

Strenge Konsistenzmodelle erlauben Benutzern einen großen Komfort, da sie die Synchronisierung von replizierten Daten in engen Grenzen festlegen, erfordern allerdings gegenüber schwächeren Konsistenzmodellen mit weniger Komfort eine häufige Synchronisierung. Aus diesem Grund sind strenge Konsistenzmodelle in Anbetracht der Geschwindigkeit in der Regel nicht so leistungsfähig wie schwächere Modelle. Strikte Konsistenz liegt beispielsweise bei allen Systemen vor, die einer UMA- oder ccNUMA-Speicherarchitektur unterliegen. Bei diesen Architekturen ist die strikte Konsistenz jedoch hardwareunterstützt, deshalb ist dort keine explizite Synchronisierung notwendig.

Transaktionen unterliegen in der Regel einer strengen Konsistenz, da geänderte Daten einer erfolgreich abgeschlossenen Transaktion für darauffolgende Transaktionen verbindlich sind und im Fall veralteter gelesener Daten einen Transaktionskonflikt provozieren. Aus Anwendungssicht ist eine strenge Konsistenz nicht immer notwendig, deswegen ist es möglich, bei Transaktionen je nach Anwendung auch schwächere Konsistenzmodelle zuzulassen, wobei die ACID-Eigenschaften der Transaktion weiterhin einzuhalten sind.

Der in dieser Arbeit behandelte verteilte gemeinsame transaktionale Speicher verwendet eine strenge Konsistenz für die Synchronisierung, um für Anwendungsentwickler möglichst transparent zu sein. Der hohe Synchronisierungsaufwand, vor allem über Ethernet-Netzwerke, die wegen ihrer im Vergleich zu rechnerinternen Bussystemen hohen Latenz um ein vielfaches langsamer sind, verursacht Verzögerungen, die mit einer verringerten Leistung einhergehen. Daher ist es Aufgabe der Commit-Protokolle, effizienten Synchronisierungsalgorithmen und Optimierungen, diese Nachteile zu kompensieren. Die Umsetzung der replizierten Objektsynchronisierung nach Vorgabe eines Konsistenzmodells bezeichnet man als *Kohärenz*. Kohärenz kann über zwei unterschiedliche Verfahren *Aktualisierung (engl. Write Update)* oder *Invalidierung (engl. Write Invalidate)* erfolgen.

2.2.1 Invalidierungsverfahren

Unter Verwendung des Invalidierungsverfahrens werden alle Kopien eines geänderten Objekts automatisch ungültig. Dies verhindert eine Verwendung veralteter Objektkopien auf anderen Rechnern. Die Rechner müssen bei einem Zugriffsversuch auf invalidierte Objekte zunächst eine aktuelle Objektversion anfordern. Es ist Aufgabe des Commit-Protokolls, anderen Rechnern mitzuteilen, welche Objekte eine erfolgreich abgeschlossene Transaktion geändert hat, damit diese eventuell bei sich vorhandene Replikate invalidieren können.

Ändert ein Rechner regelmäßig Objektreplikate, auf die andere Rechner aktuell nicht zugreifen, eignet sich das Invalidierungsverfahren besser gegenüber dem Aktualisierungsverfahren. Invalidierungsnachrichten für Objekte sind im Vergleich zu Aktualisierungsnachrichten kleiner, da sie keine Objektinhalte transportieren. Zudem muß ein Rechner Invalidierungsnachrichten nur an die Rechner schicken, die ein zu invalidierendes Objektreplikat besitzen. Aktualisierungsnachrichten halten zwar die Objekte auf den Empfängerknoten aktuell, allerdings ist die regelmäßige Aktualisierung von Objekten nutzlos, sofern die empfangenen Knoten die Replikate nicht verwenden. Das Invalidierungsverfahren spart in diesem Fall Netzwerkbandbreite und durch Interrupts und Umkopieren von Daten verursachte Prozessorauslastung.

2.2.2 Aktualisierungsverfahren

Beim Aktualisierungsverfahren erhalten Rechner, die ein Replikat eines geänderten Objekts besitzen, automatisch die aktuelle Version des Objekts. So müssen die Rechner, die Replikate besitzen, nicht erst die aktuelle Version eines Objekts anfordern.

Das Aktualisierungsverfahren benötigt gegenüber dem ersten Verfahren augenscheinlich mehr Netzwerkbandbreite. Greifen die Knoten, welche aktualisierte Objektinhalte erhalten, regelmäßig auf diese Objekte zu, relativiert sich die höhere Netzwerkbandbreite. Möchten Knoten auf invalidierte Objektreplikate zugreifen, müssen sie zunächst explizit die aktuellen Objektinhalte nachfordern. Zudem erkennen Rechner fehlende Objektreplikate erst beim ersten Zugriff. Somit kann die genutzte Bandbreite des Netzwerks aufgrund des Mehraufwands bei Anfragen höher als unter Verwendung des Aktualisierungsverfahrens sein. Vorausgesetzt, die empfangenen Knoten greifen auf jedes aktualisierte Objekt zu. Weiterhin begünstigt dieses Verfahren die Leistungsfähigkeit durch die nicht benötigte Objektanforderung und der damit verbundenen Latenz. Zum Zeitpunkt des Zugriffs liegt dem Knoten immer eine gültige Objektversion vor. Muß ein Knoten die aktuelle Objektversion erst anfordern, wie das bei dem ersteren Verfahren der Fall ist, blockiert er wegen der Netzwerklatenz und der Nachrichtenverarbeitung solange, bis er die aktuellen Objektdaten empfangen hat.

Sowohl das Invalidierungs- als auch das Aktualisierungsverfahren haben jeweils Vor- und Nachteile. Demzufolge ist es sinnvoll, beide Verfahren gemeinsam einzusetzen, je nachdem, welches Speicherzugriffsmuster gerade vorliegt. Mittels Beobachtung von Zugriffsmustern kann das System dann individuell einzelne Objektreplikate invalidieren oder aktualisieren. Das DSM-System *Munin* [13] kombiniert beide Verfahren und synchronisiert Objekte basierend auf Häufigkeiten der Zugriffsart (lesend oder schreibend) entweder über das Invalidierungs- oder Aktualisierungsverfahren. Die adaptive Objektsynchronisierung ist nicht Bestandteil dieser Arbeit, daher erfolgt die Objektsynchronisierung hier primär mittels des Invalidierungsverfahrens, falls das Aktualisierungsverfahren nicht explizit erwähnt wird.

2.3 Transaktionsobjekte

Der transaktionale Speicher enthält Transaktionsobjekte, wobei der Begriff *Objekt* im Transaktionskontext zunächst einer genauen Definition bedarf. Ein transaktionales Objekt definiert den kleinsten zusammenhängenden Speicherbereich, für den der transaktionale Speicher Lese- und Schreibzugriffe verwaltet. Dieses System notiert nur, ob Zugriffe auf ein transaktionales Speicherobjekt stattgefunden haben, jedoch nicht in welchem Bereich seines abgedeckten Speicherbereichs. Dies ist durch die Objektzugriffserkennung bedingt (siehe Kapitel 2.4). Es existieren aber auch Systeme, die eine wortbasierte Zugriffserkennung realisieren [18]. Weiterhin enthalten die Verwaltungsinformationen eines Objekts dessen Zustand, welcher unter anderem festlegt, ob die Objektkopie gültig ist oder invalidiert wurde. Jedes Objektreplikat besitzt eine Versionsnummer, die sich mit der Änderung des Objektinhalts ändert (siehe Kapitel 4.3.3 und 5.2.3). Transaktionsobjekte sind für Anwendungen transparent, da diese nur als lineare Verkettung von Speicherabschnitten in allozierten transaktionalen Speicherbereichen vorliegen.

Die Zugriffserkennung von Transaktionsobjekten dient der Konflikterkennung der Transaktion folgenden Validierungsphase und der Rückabwicklung von Transaktionen (siehe Kapitel 3) im Konfliktfall. Die Granularität der Objekte spielt hinsichtlich des Verwaltungsaufwandes und der Konflikterkennung eine wichtige Rolle. Die Granularität auf einzelne Bytes festzulegen, erweist sich beispielsweise als wenig sinnvoll, da die Verwaltungsinformationen in diesem Fall viel mehr Speicher in Anspruch nehmen, als die zu verwaltenden Daten selbst. Allerdings begünstigt eine feine Granularität die Konflikterkennung. Im umgekehrten Fall verursacht eine grobe Granularität wenig Verwaltungsinformationen, dafür aber generell vermehrt falsche Konflikte. Bei einer groben Granularität können semantische Konflikte entstehen, obwohl physisch kein Zugriff auf dieselben Daten stattgefunden hat. Dieses Verhalten ist in der Literatur auch als *False Sharing* oder *False Conflicts* bekannt [19, 5]. False Sharing ist ein zeitabhängiges Phänomen und tritt auf, wenn zur selben Zeit mehrere Rechner auf disjunkte Datenbereiche desselben Objekts zugreifen und mindestens ein Rechner schreibend auf ein Datum zugreift. In der Abbildung 2.1) greifen zwei Rechner gleichzeitig auf den transaktionalen Speicher zu, wobei ein Rechner schreibt und der andere liest. Besteht der dargestellte Speicherblock beispielsweise aus nur einem Objekt, kommt es aufgrund von False Sharing zu Konflikten. Besteht der Block dagegen aus drei Objekten (rot gestrichelte Linie), so tritt kein False Sharing auf.

Die feine Granularität kann neben dem höheren Verwaltungsaufwand auch weitere Nachteile haben, da hier der bei der groben Granularität vorhandene Cachingeffekt von Objektdaten verloren geht. Eine grobe Granularität kann wegen des Cachingeffekts eine Leistungssteigerung bewirken aber genauso auch die Wahrscheinlichkeit für False Sharing erhöhen. Eventuell muß ein Knoten bei Zugriffen auf aufeinanderfolgende feingranulare Objekte nacheinander mehrere Objektkopien anfordern, sofern dieser keine gültigen Replikate besitzt. Bei einer groben Granularität entspricht dies weniger Objektanforderungen, da bei einem Zugriff auf einen Teil eines Objekts ein Knoten stets das gesamte Objekt anfordert, sofern es lokal nicht als gültiges Replikat vorliegt. Aufgrund der Netzlatenz erfolgt der Zugriff hier unter der groben Granularität in kürzerer Zeit. Die Objektgranularität läßt sich aber auch durch Ausrichtung der Datenstrukturen an Transaktionsobjekten oder adaptiv über die Beobachtung von True- und False-Sharing-Situationen steuern [39, 82].

Die Objektgranularität beträgt im Rahmen dieser Arbeit 4 KB (Kilobyte), bedingt durch den Mechanismus der Zugriffserkennung (siehe Kapitel 2.4.4).

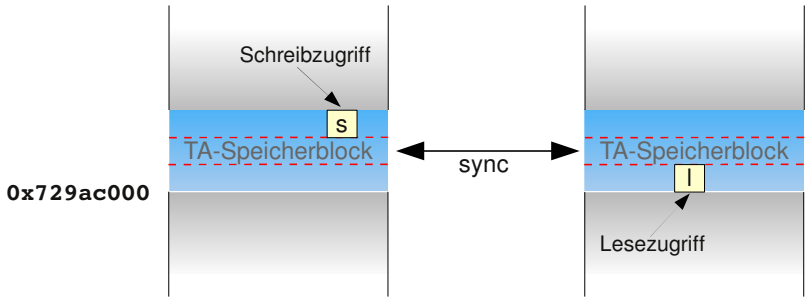


Abbildung 2.1: Replikatsynchronisierung von Transaktionsobjekten und Auswirkung auf die Konfliktwahrscheinlichkeit bei unterschiedlicher Ganularität.

2.3.1 Objektverwaltung

Für die korrekte Validierung von Transaktionen und Synchronisierung von Objektinhalten unter Vorgabe eines Konsistenzmodells muß dem System bekannt sein, auf welche Objekte eine Transaktion lesend und schreibend zugreift. Hierzu existieren mit *expliziter* und *impliziter Objekterkennung* zwei Ansätze, die sich hauptsächlich in dem Aufwand für den Anwendungsentwickler unterscheiden.

Bei der expliziten Objekterkennung muß der Anwendungsentwickler dem System vor dem ersten Zugriff eines transaktionalen Objekts in einer Transaktion einen bevorstehenden Lese- oder Schreibzugriff mitteilen. Auf diese Weise kann das System gelesene und geschriebene Objekte einer Transaktion zuordnen. Falls bei der Registrierung eines Objekts kein gültiges Replikat vorliegt, fordert das System dieses vor Ausführung des Zugriffs an. Diese Prozedur ist für den Anwendungsentwickler aufwendig und fehleranfällig, insbesondere wenn es sich um dynamische Objektzugriffe handelt. Vergißt der Entwickler ein Objekt zu registrieren, bleibt dieses in der Validierungsphase unberücksichtigt und verhindert eine zuverlässige Konflikterkennung. Schreibzugriffe auf nicht registrierte Objekte erfolgen aber dennoch und außerhalb des Transaktionskontexts. Das transaktionale Speichersystem kann solche Zugriffe jedoch nicht erkennen. Dies wiederum führt zur unerkannten Abänderung von Objektreplikaten, falls aktuell ein gültiges Replikat vorliegt. Andere Knoten können diese gültigen Replikate anfordern und eigene Transaktionen abschließen, die diese Änderungen enthalten. Dies führt zur Konsistenzverletzung.

Bei der impliziten Objekterkennung kann die beschriebene Prozedur der expliziten Objekterkennung ein spezieller Compiler übernehmen. Der Compiler muß hierfür wissen, welche Speicherbereiche als transaktionaler Speicher zu behandeln sind. Dies kann beispielsweise durch Anmerkungen (Compiler-Direktiven) im Quelltext erfolgen [47]. Eventuell muß der Compiler aber bei der Dereferenzierung von Zeigern prüfen, ob diese den transaktionalen Speicher betreffen beziehungsweise eine Transaktion läuft. Dies bietet dem Entwickler mehr Komfort und ist sicher gegenüber fehlerhaft registrierte Objekte der expliziten Objekterkennung.

Eine andere Variante, die für Anwendungsentwickler und Compiler vollständig transparent ist, basiert auf der Erkennung über die prozessoreigene Speicherverwaltungseinheit (siehe Kapitel 2.4). Bei allen Verfahrensweisen muß der Entwickler die Transaktionsgrenzen festlegen – die Anwendungssemantik bestimmt die Transaktionsgrenzen. Dies kann beispielweise über Compiler-Direktiven oder Funktionsaufrufe (siehe Kapitel 2.8.1) erfolgen. Damit weiß das System, welche Speicheroperationen zu einer Transaktion zu bündeln sind.

2.3.2 Schattenkopien

Basieren Transaktionen auf einer optimistischen Synchronisierung (siehe Kapitel 2.5.3), finden Änderungen an Objektreplikaten bereits statt, bevor bekannt ist, ob eine Transaktion erfolgreich abschließen kann. Muß ein Knoten seine Transaktion wegen eines Konflikts abbrechen, so muß er den ursprünglichen Speicherinhalt wiederherstellen. Vor dem ersten Schreibzugriff auf ein Objekt innerhalb einer Transaktion legt das System davon zunächst eine Schattenkopie an. Somit kann der Knoten, der seine Transaktion wegen eines Transaktionskonflikts abbrechen muß, den Ursprungszustand bereits geänderter Objekte mithilfe ihrer Schattenkopien wiederherstellen. Im Endeffekt hat die nicht abgeschlossene Transaktion keine Änderung an dem transaktionalen Speicher durchgeführt.

Das System kann nach dem Anlegen einer Schattenkopie prinzipiell auf der neu angelegten Kopie arbeiten. Es kann die Änderungen an einem transaktionalen Objekt also dort oder auf dem originalen Objektreplikat durchführen. Beide Vorgehensweisen erlauben es einem Knoten, die temporären Änderungen einer abzubrechenden Transaktion rückgängig zu machen. In Anbetracht der Adressierung von Speicherinhalten bestehen diesbezüglich Einschränkungen. Es ist unkomplizierter, vorläufige Änderungen an Objekten einer laufenden Transaktion an der originalen Version und nicht an der Schattenkopie durchzuführen. Führt der transaktionale Programmcode – innerhalb von Transaktionsgrenzen ausgeführter Programmcode – Änderungen an dem originalen Objektreplikat durch, behalten alle darauf verweisende Zeiger ihre Gültigkeit. Dies wäre bei der neu angelegten Objektkopie nicht der Fall, da diese an einer anderen Adresse im Speicher liegt.

Ändert man die Zeiger beziehungsweise Referenzen, die auf das originale Replikat zeigen, auf die Schattenkopie ab, kann der transaktionale Programmcode auch auf der Objektkopie arbeiten. Dies ist beispielsweise bei typsicheren Programmiersprachen, bei denen Objektreferenzen einer mehrfachen Indirektion (Zeiger auf Zeiger) entsprechen, möglich. Allerdings sollten die programmiersprachlichen Objekte bezüglich Größe und Lage im Speicher mit den transaktionalen Objekten deckungsgleich sein. Ansonsten nimmt die Komplexität zu, da ansonsten die Schattenkopien mehrerer aufeinanderfolgender transaktionaler Objekte auch aufeinanderfolgend im Speicher liegen müssen und gebündelt anzulegen sind, sofern sich das programmiersprachliche Objekt über mehrere transaktionale Objekte erstreckt. Gerade bei nicht typsicheren Programmiersprachen ist die Arbeit auf Schattenkopien nicht möglich, da sich zur Laufzeit nicht alle darauf zeigenden Referenzen identifizieren lassen.

Arbeitet ein Knoten auf dem originalen Replikat, muß er bei einer abzubrechenden Transaktion in einer weiteren Kopieroperation die Schattenkopie an die Adresse des originalen Replikats zurückkopieren. Allerdings entfällt in diesem Fall eine komplizierte Anpassung von Zeigern, als wenn der Knoten auf der Schattenkopie arbeiten würde. Zudem müßte der Knoten im letzteren Fall die Zeiger im Abbruchfall der Transaktion erneut anpassen, damit sie wieder auf das originale Replikat verweisen. Bei Arbeit auf den originalen Replikaten kann ein Zurückkopieren der Schattenkopie entfallen. Stattdessen kann das System in der Speicherverwaltungseinheit des Prozessors den Verweis der virtuellen Speicherseite des transaktionalen Objekts auf die physische Kachel der Schattenkopie abändern. Das bedingt jedoch, daß eine Speicherseite von höchstens einem transaktionalen Speicherobjekt belegt ist. Ein transaktionales Speicherobjekt darf aber uneingeschränkt mehrere Speicherseiten belegen. Diese Art der Manipulation ist nur auf Ebene des Betriebssystemkerns und nicht aus dem Anwendungskontext möglich, so daß Teile einer Implementierung des Systems im Betriebssystemkern erfolgen muß.

2.4 Speicherzugriffserkennung

Die *Speicherverwaltungseinheit*, in der Literatur als Memory Management Unit (MMU) bezeichnet, ist eine Logikeinheit von Prozessoren und unterstützt die Betriebssysteme in ihrer Speicherverwaltung. Hauptzweck ist die Regelung des Zugriffs auf den Arbeitsspeicher und Virtualisierung von Speicheradressen, um dem Betriebssystem und dessen Anwendungen einen größeren virtuellen Speicher zur Verfügung zu stellen, als physisch tatsächlich zur Verfügung steht. Die Idee, einen gegenüber dem physischen Speicher größeren virtuellen Speicher zu nutzen, wurde erstmals von John Fotheringham vorgestellt [38]. Ist die Speicherverwaltungseinheit aktiviert, erfolgt die Codeausführung statt im physischen nunmehr in einem virtuellen Adreßraum. Mittels Tabellen bildet die Speicherverwaltungseinheit die beiden Adreßräume aufeinander ab. Die Speicherverwaltungseinheit ist je nach verwendeter Prozessorarchitektur unterschiedlich aufgebaut und kann daher unterschiedlich organisiert sein.

Da dieser Entwicklung die *IA-32-* und *x86-64-Architekturen* mit dem Betriebssystem *Linux* zugrundeliegen, beziehen sich etwaige im folgenden betrachteten architekturspezifischen Eigenschaften auf diese Hardware und das oben genannte Betriebssystem. IA-32 und x86-64 sind Bezeichnungen für unterschiedliche Ausprägungen der x86-Architektur. IA-32 bezeichnet die 32-Bit-Architektur, x86-64 ist dagegen eine allgemeine Bezeichnung für die 64-Bit-Erweiterungen der IA-32-Architektur, die bei den Prozessorherstellern auch unter den Bezeichnungen *Intel 64* und *Amd 64* bekannt ist.

Die MMU unterteilt den virtuellen und physischen Adreßraum in Bereiche gleicher Größe. Zur klaren Abgrenzung voneinander bezeichnet man die Bereiche des virtuellen Adreßraums als *Seiten*, die des physischen dagegen als *Kacheln*. Bei der x86-Architektur bildet die MMU mittels einer Seitentabelle die Speicherseiten des virtuellen Adreßraumes surjektiv auf die Kacheln des physischen Adreßraums ab (siehe Abbildung 2.2) [53]. Die Standardgröße für Seiten und Kacheln, welche auch das Betriebssystem Linux verwendet, beträgt 4 KB[1]. Die Abbildung von Seiten auf Kacheln steuert das Betriebssystem.

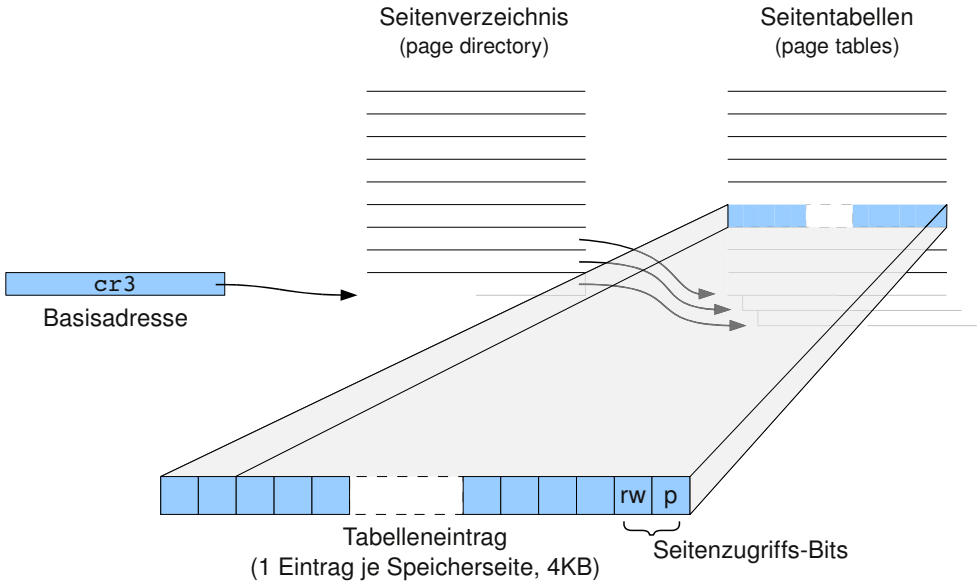


Abbildung 2.2: Virtuelle Speicherverwaltung der x86-Architektur.

[1] Die x86-Architektur unterstützt Seitengrößen von 4 KB (Kilobyte), 2 MB (Megabyte) und 4 MB. Das Betriebssystem Linux verwendet auf der x86-Architektur eine einheitliche Seitengröße von 4 KB.

Jede Seite ist individuell über Zugriffsrechte kontrollierbar. Dies ist notwendig, da das Betriebssystem wegen des im Vergleich zum physischen größeren virtuellen Speichers nicht alle Seiten gleichzeitig auf Kacheln abbilden kann. Mit diesem Mechanismus kann das Betriebssystem Speicherinhalte von länger ungenutzten Kacheln auf Festwertspeicher auslagern, um deren zugewiesenen physischen Speicher anderen Prozessen zur Verfügung zu stellen, indem es angeforderte auf dem Festwertspeicher ausgelagerte Inhalte in beliebige Kacheln wieder einlagert. So kann das Betriebssystem insgesamt mehr Speicher nutzen, als physisch vorhanden ist. Mittels der Steuerung von Zugriffsrechten und Verletzung dieser Rechte kann sich das Betriebssystem über einen Zugriffsversuch auf eine Seite informieren lassen und vor dessen Ausführung eingreifen. Nebenbei benutzen moderne Betriebssysteme diesen Mechanismus ebenfalls, um Prozesse vor gegenseitigen Speicherzugriffen voneinander abzusichern [98]. Tritt eine Zugriffsverletzung auf eine Seite auf, informiert der Prozessor das Betriebssystem durch Auslösen einer *Exception*.

2.4.1 Exceptions

Bei Exceptions handelt es sich um spezielle Unterbrechungsanforderungen (Interrupts) des Prozessors, welche die Hilfe des Betriebssystems für Ausnahmesituationen anfordern, die der Prozessor selbst nicht behandeln kann. Exceptions unterbrechen die aktuelle Codeausführung, um eine Betriebssystemfunktion aufzurufen. Nach Abbarbeitung der Interruptroutine führt der Prozessor die Codeausführung fort, an der er sie zuvor unterbrochen hat. Dieses Verfahren findet beispielsweise auch bei der virtuellen Speicherverwaltung Anwendung.

Die MMU erlaubt für jede Seite, den Zugriff über das *Present-Bit* (*p*-Bit in Abbildung 2.2) vollständig zu unterbinden. Das Betriebssystem verwendet dieses Bit zur Erkennung nicht vorhandener oder ausgelagerter virtueller Speicherseiten (gelöschtes Bit). Weiterhin kann das Betriebssystem den Zugriff auf die Seite über das *Read/Write-Bit* (*rw*-Bit in Abbildung 2.2) auf Lesezugriffe einschränken. Ist das Bit gelöscht, ist die Seite nur lesbar, und jeder Schreibzugriff löst eine Exception aus [53].

Tritt eine Zugriffsverletzung auf, so löst der Prozessor automatisch die *Pagefault*-Exception aus und ruft die damit verbundene Funktion (Handler) im Betriebssystem auf. Die Exception liefert dem Betriebssystem weitere Informationen, die Aufschluß über die Zugriffsverletzung geben. Dies sind zum einen die Speicheradresse, welche die Zugriffsverletzung ausgelöst hat, und zum anderen der Fehlercode, der Information über die Art der Zugriffsverletzung (Present- oder Read/Write-Eigenschaft) beinhaltet. Die Pagefault-Exception gehört zur Klasse der *Faults*. Dies bedeutet, der Prozessor führt die Maschineninstruktion, welche die Exception ausgelöst hat, nach Beendigung des Handlers automatisch erneut aus. Dies ist einerseits notwendig, damit der Fehler den Programmfluß nicht verändert. Andererseits stellt diese Exception einen durch das Betriebssystem isoliert korrigierbaren Fehler da, der keinen weiteren Einfluß auf den verursachenden Programmkontext ausübt. Dies hat zur Folge, daß der Exception-Handler die angeforderten Zugriffsrechte vor seiner Beendigung gewähren muß, da die Exception ansonsten erneut auslösen würde.

Das Linux-Betriebssystem stellt den Mechanismus zur Kontrolle der Zugriffsrechte über eine weitestgehend hardwareunabhängige Schnittstelle Anwendungen im Userspace zur Verfügung. Anwendungen können über die Betriebssystemfunktion `mprotect()` die Zugriffsrechte selbst steuern. Die Zugriffsrechte des Betriebssystemkerns bleiben davon unberührt. Der Kern bildet die im Userspace gesetzten Rechte auf die Seitenrechte in der Seitentabelle ab. Ist eine ausgelagerte Seite für einen ausgelösten Seitenfehler nicht ursächlich, kann die Ursache aber dem Userspace zuordnen, so schickt er ein Systemsignal (siehe Kapitel 2.4.2) an den verursachenden Prozeß. Es ist zu beachten, daß ein Zugriff auf nicht abgebildete (ungültige) Speicherseiten ebenfalls ein Systemsignal an den Prozeß generieren.

Seitenzugriff	Bits in Seitentabelle			
	ø	rw	p	p+rw
Lesen	–	–	✓	✓
Schreiben	–	–	–	✓

Tabelle 2.1: Seitenzugriff in Abhängigkeit gesetzer Zugriffsbits in der Seitentabelle.

2. Zugriff	1. Zugriff		
	kein Zugriff	Lesezugriff	Schreibzugriff
Lesen	✓	–	–
Schreiben	✓	✓	–

Tabelle 2.2: Erkennung aufeinanderfolgender Seitenzugriffe durch Exception-Handler.

Tabelle 2.1 zeigt, welche Seitenzugriffe in Abhängigkeit der für eine Seite gesetzten Zugriffsbits erlaubt sind, *rw* steht für das Read/Write-Bit und *p* für das Present-Bit. Ist kein oder nur das Read/Write-Bit gesetzt, sind sowohl Lese- als auch Schreibzugriffe auf die Seite nicht gestattet und lösen eine Exception aus. Das Present-Bit hat gegenüber dem Read/Write-Bit eine höhere Priorität, daher hat das Read/Write-Bit keinen Einfluß, wenn das Present-Bit nicht gesetzt ist. Ist allein das Present-Bit gesetzt, sind Lesezugriffe erlaubt. Schreibzugriffe lösen dagegen eine Exception aus. Sind beide Zugriffsbits gesetzt, sind Lese- und Schreibzugriffe erlaubt. Durch diese Abbildung von Zugriffsrechten lassen sich nicht alle Kombinationen von Seitenzugriffen eindeutig feststellen, da die Speicherverwaltungseinheit einen Schreibzugriff auf eine Seite immer nur zusammen mit einem Lesezugriff gewähren kann (siehe Tabelle 2.2).

Tabelle 2.2 zeigt auf, daß die Speicherverwaltungseinheit einen aufeinanderfolgenden ersten Lese- und Schreibzugriffe auf dieselbe Speicherseite nur erkennen kann, falls der Lesezugriff vor dem Schreibzugriff stattgefunden hat. Ein Lesezugriff läßt sich nach einem ersten Schreibzugriff nicht erkennen, da die Schreibrechte gleichzeitig Leserechte implizieren. Daher muß der transaktionale Speicher bei einem Schreibzugriff auf ein transaktionales Objekt immer davon ausgehen, daß auf dieses auch Lesezugriffe stattgefunden haben (Hidden Read).

2.4.2 Systemsignale

Signale in Unix-Betriebssystemen sind Nachrichten, die entweder der Betriebssystemkern an Prozesse oder Prozesse sich untereinander zuschicken können und synchron oder asynchron auftreten. Signale können im Fall von Multithreading-Prozessen statt des gesamten Prozesses auch nur einzelne Threads betreffen. Dies hängt davon ab, ob das vom Kern gesendete Signal synchron oder asynchron ist beziehungsweise einen fatalen Fehler identifiziert. Synchrone Signale treten synchron zur Programmausführung eines Threads auf und sind daher zumeist einem bestimmten Thread zugeordnet. Asynchrone Signale dagegen lassen sich oftmals keinem Thread zuordnen, daher liefert der Kern diese entweder an den gesamten Prozeß oder an einen beliebigen Thread in der Threadgruppe eines Prozesses aus. Unter anderem kann der Betriebssystemkern Exceptions, die im Kernkontext laufen, über Signale an Prozesse weiterleiten, wie dies auch bei einer anwendungsgesteuerten (`mprotect()`) Zugriffsverletzung auf Speicherseiten der Fall ist. Löst der Prozessor beispielsweise eine Pagefault-Exception aufgrund einer durch die Anwendung kontrollierten Zugriffsrechteverletzung oder eines ungültigen Speicher-

zugriffs aus, gibt der Kern dieses Ereignis in Form eines Signals vom Typ *SIGSEGV (Signal – Segmentation Violation)* an den Prozeß weiter. Da das Signal synchron zur Programmausführung innerhalb eines Threads auftritt, ist das Signal nur dem verursachenden Thread zugeordnet und unterbricht diesen. Alle anderen Threads laufen dagegen im allgemeinen weiter. Da der gesamte Signalmechanismus in Linux sehr komplex ist, sei diesbezüglich auf weiterführende Literatur verwiesen [20, 76].

Jedes an einen Prozeß ausgelieferte Signal führt standardmäßig eine vom System vordefinierte Operation aus. Beim SIGSEGV-Signal ist dies die Beendigung des Prozesses. Für bestimmte Signale erlaubt das Betriebssystem, diese Operation mit einem eigenen Handler zu überschreiben. So kann ein Prozess das SIGSEGV-Signal abfangen und so anstatt der Terminierung eine eigene Funktion (im verursachenden Threadkontext) aufrufen. In Zusammenhang mit der vom Betriebssystem gestatteten Kontrolle von Zugriffsrechten für prozeßeigene Speicherseiten kann ein Prozess individuell auf eine Verletzung der Zugriffsrechte reagieren. Die Systemfunktion `sigaction()` installiert einen benutzerdefinierten Signalhandler, dargestellt in Abbildung 2.3. Das erste Argument übergibt die Signalnummer, die den Handleraufruf verursacht hat. Das zweite Argument liefert einen Zeiger auf eine Datenstruktur, die im Kontext des SIGSEGV-Signals die Art der Zugriffsverletzung (lesend oder schreibend) und die zugegriffene Speicheradresse enthält. Das dritte Argument kann optional einen Zeiger auf eine Datenstruktur liefern, welche den aktuellen Anwendungskontext – vor Unterbrechung durch das Signal – enthält.

```
1  ...
2  void (*handler)(int, siginfo_t *, void *);
```

Abbildung 2.3: Prototyp einer Callback-Funktion für einen Signalhandler.

Die Eigenschaften der Pagefault-Exception (Fault-Klasse) spiegeln sich auch in dem SIGSEGV-Signalhandler im Userspace wider. Dieser muß also im Falle einer Zugriffsrechteverletzung vor seiner Beendigung entweder die entsprechenden Zugriffsrechte gewähren, da er ansonsten erneut auslöst, oder entsprechend seinem Standardverhalten den Prozeß beenden. Für den transaktionalen Speicher kommt eine Beendigung des Prozesses nur dann in Betracht, wenn der Handler einen unerwünschten Zugriff außerhalb des gemeinsamen verteilten transaktionalen Speichers feststellt. Dies kann etwa bei einem Zugriff auf eine nicht vorhandene Speicherseite wie eine Dereferenzierung eines Null-Zeigers auftreten. Dieser Mechanismus erlaubt es, transaktionale Objektzugriffe getrennt nach Lese- und Schreibzugriffen mithilfe der Prozessorhardware aufzuzeichnen, unter Einschränkung der in Tabelle 2.2 dargestellten Zugriffserkennung (siehe Abbildung 2.4).

2.4.3 Eintrittsinvarianz

Ein auftretendes Signal kann einen Prozeß beziehungsweise Thread an beliebigen Stellen in der Codeausführung unterbrechen. So unterbricht ein Signal die Programmausführung etwa auch, während sich der Prozeß oder Thread in System- oder Bibliotheksfunktionen befindet. Aus diesem Grund darf ein Signalhandler nicht beliebige Funktionen aufrufen, da es sonst entweder zur Verklemmung oder zu einem undefinierten Verhalten kommen kann. Dies ist dadurch begründet, daß die unterbrochene und die durch den Handler gerufene Funktion möglicherweise unkontrolliert auf gemeinsamen Datenbereichen lesen und schreiben.

Daher darf ein Signalhandler nur wiedereintrittsfähige Funktionen[2] aufrufen, da bei diesen

[2] *Eintrittsinvarianz/Wiedereintrittsfähigkeit*: Eine Funktion kann in demselben Thread mehrfach asynchron rekursiv aufgerufen werden, ohne Auswirkung auf ihr Verhalten.

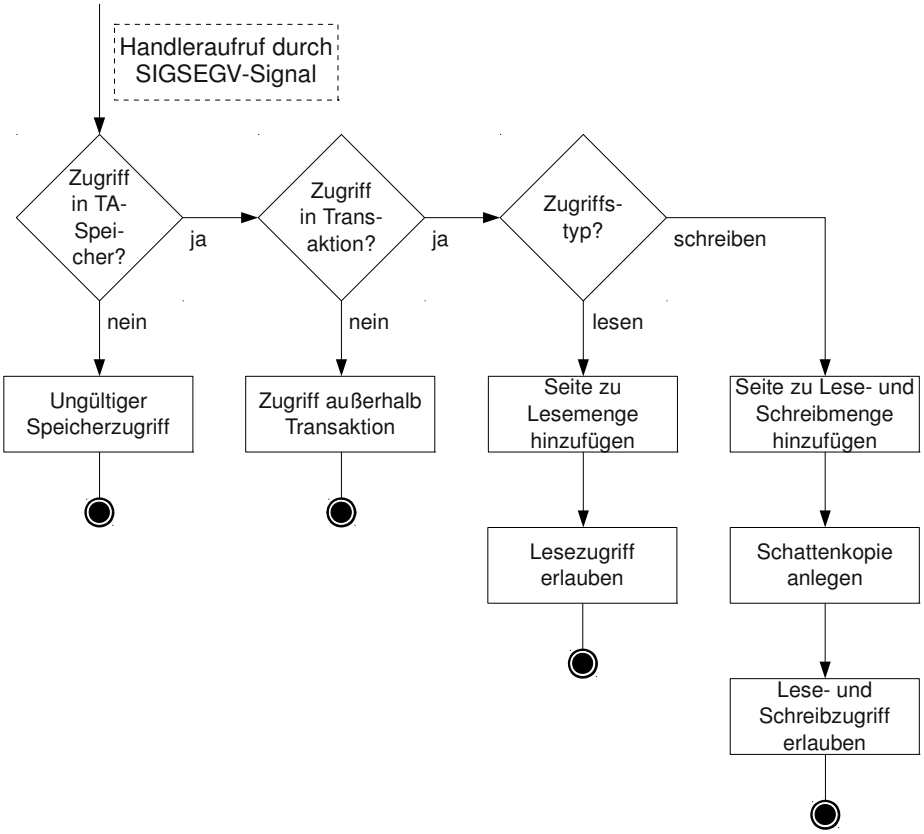


Abbildung 2.4: Schematischer Ablauf des Signalhandlers bei Auftreten eines SIGSEGV-Signals.

Funktionen garantiert ist, daß diese unabhängig von dem unterbrochenen Threadkontext nicht auf gemeinsame Daten zugreifen. Daher können diese Funktionen keine Verklemmung oder undefiniertes Programmverhalten verursachen [64]. Solche Funktionen bezeichnet man auch als *Async signal safe*. Es ist zu beachten, daß diese Funktionen threadsicher, das heißt für Multithreading verwendbar sind, der umgekehrte Fall gilt jedoch nicht uneingeschränkt.

2.4.4 Transaktionale Objektzugriffserkennung

Bei der transaktionalen Objektzugriffserkennung über den Signalhandler registriert das System nach Lese- und Schreibzugriffen getrennt, welche Objekte eine Transaktion innerhalb ihrer Transaktionsgrenzen zugreift. Die Transaktionsobjekte des von der Anwendung allozierten transaktionalen Speicherbereiches sind standardmäßig zugriffsgeschützt. Bei einem erstmaligen Zugriff innerhalb einer Transaktion löst daher zunächst der Betriebssystemkern das SIGSEGV-Signal aus und ruft den dafür registrierten Handler im Userspace auf. Der Handler prüft zunächst, ob gegenwärtig eine Transaktion läuft. Läuft eine Transaktion, und liegt der Zugriff in einem transaktionalen Speicherbereich, so fügt der Handler das Objekt seiner Lesemenge *(engl. read set)* und eventuell seiner Schreibmenge *(engl. write set)* (siehe Abbildung 2.5[3]) hinzu. Wegen der Eigenschaften der reihenfolgebasierten Zugriffserkennung durch die Speicherverwaltungseinheit (siehe Tabelle 2.2) ist jedes Objekt in der Schreibmenge ebenso

[3]Zustände: Objekt ungebunden *(U)*, Objekt in Lesemenge *(R)*, Objekt in Lese- und Schreibmenge *(RW)*.

in der Lesemenge der Transaktion enthalten. Bei einem Schreibzugriff erstellt der Handler vor dessen Ausführung noch eine Schattenkopie (siehe Kapitel 2.3.2), um im Fall eines Transaktionsabbruchs den originalen Objektinhalt wiederherstellen zu können. Vor seiner Beendigung gewährt der Handler dem Objekt die durch das Signal angeforderten Zugriffsrechte, Leserechte bei einem Lesezugriff und vollen Zugriff (lesen/schreiben) bei einem Schreibzugriff. Kosten der Zugriffserkennung entstehen daher nur beim erstmaligen Zugriff auf ein Objekt innerhalb einer Transaktion, nachfolgende Zugriffe laufen mit voller Geschwindigkeit ab.

Die inverse Verfahrensweise, den Zugriffsschutz für alle Seiten beim Beginn einer Transaktion zu setzen und nach einem Commit wieder zu entfernen, ist dem Verfahren mit standardmäßig zugriffsgeschützten Seiten gleichwertig. Allerdings ist dieses Verfahren teurer, da das System bei Beginn und Ende einer Transaktion bei nahezu allen Speicherseiten die Zugriffsrechte ändern muß. Eine Transaktion greift in der Regel aber nicht alle Seiten des transaktionalen Speichers zu. Daher muß das transaktionale Speichersystem bei dem ersteren der beiden Verfahren die Zugriffsrechte von nur wenigen Speicherseiten ändern. Außerdem kann das zweite Verfahren ungültige Zugriffe außerhalb von Transaktionen nur erkennen, falls in einem weiteren Thread eine Transaktion läuft und diese das außerhalb der Transaktion zugegriffene Objekt selbst noch nicht angetastet hat. Daher ist das erste Verfahren gegenüber dem zweiten geeigneter, auch wenn das zweite Verfahren bei einem gestatteten transaktionalen Objektzugriff außerhalb von Transaktionen effizienter ist, da der Zugriff dort meistens keine Zugriffsverletzungen auslöst.

Angesichts der hier verwendeten Hardwarearchitektur lassen sich Objektzugriffe nur mit der durch die MMU und dem Betriebssystem fest vorgegebenen Granularität der Speicherseiten (4 KB) erkennen. Folglich unterliegt das minimale Erkennungsraster dieser Limitierung, um Zugriffe auf Objekte eindeutig voneinander unterscheiden zu können. Eine Zugriffserkennung für kleinere transaktionale Objekte läßt sich nur erreichen, wenn jedes Objekte einer disjunkten Speicherseite zugeordnet wird. Dies ist indessen mit dem Nachteil verbunden, daß die Objekte nicht mehr fortlaufend im Speicher vorliegen und Lücken im Speicher eine Verschwendung physischen Speichers darstellen. Aufgrund der Surjektivität der Speichervirtualisierung ist mit dem Ansatz nach Itzkovitz et al. über eine mehrfache Abbildung virtueller Speicherseiten auf dieselbe Speicherkachel auch eine beliebig kleine Granularität erreichbar, ohne physischen Speicher zu verschwenden [55]. Die weiterhin bestehende Verschwendung virtuellen Speichers ist unter Verwendung von 64-Bit Systemen vernachlässigbar. Allerdings läßt sich das Problem mit den Lücken zwischen den feingranularen Objekten im Speicher mit diesem Ansatz nicht lösen. Dies ist nur durch eine feingranularere Hardwareerkennung möglich. Weiterhin ist zu beachten, daß der Ansatz von Itzkovitz durch die feinere Garanularität mehr Seitenfehler und eine schlechtere Bilanz in der Ausnutzung des *Translation Lookaside Buffers (TLB)* [53] verursacht, die im Vergleich zur Netzwerklatenz allerdings vernachlässigbar ist.

2.4.5 Objektzugriffserkennung in Multithreadingumgebungen

Viele der heutigen modernen Betriebssysteme unterstützen neben mehreren Prozessen auch präemptives Multithreading von Anwendungen. Die nebenläufige Ausführung von Prozessen und Threads gewinnt aufgrund der stetig steigenden Leistungen der Prozessoren, aber insbesondere durch die steigende Anzahl an Multiprozessorsystemen oder Systemen mit Mehrkernprozessoren mehr an Bedeutung. Aus diesem Grund sind nebenläufige transaktionale Speicherzugriffe in Multithreadingprozessen von wesentlicher Bedeutung hinsichtlich Leistung und Skalierbarkeit.

Bei der Objekterkennung gilt es allerdings zu beachten, daß sich alle Threads eines Prozesses denselben Adreßraum teilen. Dies begründet eine gemeinsame Speichervirtualisierung für alle Threads eines Prozesses durch die MMU. Demzufolge gelten Zugriffsberechtigungen für

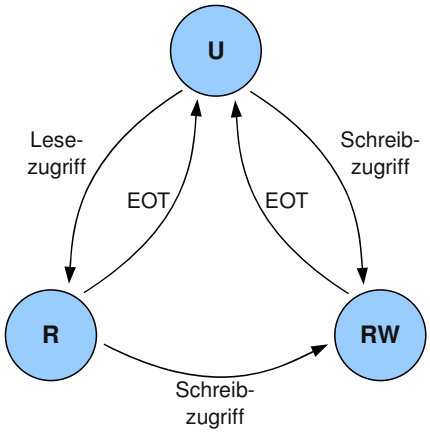


Abbildung 2.5: Zustandsüberführungsgraph eines Transaktionsobjekts während eines Zugriffs innerhalb einer Transaktion mittels MMU-Speicherzugriffserkennung.

beliebige Speicherseiten des Prozesses auch für alle seine Threads. Dies wiederum führt zu einigen Einschränkungen bei der transaktionalen Objekterkennung in Multithreadingprozessen. Für die Knoten des transaktionalen Speichers gilt diese Einschränkung nicht, da dort eigenständige Prozesse laufen, die ihrer eigenen Speichervirtualisierung unterliegen. Für die Unterstützung von Multithreading ist die Transaktionsprotokollierung (Transaktionsgrenzen und Objektzugriffserkennung) an den ausführenden Thread gekoppelt. So lassen sich die zugegriffenen Objekte der Transaktion zuordnen, die zu demselben Thread gehört, ober Objektzugriffe in einem Thread außerhalb der Transaktionsgrenzen erfolgt.

Wie bereits diskutiert, erfolgt eine Zugriffserkennung innerhalb einer Transaktion nur für den ersten Zugriff auf ein transaktionales Speicherobjekt und bei aufeinanderfolgenden Zugriffen nur bei einem Schreibzugriff, der nach einem Lesezugriff auftritt. Somit ist eine eindeutige Zugriffserkennung desselben Objekts durch zeitlich überlappende Transaktionen innerhalb mehrerer Threads desselben Prozesses nicht möglich. Hat beispielsweise eine laufende Transaktion TA_1 im Thread T_1 laufend ein Objekt x_1 gelesen, und greift eine Transaktion TA_2 im Thread T_2 gleichfalls auf dieses Objekt zu, so erfolgt eine Zugriffserkennung nur für einen Schreibzugriff.

Liest TA_2 das Transaktionsobjekt x_1, führt das System diesen problemlos aus, die Zugriffserkennung registriert dies nicht und nimmt dieses Objekt auch nicht in die Lesemenge von TA_2 auf. Allerding kann TA_2 die Datenkonsistenz verletzen, falls dieses Objekt in TA_1 konfliktbehaftet (veraltet) ist. Basieren geschriebene Objekte der zweiten Transaktion auf gelesenen veralteten Daten des bereits in der Transaktion TA_1 gelesenen Objekts, wäre dies ein Konflikt, den das System allerdings nicht erkennt, da sich das veraltete Objekt nicht in der Lesemenge der Transaktion TA_2 befindet.

Ähnlich verhält es sich mit Schreibzugriffen. Versucht TA_2 auf das Objekt x_1 zu schreiben, so registriert dies die Objektzugriffserkennung entsprechend Tabelle 2.2. Das System stellt ebenfalls fest, daß zwei verschiedene laufende Transaktionen dieses Objekt gleichzeitig verwenden. Wegen auch hier möglicher Konsistenzverletzungen muß das System diesen Zugriff unterbinden und den Prozeß beenden, da sich die Operation nicht rückgängig machen läßt.

Schreibt Transaktion TA_1 das Objekt x_1, und greift TA_2 dieses Objekt zu, erkennt es dies unabhängig vom Lese- oder Schreibzugriff nicht. Hier kann es ähnlich wie im ersten Fall zur Konsistenzverletzung kommen. Daher ist es einer Anwendung nicht gestattet, in überlappenden Transaktionen in demselben Prozess auf dieselben Objekte zuzugreifen. Die Vorgaben einzuhalten, liegt in der Verantwortung des Anwendungsentwicklers.

Eine Objekterkennung pro Thread ließe sich technisch realisieren, wenn das System die Zugriffsrechte für eine Speicherseite pro Thread steuern kann. Dies wäre allerdings mit sehr viel Aufwand und großen Eingriffen in den Betriebssystemcode verbunden, da jeder Thread seine eigene Seitentabelle besitzen und der Dispatcher beim Threadscheduling die Seitentabelle umschalten müßte. Weiterhin müßte das System alle Seitentabellen eines Prozesses bezüglich Speicherallozierungen, -freigaben und -auslagerungen konsistent halten. Alternativ könnte der Threadscheduler die modifizierten Seitenzugriffsrechte bei einer Threadumschaltung sichern und wiederherstellen. Beide Verfahren sind sehr teuer. Sie erlauben zwar eine aufwendige aber eindeutige Objekterkennung zwischen den Threads eines Prozesses, meistens sind solche Zugriffe aber immer die Ursache für Transaktionskonflikte. Zukünftig läßt sich diese Einschränkung möglicherweise mit besser etablierten Transaktionsmechanismen für SMP-Systeme (engl. Symmetric Multi-Processing) umgehen, dies liegt aber nicht im Fokus dieser Arbeit und findet daher keine weitere Berücksichtigung.

2.5 Transaktionsvalidierung

Wichtig bei überlappenden Transaktionen ist, Konflikte zuverlässig zu erkennen, um eine mögliche Verletzung der Datenkonsistenz aufgrund veralteter Daten zu vermeiden. Zunächst bedarf es einer genauen Definition des Begriffs *überlappend*, welcher im Kontext von Transaktionen eine zeitliche Überschneidung meint. Bei dem replikatbasierten transaktionalen Speichersystem können je nach Validierungsstrategie auch Transaktionen in Konflikt stehen, die sich zeitlich nicht überlappen, aber über unterschiedliche Versionen desselben Objekts miteinander in Beziehung stehen. Daher steht dieser im weiteren Verlauf dieser Arbeit gebrauchte Begriff auch für zeitlich nicht überlappende aber über Objektreplikate miteinander in Beziehung stehende Transaktionen. Endet eine Transaktion, so folgt anschließend die Validierungsphase, in der eventuelle Transaktionskonflikte geprüft werden. Die weitere Diskussion setzt an dieser Stelle voraus, daß eine Transaktion bei einem ersten Objektzugriff immer die zuletzt festgeschriebene Objektversion verwendet (Ausnahmen hierzu siehe Kapitel 6.1). Ist eine Transaktion konfliktfrei, so kann diese abschließen. Besteht dagegen ein Konflikt, bestimmt das Commit-Protokoll und dessen Strategie, welche der konkliktbehafteten Transaktionen abzubrechen sind. Eine abschließende Transaktionsvalidierung ist nur bei optimistischen Synchronisierungverfahren notwendig, nicht jedoch bei Verfahren, die auf Sperren basieren. Optimistische Synchronisierungsverfahren eignen sich besser für den verteilten Speicher (siehe Kapitel 2.5.2). Zunächst folgt zum Vergleich ein kurzer Überblick über pessimistische Sperrverfahren, damit die Vorteile der optimistischen Ansätze deutlich werden.

Greifen überlappende Transaktionen auf dieselben Objekte zu, kann es zu Transaktionskonflikten kommen, welche sie vor ihrem Commit auflösen müssen. Konflikte können als *Lese-Schreib-Konflikte* und *Schreib-Schreib-Konflikte* auftreten. Lesezugriffe auf gemeinsame Objekte allein führen dagegen nicht zu Konflikten. Ein Lese-Schreibkonflikt tritt auf, falls eine Transaktion TA_1 Daten liest, die eine andere Transaktion TA_2 schreibt und die schreibende Transaktion zuerst abschließt. In diesem Fall hätte TA_1 veraltete Daten gelesen und verursacht einen Konflikt mit der schreibenden Transaktion (siehe Abbildung 2.6)[4]. Zur Konfliktlösung muß eine der beiden Transaktionen abbrechen. Welche der Transaktionen zur Konfliktlösung abbricht, bleiben der Validierungsstrategie und dem Commit-Protokoll vorbehalten. Ein Schreib-Schreib-Konflikt kann im replikatbasierten verteilten Speicher im Gegensatz zu Systemen, die auf einem gemeinsamen physischen Speicher arbeiten – wenn zwei Transaktionen TA_1 und TA_2 gleichzeitig ihre Validierungsphase ausführen und die Transaktion festschreiben, nicht auftreten [27]. Die im weiteren Verlauf der Arbeit vorgestellten Commit-Protokolle führen zudem ausnahmslos eine strenge Serialisierung durch, daher kann der genannte Schreib-

[4] *BOT/EOT:* Transaktionsgrenzen, *r(x)/w(x):* Lesen beziehungsweise schreiben des Objekts *x*.

Schreib-Konflikt nicht eintreten [27]. Allerdings können Schreib-Schreib-Konflikte dennoch auftreten, da Schreibzugriffe bei der Zugriffserkennung gleichzeitig Lesezugriffe implizieren (siehe Kapitel 2.4.1 und 2.5.1).

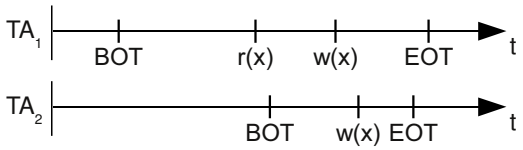


Abbildung 2.6: Lese-Schreib-Konflikt: TA_1 verwendet veraltete Daten (x), wenn TA_2 abschließt.

2.5.1 Partiell geschriebene Transaktionsobjekte

Auch wenn Schreib-Schreib-Konflikte wegen der strengen Transaktionsserialisierung nicht auftreten, gilt dies jedoch nicht für den gemeinsamen verteilten transaktionalen Speicher. Einerseits arbeitet dieser mit Objektreplikaten. Andererseits kann die Speicherzugriffserkennung nicht feststellen, welche Bereiche eines transaktionalen Speicherobjekts eine Transaktion geändert, noch ob sie das Objekt vollständig beschrieben hat. Die folgende Abbildung 2.7 zeigt einen Schreib-Schreib-Konflikt von zwei überlappenden Transaktionen auf unterschiedlichen Rechnern, die dasselbe Objekt partiell beschreiben. Der Konflikt tritt auf, da sich die modifizierten Replikate des Objekts nicht fehlerfrei miteinander synchronisieren lassen. Die objektbasierte Synchronisierung (siehe Abbildung 2.7a) synchronisiert den vollständigen Objektinhalt. Da der Schreibzugriff in der zweiten abzuschließenden Transaktion den zuerst modifizierten Objektinhalt nicht vollständig überdeckt, würde die Synchronisierung geschriebene Daten des ersten Commits mit veralteten Daten der beim Transaktionsbeginn vorliegenden Objektversion überschreiben. In ähnlicher Weise verhält es sich, wenn die Objektsynchronisierung wie in Abbildung 2.7b differentiell (Unterschiede zwischen der Schattenkopie des Objekts und dessen modifizierten Version) erfolgt. Dieses Verfahren löst zwar das Problem, daß veraltete Daten neuere überschreiben, kann aber ebenso zur Dateninkonsistenz führen. Ist ein Teil der geschriebenen Daten deckungsgleich mit denen des Ausgangsobjekts, kann die Diff-Operation nicht vollständig erkennen, welche Daten die Transaktion geschrieben hat. Demzufolge bleiben beim zweiten Commit Daten vom ersten Commit bestehen, obwohl die zuletzt abgeschlossene Transaktion diese eigentlich überschrieben hat.

Daher würde der verteilte transaktionale Speicher nicht davon profitieren, wenn die Zugriffserkennung auf Basis der prozessoreigenen Speicherverwaltungseinheit Lese- und Schreibzugriffe auf eine Speicherseite vollständig voneinander getrennt, sprich Lesezugriffe nach der Zuteilung von Schreibrechten, erkennen könnte.

2.5.2 Pessimistische Sperrverfahren

Pessimistische Sperrverfahren gehen davon aus, daß Transaktionen relativ häufig Konflikte auslösen. Daher versucht diese Methode Konflikte im Vorfeld zu vermeiden. Jedes in einer Transaktion verwendbare Objekt besitzt eine *Sperre (Lock)*, welche eine Transaktion vor der Verwendung des Objekts anfordern muß. Eine Transaktion ist dabei in zwei Phasen unterteilt, in der ersten fordert sie zunächst Sperren an, in der zweiten gibt sie diese wieder Frei. Eine Transaktion darf keine weiteren Sperren mehr anfordern, sofern sie mindestens eine bereits freigegeben hat [35]. Für mehr Nebenläufigkeit sind Sperren in Lese- und Schreibsperren

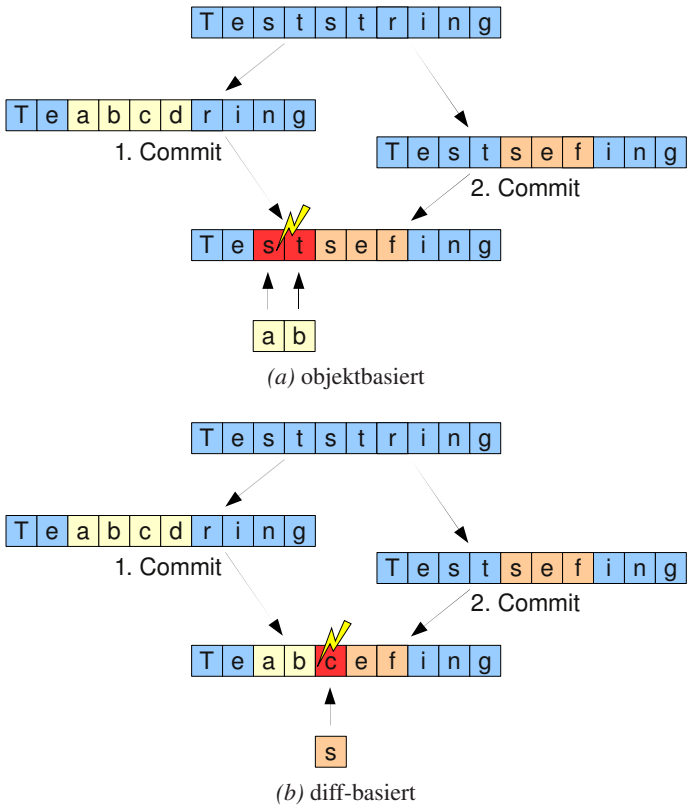


Abbildung 2.7: Schreib-Schreib-Konflikt: Fehlerhafte Zusammenführung von geänderten Replikaten eines gemeinsamen Transaktionsobjekts.

unterteilt. So können mehrere Transaktionen gleichzeitig von einem Objekt lesen, eine Transaktion sich beim Schreiben gegenüber anderen aber das alleinige Zugriffsrecht auf ein Objekt sichern. Andere Transaktionen, die eine bereits vergebene Sperre ebenfalls anfordern, müssen eventuell warten, bis die Sperre wieder verfügbar ist. Die entsprechenden Szenarien sind Anforderung einer Schreibsperre, falls eine oder mehrere Transaktionen eine Lesesperre auf dasselbe Objekt besitzen. Umgekehrt gilt das auch für die Anforderung einer Lesesperre, wenn eine andere Transaktion bereits mit einer Schreibsperre exklusiven Zugriff auf dasselbe Objekt besitzt. Transaktionen können immer Lesesperren auf ein Objekt erhalten, sofern niemand eine Schreibsperre darauf besitzt. Aus Fairneßgründen ist es jedoch sinnvoll, keine weiteren Lesesperren zu vergeben, falls eine Transaktion bereits auf eine Schreibsperre wartet, damit dieser nicht verhungert.

Für die Anforderung von Objektsperren durch Transaktionen gibt es unterschiedliche Verfahren. Eine Transaktion kann beispielsweise die Sperren aller verwendeten Objekte entweder zu Beginn der Transaktion atomar oder sukzessive während der Ausführung anfordern. Eine Anforderung zu Beginn einer Transaktion setzt voraus, daß alle verwendeten Objekte zuvor bekannt sind. Dies ist nur möglich, wenn die Kontrolle der Sperren beim Applikationsentwickler liegt. Denn nur er kann wissen, welche Objekte potenziell in der Transaktion verwendet werden. Weiterhin ist die Anwendung der Strategie auch von der Applikation selbst abhängig. Falls die verwendeten Objekte erst zur Laufzeit bekannt werden, ist eine Anforderung der Objektsperren zum Transaktionsbeginn nicht möglich. Abhilfe schafft hier die sukzessive Anforderung der Sperren während der Laufzeit. Bei diesem Verfahren kann die Sperrverwaltung für den Entwickler auch transparent erfolgen, sofern der Rechner die Sperre automatisch vor

einem bevorstehenden Objektzugriff anfordert.

Bei der sukzessiven Sperranforderung besteht das Problem, daß es zu einer verteilten Verklemmung kommen kann, die sich mittels Zeitüberschreitungen oder Wartegraphen erkennen läßt. Läuft eine Transaktion bis zum Ende durch, ist diese in jedem Fall konfliktfrei und kann abschließen, da ein gleichzeitiger Objektzugriff von verschiedenen Transaktionen durch die Objektsperren ausgeschlossen ist. Tritt dagegen eine Verklemmung auf, müssen einige hierfür ursächliche Transaktionen abbrechen und ihre Sperren freigeben. Verteilte Sperren (Anforderung, Freigabe und Benachrichtigung wartender Transaktionen) in Verbindung mit der Prüfung auf Verklemmungen in verteilten Systemen sind wegen der hohen Netzkommunikation sehr teuer. Zwar erlauben Verfahren wie *Zwei-Version-Sperren* oder *Hierarchische Sperren* mehr Nebenläufigkeit (siehe auch [27]), können die Einschränkungen in verteilten Systemen aber nicht umgehen. Der gemeinsame verteilte transaktionale Speicher geht von einer geringen Konfliktrate aus, was sich hauptsächlich im Softwaredesign der Anwendung widerspiegelt, deswegen eignen sich optimistische Synchronisierungsverfahren unter diesen Umständen besser und werden ausschließlich verwendet.

2.5.3 Optimistische Synchronisierung

Optimistische Synchronisierungsverfahren, erstmals vorgestellt von Kung und Robinson [58], gehen im Gegensatz zu Sperrverfahren von einer geringen Konfliktrate aus. Bei diesem Ansatz spekulieren die Transaktionen darauf, während ihres Ablaufs keinen Konflikt zu verursachen und prüfen daher erst in der Validierungsphase nach Beendigung der Transaktion, ob ein Konflikt besteht. Dieser Ansatz hat gegenüber dem pessimistischen Ansatz den Vorteil, weil er keine verteilten Objektsperren benötigt. Stattdessen führt die Transaktion eine Historie über gelesene und geschriebene Objekte, die sie in der anschließenden Validierungsphase für die Konflikterkennung heranzieht. In der Validierungsphase kann die Konfliktprüfung mit der Vorwärts- oder Rückwärtsvalidierung stattfinden. Tritt kein Konflikt auf, kann die Transaktion gleich zur Commitphase übergehen und ihre Änderungen bekanntgeben, so daß andere Knoten die neuen Objektversionen zugreifen können.

Vorwärts- und Rückwärtsvalidierung

Die Vorwärtsvaliderung stellt konfliktprüfende Transaktionsvergleiche in Vorwärtsrichtung an. Das heißt, es findet ein Vergleich der abzuschließenden Transaktion mit allen gegenwärtig laufenden Transaktionen statt. Dies gilt ebenso für Transaktionen, die ihrerseits gerade abschließen wollen, aber aufgrund der strengen Serialisierungsreihenfolge erst später in ihre Validierungsphase eintreten können. Im Detail vergleicht die Konfliktprüfung die Schreibmenge der abzuschließenden Transaktion mit der Lesemenge der anderen laufenden Transaktionen, um so festzustellen, ob die laufenden Transaktionen bei einem Commit auf veralteten Daten arbeiten würden. Im Konfliktfall muß die Validierungsphase eine minimale Anzahl von Transaktionen abbrechen, bis kein Konflikt mehr besteht. Das Vorwärtsvalidierungsschema schreibt nicht vor, welche der verglichenen Transaktionen abzubrechen sind, dies liegt allein im Ermessen des Commit-Protokolls. Dabei ist es ebenso erlaubt, die abzuschließende Transaktion abzubrechen. Allein unter Laufzeitzeitaspekten der Transaktionen ist es sinnvoll, möglichst wenige und kurzzeitig laufende Transaktionen abzubrechen. Unter Einbezug des gesamten Commit-Protokolls und der transaktionalen Objektreplikation ist diese Vorgehensweise jedoch nicht als effizienteste haltbar (siehe Kapitel 4.3).

Die Rückwärtsvalidierung vergleicht die festzuschreibende Transaktion mit allen zu ihrer Laufzeit überlappenden Transaktionen, die selbst bereits festgeschrieben sind. Hierbei vergleicht die abzuschließende Transaktion ihre Lesemenge mit den Schreibmengen zu ihrer Laufzeit

überlappenden Transaktionen. So kann die Transaktion feststellen, ob sie selbst veraltete Transaktionsobjekte gelesen hat. Stellt sich bei der Prüfung auf Konflikte heraus, daß die laufende Transaktion die Konfliktregeln (siehe Kapitel 2.5) verletzt, so muß sie abgebrochen werden. Bei der Rückwärtsvalidierung sind die in der Konfliktprüfung zum Vergleich herangezogenen Transaktionen bereits abgeschlossen und festgeschrieben. Im Konfliktfall bleibt nur der Weg, die laufende abzuschließende Transaktion abzubrechen, um den Konflikt aufzulösen. Der gemeinsame verteilte transaktionale Speicher arbeitet auf Objektreplikaten, daher ist es ausreichend, wenn sich die Konfliktprüfung darauf beschränkt, ob die abzuschließende Transaktion veraltete Replikate verwendet. Die Knoten müssen daher keine Historie über abgeschlossene Transaktionen führen, da neben einem Vergleich von Objektreplikatsversionen nebensächlich ist, welche bereits abgeschlossene Transaktion ein Replikat festgeschrieben hat. Die Vorwärtsvalidierung hat im Vergleich eine größere Auswahl, Transaktionen für die Auflösung von Konflikten abzubrechen, was jedoch keineswegs bedeutet, daß die Vorwärtsvalidierung im verteilten System der anderen vorzuziehen ist (siehe Kapitel 5.2).

2.5.4 Commit

Ist eine Transaktion nach ihrer Validierung konfliktfrei, so kann sie abschließen und ihre geänderten Transaktionsobjekte festschreiben, indem andere Knoten mittels des Invalidierungs- beziehungsweise Aktualisierungsverfahrens (siehe Kapitel 2.2.1 und 2.2.2) andere Knoten über den Commit und geänderte Transaktionsobjekte in Kenntnis setzt. Anschließend löscht die Transaktion alle Schattenkopien der geschriebenen Objekte und entzieht allen Objekten in ihrer Lese- und Schreibmenge die Zugriffsrechte (Ursprungszustand), damit folgende Transaktionen Zugriffe auf diese Objekte erneut erkennen können. Damit ist die Transaktion erfolgreich abgeschlossen.

2.6 Transaktionale Speichermodelle

Neben den Grundlagen von Transaktionen in Linux-Betriebssystemen und der hardwareunterstützten Objekterkennung, ist es ebenso wichtig, wie eine Integration des transaktionalen Speichers in das Betriebssystem aussehen kann. Ein Betriebssystem kann den transaktionalen Speicher auf unterschiedlichen Ebenen implementieren, wobei die Komplexität mit zunehmender Verlagerung betriebssystemrelevanter Bestandteile in den gemeinsamen transaktionalen Speicher zunimmt.

- Betriebsystem im gemeinsamen verteilten transaktionalen Speicher
- Prozesse im gemeinsamen verteilten transaktionalen Speicher
- Dedizierte Speicherblöcke im Prozeßadreßraum als gemeinsamer verteilter transaktionaler Speicher

Das Linux-Betriebssystem ist zwar für Mehrkern- und Multiprozessorsysteme in Verbindung mit UMA- und ccNUMA-Speicherarchitekturen, aber nicht für einen verteilten transaktionalen Speicher konzipiert. Eine Portierung des Betriebssystems in einen gemeinsamen verteilten Speicher ist ohne systemweite Umbauten nicht möglich und in vielen Fällen auch nicht erwünscht. Diverse lokale Datenstrukturen wie die Prozeßverwaltung müßten rechnerübergreifend vorliegen, wobei Rechner einige Datenstrukturen zwingend nicht gemeinsam verwenden dürfen. Dies gilt vor allem für den hardwarenahen Systemkontext eines Rechners (zum Beispiel Treiber und deren Interruptroutinen). Weiterhin müßte der gemeinsame transaktionale Speicher oberhalb des Betriebssystemkerns auch dessen Speicherverwaltung übernehmen, und zudem müßten alle relevanten Teile des Betriebssystems in Transaktionen gekapselt sein.

Eine Verlagerung von Prozessen in den gemeinsamen Speicher unterliegt ähnlichen Einschränkungen wie bei dem Betriebssystem. Das Betriebssystem muß dies explizit unterstützen. Prozesse agieren mit ihrem Betriebssystem. Bei einer Ausführung von Prozessen über mehrere Rechner hinweg, würden diese mehreren autonomen Betriebssystemen unterliegen, so daß es hier Inkonsistenzen durch die nicht miteinander synchronisierten Betriebssysteme auftreten würden. Weiterhin muß auf allen Rechnern das Speicherlayout der Anwendung übereinstimmen. Linux stellt jeder Anwendung einen virtuellen linearen Adreßraum zur Verfügung, welcher sich in unterschiedliche Abschnitte für Programmcode, Bibliotheken, Konstanten, vorinitialisierte statische Daten, Halde und Keller *(engl. Stack)* unterteilt. Außerdem blendet das Betriebssystem einen Teil des Betriebssystemkerns in den Speicher eines jeden Prozesses ein. Deshalb müssen auch lokale Systemaufrufe wie lokale dynamische Speicherallozierung streng synchronisiert und transparent als transaktionale Allozierung erfolgen.

Ebenso muß das Betriebssystem bei seiner Verlagerung in den gemeinsamen verteilten transaktionalen Speicher weitere Fehlertoleranzmechanismen integrieren, damit eine fehlerhafte Betriebssysteminstanz oder Netzwerkfehler nicht zum Absturz anderer Betriebssysteminstanzen führt. Vergleichbares gilt für die Ausführung einzelner Prozesse im transaktionalen Speicher. Systemdateien und Programmbibliotheken müssen zudem auf allen Rechnern in derselben binärkompatiblen Version vorliegen. Der Fokus dieser Arbeit liegt in der einfachen Entwicklung verteilter Anwendungen über gemeinsame verteilte transaktionale Speicher, daher ist eine Verlagerung des Betriebssystems oder Prozesse in den transaktionalen Speicher kontraproduktiv. Dies würde hinsichtlich Synchronisierung einen großen Mehraufwand verursachen, der vor allem unter der Prämisse einer hohen Netzwerklatenz in Gridumgebungen und Weitverkehrsnetzen zu einer schlechten Skalierbarkeit führt.

Im dritten Szenario laufen die Prozesse von verteilten Anwendungen auf einzelnen autonomen Rechnern, wobei jeder Prozeß dynamisch verteilten transaktionalen Speicher (siehe Kapitel 2.6.1) allozieren kann. Dies ist für Anwendungsentwickler die einfachste und bequemste Art eines gemeinsamen verteilten transaktionalen Speichers. Der Entwickler muß gemeinsame transaktionale Speicherblöcke zwar explizit allozieren, braucht für Datenstrukturen, die nicht in diesem Speicher liegen, keine weiteren Vorkehrungen hinsichtlich Synchronisierung treffen, da die nur im lokalen Prozeßkontext sichtbar sind. Diese Vorgehensweise ermöglicht es, vollkommen ohne Modifikationen des Linux-Betriebssystems auszukommen (siehe Kapitel 7). Im Gegensatz zu den beiden erstgenannten kann dieses Verfahren besser mit heterogenen Systemen umgehen, da die Betriebssysteme voneinander isoliert auf unterschiedlichen Rechnern laufen und diese zudem von heterogener Hardware abstrahieren. Zu berücksichtigen ist nur, daß die gemeinsam verwendeten Daten zum lokalen Anwendungskontext kompatibel sind. Dies betrifft in erster Linie die Datenwort- und Adreßbreite (Zeiger) auf Maschinenebene und die Byte-Reihenfolge (*Little-Endian* oder *Big-Endian*). Vorteil dieser Variante ist, daß nur notwendige Speicherbereiche dem Transaktionskonzept unterliegen und über das Netzwerk synchronisiert werden müssen, was gegenüber den anderen beiden Verfahren leistungsfähiger ist. Abgesehen von Transaktionen ähnelt dieses Konzept des verteilten Speichers den von Linux bereitgestellten *Shared-Memory-Segmenten* für die Interprozeßkommunikation.

2.6.1 Dynamische Speicherallozierung

Der virtuelle Adreßraum eines Prozesses ist in Abschnitte gliedert (siehe Abbildung 2.8), die neben dem Programmcode (Text), statische Daten (Data und BSS), Keller und Kern auch zwei Bereiche für dynamische Speicherallokationen (Halde und Mmap) beinhalten. Pfeile zeigen die Wachstumsrichtung dynamischer Speicherbereiche an. Das genaue Layout ist betriebssystemabhängig und kann von Rechner zu Rechner variieren. Zudem sind die Grenzen der einzelnen Abschnitte mitunter auch von der Anwendung abhängig [85, 20].

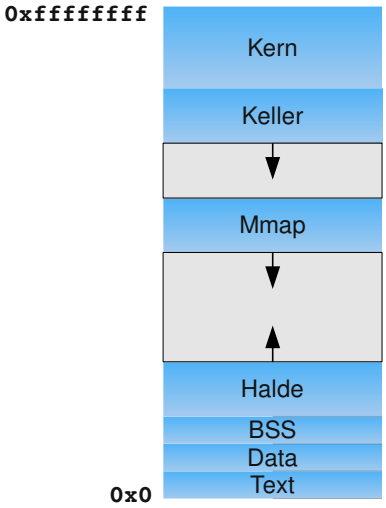


Abbildung 2.8: Schematische Adreßraumaufteilung eines Linuxprozesses (32-Bit).

Speicherblöcke des gemeinsamen transaktionalen Speichers müssen auf Rechnern nicht zwingend an derselben virtuellen Adresse liegen, da die Synchronisierung an sich betrachtet unabhängig von lokalen virtuellen Speicheradressen ist. Dieses Verhalten entpricht aus Anwendungssicht dem gleichen Verhalten, wenn Prozesse eines lokalen Rechners über gemeinsame System-V-Shared-Memory-Segmente kommunizieren, diese aber an unterschiedlichen Adressen ihres Prozeßadreßraumes einblenden. Eine weniger komplexe Variante ist, transaktionale Speicherblöcke mit einer global eindeutigen virtuellen Adresse zu verknüpfen (siehe Abbildung 2.9). Bei diesem Verfahren entfällt zwar die Adreßübersetzung, allerdings müssen alle Rechner bei Allozierung eines neuen Speicherblocks an derselben Adresse einen freien Speicherblock besitzen. Allozieren die auf den Rechnern laufenden Prozesse die transaktionalen Speicherblöcke direkt nach ihrem Start, stehen die Erfolgschancen gut, da diese zu diesem Zeitpunkt noch keine lokalen dynamischen Speicherblöcke alloziert haben. Je mehr Rechner beteiligt sind, desto schwieriger ist es, unter Umständen einen gemeinsamen freien virtuellen Speicherbereich zu finden. Diese Problematik gilt insbesondere für 32-Bit-Prozesse, während 64-Bit-Prozesse einen wesentlich größeren virtuellen Adreßraum besitzen, mit geringerer Kollisionswahrscheinlichkeit bei der Suche nach freien virtuellen Speicherbereichen. Daher sind 64-Bit-Systeme mit 64-Bit-Prozessen vorzuziehen. Eine weitere Beleuchtung der Speicherverwaltung erfolgt in dieser Arbeit nicht, da der Schwerpunkt auf der Synchronisierung replizierter transaktional verteilter Daten liegt.

Der in der Entwicklung verwendete Prototyp für die Analyse des Systems (siehe Kapitel 7) verwendet für die Allokation transaktionaler Speicherblöcke eine zur Funktion `malloc()`[5] ähnliche Semantik. Dies hat den Vorteil, daß Entwickler bereits vertraute Speicherallokationsmechanismen verwenden können. Alloziert ein Rechner einen Speicherblock, so steht dieser über die global zugewiesene virtuelle Adresse automatisch auch allen anderen Rechnern des transaktionalen Speichersystems zur Verfügung. Eine besondere Berücksichtigung erfordern Speicherblöcke in unterschiedlichen Konsistenzdomänen (siehe Kapitel 6.4).

Abbildung 2.10 zeigt hierzu ein vereinfachtes Anwendungsbeispiel. Die in Zeile [2] dargestellten Zugriffe auf den Namensdienst übermitteln die Speicheradresse des von dem ersten Rechner allozierten Speicherbereichs an den zweiten. Die Barriere dient dazu, daß der zweite Rechner seine Transaktion erst ausführt, wenn der erste Rechner die Variable im transaktionalen

[5] Funktion zur Allozierung dynamischer Speicherblöcke in der Programmisprache *C*.

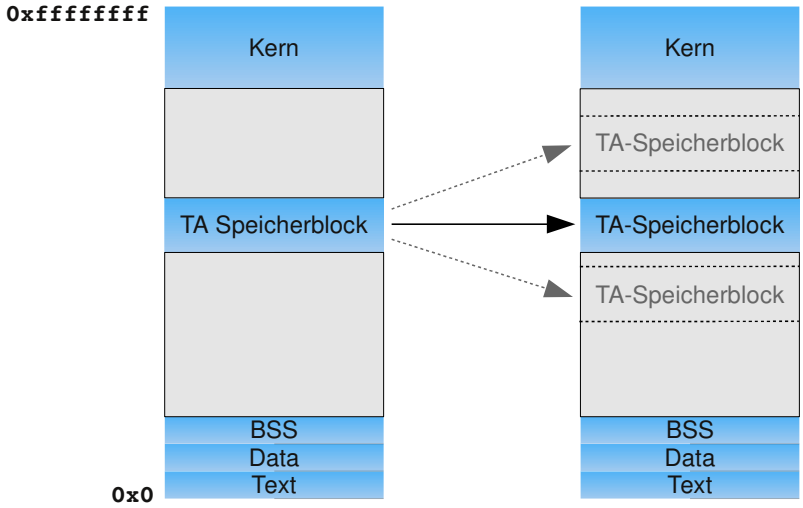


Abbildung 2.9: Allozierte Speicherblöcke des transaktionalen Speichers an variablen Basisadressen (32-Bit).

```
1  int *x = tm_alloc(4);
2  nameservice_set("p", x);
3
4  BOT();
5    *x = 3;
6  EOT();
7  signal_barrier(b);
```

```
1  wait_barrier(b);
2  int *x = nameservice_get("p");
3
4  BOT();
5    y = *x;
6  EOT();
7  printf("%i\n", y);    (y -> 3)
```

Abbildung 2.10: Nutzung des transaktionalen Speichers mit zwei Rechnern.

Speicherblock geschrieben hat. Somit braucht der zweite Rechner nicht in einer Schleife abzufragen, ob der erste Rechner seine Operation bereits durchgeführt hat. Eine weiterführende Diskussion zu Namensdiensten und Mechanismen zur ereignisgetriebenen Informationsverarbeitung findet in dieser Arbeit nicht statt. Der erste Rechner schreibt einen Wert in den transaktionalen Speicher. Ist der Commit erfolgreich, verläßt der zweite Rechner die Barriere und überträgt den Wert des verteilten transaktionalen Speichers an eine lokale Variable. Es ist ersichtlich, daß die lokale Allozierung des transaktionalen Speicherblocks nicht notwendig ist, da diese für die Anwendung transparent beim ersten Zugriff erfolgt. Ist die Transaktion erfolgreich, erfolgt eine Bildschirmausgabe des vom ersten Rechner geschriebenen Wertes. Der Umweg über die lokale Variable ist wegen der Verwendung von nicht idempotenten Funktionen (siehe Kapitel 3.2.3) in Zeile [7] notwendig.

2.7 Transaktionale Speicherinhalte

Daten im transaktionalen verteilten Speicher sind genauso wie Daten im lokalen Anwendungskontext üblicherweise Skalare, Arrays, Strings, komplexe Datenstrukturen oder Zeiger. Der transaktionale Speicher behandelt jedwede Daten als binäre Rohdaten und arbeitet daher unabhängig von Datentypen, es findet seinerseits also keine Interpretation der Daten statt. Interpretation und Struktur der gespeicherten Daten unterliegen nur der verwendeten Hardwarearchitektur, dem Betriebssystem und der Anwendung.

2.7.1 Skalare

Die Hardware bestimmt die Bytereihenfolge. Kommunizieren verteilte Anwendungen mit gegensätzlicher Bytereihenfolge über den transaktionalen Speicher, so ist es Aufgabe der Anwendung, die Bytereihenfolge von Basisdatentypen und Strukturelementen anzupassen. Die Basisdatentypen einer Programmiersprache können je nach Ausführungsumgebung – 32- oder 64-Bit bei der x86-64-Architektur mit einem 64-Bit Linuxsystem – eine unterschiedliche Größe haben. Ebenso können Elemente von Datenstrukturen aufgrund von Füllbytes (Padding) relativ zum Beginn einer Datenstruktur an unterschiedlichen Adressen liegen. Bei Strings hängt die Bytereihenfolge von der Zeichenkodierung ab. Bei Ein-Byte-Kodierungen wie zum Beispiel ASCII (American Standard Code for Information Interchange) ist die Bytereihenfolge irrelevant. Bei Zeichenkodierungen, die aus mehreren Bytes bestehen kann die Bytereihenfolge mitunter wichtig sein. Dies ist beispielsweise bei UCS-2 und UCS-4 (Universal Character Set) der Fall.

2.7.2 Zeiger

Die Größe von Zeigern bestimmt die Ausführungsumgebung des Prozesses und entspricht dem Adressierungsmodus des Prozessors. Zeiger im transaktionalen Speicher, die auf andere Bereiche desselben oder andere allozierte transaktionale Speicherblöcke verweisen, bleiben rechnerübergreifend gültig, sofern jeder Speicherblock global mit einer festen virtuellen Adresse verknüpft ist (siehe Kapitel 2.6.1). Gleichermaßen ist dies bei Zeigern aus dem lokalen Prozeßkontext in transaktionale Speicherblöcke der Fall. Umgekehrt gilt das jedoch nicht. Verweist ein Zeiger in einem transaktionalen Speicherblock auf ein Datum in dem lokalen Prozeßkontext, so ist dieser Zeiger in einem anderen Prozeß nicht gültig, da sich an der referenzierten Adresse nicht unbedingt ein vergleichbares Datum in dem anderen Prozeßkontext befinden muß.

Möchte eine verteilte Anwendung prozeßübergreifend aus dem gemeinsamen verteilten transaktionalen Speicher lokale Daten referenzieren, so kann sie diese Zeiger durch UUIDs (Universal Unique Identifier) substituieren. So kann jeder zugehörige Prozeß die Zeiger mittels einer Übersetzungstabelle zu lokalen Referenzen auflösen. Ebenso können veraltete Zeigerdereferenzierungen zwischen mehreren transaktionalen Speicherobjekten bei der verzögerten Abbruchstrategie von Transaktionen (siehe Kapitel 3.2.2) zum Programmabsturz führen, bevor die Transaktion wegen eines Konflikts abbricht. Diesen Fall verhindert die Verwaltung von Multiversion-Objekten (siehe Kapitel 6.1).

2.7.3 Programmiersprachliche Objekte mit variabler Größe

Programmiersprachliche Objekte mit variabler Größe im transaktionalen Speicher unterliegen im Vergleich zu denen im lokalen Speicher der Anwendung keinen besonderen Einschränkungen. Benötigen programmiersprachliche Objekte mehr Speicher, als ihnen im zugewiesenen Speicherblock zur Verfügung steht, kopiert die Anwendung diesen normalerweise in einen größeren Speicherblock oder referenziert auf einen neu allozierten Speicherblock. Wenn sich ein Objekt in der Größe verkleinert, kann die Freigabe dessen Speichers analog erfolgen. Liegen die programmiersprachlichen Objekte im gemeinsamen verteilten transaktionalen Speicher, müssen die Restrukturierungsoperationen innerhalb einer Transaktion stattfinden, damit trotz der Nebenläufigkeit die Atomarität gewahrt bleibt.

2.7.4 Dateien

Linux bietet die Möglichkeit, über die Systemfunktion `mmap()` Dateien in den Speicher abzubilden. Dateien lassen sich so auch mit der Allozierung transaktionalen Speichers verknüpfen, so daß dieser mit dem Dateiinhalt vorinitialisiert ist. Bildet ein Rechner bei der Speicherallozierung eine Datei in den transaktionalen Speicher ab, können andere Rechner dessen Inhalt lesen und schreiben. Dabei bleibt den anderen Rechnern verborgen, daß es sich bei dem transaktionalen Speicherinhalt um eine verknüpfte Datei handelt. Unterschiede bestehen in dem Verhalten durch Schreibzugriffe im Vergleich zu lokalen speicherbasierten Dateien. Bei lokal geladenen Dateien sind Schreibzugriffe über den Speicher in der Regel zwischengespeichert. Dies bedeutet, Änderungen an der Datei schreibt das Betriebsystem nicht sofort auf die Datei durch, sondern erst nach einer unbestimmten Zeit, wie das auch bei Schreibcaches von Datenträgern der Fall ist.

Führt ein Rechner, der die Datei nicht geladen hat, einen Schreibzugriff auf die Datei durch, so legt die Synchronisierungsstrategie fest, wann und ob die Änderungen auch in der Datei erfolgen. Hierfür ist maßgeblich, daß dem Rechner, der die Datei geladen hat, etwaige Änderungen bewußt sind. Herfür muß er Aktualisierungen für geschriebene transaktionale Speicherobjekte erhalten. Beim Invalidierungsverfahren kann der Rechner in regelmäßigen Zeitabständen aktualisierte Objektreplikate anfordern, falls erhaltene Commit-Nachrichten zur Datei zugehörige transaktionale Speicherobjekte invalidieren. Das Aktualisierungsverfahren bietet dagegen den Komfort, daß der Rechner automatisch aktualisierte Objektreplikate erhält, die das Betriebssystem auf die Datei durchschreiben kann. Finden häufige entfernte Dateiänderungen statt, kann sich die Aktualisierungsstrategie auch in diesem Fall als hinderlich erweisen, so daß hier eine Mischstrategie aus beiden Verfahren anzuraten ist.

Daher ist es sinnvoll, daß Rechner Synchronisierungsstrategien bei der Speicherallozierung festlegen können, so daß entfernt getätigte Änderungen an mittels `mmap()` geladener Dateien häufiger synchonisiert werden. Es ist daher sinnvoll, daß das transaktionale Speichersystem Dateien im transaktionalen Speicher explizit unterstützt. Als Erweiterung bietet es sich an, die API (Application Programming Interface) auch für entfernte Rechner zur Verfügung zu stellen. So können andere Rechner beispielsweise einen anderen Abschnitt einer Datei bei nicht vollständig geladenen Dateien in den transaktionalen Speicher einblenden. Eine detaillierte Erläuterung von Dateien im verteilten transaktionalen Speicher erfolgt an dieser Stelle nicht.

2.8 Transaktionen

2.8.1 Deklaration

Transaktionsgrenzen definieren den innerhalb einer Transaktion auszuführenden Programmcode. Dieser Festlegung kommt im Fall einer Transaktionsrücksetzung (siehe Kapitel 3) eine wichtige Bedeutung zu. Prinzipiell kennzeichnen Zugriffe auf transaktionale Speicherblöcke bereits, daß diese in einem transaktionalen Kontext stattfinden. Dennoch muß der Entwickler genau bestimmen, wann eine Transaktion endet und die Validierungsphase einzuleiten ist, da eine Transaktion zumeist aus mehreren aufeinanderfolgenden Speicherzugriffen besteht, die zusammen den ACID-Eigenschaften von Transaktionen unterliegen. Ebenso ist es notwendig, daß die Anwendung den Beginn einer Transaktion festlegt, da dieser den Programmcodeabschnitt definiert, der nach der Anwendungssemantik zum Transaktionskontext gehört. Diese Markierung ist ferner notwendig, damit eine wegen eines Konflikts rückzusetzende Transaktion wieder ab dem Programmcodeabschnitt beginnt, der nach der Anwendungssemantik zum Transaktionskontext gehört.

Zwar ließe sich der Beginn einer Transaktion auch über den ersten erkannten Speicherzugriff durch die Speicherverwaltungseinheit definieren. Das hat aber den Nachteil, daß die Anwendung vor dem ersten Zugriff keine Instruktionen im Transaktionskontext ausführen kann. Der Entwickler müßte stattdessen Ergebnisse solcher Operationen, die vor dem ersten transaktionalen Speicherzugriff stattfinden, zunächst in globalen Variablen zwischenspeichern, damit deren Ergebnisse im Fall einer Transaktionsrücksetzung mit Neuausführung nicht verloren gehen. Eine Zwischenspeicherung in lokalen Variablen ist aufgrund der Rücksetzung des Threadkontexts bei einer wiederholten Transaktionsausführung nicht möglich (siehe Kapitel 3.3). Auch eine direkte Zuweisung eines Operationsergebnisses an eine Variable des transaktionalen Speichers führt aus einer Hochsprache (zum Beispiel *C*) meistens zu einer fehlerhaften Rücksetzung von Transaktionen. Dies gilt gleichermaßen, wenn die Anwendung für die Zuweisung eine lokale Hilfsvariable verwendet (siehe Abbildung 2.11).

```
1   tmp = 0;
2     ta_mem = sin(2.5);
3     ...
4   Commit();
```

```
1   tmp = sin(2.5);
2     ta_mem = tmp;
3     ...
4   Commit();
```

Abbildung 2.11: Ungültige Programmausführung bei Transaktionsrücksetzung.

Die Abbildung zeigt beispielhaft eine Transaktion, deren Beginn implizit durch den ersten transaktionalen Speicherzugriff stattfindet. Die Variable *ta_mem* im transaktionalen Speicherblock erhält als Zuweisung das Ergebnis einer mathematischen Operation, entweder direkt oder über eine lokale Hilfsvariable. Die Operation in Zeile 1 ist nicht Bestandteil der Transaktion. Die Operation in Zeile 2 gehört nicht vollständig zur Transaktion[6]. Erfolgt eine Transaktionsrücksetzung, ist die in Zeile 2 dargestellte Zuweisung undefiniert, die Variable enthält nach der Zuweisung eventuell ein fehlerhaftes Ergebnis.

Die fehlerhafte Rücksetzung ist aus Hochsprachen nicht klar ersichtlich, hierzu ist eine Betrachtung auf Ebene des Maschinensprachecodes unerläßlich. Der Prozessor kann nicht zwischen einzelnen Operationen einer Hochsprache differenzieren, da er nur Maschinencode ausführt. Eine Hochsprachenoperation besteht normalerweise aus mehreren Maschinensprachebefehlen, die der Prozessor zudem nicht atomar ausführen kann (Hardwareinterrupts, Exceptions, präemptives Multitasking/-threading). Der folgende Codeausschnitt (siehe Abbildung 2.12) zeigt die direkte Zuweisung der mathematischen Operation an die Variable im transaktionalen Speicher als x86-Assemblercode[7]. Hierbei ist anzumerken, daß es sich bei dem Beispiel nur um eine von vielen möglichen Übersetzungen in Maschinencode handelt.

```
1   movsd   xmm0,QWORD PTR [rip+0x291cd2]
2   call    401060 <__sin>
3   movsd   QWORD PTR [rip+0x2929a5],xmm0
```

Abbildung 2.12: Hochsprachenoperation *ta_mem := sin(2.5)* in x86-Assembler.

An dieser Stelle wird deutlich, warum es zu einer fehlerhaften Rücksetzung kommen kann. Die endgültige Zuweisung an die Variable `ta_mem` erfolgt erst in Zeile 3, an dieser Stelle erfolgt die Unterbrechung durch die Pagefault-Exception, die wiederum den Signalhandler aufruft. Dieser würde bei einem impliziten Transaktionsstart die Operation in dieser Codezeile

[6] Innerhalb der Transaktion ausgeführte Operationen oder Teiloperationen sind in grüner Farbe, andere in roter Farbe dargestellt.

[7] Hier dargestellte Fragmente der x86-Assemblersprache verwenden für eine einfachere Lesbarkeit die Intel-Syntax, obwohl für Linux die AT&T-Syntax gebräuchlicher ist.

als Transaktionsbeginn registrieren, obwohl die Hochsprachenoperation in Zeile [1] beginnt. Bei einer wiederholten Ausführung der Transaktion erfolgt nur eine Wiederholung der letzten Maschinencodeinstruktion, das Prozessorregister `xmm0` kann aber wegen anderweitiger Nutzung bereits einen anderen Wert beinhalten. Deswegen kann die erneut ausgeführte Transaktion auf diese Weise kein korrektes Ergebnis garantieren.

Muß die Anwendung den Beginn einer Transaktion über einen expliziten Funktionsaufruf einleiten, liefert eine Transaktion auch ein korrektes Ergebnis, wenn sie aufgrund eines Konflikts neu startet. Statt eines Funktionsaufrufs kann auch eine äquivalente Hochsprachenanweisung den Beginn einer Transaktion definieren. Entscheidend ist an dieser Stelle, daß der Entwickler den Transaktionsbeginn explizit über eine Hochsprachenanweisung einleitet. Damit ist sichergestellt, daß das System den Maschinencode der darauffolgenden Hochsprachenanweisung bei einer Transaktionsrücksetzung mit Wiederholung vollständig ausführt.

Die Funktion, die den Beginn einer Transaktion definiert, kann über ihre Rücksprungadresse genau den Beginn der nachfolgenden Operationen erkennen. Bei einem expliziten deklarierten Beginn einer Transaktion vor der Anweisung `ta_mem = sin(2.5);` startet die Transaktion nach Abbildung 2.12 bei einem Neustart in Zeile [1] statt [3]. Deswegen müssen Anwendungen nach dem Programmiermodell Transaktionsgrenzen explizit durch spezielle Funktionsaufrufe kenntlich machen (siehe Abbildung 2.13). `BOT()` kennzeichnet den hier Anfang einer Transaktion und steht für *Begin Of Transaction* und `EOT()` für *End Of Transaction*. `EOT()` entspricht der `Commit()`-Anweisung aus Abbildung 2.11 und leitet die Validierungsphase ein. Eine detaillierte Diskussion über die Rücksetzung von Transaktionen findet sich in Kapitel 3.

```
1  tmp = 0;
2  BOT();
3    ta_mem = sin(2.5);
4    ...
5  EOT();
```

Abbildung 2.13: Explizit definierte Transaktionsgrenzen mittels BOT/EOT.

2.8.2 Operationen auf Transaktionen

Standardmäßig ist der Anwendungszweck von Transaktionen, nebenläufige Zugriffe auf gemeinsame Daten automatisch zu serialisieren. Demzufolge ist ein Transaktionskonflikt damit verbunden, daß einige beteiligte Rechner ihre Transaktion neu starten müssen. Jedoch kann es für manche Anwendungen sinnvoll sein, bei einem Transaktionskonflikt statt eines Neustarts die Kontrolle unmittelbar an die Anwendung zurückzugeben, ebentuell mit einem Hinweis auf die konfliktverursachenden Objekte. Dies kann beispielsweise bei lang laufenden Transaktionen der Fall oder über eine durch Benutzerinteraktionen gesteuerte Transaktionsausführung sinnvoll sein. Ebenso können Situationen eintreten, in denen eine Anwendung ihre Transaktion aus Sicht der Anwendungslogik abbrechen möchte, auch wenn am Transaktionsende kein Konflikt aufgetreten wäre. Für den manuellen Abbruch ließe sich wiederum entscheiden, ob das System die Transaktion erneut ausführen soll oder nicht. Folgende zusätzliche Operationen neben der Definition von Transaktionsgrenzen und der automatischen Neuausführung abzubrechender Transaktionen erhöhen die Flexibilität bei der Anwendungsentwicklung

- Kein Transaktionsneustart bei konfliktspezifischem Transaktionsabbruch
- Manueller Abbruch von Transaktionen durch Anwendung
- Manueller Neustart von Transaktionen durch Anwendung

2.8.3 Systemaufrufe

Während der Anwendungsausführung treten zwischenzeitlich Aufrufe von Systemfunktionen und Programmbibliotheken auf, über die der Anwendungsentwickler keine direkte Kontrolle hat. Systemfunktionen bezeichnen im Kontext dieser Arbeit sowohl Kernsystemaufrufe als auch die zugehörigen Containerfunktionen im Anwendungskontext *(engl. Wrapper)*, die das Betriebssystem über seine Standardbibliotheken zur Verfügung stellt. Funktionen liefern allgemein auf eine Eingabe eine Ausgabe zurück und modifizieren eventuell den Anwendungs- und Systemzustand. Die detaillierte Arbeitsweise von Funktionen bleibt dagegen verborgen. Die Ausführung von System- oder Bibliotheksfunktionen ist innerhalb von Transaktionen unproblematisch, solange die Transaktionen erfolgreich abschließen können. Bei der Rücksetzung dieser Transaktionen im Konfliktfall läßt sich jedoch nicht mit vollständiger Sicherheit gewährleisten, ob die Funktionen ebenfalls vollständig zurückgesetzt werden[8]. Dies hängt davon ab, ob die Funktionen nur auf dem transaktionalen Speicher arbeiten oder nebenbei auch mit dem lokalen Prozeßkontext und Betriebssystem interagieren und deren Zustände auf diese Weise verändern. Ist letzteres der Fall, so kann das System diese Funktionen nicht oder nicht vollständig zurücksetzen, da die Rücksetzung auf den transaktionalen Speicher und den lokalen Threadkontext beschränkt ist. Ebenfalls ist die Rücksetzung von Funktionen, die sich nicht vollständig zurücksetzen lassen, aber deren mehrfache aufeinanderfolgende Ausführung dasselbe Ergebnis liefern, unkritisch. Diese Funktionen bezeichnet man auch als idempotente Funktionen[9] (siehe Kapitel 3.2.3). Weiterhin können Transaktionsabbrüche innerhalb von Systemfunktionen zu einem undefinierten Programmverhalten bis hin zur Verklemmung oder Absturz führen (siehe Kapitel 3.2.1).

2.8.4 Transaktionaler Objektzugriff außerhalb von Transaktionen

Bei der transaktionalen Konsistenz handelt es sich hier um ein sehr strenges Konsistenzmodell, da die Transaktionen den ACID-Eigenschaften und einer strengen Serialisierung unterliegen. Eventuell benötigen Anwendungen nicht immer stark konsistent gehaltene Daten. So ist es im Hinblick der Leistungsfähigkeit des transaktionalen Systems – strengere Konsistenzmodelle benötigen eine häufigere Datensynchronisierung gegenüber schwächeren Konsistenzmodellen und sind dadurch erfahrungsgemäß langsamer – praktikabel, auf Anforderung von Anwendungen auch außerhalb von Transaktionen Zugriffe auf den transaktionalen Speicher zuzulassen. Sinnvoll sind hier allenfalls Lesezugriffe, da bei Schreibzugriffen fremde Rechner ansonsten die gleiche Version eines Objekts mit unterschiedlichen Dateninhalten ausgeliefert bekommen können. Zudem ist nicht sichergestellt, daß andere Rechner aus Gründen der Nebenläufigkeit außerhalb von Transaktionen geschriebene Objektinhalte in eigenen Transaktionen verwenden. Ebenso können außerhalb von Transaktionen geschriebene Objektinhalte auch durch ältere Objektinhalte überschrieben werden, sofern das innerhalb der Transaktion geschriebene Objekt auf einer zeitlich früheren aber im Transaktionskontext weiterhin gültigen Version basiert.

Neben dem Zugriff auf veraltete Daten ist beim Aktualisierungsverfahren zu beachten, daß Objektaktualisierungen auch während des Lesezugriffs auf ein Objekt stattfinden können. Demnach können bei einem Zugriff auf Skalare und komplexe Datenstrukturen die Daten aus unterschiedlichen Transaktionen hervorgehen. Anwendungsgebiete können periodisch aktualisierte Datenfelder wie zum Beispiel Bildschirminhalte sein.

[8] UNIX-Systemsignale können Funktionen im Anwendungskontext unterbrechen, Kernsystemaufrufe dagegen nicht. Die Systemsignale können jedoch einige wenige blockierende Kernsystemaufrufe abbrechen [20].

[9] Eine Operation liefert sowohl bei einfacher als auch mehrfacher aufeinanderfolgender Ausführung dasselbe Ergebnis $f(x) = f(f(x))$.

2.8.5 Nur-Lese-Transaktionen

Nach Kapitel 2.5 können Transaktionskonflikte unter anderem dann auftreten, wenn mindestens eine Transaktion schreibend zugreift. Die Konflikte verhindern, daß veraltete gelesene Daten in geschriebene Transaktionsobjekte derselben Transaktion einfließen. Entsprechend der Konfliktkriterien müssen Transaktionen, die nur Objekete gelesen haben, auch dann nicht abbrechen, wenn sie aufgrund veralteter gelesener Daten mit anderen Transaktionen in Konflikt stehen. Dies ist deshalb allgemein zulässig, da das Verhalten solcher Transaktionen dasselbe ist, als hätten sie nicht stattgefunden.

Transaktionen mit nur gelesenen Objekten benötigen abhängig vom Applikationskontext teilweise dennoch Objektaktualisierungen. Das liegt darin begründet, daß der transaktionale Speicher immer mit dem lokalen Kontext in Beziehung steht beziehungsweise diesen durch programmiersprachliche bedingte Anweisungen beeinflußt. Die folgende Abbildung 2.14 zeigt ein Beispiel, in dem Transaktionen auf einem Rechner eine lokale Verklemmung verursachen, wenn diese keine aktuellen Objektreplikate anfordern. Angenommen, der erste Rechner schreibt den Wert *3* in den transaktionalen Speicher. Der zweite Rechner führt in einer Schleife mit Abbruchbedingung eine Transaktion aus, die jederzeit einen veralteten Wert liest. Die Transaktion kann durchlaufen, da sie nur Objekte liest. Allerdings ist durch die veralteten gelesenen Daten die Abbruchbedingung der Schleife nicht erfüllt, so daß die Anwendung in der Schleife verharrt. Daher stellen die Commit-Protokolle sicher, daß Transaktionen von invalidierten Transaktionsobjekten aktuelle Replikate anfordern. So entstehen bei diesen Transaktionen keine Verklemmungen, aber dennoch brauchen die Transaktionen nicht streng serialisiert abschließen. Bei geforderter strenger Konsistenz ist aber weiterhin ein Commit erforderlich (siehe hierzu Kapitel 6.1.1).

```
1  int *x = tm_alloc(4);
2  nameservice_set("p", x);
3
4  BOT();
5    *x = 3;
6  EOT();
7  signal_barrier(b);
```

```
1  wait_barrier(b);
2  int *x = nameservice_get("p");
3
4  do {
5    BOT();
6      y = *x;   (liest Wert: 2)
7    EOT();
8  while (y != 3);
```

Abbildung 2.14: Verklemmung aufgrund veralteter gelesener Transaktionsobjekte.

2.9 Anwendungsschnittstelle

Die Anwendungsschnittstelle stellt dem Entwickler Funktionen zur Verfügung, um den verteilten transaktionalen Speicher verwenden zu können. Dieser Abschnitt stellt abstrakt die wichtigsten Funktionen für die Erstellung verteilter Anwendungen vor. Im Detail handelt es sich um die Verwaltung transaktionaler Speicherblöcke, Deklaration von Transaktionen in der Anwendung und einen Namensdienst. In dieser Arbeit vorgestellte Anwendungsbeispiele enthalten der Übersichtlichkeit halber teilweise nicht alle Funktionsparameter.

2.9.1 Transaktionsdeklaration

```
taid_t BOT(ta_attr_t *attr);
```

Die Funktion `BOT()` definiert den Beginn einer Transaktion innerhalb des Programmcodes. Gleichzeitig sichert sie den gegenwärtigen Threadkontext (Kellerspeicher und Prozessorregister). Über Attribute lassen sich die Eigenschaften einer Transaktion anpassen, so daß diese beispielsweise im Konfliktfall nicht automatisch erneut startet. Der Rückgabewert der Funktion liefert entweder eine Transaktions-ID zurück, wenn der Funktionsaufruf erfolgreich war, oder einen Fehlercode. Fehlercodes zeigen beispielsweise einen Transaktionsabbruch aufgrund eines Konflikts an, wenn gleichzeitig eine automatische Neuausführung der Transaktion nicht gewünscht ist. Ebenfalls zeigt der Rückgabewert auch einen manuellen Transaktionsabbruch an.

```
int EOT(taid_t taid);
```

Die Funktion `EOT()` definiert das Ende einer Transaktion innerhalb des Programmcodes. Der Funktionsaufruf startet unmittelbar die Validierungsphase und im konfliktfreien Fall die Commit-Phase. Der Rückgabewert der Funktion zeigt an, ob der Funktionsaufruf erfolgreich war. In diesem Fall haben Validierungs- und Commit-Phase die Transaktion erfolgreich abgeschlossen. Ein Aufruf der Funktion ohne zuvor einleitenden Transaktionsbeginn quittiert die Funktion mit einem Fehlercode als Rückgabewert. Im Konfliktfall stellt die Funktion den Threadkontext wieder her und kehrt zur Funktion `BOT()` zurück. In diesem Fall kehrt die Funktion `EOT()` selbst nicht zurück. Falls ein automatischer Neustart der Transaktion nicht gewünscht ist, kehrt `BOT()` statt mit der ursprünglichen Transaktions-ID mit einem Fehlercode zurück.

```
int ABORT();
```

Die Funktion `ABORT()` erlaubt Transaktionen, inmitten ihrer Ausführung abzubrechen. Der Abbruch bezieht sich dabei immer auf die gerade aktive Transaktion eines Threads. Der weitere Verlauft ist derselbe wie bei einem Transaktionsabbruch aufgrund eines Konflikts. Die Funktion liefert einen Fehlercode zurück, falls der Aufruf nicht erfolgreich war, ansonsten kehrt der Aufruf nicht zurück.

2.9.2 Speicherverwaltung

```
void *tm_alloc(size_t size, tm_attr_t *attr);
```

Die Funktion `tm_alloc()` alloziert dynamisch transaktionale Speicherblöcke und arbeitet ähnlich wie die dynamische Speicherallozierung mittels `malloc()`. Als Parameter erwartet sie die Größe des Speicherblocks in Bytes. Mittels Attribute lassen sich dem Speicherblock bestimmte Eigenschaften wie beispielsweise eine eigene Konsistenzdomäne (siehe Kapitel 6.4) zuweisen. Als Rückgabewert liefert die Funktion einen Zeiger auf den allozierten Speicherbereich. Im Fehlerfall liefert sie einen Nullzeiger zurück. Allozierte Speicherbereiche stehen automatisch auch anderen Rechnern zur Verfügung, indem diese die Speicherblöcke beim erstmaligen Zugriff transparent in ihrem eigenen Adreßraum deckungsgleich abbilden.

```
void tm_free(void *ptr);
```

Die Funktion `tm_free()` ist das Pendant zur Speicherallozierung und gibt allozierte Speicherblöcke wieder frei. Als Parameter erwartet sie einen Zeiger auf einen allozierten Speicherblock.

2.9.3 Namensdienst

Der Namensdienst speichert rechnerübergreifend Schlüssel-Werte-Paare, so daß Rechner hierüber beispielsweise elementare Informationen wie Adressen von allozierten transaktionalen

Speicherblöcken austauschen können.

```
void *nameservice_get(const char *id);
```

Die Funktion `nameservice_get()` liefert den Wert eines im Aufrufparameter referenzierten Schlüssels zurück. Bei dem Schlüssel handelt es sich um einen hierarchischen Pfad in Form eines Strings (beispielsweise »`/xxx/yyy`«). Bei dem zurückgelieferten Wert handelt es sich um einen Zeiger, der im Kontext der Anwendung frei interpretierbar ist (zum Beispiel durch Umwandlung auf elementare numerische Datentypen). Für nicht existierende Schlüssel liefert die Funktion einen Nullzeiger zurück.

```
void nameservice_set(const char *id, void *val);
```

Die Funktion `nameservice_set()` schreibt analog zur Funktion `nameservice_get()` den Wert eines referenzierten Schlüssels in den Namensdienst. Als zusätzlichen Aufrufparameter erwartet sie neben dem Pfad als Schlüssel den zu schreibenden Wert.

2.10 Verwandte Arbeiten

Bei *Typed Grid Object Sharing (TGOS)* [94] handelt es sich um ein Programmiermodell für die Entwicklung verteilter Anwendungen, wobei das Hauptaugenmerk auf virtuellen verteilten Welten liegt. TGOS abstrahiert genauso wie diese Arbeit datenzentrierte Anwendungsaspekte von der Netzwerkarchitektur und -kommunikation. Der zentrale Unterschied beider Arbeiten liegt darin, daß TGOS statt einer transaktionsbasierten Datenkommunikation eine Handvoll von Grundoperationen definert, die auf typisierte programmiersprachliche Objekte anwendbar sind. Mittels dieser Operationen lassen sich unterschiedliche Konsistenzmodelle bauen, welche Objektreplikate über Datenknoten zwischen mehreren Clientknoten synchronisieren. TGOS bietet über seine Basisoperationen auch transaktionale Konsistenz an. Diese basiert auf einer Rückwärtsvalidierung bei gleichzeitiger Aktualisierung von Objektreplikaten. TGOS stellt demnach andere Anforderungen an die Datenkonsistenz, da vor allem bei verteilten virtuellen Welten eine strenge Synchronisierung aus Leistungsgründen nicht immer vorteilhaft beziehungsweise notwendig ist. Daher lassen sich beide Ansätze nicht direkt miteinander vergleichen, da sie unterschiedliche Zielsetzungen haben.

Sinfonia [4] ist ein verteiltes Speichersystem. Anwendungen können über sogenannte Minitransaktionen mit dem verteilten Speicher kommunizieren und und hierüber durch den Austausch von Daten verteilte Anwendungen erstellen. Der gemeinsame Speicher ist hierbei über mehrere Speicherknoten verteilt. Sinfonia ist mit dieser Arbeit nur schwer vergleichbar, da dessen Zielsetzung auf Datenzentren liegt, in denen viele Rechner über Hochgeschwindigkeitsnetzwerke mit geringen Latenzen und geringer Ausfallwahrscheinlichkeit miteinander verbunden sind. Im Vergleich adressiert diese Arbeit explizit auch Weitverkehrsnetze, wodurch sich besondere Anforderungen an Skalierbarkeit bei hohen Latenzen und Fehlerszenarien ergeben.

Ein anderes Betriebssystem, das komplett verteilt im transaktionalen Speicher liegt, ist *Plurix* [92]. Dieses System hat eine andere Architektur als Betriebssysteme, die Anwendungen als Prozesse ausführen. Plurix behandelt aufrufbare Funktionen als Transaktionen. Es delegiert diese beim Aufruf an einen Transaktionsscheduler, der diese dann ausführt. Die verteilte Kommunikation erfolgt über einen *Pageserver*, der den verteilten Speicher verwaltet und synchronisiert. Plurix hat seinen Fokus ebenfalls nur auf Clustersysteme.

Der gemeinsame verteilte transaktionale Speicher kann wie hybride transaktionale Speicher durch Mechanismen von HTM-Systemen profitieren. Matveev et al. [67] haben einen virtuellen Filter für Speicherzugriffe (*Virtual Memory Filter (VMF)*) entwickelt, der Zugriffe auf

unterschiedlichen Granulariätsstufen erkennen kann. Da die Erkennung von Speicherzugriffen mithilfe der Speicherverwaltungseinheit mehreren Einschränkungen unterliegt kann der Filter Speicherzugriffe effizienter erkennen. Angesichts des Verteilungsaspekts des gemeinsamen verteilten transaktionalen Speichers kann der Filter in Abhängigkeit von Zugriffsmustern (False Sharing, Caching und Aggregation von Daten) die Granularität variieren und so unnötige Transaktionskonflikte durch Hardwareunterstützung vermeiden.

Vorarbeiten zu dem gemeinsamen verteilten transaktionalen Speicher basieren hinsichtlich des Verteilungsaspekts überwiegend auf DSM-Systemen. Die strenge Synchronisierung von Daten fußt dagegen auf transaktionalen Speichersystemen für Mehrkern- und Mehrprozessorarchitekturen. Bennett et al. haben mit *Munin* [13] einen DSM entwickelt, welcher nur explizit durch den Programmierer markierte Variablen über Rechnergrenzen verteilt. In ähnlicher Weise arbeiten objektbasierte Systeme wie *Orca* [11] und *Rthreads* [32], indem sie statt Variablen programmiersprachliche Objekte verteilt verwenden. Alle drei Systeme unterliegen großen Einschränkungen da sie entweder einen speziellen Compiler oder eine spezielle Laufzeitumgebung benötigen. Die Objekterkennung dieser Arbeit ist dagegen aufgrund der Unterstützung durch die Speicherverwaltungseinheit transparent und unabhängig von Programmiersprachen und Laufzeitumgebungen. Dies bietet Anwendungsentwicklern wesentlich mehr Flexibilität, da Programmiersprachen Implementierungen auf dieser Ebene leicht integrieren können (zum Beispiel als native Systembibliothek in Java).

Die drei genannten Ansätze vermeiden False-Sharing-Situationen, da sie die Objektgranularität implizit über programmiersprachliche Objekte beziehungsweise Variablen definieren. Der seitenbasierte Ansatz in dieser Arbeit kann zwar False Sharing verursachen, dieser läßt sich aber – wie in Kapitel 2.3 erwähnt – mittels Aggregation von Transaktionsobjekten und variabler Objektgranularität vermeiden. Der seitenbasierte Ansatz hat gegenüber den programmiersprachlich objektbasierten jedoch den Vorteil, daß bei der Synchronisierung automatisch ein Caching noch nicht zugegriffener Daten stattfindet, sofern sich mehrere Objekte oder Variablen auf derselben Speicherseite befinden. Munin verwendet für die Zugriffserkennung auf programmiersprachliche Objekte genauso wie diese Arbeit die Speicherverwaltungseinheit des Prozessors. In Rthreads erfolgt die Zugriffserkennung und Operationen auf gemeinsame Objekte mittels expliziter Funktionsaufrufe.

Eine weitere Eigenschaft von Munin ist, mehrere kleine Objektaktualisierungen verzögert, dafür aber gebündelt über das Netzwerk zu übertragen. Diese Eigenschaft ist in dieser Arbeit bereits implizit durch die Transaktionen gegeben, da ein Commit alle Objekte einer Transaktion gebündelt invalidiert beziehungsweise aktualisiert. Zudem ist die Bündelung unabhängig von Commit-Verfahren. Orca modifiziert Objekte mittels Funktionen. Dabei schickt es die Funktionsparameter an entfernte Rechner und führt die Funktionen dort aus. Dies gleicht einem Aktualisierungsverfahren. Rthreads verwendet ebenfalls eine Aktualisierungssynchronisierung. Munin dagegen verwendet das Aktualisierungsverfahren, zusätzlich aber auch eine Objektinvalidierung. Ähnlich zu Munin ist Treadmarks [6], verwendet stattdessen aber ein leicht modifiziertes Konsistenzprotokoll und das Invalidierungsverfahren. Optimal wäre, sowohl das Invalidierungs- als auch das Aktualisierungsverfahren dynamisch einsetzen zu können, je nachdem wieviele andere Rechner an Objektaktualisierungen interessiert sind. Diese Arbeit stützt sich zwar ebenfalls auf das Invalidierungsverfahren, aber nur um Commits möglichenfalls zu vermeiden und Lokalität auszunutzen. Bei lokalen Commits handelt es sich um Protokolloptimierungen, deren Diskussion in Kapitel 6.2 stattfindet.

Anwendungen können ebenso verteilt über die Kapselung der Verteilungsaspekte durch Betriebssysteme erfolgen. Zu rechnerübergreifenden Betriebssystemen, welche über einen DSM kommunizieren, zählt beispielsweise *Kerrighed* [101]. Dieses bündelt die Ressourcen über auf den Rechnern laufende Betriebssysteminstanzen und repräsentiert sich durch das Konzept eines *Single System Image* nach außen als einen einzelnen leistungsstarken Mehrprozessorrechner. Kerrighed unterstützt keine Transaktionen und kommuniziert auch nur auf Betriebs-

systemebene über den DSM, um Ressourcen zu koordinieren und diese Prozessen zuzuweisen. Durch die transparente Verteilung von Prozessen obligen etwaige Synchronisierungsmaßnahmen bei der Entwicklung verteilter Anwendungen deshalb dem Programmierer. Kerrighed ist ein Betriebssystem für Clustersysteme und hat seinen Fokus deswegen auch nicht auf Weitverkehrsnetze.

Ein ähnlichen Ansatz verfolgen Chapman et al. mit *vNUMA*, indem sie mittels Virtualisierung und DSM für Betriebssysteme transparent eine ccNUMA-Multiprozessorarchitektur simulieren [23]. Dieses System setzt oberhalb von Betriebssystemen an und simuliert im Falle von Linux nur ein Multiprozessorsystem mit einer ccNUMA-Speicherarchitektur. Deshalb bietet Linux unter diesem System keine weitere Funktionalität und ebenso keine Transaktionsunterstützung. Entwickler verteilter Anwendungen müssen die Synchronisierung von Prozessen und Threads bei diesem System genauso wie bei Anwendungen auf SMT- (engl. Simultaneous Multithreading) oder SMP-Systemen vollziehen. Dieses System hat einen ähnlichen Kofus wie Kerrighed, indem es Hardwareressourcen zu einem Gesamtsystem bündelt, daher ist eine Vergleich mit diesem System nicht vollständig möglich.

2.11 Zusammenfassung

Das in diesem Kapitel diskutierte Programmiermodell erlaubt Anwendungsentwicklern das einfache Erstellen verteilter Anwendungen und ergänzt somit bestehende Verfahren wie Message Passing und entfernter Methodenaufruf um eine weitere Technik. Bei dem gemeinsamen verteilten transaktionalen Speicher müssen sich Entwickler nicht mit Netzwerkkommunikation, relevanten Protokollen und deren Entwicklung sowie der Programmierung von Sockets als Kommunikationsendpunkte auseinandersetzen. Die Absicht besteht darin, daß verteilte Anwendungen auf beliebigen Linux-Betriebssystemen laufen und andere lokale Prozesse nicht beeinflussen. Daher ist das einzige sinnvolle Speichermodell, dedizierte Speicherblöcke in Anwendungsprozesse einzublenden und über Prozeßgrenzen unsichtbar miteinander zu synchronisieren. Ein wesentlicher Punkt der Entwicklung war es, möglichst unabhängig von Programmiersprachen und Laufzeitumgebungen zu sein, was sich maßgeblich auf die Erkennung von Speicherzugriffen auf Variablen und Datenstrukturen von Hochsprachen im verteilten Speicher auswirkt. Für die Unabhängigkeit kommt eine Erkennung auf Variablen- oder Objektebene nicht in Frage. Daher erfolgt die Zugriffserkennung auf Hardwareebene mittels der Speicherverwaltungseinheit des Prozessors. Jene bietet überdies den Vorteil, daß sie von anderen Rechnern allozierte transaktionale Speicherblöcke für einen anderen Anwendungsprozeß transparent im Hintergrund auf den lokalen Speicher abbildet. Somit ist eine explizite Allozierung eines Speicherblocks systemweit nur einmal notwendig und steht somit automatisch allen Rechnern zur Verfügung. Die Speicherverwaltung ist demgemäß ähnlich zu lokalen Multithreading-Anwendungen.

Ebenso ist die Deklaration von Transaktionsgrenzen für die Transparenz für Programmiersprachen wichtig. Transparenz bedeutet diesseits, daß die Transaktionen nicht auf Konstrukte einzelner Programmiersprachen wie Schleifen, Schlüsselwörter und Makros basieren. So können anderen Programmiersprachen die Implementierung eines softwareverteilten transaktionalen Speichers, der sich nur über Funktionen steuern läßt, leicht verwenden. Die Implementierung kann beispielsweise als Systembibliothek erfolgen, die andere Anwendungen einbinden. Dies ist generell unproblematisch, da viele Systemfunktionen in Linux in Bibliotheken gekapselt sind. Programmiersprachen können daher teilweise diese Bibliotheken laden, um ihre Funktionalität zu erweitern.

Die mit der Zugriffserkennung einhergehenden Nachteile wie False Sharing, die falsche Konflikte bei der Transaktionsvalidierung provozieren können, lassen sich durch eine variable Objektgranularität und Aggregation von Daten vermeiden. So besteht in dieser Hinsicht kein

Nachteil gegenüber objekt- und variablenbasierten Systemen, aber dennoch ist diesen gegenüber als Vorteil ein automatisches Zwischenspeichern von Daten über die Granularität der Zugriffserkennung möglich. Optimistische Synchronisierung und Schattenkopien machen es möglich, daß die Transaktionen als Synchronisierungskonstrukte ohne Sperren auskommen. Transaktionen reduzieren außerdem die Fehleranfälligkeit in der Entwicklung verteilter Anwendungen im Vergleich zu traditioneller Synchronisierung mittels Sperren wie beispielsweise bei Message Passing [88]. Verteilte Sperren sind zudem schwieriger als solche handzuhaben, die mehrere Threads innerhalb eines Prozesses synchronisieren. Weiterhin vermeiden die fehlenden Sperren bei der optimistischen Synchronisierung auch Verklemmungen von verteilten Anwendungen.

Transaktionen nehmen dem Programmierer die Arbeit der Datensynchronisierung ab. Außer der expliziten Definition von Transaktionsgrenzen sind keine weiteren Arbeiten nötig, da die Transaktion Zugriffe auf den transaktionalen Speicher automatisch erkennt. Ebenso findet standardmäßig für die Anwendung transparent eine automatische Neuausführung von abgebrochenen konfliktbehafteten Transaktionen statt.

Berücksichtigen muß der Programmierer im Vergleich dazu Aspekte der Heterogenität und der im Speicher abgelegten Daten, da der transaktionale Speicher die abgelegten Daten als nicht typsichere Rohdaten betrachtet. Vor allem Zeiger im transaktionalen Speicher verlangen je nach Zielreferenz besondere Aufmerksamkeit, da sie ansonsten teilweise ungültig sind. Dieses Problem läßt sich aber durch die Substitution der Zeiger durch UUIDs vermeiden.

3 Transparente Transaktionsrücksetzung

Die Rücksetzung von Transaktionen dient der Auflösung von Konflikten, die zwischen überlappenden Transaktionen entstehen können. Welche Transaktionen zurückzusetzen sind, entscheidet dabei die Transaktionsvalidierung in Abhängigkeit des verwendeten Commit-Protokolls. Entsprechend des Fokus dieser Arbeit umfaßt die Transaktionsrücksetzung in erster Linie nur transaktionale Speicherinhalte. Eine Erläuterung von Ein- und Ausgabeoperationen (E/A) des Systems in Transaktionen findet in Kapitel 3.2.3 statt. Bei der Transaktionsrücksetzung verwirft das System in einem ersten Schritt alle bisher in der Transaktion getätigten Änderungen, indem es den ursprünglichen Zustand mithilfe der angelegten Schattenkopien (siehe Kapitel 2.3.2) wiederherstellt. Anschließend befindet sich der transaktionale Speicher, den die rückzusetzende Transaktion zugegriffen hatte, in dem vor Transaktionsbeginn vorliegenden Zustand. Der folgende Schritt bestimmt die weitere Vorgehensweise und definiert, wie die weitere Programmcodeausführung erfolgt. In der Regel folgt auf zurückgesetzte Transaktionen wie in DBS automatisch deren Neuausführung. Allerdings kann es in Anwendungen auch Situationen geben, in denen eine Neuausführung nicht sinnvoll ist. Deswegen kann das System auch auf eine automatische Neuausführung verzichten und die weitere Vorgehensweise dem Anwendungsentwickler überlassen (siehe Kapitel 2.8.2). Der Abbruch ohne Neuausführung erlaubt somit eine effizientere Nutzung von Transaktionen im Umfeld der verteilten Anwendungsentwicklung, dessen Diskussion gesondert in Kapitel 3.4 erfolgt.

3.1 Transparente Rücksetzung

Neben der Objektzugriffserkennung soll die Transaktionsrücksetzung im Hinblick auf Programmiersprachen ebenso einen hohen Grad an Transparenz aufweisen. Ziel ist es, daß Transaktionen für ihre Rücksetzung keine zusätzlichen programmiersprachlichen Konstrukte benötigen. Dies wurde auch bereits ansatzweise in Kapitel 2.8.1 diskutiert, indem Funktionsaufrufe (`BOT()` und `EOT()`) neue Transaktionen initialisieren beziehungsweise deren Kontext aufräumen und gleichzeitig die Transaktionsgrenzen festlegen. Die hier vorgestellte Rücksetzung nimmt auf die Implementierung eines Prototyps in der Programmiersprache *C* Bezug, die auch bei der Evaluation in Kapitel 7 zum Einsatz kommt. Viele andere Programmiersprachen können in der Programmiersprache C geschriebene Funktionen aufrufen. Folglich kann die Implementierung des gemeinsamen verteilten transaktionalen Speichers in dieser Sprache anderen Programmiersprachen als Basis für eigene verteilte Anwendungen dienen. Abbildung 3.1 zeigt ein Beispiel einer mittels Funktionsaufrufe deklarierten Transaktion.

Eine einfache Umsetzung einer Transaktionsvalidierung mit Rücksetzung läßt sich auch durch Programmschleifen mit Abbruchbedingung realisieren (siehe Abbildung 3.2) [94]. Die Abbruchbedingung prüft hierbei, ob die Validierungsphase der Transaktion erfolgreich war oder nicht. Im Falle eines Abbruchs – gesteuert durch die Funktion `validate()` – startet die Transaktion über die *do*- und *while*-Konstrukte neu. Dieser Ansatz ist weder für Anwendungsentwickler oder Hochsprachen transparent, noch können andere Hochsprachen auf diese Weise realisierte Implementierungen verwenden. Die gleiche Problematik besteht durch Verwendung

```
1  BOT();
2    TA-Operation 1;
3    TA-Operation 2;
4      ...
5    TA-Operation n;
6  EOT();
```

Abbildung 3.1: BOT/EOT-basierte Transaktion.

eines modifizierten Compilers, der die Transaktionsrücksetzung selbst verwaltet und eigenen Code an ensprechend markierten Stellen im Programmcode (zum Beispiel durch *Pragmas*) einfügt.

```
1  do {
2    TA-Operation 1;
3    TA-Operation 2;
4      ...
5    TA-Operation n;
6  } while (validate() != 0);
```

Abbildung 3.2: Schleifendarstellung einer Transaktion.

Die mittels Funktionen deklarierten Transaktionen sind zudem funktionsübergreifend definierbar (siehe Abbildung 3.3). Startet eine Transaktion beispielsweise innerhalb einer Funktion, so kann der zugehörige `EOT()`-Aufruf in einer untergeordneten oder in der hierarchischen Aufrufreihenfolge weiter oben stehenden Funktion erfolgen. An dieser Stelle ist wichtig, daß eine Anwendung die beiden Funktionen in demselben Threadkontext ausführt. Zum anderen werden lokale Variablen (beispielsweise Zählvariablen) in Funktionen bei der Deklaration von Transaktionen über Funktionen bei einer Rücksetzung ebenfalls zurückgesetzt (siehe Kapitel 3.3.2). Dies ist mittels Schleifen wie bei der do-while-Variante nicht möglich. Die funktionsübergreifende Defintion von Transaktionen ist beispielsweise wichtig, wenn deren Funktionen (`BOT()` und `EOT()`) selbst in Funktionen von Programmiergerüsten *(engl. Framework)* oder denen anderer Programmiersprachen, die C-Funktionen aufrufen können, eingekapselt sind.

```
1  void func_a()
2  {
3    BOT();
4      TA-Operation 1;
5      func_b(1);
6  }
7
8  void func_b(int x)
9  {
10     TA-Operation 2;
11       ...
12     TA-Operation n;
13   EOT();
14 }
```

Abbildung 3.3: Transaktion, die sich über mehrere Hochsprachenfunktionen erstreckt.

3.2 Rücksetzungsstrategien

Abhängig von der verwendeten Validierungsstrategie ergeben sich verschiedene Szenarien für Transaktionsabbrüche. Bei der Rückwärtsvalidierung kann eine abzuschließende Transaktion nur selbst abbrechen (siehe Kapitel 2.5.3). Der Abbruch einer Transaktion erfolgt dort immer nach Beendigung derselbigen (`EOT()`-Aufruf). Dies bedeutet, der Abbruch erfolgt synchron zur Programmausführung. Dieses Szenario ist weniger komplex als asynchrone Transaktionsabbrüche, da der Abbruch immer in einem unkritischen Programmabschnitt (siehe Kapitel 3.2.1) stattfindet. Für die automatische Wiederholung der Transaktion ist neben Wiederherstellung der Schattenkopien hinreichend, den Threadkontext auf den Zustand zurückzusetzen, der zu Beginn der Transaktion vorlag (siehe Kapitel 3.3).

Die Rücksetzung von Transaktion unter Anwendung der Vorwärtsvalidierung und der First-Wins-Strategie (siehe Kapitel 4.3) ist dagegen komplizierter, da die Rücksetzung hier asynchron zur Programmausführung erfolgt. Dies bedeutet, die Rücksetzung einer Transaktion kann jederzeit inmitten ihrer Ausführung auftreten.

3.2.1 Unmittelbare Transaktionsrücksetzung

Bei der Vorwärtsvalidierung mit First-Wins-Strategie startet die Validierungsphase auf einem Rechner unmittelbar, wenn er eine Commit-Benachrichtigung eines anderen erhalten hat. Die Validierungsphase vollzieht sich nebenläufig für alle derzeit laufenden Transaktionen. Ist eine Transaktion konfliktbehaftet, sendet der Validierungsthread ein Systemsignal (siehe Kapitel 2.4.2) an den Thread, der die konfliktbehaftete Transaktion ausführt. Ähnlich wie bei der automatischen Zugriffserkennung unterbricht der Thread die Programmausführung und ruft einen Signalhandler auf. Der Signalhandler macht daraufhin die Änderungen an dem transaktionalen Speicher rückgängig, stellt den Threadkontext wieder her und startet die Transaktion erneut. Der Signalhandler muß bei der Wiederherstellung zudem sicherstellen, eigenständig seinen Signalkontext aufzuräumen, da dies durch den unmittelbaren Rücksprung zum Transaktionsbeginn nicht automatisch durch das Betriebssystem erfolgt. Der Signalkontext beinhaltet im allgemeinen den Prozessorkontext (Prozessorregister) des unterbrochenen Threads, Signalsperren zur unterbrechungsfreien Ausführung des Signalhandlers und eventuell einen eigenen Kellerspeicher.

Transaktionaler Programmcode kann auch Funktionen enthalten, die nicht eintrittsinvariant sind. Bricht die Transaktion asynchron in solchen Funktionen ab, kann es genauso wie beim Aufruf solcher Funktionen innerhalb von Signalhandlern zu einem undefinierten Programmverhalten oder zur Verklemmung kommen (siehe Kapitel 2.4.3). Diese Beschränkung gilt zugleich auch für jeglichen Programmcode, der nicht wiedereintrittsfähig ist.

Rücksetzung in Systemfunktionen

An dieser Stelle findet eine genauere Beleuchtung statt, warum nicht eintrittsinvariante System- und Bibliotheksfunktionen (siehe Kapitel 2.8.3) eine asynchrone Transaktionsrücksetzung verbieten. Nicht wiedereintrittsfähiger Programmcode greift auf statische Variablen oder Felder außerhalb des transaktionalen Speichers zu. Befinden sich solche Zugriffe innerhalb von Systemfunktionen oder Bibliotheken, so sind sie für den Programmierer nicht sichtbar. Bricht die Transaktion in solchen Codeabschnitten ab, bleiben diese oder deren Funktionen in einem undefinierten Zustand, der zu einem Fehlverhalten bei einem erneuten Aufruf führen kann. Der folgende Codeabschnitt in Abbildung 3.4 zeigt beispielhaft, wie ein asynchroner Transaktionsabbruch innerhalb einer nicht eintrittsinvarianten aber dennoch threadsicheren Funktion zu einer Verklemmung führt.

```
1  void mt_sysfunc()
2  {
3    static mutex_t syslock = MUTEX_INITIALIZER;
4
5    mutex_lock(&syslock);
6      Operation 1;            <-- Transaktionsabbruch
7      Operation 2;
8    mutex_unlock(&syslock);
9  }
```

```
1  void func()
2  {
3    BOT();
4      TA-Operation 1;
5      mt_sysfunc();
6      TA-Operation 2;
7    EOT();
8  }
```

Abbildung 3.4: Transaktionsabbruch in einer von einer Transaktion aufgerufenen Systemfunktion.

Die Funktion `func()` führt in der Anwendung eine Transaktion aus, die neben mehreren Operationen die Systemfunktion `mt_sysfunc()` aufruft. Bei dieser Funktion handelt es sich um eine threadsichere Funktion, weswegen sie für ihre serialisierte Ausführung Sperren benutzt. Die Sperre liegt statisch im nicht transaktionalen Speicher, und ihr Zustand bleibt über den Funktionsaufruf hinaus erhalten. Setzt in Zeile [6] der Systemfunktion `mt_sysfunc()` ein Transaktionsabbruch ein, beginnt die Transaktion erneut bei der ersten Hochsprachenanweisung nach Transaktionsbeginn (Zeile [5] der Funktion `func()`).

```
1    BOT();
2      TA-Operation 1;
3           mutex_lock(&syslock);
4             Operation 1;
5      TA-Operation 1;
6           mutex_lock(&syslock);     <-- Verklemmung
7             Operation 1;
8             Operation 2;
9           mutex_unlock(&syslock);
10     TA-Operation 2;
11   EOT();
```

Abbildung 3.5: Bildliche Abfolge von Hochsprachenanweisungen einer neu gestarteten Transaktion.

Demnach ergibt sich nach Abbildung 3.5 folgende Abfolge von Hochsprachenanweisungen, die Operationen der Systemfunktion `mt_sysfunc` sind in blauer Farbe dargestellt. Funktionsaufrufe sind der Übersichtlichkeit halber nicht aufgeführt. Nach Abarbeitung der Anweisung in Zeile [4] startet die Transaktion neu und führt die erste Anweisung der Transaktion erneut aus. In Zeile [6] tritt die Transaktion erneut in die Systemfunktion ein und verursacht eine Verklemmung, da sie in demselben Thread ein weiteres Mal die bereits erhaltene Sperre akquirieren will. Die Sperrfunktion blockiert solange, bis der Thread die bereits in Zeile [3] erfolgreich erhaltene Sperre wieder freigibt, was dieser aber nicht leisten kann.

Ein klassisches Beispiel ist die Funktion `printf()`, die threadsicher ist, damit die Ausgabe eines einzelnen Funktionsaufrufs atomar und nicht verschachtelt mit denen aus anderen

Threads erfolgt. Hierzu serialisiert die Funktion die Ausgaben mittels Sperren, indem sie vor der Ausgabe die Sperre setzt und anschließend wieder freigibt. Bricht eine Transaktion nach Setzen der Sperre ab, führt ein erneuter Aufruf der Funktion `printf()` zu einer Verklemmung.

3.2.2 Verzögerte Transaktionsrücksetzung

Eine verzögerte Transaktionsrücksetzung bis zum `EOT()`-Aufruf vermeidet ein undefiniertes Anwendungsverhalten beziehungsweise Absturz oder eine Verklemmung aufgrund eines Transaktionsabbruchs in einer nicht eintrittsinvarianten Funktion, da der Transaktionsabbruch hier analog zur Rückwärtsvalidierung in einem unkritischen Bereich erfolgt. Dies führt zwar zu einem Zeitverlust, da Transaktionen trotz ihres bereits vorgesehenen Abbruchs dennoch bis zum Aufruf von `EOT()` zu Ende laufen. Dies ist aber angesichts der zuverlässigen Ausführung von System- und Bibliotheksfunktionen vertretbar.

Muß das System eine Transaktion aufgrund eines Konflikts abbrechen, merkt sich der Transaktionsmanager die Transaktion lediglich für einen Abbruch vor, statt sie unverzüglich abbrechen zu lassen. Nach Beendigung der Transaktion mittels `EOT()` prüft der Transaktionsmanager vor Einleitung der Validierungsphase, ob er die Transaktion bereits früher für einen Abbruch vorgemerkt hat. Ist dies der Fall, startet er die Transaktion automatisch neu oder bricht sie endgültig ab.

Beide Rücksetzungsstrategien lassen sich miteinander kombinieren, indem der Entwickler innerhalb einer Transaktion explizit kritische Codeabschnitte markieren kann. Im Falle eines Transaktionsabbruchs verfährt das System zunächst wie in Kapitel 3.2.1. Befindet sich die Transaktion aber während eines vorgesehenen Abbruchs gerade in einem kritischen Bereich, wendet das System die synchrone (verzögerte) Rücksetzungsstrategie an. Die Transaktion braucht bei der verzögerten Abbruchstrategie jedoch nicht bis zum Aufruf von `EOT()` zu Ende laufen, sondern kann unverzüglich nach Verlassen des kritischen Codeabschnitts abbrechen. Dies spart im Vergleich zur ausschließlich verzögerten Abbruchstrategie Zeit.

3.2.3 Nichtrücksetzbare Operationen

Besondere Berücksichtigung benötigen Operationen, die sich im Konfliktfall nicht vollständig zurücksetzen lassen. Verbietet man solche Operationen in Transaktionen, ist deren Rücksetzung unproblematisch, schränkt aber die Verwendung von Transaktionen ein. Die Nichtrücksetzbarkeit betrifft generell ungepufferte E/A-Operationen und Funktionen, die auf nicht transaktionalem Speicher arbeiten, wie dies beispielsweise bei der *Halde* (siehe Abbildung 2.8) der Fall ist. Bei E/A-Operationen können Eingaben bei einer wiederholten Ausführung der Transaktion entweder verloren gehen (zum Beispiel Tastatureingaben) oder andere Werte liefern. Ausgabeoperationen können im Vergleich mehrfach erfolgen (beispielsweise Bildschirmausgaben). Sofern Ein- und Ausgabeoperationen nicht verschachtelt in Transaktionen auftauchen, sind diese Einschränkungen mithilfe von E/A-Pufferung *(Smart Buffers)* umgehbar [15]. Eine Transaktion hält Eingabedaten, die innerhalb einer Transaktion anfallen, als Kopie in einem Puffer vor. So kann sie dieselben Eingabedaten bei einem Neustart der Transaktion wiederverwenden. Ohne Zwischenpufferung wären diese Daten bei der wiederholten Ausführung der Transaktion verloren. Für die Ausgabedaten geschieht dies analog. Die Transaktion hält diese Daten in einem Puffer vor und gibt diese erst nach einem erfolgreichen Commit an das Ausgabegerät weiter. So erhält das Ausgabegerät keine Daten von abzubrechenden Transaktionen.

Smart Buffers funktionieren nicht, wenn sich die verschachtelte Ein- und Ausgabe gegenseitig bedingen und beispielsweise die Eingabe von der vorangehenden Ausgabe abhängig ist. Ein

einfaches Beispiel ist eine Tastatureingabe, auf die eine Anwendung nach Ausgabe einer Bildschirmmeldung wartet. Puffert ein Smart Buffer die Ausgabe, weiß der Anwender nicht, daß die Anwendung eine Tastatureingabe erwartet.

Eine weitere Möglichkeit, E/A-Operationen zurückzusetzen, ist, kompensierende Operationen bei der Rücksetzung von Transaktionen auszuführen. Diese Technik setzen Volos et al. in *xCalls* [103] für die Rückabwicklung bestimmter Systemaufrufe des Linux-Betriebssystemkerns ein. Die Kompensationsmethode ist nur auf solche E/A-Operationen anwendbar, die sich vollständig kompensieren lassen. Damit die Transaktionsrücksetzung die Operationen kompensieren kann, muß es diese während der Transaktionsausführung genauso wie die Speicherzugriffe aufzeichnen.

3.3 Kontextverwaltung für Transaktionsrücksetzung

Damit die Rücksetzung von Transaktionen für Anwendungen transparent ist, muß es möglich sein, von der Funktion `EOT()` die Programmablaufkontrolle direkt an die Funktion `BOT()` zu übertragen, so daß die Programmausführung bei der ersten Instruktion fortfährt. Neben dem Rücksprung muß der Threadkontext so modifiziert werden, daß dieser zum geänderten Programmablauf paßt, da ansonsten ein undefiniertes Programmverhalten bis hin zum Absturz auftreten kann. Die zentralen Bestandteile eines Threadkontexts sind Prozessor- und Kellerspeicherzustand. Der Ablauf einer Tranaktionsrücksetzung ist schematisch in Abbildung 3.6 dargestellt.

```
1  BOT();
2            <Kellerspeicher sichern>
3            <Prozessorregister sichern>
4  <Beginn des Transaktionskontexts>
5      TA-Operation 1;
6        ...
7      TA-Operation n;
8  <Ende des Transaktionskontexts>
9  EOT();
10           <Kellerspeicher wiederherstellen>
11           <Prozessorregister wiederherstellen>
```

Abbildung 3.6: Wiederherstellung des Threadkontexts bei einer Transaktionsrücksetzung.

Die bei einer Transaktionsrücksetzung betroffenen Prozessorregister und Operationen auf dem Kellerspeicher sind vom Betriebssystem (Linux), der Hardwarearchitektur (IA-32 und x86-64) und der verwendeten Aufrufkonvention für Funktionen abhängig. Die Konventionen sind in dem *Application Binary Interface (ABI)* festgelegt [9, 10, 68].

3.3.1 Prozessorkontext

Der Prozessorkontext besteht aus den in der Anwendung verwendeten Prozessorregistern, die für eine Restaurierung des Threadkontexts von Bedeutung sind. Alle anderen Register des Prozessors sind für eine Transaktionsrücksetzung nicht relevant, da Anwendungen diese nicht verändern, respektive wegen ihrer niedrigen Privilegienstufe teilweise auch nicht zugreifen können. IA-32- und x86-64-Register, welche die Transaktionsrücksetzung berücksichtigen muß, sind in Tabelle 3.1 aufgeführt. Die Anzahl der zu sichernden und wiederherzustellenden Register sind von der Prozessorhardware abhängig, daher können je nach Prozessor auch noch

x86-64	IA-32	Beschreibung
RAX, RBX, RCX, RDX	EAX, EBX, ECX, EDX	Vielzweckregister
RSI, RDI	ESI, EDI	Indexregister
RBP	EBP	Basiszeiger (für Kellerrahmen)
RSP	ESP	Kellerzeiger
RIP	EIP	Befehlszeiger
RFlags	EFlags	Flagregister
r8, . . . , r15	—	Vielzweckregister (nur 64-Bit)
ST(0), . . . , ST(7)	ST(0), . . . , ST(7)	Coprozessorregister

Tabelle 3.1: Relevante Prozessorregister eines anwendungsbezogenen Threadkontexts (Auszug).

Register durch Erweiterungen wie *Single Instruction Multiple Data (SIMD)* [52] hinzukommen. Segmentregister brauchen bei der Transaktionsrücksetzung nicht berücksichtigt werden, da Linux für Anwendungsprozesse ein flaches Speichermodell *(engl. Flat Memory Model)* verwendet, in der jedes Segment eines Typs nur einmal vorhanden ist und eine maximale Ausdehnung über den virtuellen Adreßraum besitzt. Somit sind Intersegmentsprünge auf Maschinencodeebene nicht notwendig. Genauso liegen die Anwendungsdaten in nur einem Segment. Dies bedeutet, der Inhalt der Segmentregister ändert sich zur Laufzeit einer Anwendung in ihrem Ausführungskontext nicht.

3.3.2 Keller-Organisation

Jeder Thread einer Anwendung besitzt einen eigenen Kellerspeicher, auf den er lokale Funktionsvariablen, Rücksprungadressen von Funktionsaufrufen oder zwischenzeitlich auch Inhalte von Prozessorregistern sichert. Je nach Aufrufkonvention für Funktionen legen Anwendungen dort auch Aufrufparameter und Rückgabewerte von Funktionen ab. Deshalb ist der Keller von den Prozessorregistern – vor allem vom Kellerzeiger – abhängig. Daher erfordert eine Rücksetzung der Prozessorregister auch immer ein Zurückschneiden des Kellerspeichers auf seine ursprüngliche Größe zu Transaktionsbeginn. Damit ist es möglich, den `EOT()`-Aufruf in derselben Funktion wie den `BOT()`-Aufruf oder einer von ihr aufgerufenen Unterfunktionen auszuführen, da der Keller beim Aufruf von Unterfunktionen nur wachsen, sich aber nicht verkleinern kann.

Umgekehrt ist es jedoch nicht möglich, `EOT()` relativ zu `BOT()` in einer hierarchisch übergeordneten Funktion der Aufrufreihenfolge auszuführen, da der Keller dann kleiner als zu Beginn einer Transaktion wäre und sich nicht zurückschneiden ließe. Dies läßt sich verhindern, indem man zu Beginn einer Transaktion ein komplettes Abbild des Kellers anlegt und beim Abbruch einer Transaktion zurückspielt, anstatt ihn zurückzuschneiden. Dies hat zudem den Vorteil, daß eine Transaktionsrücksetzung ebenso die geänderten lokalen Variablen einer Funktion zurücksetzt, da sich diese auf dem Keller befinden. Dies ist vorteilhaft, wenn eine Transaktion beispielsweise Zählschleifen benutzt, da der Entwickler die Zählvariable dann nicht manuell zurücksetzen braucht.

Ein Nachteil der Kellerrückschneidung, der in Programmiersprachen auftreten kann, ist, daß eine Variablendeklaration bei gleichzeitiger Initialisierung derselbigen nicht immer der Semantik entspricht, wie vom Programmierer angegeben. In dem zweiten Beispiel in Abbildung 3.7 kann die Deklaration der Variable bei manchen Programmiersprachen am Anfang eines Funktionsaufrufs erfolgen, obwohl die Deklaration innerhalb der Transaktion erfolgt. Somit würde das zweite Beispiel dem Verhalten des ersten entsprechen. In dem zweiten Beispiel würde demnach eine zurückgesetzte Transaktion die Variable weiterhin hochzählen, obwohl dies der

```
1  int x = 3;
2
3  BOT();
4    x++;
5  EOT();
```

```
1  BOT();
2    int x = 3;
3    x++;
4  EOT();
```

Abbildung 3.7: Nutzung des transaktionalen Speichers mit zwei Rechnern.

Programmsemantik widerspricht. Daher ist es generell sinnvoll, anstatt den Keller zurückzuschneiden, diesen ebenfalls wie Änderungen im transaktionalen Speicher zurückzusetzen. Im letzteren Fall tritt die fehlerhafte Hochzählung der Variable x im Speicher nicht auf.

Erkennung des Kelleranfangs in Linuxanwendungen

Auf der x86-Architektur wächst der Keller von höheren zu niedrigeren Adressen. In der Regel besteht der Keller in Anwendungen aus einer Verkettung von Kellerrahmen *(engl. Stackframe)*, wobei jeder Rahmen den Kontext einer Funktion (Rücksprungadresse, gesicherte Register, lokale Variablen und Parameter aufgerufener Funktionen) speichert. Ruft eine Anwendung eine neue Funktion auf, so legt sie auf dem Keller einen neuen Rahmen an. Verläßt sie dagegen eine Funktion, löscht sie den zugehörigen Rahmen und kehrt zum vorherigen zurück. Abbildung 3.8 zeigt schematisch den Aufbau eines Threadkellers. Als *lokale Daten* deklarierte Abschnitte dienen lokalen Daten (Skalare oder Feldern) der Funktion, wobei die ABI weder das Format noch die Größe der abgelegten Daten vorschreibt. Eventuell muß die aufgerufene Funktion Prozessorregister sichern, deren Inhalte sie ebenfalls auf dem Keller ablegt.

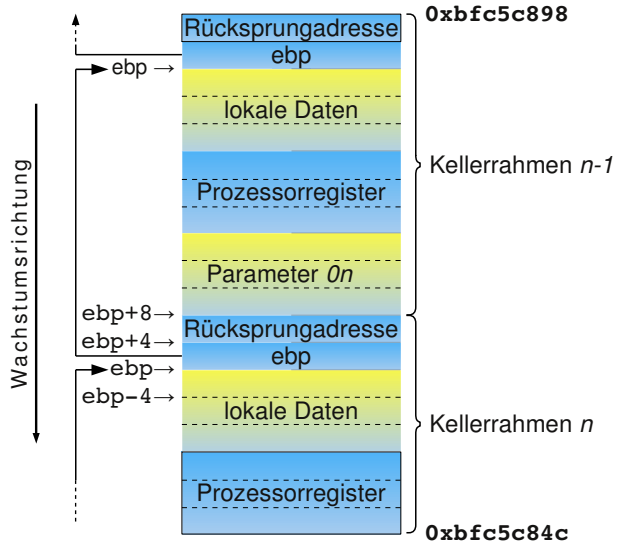


Abbildung 3.8: Schematischer Aufbau des Aufrufkellers eines Threads nach ABI (32-Bit).

Das Register `ebp` enthält einen Zeiger auf den aktuellen Kellerrahmen. Nach dem Aufruf einer Funktion sichert die aufgerufene Funktion den Inhalt des `ebp`-Registers auf dem Keller, bevor es dessen Inhalt an den neuen Kellerrahmen anpaßt. Somit ergibt sich über den aktuellen Wert des `ebp`-Registers und die auf dem Keller gesicherten Inhalte des Registers eine Verkettung der Kellerrahmen. Über eine Dereferenzierung ließe sich so der Beginn des Kellers ermitteln. Jedoch ist die Sicherung des Registers auf dem Keller nach der ABI optional, so daß der Keller

dann keine Zeiger auf die Kellerrahmen enthält. Wenn die Anwendung keine Zeiger auf die Kellerrahmen auf dem Keller speichert, aber auch bei optimiert kompiliertem Code, können Anwendungen das `ebp`-Register als weiteres Vielzweckregister verwenden. In diesem Fall läßt sich über dieses Register der Beginn des Kellers nicht herausfinden, da dessen Inhalt nicht immer auf einen Kellerrahmen zeigt beziehungsweise die Rückwärtsverkettung der Rahmen zeitweise unterbrochen sein kann. Daher ist die Ermittlung des Kelleranfangs nur mithilfe des Betriebssystems durch Aufruf einer Systemfunktion zuverlässig.

3.3.3 Kontextsicherung und -wiederherstellung

Bei der Sicherung und Wiederherstellung des Threadkontexts können sich die Operationen auf den Kellerinhalt und die Prozessorregister gegenseitig beeinflussen. Die Sicherungsfunktion für den Keller ändert den Inhalt der Prozessorregister. Genauso kann die Sicherungsoperation für die Prozessorregister auch den Kellerspeicher verändern. Hier müssen die Operationen so aufeinander abgestimmt sein, daß beide Teilzustände ein konsistentes Abbild des Threadkontexts erzeugen. Bezüglich Multithreading müssen die Sicherungsoperationen keine Vorkehrungen treffen, da jeder Thread einen eigenen Keller besitzt und die Inhalte der Prozessorregister nur für den gerade aktiven Thread Gültigkeit besitzen. Daher sichert jeder Thread seinen Kontext selbst, dies gilt ebenso für die Wiederherstellung.

Bei der Sicherung legt der Thread zunächst eine Kopie seines Kellers an und sichert anschließend seine Prozessorregister. Dies hat den Vorteil, daß der Thread die Sicherungsoperation nicht erneut ausführt, wenn er eine Transaktion nach einem Abbruch wiederholt ausführt. Der gesicherte Kellerinhalt bleibt unabhängig von der Anzahl der wiederholten Transaktionsausführungen weiterhin gültig. Für die Prozessorregister analog erfolgt die Sicherung des Befehlszeigers zuletzt, damit sich bei der wiederholten Ausführung der Transaktion die Sicherung der Register nicht wiederholt. Die Sicherung des Befehlszeigers (`eip`-Register) ist nicht trivial, da sich dieser mit jeder Maschinenspracheanweisung ändert. Daher ist es einfacher, stattdessen eine Sprungadresse zu sichern, die hinter dem Maschinencode liegt, der die Prozessorregister sichert. Der gesicherte Wert für den Befehlszeiger legt den Einsprungpunkt in den Maschinencode fest, falls der Thread eine Transaktion wiederholt ausführt.

Eine Transaktion startet unmittelbar erneut, wenn die Wiederherstellungsfunktion den Befehlszeiger (`eip`-Register) wiederherstellt, deshalb darf sie dieses Register in der Reihenfolge nur als letztes restaurieren. Zuvor stellt sie noch den gesicherten Kellerinhalt wieder her (siehe auch Abbildung 3.6). Die Adressierung des Kellers ist nach der Wiederherstellung des Threadkontexts automatisch korrekt, da der Basiszeiger auf den Kellerrahmen (`ebp`-Register) und der Zeiger auf das oberste Kellerelement (`esp`-Register) ihren zum wiederhergestellten Kellerinhalt korrespondierenden Inhalt haben.

3.4 Transaktionsabbruch ohne Neuausführung

Bei einem Transaktionsabbruch ohne wiederholte Ausführung, entweder manuell auf Anforderung der Anwendung oder konfliktbedingt, ist der Ausstieg aus dem Transaktionskontext auf unterschiedliche Weise möglich. Zunächst ist anzumerken, daß bei einem inmitten der Transaktion auftretenden Transaktionsabbruch kein transparenter Sprung zur `EOT()`-Funktion folgenden Hochsprachenanweisung möglich ist, da dessen Adresse im vorhinein nicht bekannt ist. Diese Vorgehensweise ließe sich nur mit Compilerunterstützung realisieren. Die Java-Laufzeitumgebung unterstützt dies zum Beispiel mittels des eigenen Exception-Mechanismus (Schlüsselwörter `try`, `catch` und `throw`).

Ein automatischer Abbruch wegen eines Transaktionskonflikts kann auf ähnliche Weise wie die Rücksetzung verfahren, bei der die Transaktion automatisch erneut startet. Das System setzt den Kellerspeicher und die Prozessorregister zurück. Bevor es aber den Befehlszeiger zurücksetzt, manipuliert es den Rückgabecode der Funktion `BOT()`, so daß die Anwendung über deren Rückgabecode erkennen kann, daß ein Transaktionsabbruch erfolgt ist. Die Hochsprache kann dann mit der der Transaktion nachfolgenden Anweisung fortfahren. Abbildung 3.9 illustriert dies durch ein Beispiel. `BOT()` kehrt nach einem Transaktionsabbruch mit dem Rückgabewert `-E_ABORT` zurück, woraufhin die Anwendung den Transaktionscode überspringt und den nachfolgenden Programmcode abarbeitet.

```
1  if (BOT() != -E_ABORT) {
2      TA-Operation 1;
3      TA-Operation 2;
4        ...
5      TA-Operation n;
6    EOT();
7   }
```

Abbildung 3.9: Integration von Transaktionen in Anwendungen, die bei einem automatischen Abbruch nicht erneut starten.

Bei einem manuellen Abbruch kann das System auf gleiche Weise wie in der Abbildung dargestellt verfahren. Auch lassen sich der automatische und manuelle Transaktionsabbruch kombinieren. Vorteil dieser Vorgehensweise ist, daß der Keller und somit auch die lokalen Variablen der Funktion dieselben Werte wie vor dem Beginn der Transaktion besitzen. Alternativ könnte die Transaktion bei einem manuellen Abbruch von der Abbruchfunktion zurückkehren (siehe Abbildung 3.10). Nachteilig ist diesbezüglich, daß eine Rücksetzung des Kellers im Gegensatz zu den Änderungen im transaktionalen Speicher nicht erlaubt ist, da dieser nicht zur weiteren Programmausführung ab der Abbruchfunktion konsistent wäre. Daher ist es sinnvoll, semantisch genauso wie bei dem automatischen Transaktionsabbruch vorzugehen und ab der Funktion `BOT()` mit der Programmausführung weiterzumachen. Ebenso ist zu berücksichtigen, daß eventuell nicht alle Programmiersprachen einen direkten Sprung aus dem Transaktionscode – wie in dem *C*-Pseudocode dargestellt – unterstützen. Damit ergäbe sich wieder eine Einschränkung auf bestimmte Programmiersprachen.

```
1  BOT();
2    TA-Operation 1;
3    TA-Operation 2;
4    if (expression) {
5      ta_abort();
6      goto label;
7    }
8      ...
9    TA-Operation n;
10 EOT();
11
12 label:
```

Abbildung 3.10: Alternative Integration von Transaktionen in Anwendungen, die bei einem manuellen Abbruch nicht erneut starten.

3.5 Absturz aufgrund veralteter Transaktionsobjekte

Beim replikatbasierten verteilten transaktionalen Speicher ist es möglich, daß Transaktionen veraltete Transaktionsobjekte lesen. Dies erkennt das System spätestens, wenn eine Transaktion unter Anwendung der Rückwärtsvalidierung in ihre Validierungsphase übergeht. Das System erkennt dies ebenso während der Transaktionsausführung, wenn es die Vorwärtsvalidierung mit First-Wins-Strategie (siehe Kapitel 2.5.3) verwendet. Somit ist zum Transaktionsende immer die Konsistenz des gemeinsamen verteilten transaktionalen Speichers garantiert. Der Zustand eines Prozesses basiert sowohl auf seinem lokalen Speicher als auch den Speicherblöcken des gemeinsamen tansaktionalen Speichers. Veraltete Inhalte in diesem können zum Absturz des transaktionsausführenden Prozesses führen, beispielsweise wenn dieser veraltete Zeiger im transaktionalen Speicher zugreift (siehe Kapitel 2.7.2).

Die Idee ist, Programmabstürze über den SIGSEGV-Signalhandler, der auch bei der transaktionalen Objekterkennung zum Einsatz kommt, abzufangen. Der Handler kann Abstürze vermeiden, wenn hierfür veraltete transaktionale Speicherobjekte ursächlich sind. Greift eine Anwendung wegen veralteter Zeiger auf ungültige Speicherbereiche (nicht vorhandene oder dem Betriebssystemkern zugeordnete Speicherseiten) zu, ruft das System genauso wie bei der Erkennung von transaktionalen Objektzugriffen den SIGSEGV-Handler (siehe Kapitel 2.4.2) auf, in diesem Fall um den Anwendungsprozeß zu beenden. Stellt der Handler fest, daß ein ungültiger Speicherzugriff für seinen Aufruf verantwortlich ist, dieser aber innerhalb einer Transaktion stattgefunden hat, kann er veraltete Replikate von Transaktionsobjekten als Ursache annehmen. In diesem Fall kann der Handler die Transaktion abbrechen und mit höchster Priorität erneut ausführen, so daß die wiederholte Transaktionsausführung keine veralteten Objektreplikate zugreift. Diese Behandlungsmethode ist jedoch nicht zuverlässig. Der Mechanismus kann nicht feststellen, ob ein Thread, für den der Signalhandler einen ungültigen Zugriff innerhalb einer Transaktion erkannt hat, nicht schon vor der Zugriffsverletzung aufgrund eines ungültigen Zeigers lokale Speicherbereiche überschrieben hat. Abbildung 3.11 verdeutlicht dies anhand eines Beispiels.

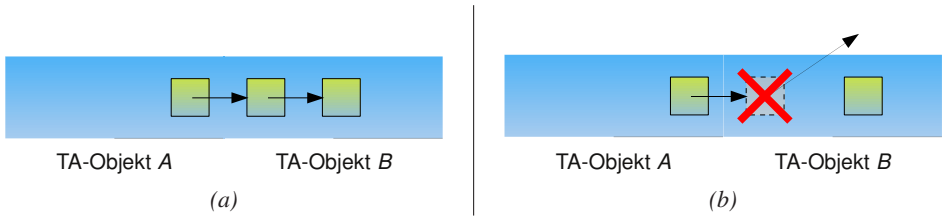


Abbildung 3.11: Programmabsturz wegen ungültiger Zeiger im gemeinsamen verteilten transaktionalen Speicher.

Abbildung 3.11a zeigt eine verkettete Liste, die sich über zwei Transaktionsobjekte erstreckt. Rechner 1 möchte das letzte Objekt der Liste in einer Transaktion aktualisieren, besitzt aber nur invalidierte Replikate der Transaktionsobjekte. Beim ersten Zugriff auf Objekt A fordert das System dieses transparent an. Nach dem Auslesen des Zeigers vom ersten auf das zweite Listenelement aktualisiert ein anderer Rechner beide Transaktionsobjekte, indem er das mittlere Listenelement entfernt und die Zeiger anpaßt. Rechner 1 traversiert nun mithilfe des Zeigers durch die Liste, um das letzte Element zu ändern. Da sich der ausgelesene Zeiger auf Rechner 1 bereits in einer lokalen Hilfsvariable oder einem Prozessorregister befindet, fordert der Rechner die neueste Version des bereits invalidierten Transaktionsobjekts A nicht an. Für Rechner 1 ergibt sich die in Abbildung 3.11b dargestellte Sicht, obwohl das erste Listenelement in der neuesten Version des Transaktionsobjektes A bereits direkt auf das letzte zeigt, da Rechner 2 das mittlere Element entfernt und diesen Speicherbereich möglicherweise für andere Daten verwendet hat.

Rechner 1 greift auf das nicht mehr existierende zweite Listenelement zu und versucht, in den ungültigen Daten einen weiteren Zeiger auf das dritte Listenelement zu dereferenzieren. Hierbei kann es vorkommen, daß dieser Zeiger außerhalb des transaktionalen Speicherbereichs auf lokale Daten zeigt, den der Rechner mit einer Änderungsoperation des dritten Listenlements überschreibt. Die Schreiboperation ist erfolgreich, da der ungültige Zeiger auf allozierte lokale Speicherbereiche verweist. Der SIGSEGV-Signalhandler wird deswegen nicht aufgerufen. Die Transaktion bricht in ihrer Validierungsphase ab, da sie eine veraltete Version des Transaktionsobjekts A verwendet hat. Die Rücksetzung stellt keine lokalen überschriebenen Daten her. Daher können Zeiger in veralteten Transaktionsobjekten lokale Daten zerstören, die sich wiederum später in einem undefinierten Programmverhalten oder einem Programmabsturz außerhalb von Transaktionen äußern können. Der Absturz muß dabei nicht notwendigerweise von demselben Thread ausgehen. Die einzige Möglichkeit, diese Problematik zu umgehen, besteht darin, mehrere Versionen eines jeden Transaktionsobjekts vorzuhalten, so daß keine Dereferenzierung veralteter Zeiger auftreten kann (siehe Kapitel 6.1).

3.6 Verwandte Arbeiten

Schoettner et al. haben mit Plurix ein Betriebssystem entwickelt, bei dem die Codeausführung vollständig transaktional erfolgt. Jeder Funktionsaufruf startet implizit eine Transaktion. Die Rücksetzung in Plurix erfolgt wie in dieser Arbeit auf gleiche Weise mittels Schattenkopien und über die MMU erkannte Seitenzugriffe. Da das System vollständig transaktional ist, operieren Systemfunktionen (E/A unberücksichtigt) ausschließlich auf dem transaktionalen Speicher. Somit ist die Rücksetzung von Systemfunktionen gegenüber Transaktionen in Linux unproblematisch. Weiterhin können in Plurix bei der Transaktionsrücksetzung auch keine Verklemmungen entstehen, da statische Daten und Felder ebenfalls im transaktionalen Speicher liegen und nicht wie in Linux außerhalb (siehe Kapitel 3.2.1).

Volos et al. haben in *xCalls* [103] mit dem Aufruf von Systemfunktionen und Ein- und Ausgabe (E/A) innerhalb von Transaktionen beschäftigt. xCalls definiert eine API, die transaktionale Semantiken für Systemaufrufe anbietet, indem sie die Aufrufe entweder verzögert ausführt oder mittels Kompensationsoperationen rückabwickelt. Weiterhin puffert xCalls Daten für Systemaufrufe. Die xCalls-Implementierung ist compilerspezifisch und im Gegensatz zu dieser Arbeit nicht transparent. Systemfunktionen die weder eine Verzögerung oder Kompensation gestatten, dürfen nur in unwiderruflichen Transaktionen aufgerufen werden, da ansonsten negative Seiteneffekte bei der Programmausführung entstehen können. Diese Problematik entspricht der in diesem Kapitel diskutierten verschachtelten Ein- und Ausgabe (siehe Kapitel 3.2.3). xCalls unterstützt zusätzlich noch Kernelobjekte. xCalls ist mit dieser Arbeit nicht direkt vergleichbar, da es darauf abzielt, Systemfunktionen transaktional auszuführen. Diesen Anspruch erhebt diese Arbeit nicht, da hier nur Speicherzugriffe auf transakionale Speicherblöcke dem Transaktionskontext unterliegen und nicht die Systemaufrufe selbst.

In *Unrestricted Transactional Memory* von Blundell et al. [17] erfolgt die Transaktionsausführung mit *restricted* und *unrestricted* in zwei unterschiedlichen Modi. Führt eine Transaktion Systemaufrufe aus oder verursacht Ein- und Ausgaben, wechselt sie automatisch vom restricted in den unrestricted Modus. Ein Prozeß kann zur selben Zeit nur eine Transaktion in dem letzteren Modus ausführen, dagegen aber viele gleichzeitig im restricted Modus. Wechselt eine Transaktion in den unrestricted Modus, so kann sie nicht mehr zurückgesetzt werden und setzt sich gegenüber anderen Transaktionen durch. Dieser Ansatz hat mit der Transaktionsausführung auf lokalen Rechnern eine andere Zielsetzung. Eine Koordination der Transaktionsmodi und der Beschränkung auf eine Transaktion im unrestricted Modus über Netzwerke hinweg wäre aufgrund der Netzwerklatenz zu kostspielig, so daß ein verteilter transaktionaler Speicher so nicht gut skalieren würde.

3.7 Zusammenfassung

Dieses Kapitel hat die Transaktionsrücksetzung für die Programmiersprache *C* gezeigt. Die Rücksetzung von Transaktionen erfolgt für Anwendungen transparent mithilfe des Threadkontexts. Entwickler brauchen nur explizit die Transaktionsgrenzen definieren. Die unmittelbare Transaktionsrücksetzung kann Transaktionen ohne zeitliche Verzögerung inmitten ihrer Ausführung abbrechen, was im Konfliktfall Zeit spart, da die Transaktionen nicht unnötig zu Ende laufen brauchen. Damit verbundene Nachteile bezüglich der Rücksetzung von Systemfunktionen kompensiert eine alternative verzögerte Rücksetzungsstrategie, so daß keine Einschränkungen bei der Verwendung von System- und Bibliotheksfunktionen entstehen. Die verzögerte Rücksetzungsstrategie verschwendet Zeit, da Transaktionen bis zu ihrer Beendigung weiterlaufen. Eine Kombination beider Abbruchstrategien verhindert dies, indem kritische Bereiche in einer Transaktion explizit markiert sind. Transaktionen können so gefahrlos Systemfunktionen ausführen, aber dennoch zeiteffizient zurücksetzen. Dies ist ein Vorteil gegenüber vielen anderen Systemen, die keine Aufrufe von Systemfunktionen innerhalb von Transaktionen zulassen. Systeme, die Systemfunktionsaufrufe in Transaktionen erlauben, lassen die Transaktionen teilweise bis zum Ende durchlaufen. Die Kombination beider Strategien spart an dieser Hinsicht Zeit.

Ebenso ist es Transaktionen möglich, partiell nicht rücksetzbare Operationen mittels der Pufferung von E/A-Operationen auszuführen. Weiterhin hat das Kapitel dargelegt, daß Programmabstürze, die durch veraltete Objektreplikate hervorgerufen werden, sich nicht mit vollständiger Sicherheit über den Signalhandler zur Erfassung transaktionaler Objektzugriffe verhindern lassen. Das System löst dieses Problem aber dennoch effizient durch die Verwaltung mehrerer Versionen von Objekten (siehe Kapitel 6.1).

Weiterhin kann der Entwickler weitreichend in die Transaktionsausführung eingreifen und Transaktionen manuell abbrechen, sofern dies durch bestimmte Programmsituationen sinnvoll ist. Dies spart wiederum Zeit, vor allem wenn die laufende Transaktion wegen eines Konfliktfalls erneut starten würde. Ebenso ist der automatische Neustart von Transaktionen in manchen Fällen nicht sinnvoll, weshalb der Entwickler das Transaktionsverhalten zu Beginn derselbigen festlegen kann. Der Entwickler kann im Abbruchfall einer Transaktion dann selbst (über die Anwendungssemantik) entscheiden, ob er eine abgebrochene Transaktion erneut ausführen möchte. Diese Mechanismen erhöhen weiter die Skalierbarkeit des verteilten transaktionalen Speichers.

4 Synchronisierung eines verteilten transaktionalen Speichers

In verteilten Systemen kommunizieren Rechner untereinander über Netzwerke durch den Austausch von Nachrichten. Dies kann entweder synchron auf Basis eines Anfrage/Antwort-Schemas oder aber auch asynchron erfolgen. Im asynchronen Fall schicken sich die Rechner gegenseitig Nachrichten, ohne unmittelbar eine Rückantwort abzuwarten oder einzufordern. Die verteilte Synchronisierung von Transaktionen über Rechnergrenzen hinweg basiert gleichermaßen auf synchroner als auch asynchroner nachrichtenbasierter Netzkommunikation, die in den folgenden Abschnitten weiter diskutiert wird. Das zugrundeliegende Commit-Protokoll führt zunächst eine Validierung der Transaktionen hinsichtlich Konflikte durch und anschließend im konfliktfreien Fall den Commit. Einzelne Transaktionen beschränken sich dabei nur auf einen einzigen Rechner. Die Transaktionsausführung findet selbst also nicht verteilt statt. Der Verteilungsaspekt betrifft dagegen die Synchronisierung unterschiedlicher Transaktionen, die überlappend im verteilten System laufen.

4.1 Netzwerkarchitekturen

Die Kommunikation verteilter Anwendungen über Netzwerke läßt sich in unterschiedliche Architekturmodelle unterteilen, welche die Rollen und die Art der Kommunikation definieren. Zwei verbreitete Netzwerkarchitekturen sind die *Client-Server-* und die *Peer-to-Peer-*Architektur (P2P-Architektur). Der folgende Abschnitt führt zunächst in die beiden Netzwerkarchitekturen ein und gibt einen Überblick über deren Organisationsformen und den damit verbundenen Eigenschaften. Der darauf folgende zweite Abschnitt diskutiert die Validierung und den Commit von Transaktionen in einem verteilten System auf Basis einer Peer-to-Peer-Architektur.

4.1.1 Client/Server-Netzwerkarchitektur

Die klassische Kommunikation basiert auf einer Client/Server-Architektur welcher auch der überwiegende Anteil der Internetkommunikation zugrundeliegt. Dies sind zum einen Webserver, die über das Hypertext-Transfer-Protokoll (HTTP) [36] kommunizieren. Ein Webserver tritt, wie der Name bereits suggeriert, als Dienstleister auf. Er wartet auf Anfragen von Konsumenten (Clients) und beantwortet diese, indem er den angeforderten Inhalt von Internetseiten zustellt. Die Clients kommunizieren dagegen untereinander nicht (siehe Abbildung 4.1). Weitere klassische Internetdienste wie Dateitransfer oder E-Mail unterliegen ebenfalls dieser Architektur.

Die klassische Client/Server-Kommunikation weist gegenüber der P2P-Kommunikation (siehe Kapitel 4.1.2) jedoch einige Schwächen bezüglich Fehlertoleranz und Skalierbarkeit auf. Stellen viele Clients gleichzeitig Anfragen an denselben Webserver, so kann dieser aufgrund der vielen Anfragen zum Flaschenhals werden. Anfragen treffen beim Server in einem kürzeren Zeitintervall ein, als er für die Beantwortung derselbigen benötigt. Demzufolge stauen

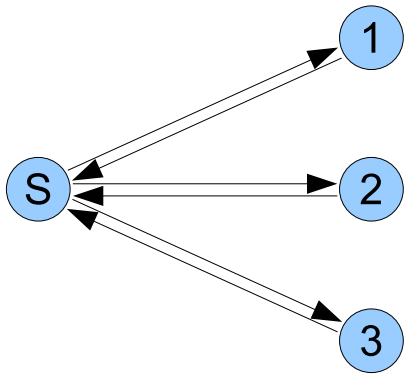


Abbildung 4.1: Client/Server-Netzwerk.

sich immer mehr Anfragen auf, die eine weitere Verzögerung verursachen. Hält der Webserver an der strikten Reihenfolgeerhaltung der Anfragen fest, würden alle auflaufenden Anfragen einer immer weiter anwachsenden Verzögerung unterliegen. Ab einer bestimmten Zeitdauer würden Clients ihre Anfrage aufgrund der augenscheinlich ausbleibenden Antwort (Timeout) gegebenenfalls verwerfen oder wiederholen. Auf dem Webserver führt dies schlimmstenfalls zu einer unnötigen Bearbeitung der Anfrage, welche den Server belastet, sofern dieser einen vorzeitigen Verbindungsabbruch nicht rechtzeitig bemerkt.

Als mögliche Lösungen bieten sich hier die Beschränkung auf eine Obergrenze gleichzeitiger Verbindungen, die Leistung des Servers zu erhöhen oder die Verteilung der Anfragen auf mehrere Rechner mittels Lastverteilung an. Weiterhin ist ein einzelner Server auch ein *Single Point of Failure (SPoF)*. Fällt der Server aus, so bedeutet dies den Ausfall des gesamten Dienstes. Anstehende Clientanfragen können dann nicht mehr beantwortet werden.

Aus diesem Grund werden Systeme mit einer hohen Anforderung an deren Verfügbarkeit (Hochverfügbarkeitssysteme) redundant ausgelegt. Für das Webserverbeispiel bedeutet dies eine mehrfache Existenz des vorgehaltenen Inhalts auf unterschiedlichen physischen Servern. Fällt ein Server aus, so können die anderen Server einspringen oder dessen Arbeit übernehmen. Je nachdem, ob die redundanten Systeme den Dienst gleichzeitig anbieten oder erst im Fehlerfall einspringen.

4.1.2 Peer-to-Peer-Netzwerkarchitektur

Eine weitere Netzwerkarchitektur bilden die Peer-to-Peer-Systeme (P2P). Diese Architektur unterteilt Geräte nicht in Clients und Server, sondern bezeichnet sie generell nur als *Peers*. In einem P2P-System sind alle Peers gleichberechtigt und vereinigen sowohl die Client- als auch die Serverrolle. Demzufolge bietet jeder Peer Dienste an und konsumiert gleichzeitig Dienste von anderen Peers. Das in diesem Kapitel diskutierte Commit-Protokoll liegt einer vollvermaschten P2P-Kommunikation zugrunde (siehe Abbildung 4.2). Daraus folgend nehmen alle Peers in dem Netzwerk die gleiche Rolle ein, indem sie Commits fremder Peers verarbeiten und eigene Transaktionen abschließen. Die vollvermaschte Netzwerkkommunikation liegt darin begründet, daß dieses Kapitel lediglich die Basisentwicklung einer Transaktionssynchronisierung in verteilten Systemen behandelt, ohne Berücksichtigung infrastruktureller Aspekte. Die Synchronisierung von Transaktionen unter Einbezug der Netzwerkinfrastruktur erfolgt in Kapitel 5.

Gegenüber der Client/Server-Architektur existiert in dem vollvermaschten P2P-Netzwerk kein

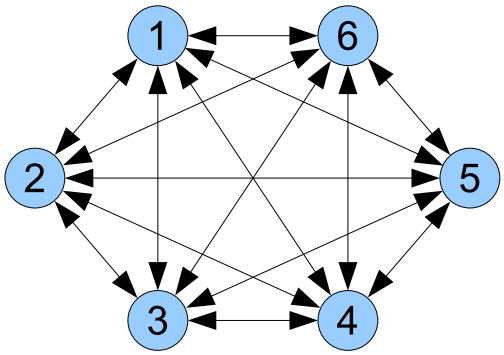


Abbildung 4.2: Vollvermaschtes P2P-Netzwerk.

SPoF, da hier keine zentrale Instanz des ausführenden Systems vorliegt. Diese Art der Kommunikation eignet sich für Cluster oder föderierte Cluster, die eine niedrige Netzwerklatenz aufweisen. Weiterhin eignet sie sich auch für Grid-Systeme in Weitverkehrsnetzen (engl. Wide Area Network (WAN)) mit höheren Latenzen. An dieser Stelle sei bereits erwähnt, daß eine globale Transaktionsserialisierung auch in einem vollvermaschten P2P-Netzwerk aufgrund der Netzkommunikation nicht für beliebig viele Peers gut skalieren kann, weshalb im weiteren Verlauf dieser Arbeit diskutierte Techniken notwendig sind.

4.1.3 Transportprotokolle und Kommunikationsformen

Für die Kommunikation in Netzwerken (lokale als auch Weitverkehrsnetze) finden oftmals die zwei im Internet gebräuchlichsten Transportprotokolle *Transmission Control Protocol (TCP)* und *User Datagram Protocol (UDP)* Verwendung. Beide Protokolle haben bezüglich ihrer Zuverlässigkeit und Reihenfolgeerhaltung unterschiedliche Eigenschaften. TCP ist ein zuverlässiges verbindungsorientiertes Transportprotokoll und dient dem Austausch von Datenströmen über virtuelle Verbindungen (Kanäle). Es garantiert ebenso die Reihenfolge aller übertragenen Daten, stellt also einen FIFO-Kanal (First-In-First-Out) zur Verfügung. UDP ist dagegen ein verbindungsloses, unzuverlässiges und nachrichtenbasiertes Transportprotokoll. Es garantiert im Vergleich zu TCP keine Reihenfolgeerhaltung bezüglich Nachrichten zwischen zwei Verbindungsendpunkten und eignet sich daher nicht ohne weitere Mechanismen zur Übertragung von Datenströmen. Ein Peer kann Nachrichten, die ihm ein anderer Peer nacheinander zusendet, daher in unterschiedlicher Reihenfolge erhalten. Ferner unterstützt UDP im Gegensatz zu TCP keine Fluß- und Staukontrolle, falls der Empfänger oder eine Zwischenstation (Router) zeitweise mit der Verarbeitung von Netzwerkpaketen überlastet ist. Unter Verwendung von UDP ist es also Aufgabe des darauf aufsetzenden Protokolls, die Reihenfolge von Nachrichten zu garantieren, sofern erforderlich.

Neben den Transportprotokollen ist die Datenübertragung auch durch unterschiedliche Kommunikationsformen des darunterliegenden Netzwerkprotokolls bestimmt, wobei die genannten Transportprotokolle überwiegend auf das Internet-Protokoll (IP) aufsetzen. Einerseits tauschen je zwei Knoten Daten miteinander über eine Punkt-zu-Punkt-Verbindung (Unicast) aus. Andererseits müssen Knoten Nachrichten teilweise auch an eine Gruppe von Knoten senden (Multicast). Während UDP beide Kommunikationsformen unterstützt, gestattet TCP dem Netzwerkprotokoll keine Multicast-Kommunikation, da es zustandsbehaftet ist und ein Zustand durch die beiden Endpunkte einer Punkt-zu-Punkt-Verbindung bestimmt ist. Für TCP spricht dessen Zuverlässigkeit, währenddessen die Multicastunterstützung UDP befürwortet.

Multicast-Unterstützung im Internet

Obwohl UDP IP-Multicast unterstützt und in lokalen Netzen teilweise auch verwendbar ist, ist dies im Internet nicht der Fall. IP-Multicast im Internet erfordert weitere Protokolle und Hardwareunterstützung zur Verwaltung und Koordination, was dessen Komplexität steigert. Zum einen müssen Router multicast-fähig sein, da diese die Multicast-Gruppen, bei denen sich Interessenten anmelden, verwalten müssen. Nur anhand dieser Informationen können Router entscheiden, an welche weitere Router oder Teilnehmer sie bestimmte Daten weiterleiten müssen. Teilweise unterstützen die Internet verwendeten Router keinen Multicast. Ebenso ist der für Multicast spezifische IP-Adreßraum derzeit unreguliert. So kann es insbesondere bei Interdomänen-Kommunikation zu Kollisionen bei der Adreßvergabe kommen. Ebenso können mehrere bereits etablierte Multicast-Sessions, welche die gleiche Multicast-Adresse verwenden, kollidieren, wenn weitere Teilnehmer beitreten oder sich Routen dynamisch ändern. Weiterhin sind Fragen bezüglich der Zugriffskontrolle und Authentifizierung von Interessenten für eine Multicast-Gruppe sowie die Protokolle für die Verteilung von Routinginformationen bisher nicht einheitlich geklärt beziehungsweise festgelegt [31].

Knoten, welche dieselben Nachrichten an eine Gruppe von Knoten schicken möchten, müssen die Multicast-Kommunikation mittels Unicast nachbilden, indem sie die gleiche Nachricht an jeden einzelnen Knoten schicken. Dabei entsteht ein höherer Mehraufwand als bei der IP-basierten Multicast-Kommunikation. Auf Basis eines strukturierten Netzwerks läßt sich auf Applikationsebene eine Multicast-Kommunikation ähnlich des IP-Multicasts realisieren (siehe Kapitel 5.4.1).

4.1.4 Verteilte Transaktionen unter Client/Server- und Peer-to-Peer-Netzarchitekturen

Bevor eine Integration eines gemeinsamen transaktionalen Speichers in ein verteiltes System überhaupt möglich ist, sind zunächst die Systemanforderungen zu definieren. Diese Anforderungen bestimmen letztendlich, welche Netzarchitektur als Kommunikationsinfrastruktur am besten geeignet ist. Der Fokus des verteilten transaktionalen Speichers liegt nicht nur auf Clusterumgebungen, sondern auch auf Grid- und Cloudsysteme sowie WANs. Da ein Grid hinsichtlich der angebundenen Rechner dynamisch ist, Teilnehmer also über die Zeit bei- und austreten können, muß der transaktionale Speicher diese Eigenschaft ebenfalls unterstützen. Aufgrund der Systemarchitekturen von Rechnern entstehen bei der Synchronisierung des Speichers zwangsläufig Replikate, die im Netz verteilt sind (siehe Kapitel 2.1). Daher ist eine ausschließlich client/server-basierte Kommunikationsinfrastruktur für den gemeinsamen verteilten Speicher mit den zuvor genannten Anforderungen eher ungeeignet.

Wie in Kapitel 4.1.1 bereits erwähnt wurde, kann der Ausfall eines einzigen Servers ein ganzes System lahmlegen. Dies gilt in diesem Fall auch für den verteilten gemeinsamen Speicher. Der Ausfall des Servers würde das gesamte System zum Stillstand bringen. Der Server müßte also wie im Webserverbeispiel redundant ausgelegt werden. Ändert eine Transaktion zudem Objekte, so werden auf einen Schlag alle vorhandenen Replikate eines Objekts ungültig, unabhängig davon, ob das transaktionale System die anschließende Synchronisierung mittels des *Invalidierungs-* oder *Aktualisierungsverfahrens* durchführt (siehe Kapitel 2.2.1 und 2.2.2). Dies bedeutet eine starke Belastung des Servers bei vielen Teilnehmern. Der serverbasierte Ansatz bietet jedoch den Vorteil, daß ein Teilnehmer Objekte nicht aufwendig suchen muß, da eine aktuelle Kopie entweder immer auf dem Server vorhanden ist (Invalidierungsverfahren) oder dem Server bekannt ist, auf welchem Knoten die aktuelle Version zu finden ist (Aktualisierungsverfahren).

Für eine P2P-Kommunikationsinfrastruktur spricht dagegen eine gute Lastverteilung unter allen teilnehmenden Peers, die unabhängig von der Netzinfrastruktur verstreuten Replikate über die Knoten[1] und der dynamische Ein- und Austritt von Knoten. Viele Replikate bieten einerseits den Vorteil, daß Peers bei einer Objektanforderung deren Lokalität (geringe Latenz) ausnutzen können, und andererseits dienen sie dem Zweck einer besseren Fehlertoleranz. Fällt ein Knoten aus, kann das System trotz Datenverlust weiterarbeiten, sofern die Daten des ausgefallenen Knotens als Objektreplikate vorhanden sind (Selbstheilung). Eine detaillierte Diskussion von Fehlertoleranzaspekten findet sich in Kapitel 4.6. Bei einem Austritt eines Knotens braucht dieser nur seine noch nicht replizierten Objekte an andere Knoten im System abgeben, das System kann anschließend fehlerfrei weiterarbeiten. Obendrein können sich overlay-basierte P2P-Netzwerke (siehe Kapitel 5) zur Leistungsverbesserung einfacher reorganisieren als dies bei einer client/server-basierten Architektur der Fall ist.

4.2 Speichersynchronisierung in Peer-to-Peer-Netzen

Peer-to-Peer-basierte (P2P) Commit-Protokolle stützen sich auf eine dezentrale Validierung von Transaktionen. Alle an dem transaktionalen System beteiligten Rechner sind gleichberechtigt und nehmen ähnlich der Client- und Serverrolle in einem klassischen P2P-System gleichzeitig zwei unterschiedliche Rollen ein. Zum einen treten Knoten als Koordinator (Master) und zum anderen als Informationsempfänger (Slave) auf. Somit existiert bei diesem Modell kein zentraler Transaktionskoordinator, da jeder beteiligte Knoten zeitweilig Koordinator oder Informationsempfänger ist. Das Protokoll muß sicherstellen, daß nicht mehrere Rechner zur gleichen Zeit die Rolle des Koordinators für sich beanspruchen und dadurch eine fehlerhafte Transaktionsvalidierung verursachen. Dies erfolgt durch eine Serialisierung aller Validierungsvorgänge (siehe Kapitel 4.2.2).

Eine Validierung in einem verteilten System ist gegenüber der auf einem einzelnen Rechner komplexer. Auf einem einzelnen Rechner lassen sich alle Transaktionen mittels einer globalen Zeit in eine feste Reihenfolge einordnen. Auch erlauben Mutexe und Semaphore eine Serialisierung von Validierungs- und Commitphase. In einem verteilten System dagegen existiert keine globale Zeit. Knoten können parallele Validierungsereignisse in einer unterschiedlichen Reihenfolge erfassen, was folglich die Datenkonsistenz des transaktionalen Speichers verletzen kann. Das Commit-Protokoll 4.3 muß daher explizit eine globale Ordnung sämtlicher Commits garantieren, damit alle Knoten die Validierungsergebnisse in derselben Reihenfolge sehen beziehungsweise verarbeiten können. Zudem erfordert das Commit-Protokoll eine Berücksichtigung der Netzwerkkommunikation, da diese einen wesentlichen zeitlichen Kostenfaktor darstellt.

4.2.1 Reihenfolgeerhaltung von Nachrichten

Die einzelnen Peers des verteilten transaktionalen Speichers kommunizieren durch den Austausch von Nachrichten. Abhängigkeiten der Nachrichten untereinander erfordern, daß Knoten diese in der Reihenfolge verarbeiten, in der sie im gesamten System auftreten. Hierbei spielen sowohl die Organisation des Netzwerkes und die Eigenschaften der Verbindung zweier Peers untereinander eine wesentliche Rolle. Den Eigenschaften der Verbindung liegen die Eigenschaften des darunterliegenden Transportprotokolls zugrunde. Wie bereits in Kapitel 4.1.3 beschrieben, hängt die Reihenfolgeerhaltung von Nachrichten einer Verbindung von den verwendeten Transportprotokollen ab.

[1] Replikate entstehen, falls Transaktionen auf Objekten lesen, diese aber nicht verändern.

Sofern voneinander abhängige Nachrichten Peers über unterschiedliche Kanäle erreichen, die Nachrichten also eine feste Reihenfolge definieren, ist die Reihenfolgeerhaltung durch das Transportprotokoll nicht mehr gegeben. An dieser Stelle müssen die auf den Transportprotokollen aufsetzende Protokolle berücksichtigen, Nachrichten entsprechend der Protokollsemantik in der korrekten Reihenfolge zu verarbeiten. Bei dem P2P-Commit können Peers voneinander abhängige Nachrichten von unterschiedlichen Peers erhalten. Die Nachrichten treffen über unterschiedliche Kanäle ein, daher müssen die Commit-Protokolle für die semantisch korrekte Verarbeitung der empfangenen Nachrichten sorgen.

Eine strikte Reihenfolgeerhaltung ist in einem P2P-System allein auf Ebene der Transportprotokolle nicht möglich. Die im weiteren Verlauf dieser Arbeit diskutierten Commit-Protokolle und Kommunikationsverfahren gehen deswegen nicht davon aus, daß unterliegende Transportprotokolle eine Nachrichtenreihenfolge zwischen zwei Peers garantieren, obwohl dies bei dem in dieser Arbeit verwendeten TCP der Fall ist. Dennoch fordern die Commit-Protokolle eine garantierte zuverlässige Kommunikation des unterliegenden Transportprotokolls.

4.2.2 Commit-Koordination

Sowohl Abstimmungsverfahren als auch tokenbasierte Serialisierung garantieren eine globale Commit-Ordnung. Abstimmungsverfahren fordern Knoten zur Stimmabgabe über ein bestimmtes Ereignis auf. Ein (zuvor gewählter) Koordinator steuert den Abstimmungsprozeß und fordert alle an der Abstimmung beteiligten Knoten zur Stimmabgabe auf. Die Knoten schicken dem Koordinator daraufhin ihre Antwort, indem sie der Umfrage zustimmen oder diese ablehnen. Nach dem Erhalt aller Antworten teilt der Koordinator den Knoten das Abstimmungsergebnis mit. Ist mit der Abstimmung der unmittelbare Eintritt in einen kritischen Abschnitt verknüpft, so tritt an Stelle einer Ergebnismitteilung eine Freigabenachricht, welches das Verlassen des kritischen Abschnitts ankündigt. Unter Annahme einer fehlerfreien Kommunikation kann bei einer Abstimmung immer ein Konsens erreicht werden, da jeder Knoten das gleiche Abstimmungsergebnis (beziehungsweise Freigabenachricht) erhält. Während des Validierungsprozesses kann beispielsweise eine Mehrheitsabstimmung entscheiden, ob eine Transaktion erfolgreich abschließen darf oder aufgrund eines Konflikts mit anderen Transaktionen abzubrechen ist. Das Abstimmungsverfahren muß ferner definieren, wie in einer Pattsituation (Fall von Stimmengleichheit) zu verfahren ist. Eine Abstimmung erfordert einen Austausch von mindestens zwei Nachrichten (Anfrage, Antwort) zwischen dem Koordinator und jedem an der Abstimmung beteiligten Knoten [84]. Eine optionale Mitteilung des Abstimmungsergebnisses an die Teilnehmer erfordert zusätzlich eine dritte Nachricht (siehe Abbildung 4.3). Da die Abstimmung über den Commit von Transaktionen dem Eintritt in einen kritischen Bereich gleichkommt, ist eine dritte Nachricht (Commit-Nachricht) erforderlich, die nicht optional sein darf. Die Mindestanzahl für eine Abstimmung erforderlichen Knoten hängt dabei von der Gesamtanzahl an dem Transaktionssystem beteiligten Knoten ab. Mamoru Maekawa und Wai-Shing Luk et al. haben gezeigt, daß Abstimmungsverfahren nicht unbedingt den Einbezug aller Knoten erfordern, wie dies bei broadcast-basierenden Algorithmen der Fall ist [66, 65].

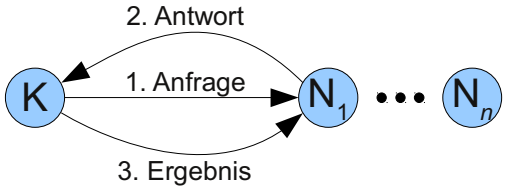


Abbildung 4.3: Kommunikationsverlauf bei Abstimmunsverfahren.

Bei tokenbasierten Verfahren [75] besteht der Nachrichtenaustausch pro Validierungsereignis im Regelfall nur aus zwei Nachrichten. An dieser Stelle wird ersichtlich, daß eine Abstimmung einen wesentlich höheren Nachrichtenaustausch als ein tokenbasiertes Verfahren verursacht. Auch wenn die Anfragen im Abstimmungsverfahren parallel ablaufen können, ist eine hohe Anzahl von auszutauschenden Netzwerknachrichten vor allem in WAN-Netzwerken aufgrund ihrer höheren Latenz kritisch zu betrachten. Zudem führt ein insgesamt vielfacher Nachrichtenaustausch zu einer höheren Netzbelastung. Aus diesem Grund eignen sich tokenbasierte gegenüber abstimmende Verfahren allgemein besser für eine Transaktionskoordination in Netzwerken. Kapitel 4.4 diskutiert unterschiedliche Tokenaustauschverfahren für die Serialisierung von Transaktionen.

4.3 Peer-to-Peer-Commit-Protokoll

Der Commit von Transaktionen kann in einem verteilten System prinzipiell sowohl über die Vorwärts- als auch über die Rückwärtsvalidierung erfolgen, da alle Verfahren ein konsistentes Ergebnis liefern (siehe Kapitel 2.5.3). Da die Netzwerkkommunikation aber einen Großteil der aufgewendeten Zeit in Anspruch nimmt, sind Umsetzung des Validierungsverfahrens und der Konfliktauflösung für eine gute Skalierbarkeit von wesentlicher Bedeutung.

Die Rückwärtsvalidierung läßt sich als P2P-Variante auf zwei Weisen umsetzen. Zum einen kann das Protokoll eine Anfrage mit der Lesemenge der abzuschließenden Transaktion an alle Knoten schicken. Diese prüfen daraufhin, ob die Transaktion einen Konflikt mit bereits abgeschlossenen Transaktionen der angefragten Knoten verursacht. Die Knoten teilen in ihrer Antwort mit, ob ein Konflikt vorliegt. So kann der transaktionsabschließende Knoten feststellen, ob seine Transaktion auf veralteten Daten basiert. Diese Variante würde dem bereits zuvor erwähnten Konsensusverfahren entsprechen und aufgrund der synchronen Anfrage/Antwort-Netzkommunikation mit einer zwischenzeitlichen Blockierung nicht gut skalieren. In diesem Fall brauchen Knoten keine Commit-Benachrichtigungen mit den Schreibmengen von Transaktionen zu verchicken. Alternativ kann ein Peer von anderen Peers versandte Schreibmengen speichern. So können Peers abzuschließende Transaktionen ohne zusätzliche Netzwerkkommunikation lokal validieren. Abschließend mit dem Commit müssen Peers den anderen Teilnehmern die Schreibmenge der Transaktion mitteilen.

Die Vorwärtsvalidierung läuft auf ähnliche Weise ab. Eine abzuschließende Transaktion schickt ihre Schreibmenge an alle anderen Knoten. Diese antworten daraufhin, ob mit ihrerseits laufenden Transaktionen ein Konflikt besteht. Im Konfliktfall kann der transaktionsabschließende Knoten entscheiden, ob er seine Transaktion im Konfliktfall selbst abbricht oder konfliktverursachende fremde Transaktionen. Diese Koordination erfordert viel Netzwerkkommunikation, weshalb sie nicht gut skaliert. Alternativ kann die Vorwärtsvalidierung von vornherein den Abbruch konfliktverursachender Transaktionen auf dem empfangenden Systemen vorherbestimmen. Diese Art der Konfliktlösung bezeichnet man auch als *First-Wins*-Strategie und benötigt daher wie die zweite Variante der Rückwärtsvalidierung keine weitere Netzwerkkommunikation. Die Vorwärtsvalidierung bietet gegenüber der Rückwärtsvalidierung aber den Vorteil, daß konfliktverursachende Transaktionen nicht bis zum Ende laufen müssen, sondern inmitten ihrer elementaren Operationen abgebrochen werden können, sofern sie sich nicht gerade in kritischen Programmabschnitten befinden (siehe Kapitel 3.2.1). Als weiterer Vorteil ergibt sich, daß Knoten empfangene Schreibmengen abgeschlossener Transaktionen nicht speichern brauchen.

Als P2P-Commit-Protokoll ist daher eine *First-Wins*-Strategie mit Vorwärtsvalidierung besser geeignet. Als Koordination findet eine tokengesteuerte Transaktionsserialisierung Anwendung (siehe Kapitel 4.4). *First-Wins* bedeutet in diesem Kontext, daß ein Knoten, welcher das Token zuerst akquirieren kann, seine Transaktion abschließen darf. Während des Commits sendet der

Knoten die Schreibmenge seiner Transaktion an alle anderen Knoten und löst damit auf allen anderen Knoten eine Validierung gegen überlappende laufende Transaktionen aus. Stellen diese Knoten Konflikte mit laufenden Transaktionen fest, müssen sie diese abbrechen. Weiterhin invalidieren alle Knoten lokale Datenreplikate, welche die abzuschließende Transaktion geändert hat. Anschließend gibt der Koordinator das Token wieder frei. Die Verfahrensweise bei der Tokenfreigabe (Rückgabe an den Tokenkoordinator oder Abholung durch andere Knoten) hängt dabei vom verwendeten Tokenverfahren ab (siehe Kapitel 4.4). Damit eine laufende Transaktion T_x konfliktfrei zu einer abgeschlossenen Transaktion T_c ist, muß folgende Bedingung[2] gelten:

$$WS_{T_c} \cap RS_{T_x} = \emptyset \tag{4.1}$$

Ein Konflikt erfordert nicht ausnahmslos den Transaktionsabbruch auf dem Zielsystem, da die Vorwärtsvalidierung generell den Abbruch einer beliebigen Transaktionsmenge gestattet. Unter Berücksichtigung der Netzwerkkommunikation und der damit verbundenen Latenz ist ein Abbruch der abzuschließenden Transaktion jedoch nicht sinnvoll. Ein Abbruch der abzuschließenden Transaktion erfordert eine zusätzliche Koordination des Transaktionsabbruchs und bedeutet deswegen einen höheren Kommunikationsaufwand. Unter der Annahme, einer höheren Netzwerklatenz gegenüber der durchschnittlichen Transaktionsdauer, würde in diesem Fall die Skalierbarkeit sinken. Aus diesem Grund sieht das Commit-Protokoll nur den Transaktionsabbruch auf dem Zielsystem vor. Der Protokollablauf ist beispielhaft in Abbildung 4.4 dargestellt. Die Commit-Serialisierung in der Abbildung basiert auf einem P2P-basierten Tokenaustausch. Bei diesem Verfahren können Tokenanfragen aufgrund der Nebenläufigkeit Knoten erreichen, die nicht mehr im Besitz des Tokens sind. Diese Knoten müssen die Anfragen dann an andere Knoten weiterleiten. Eine detaillierte Beschreibung der Tokenverfahren findet sich im Kapitel 4.4.

Anstatt die Zielsysteme anzuweisen, ihre Datenreplikate zu invalidieren (*Invalidierungsverfahren*), kann der transaktionsabschließende Peer auch mittels des *Aktualisierungsverfahrens* alle innerhalb der Transaktion geänderten Objekte über die Commit-Benachrichtigung auf den Zielsystemen aktualisieren. Die Peers, die nachfolgend auf ein geändertes Objekt zugreifen, müssen es dann nicht erst beim zuletzt festschreibenden Peer anfordern. Dies verhindert Verzögerungen bei der Transaktionsausführung, da die Peers nicht explizit neue Objektversionen anfordern müssen. Allerdings kann die Aktualisierungsstrategie auch eine unnötige Netzbelastung verursachen. Nicht alle Peers benötigen die in der Benachrichtigung mitgesendeten Replikate, falls sie diese nicht verwenden. Für den Optimalfall ist an dieser Stelle ein Kompromiß zwischen beiden Strategien sinnvoll. Peers, die bestimmte Objekte oft zugreifen, sollten aktualisierte Versionen in der Commit-Nachricht geliefert bekommen, während bei einem seltenen Zugriff eine Invalidierung ausreicht. Für die Effizienz beider Strategien ist auch zu berücksichtigen, ob das unterliegende Netzwerk dieselbe Nachricht an mehrere Empfänger gleichzeitig verteilen kann (Multicast).

4.3.1 Bestätigter Commit

Da Nachrichten in Netzwerken generell keiner globalen Ordnung unterliegen, ist es möglich, daß sich Nachrichten gegenseitig überholen und Knoten Nachrichten generell auch in unterschiedlicher Reihenfolge empfangen können. Bei der tokengesteuerten Validierung kann es also durchaus vorkommen, daß Knoten Commit-Benachrichtigungen von verschiedenen Knoten in unterschiedlicher Reihenfolge erhalten (siehe Abbildung 4.5). So ist es ebenfalls möglich,

[2] RS = Lesemenge (read set), WS = Schreibmenge (write set).

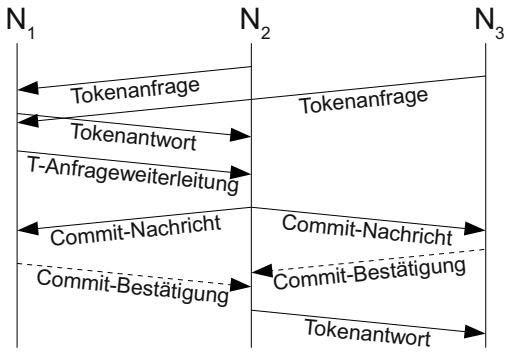


Abbildung 4.4: Bestätigtes und unbestätigtes P2P-Commit-Protokoll.

daß ein Knoten das Token für eigene Commits erhält, bevor er alle zuvor versandten Commit-Nachrichten verarbeitet hat, da sich diese noch im Transit befinden können. Für die Datenkonsistenz ist es aber notwendig, daß ein Knoten vor dem Commit eigener Transaktionen alle ausstehenden Commit-Benachrichtigungen verarbeitet, da er sonst eventuelle Transaktionskonflikte nicht erkennt. Diese Probleme lassen sich verhindern, indem jeder Knoten, nachdem er eine Commit-Nachricht verarbeitet hat, eine Bestätigung an den Koordinator zurückschickt und dieser das Token erst nach dem Empfang aller Bestätigungen freigibt (siehe Abbildung 4.6). Auf diese Weise besteht eine totale Ordnung sämtlicher Commit-Benachrichtigungen auf allen Knoten und weiterhin eine Ordnung zwischen dem Token und allen zuvor ausgeführten Commits. Dies bedeutet, eine Verarbeitung von Commit-Benachrichtigungen in falscher Reihenfolge auf einem Knoten ist ausgeschlossen, und bei Erhalt des Tokens hat ein Knoten bereits alle Commit-Benachrichtigungen früherer abgeschlossener Transaktionen verarbeitet.

Für die Validierung und korrekte Invalidierung von Objektreplikaten ist es auf den Zielsystemen ferner notwendig, Abhängigkeiten zwischen Objektreplikaten und Commit-Benachrichtigungen zugehöriger Transaktionen festzustellen. Ansonsten kann es durch nicht erkannte veraltete Objektreplikate zu Validierungsfehlern kommen, und die Datenkonsistenz in dem transaktionalen Speicher ist nicht mehr gewährleistet. Dies kann das System einerseits dadurch erreichen, indem es Objektreplikate, die ein Knoten vor einer Invalidierung angefordert hat, aber erst nach dieser erhalten hat, als ungültig annimmt. Denn diese können potentiell veraltet sein, was der Knoten allerdings nicht zuverlässig feststellen kann. Angenommen Knoten N_2 fordert ein Objektreplikat bei Knoten N_1 an. Knoten N_1 schickt eine Kopie des Objekts und kurz danach die Schreibmenge einer abgeschlossenen Transaktion, welche das angeforderte Objekt enthält, an Knoten N_2. Knoten N_2 kann nicht feststellen, ob das Replikat gültig ist, da er von der Reihenfolge der empfangenen Nachrichten nicht auf deren Sendereihenfolge schließen kann (siehe Kapitel 4.2.1). Andererseits kann das System auch von Transaktionen erzeugte Objektreplikate und die zugehörigen Commit-Benachrichtigungen über eine zusätzliche ID miteinander verknüpfen. Eine detaillierte Betrachtung der Synchronisierung von Objektreplikaten in Verbindungen mit Commits von Transaktionen behandelt Kapitel 4.3.4.

4.3.2 Unbestätigter Commit

Der bestätigte Commit wahrt die Reihenfolge der zu verarbeitenden Commit-Nachrichten entsprechend der Commit-Reihenfolge, hat aber Schwächen im Hinblick auf die Skalierbarkeit. Der Koordinator muß nach dem Commit so lange mit der Tokenfreigabe warten, bis alle Zielsysteme die Validierung beendet und die Bestätigung zurückgeschickt haben (farblicher Balken in Abbildung 4.6). Erst dann kann ein anderer Knoten das Token erhalten und sei-

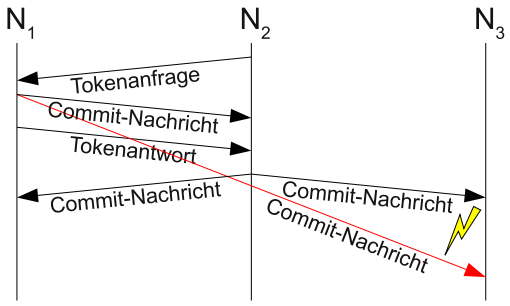


Abbildung 4.5: Konsistenzverletzung durch vertauschte Commit-Nachrichten.

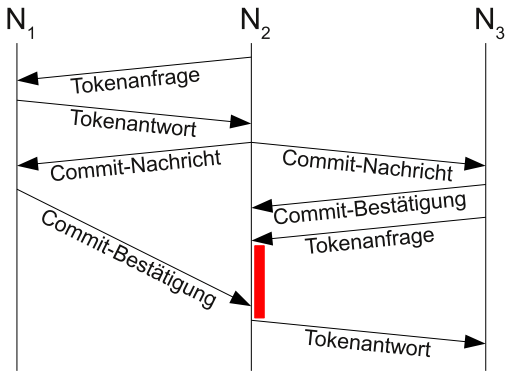


Abbildung 4.6: P2P-Commit-Protokoll mit Bestätigungsnachrichten.

nerseits einen Commit durchführen. Ein effizienterer Ansatz arbeitet mit einem unbestätigten Commit [74]. Dieser erlaubt eine Tokenfreigabe unmittelbar nach dem Versand der eigenen Commit-Nachrichten. Demnach müssen Tokenanfragen anderer Knoten nicht die Commit-Bestätigungen abwarten, sondern können das Token unmittelbar nach dem Commit erhalten. Diese Protokolloptimierung verhindert unnötige Verzögerungen, die zuvor bei der Tokenfreigabe entstanden sind, und verbessert damit die Skalierbarkeit des Systems. Allerdings geht damit auch die totale Ordnung von Commit-Nachrichten und dem Token verloren. Der unbestätigte Commit muß die Ordnung daher auf andere Weise sicherstellen.

Statt der Bestätigungsnachrichten, die eine ausgeführte Validierung auf dem Zielsystem signalisieren, erhält jeder Commit eine eindeutige globale ID (Commit-ID). Die ID entspricht einem inkrementierenden Commit-Zähler, basierend auf logischen Zeitstempeln [60]. Dieser spiegelt die Commit-Reihenfolge im System wider. Die Zielsysteme sind so in der Lage, die Reihenfolge von Commit-Nachrichten unabhängig von der Reihenfolge ihres Empfangs zu erkennen. Der aktuelle Wert des globalen Commit-Zählers wird zwecks Synchronisierung zwischen den Knoten im Token übermittelt. So können die Knoten nach dem Erhalt des Tokens ebenfalls erkennen, ob sie vor dem eigenen Commit noch ausstehende Commit-Benachrichtigungen verarbeiten müssen und so die Ordnung zwischen Token und Commit-Benachrichtigungen garantieren. Im Gegensatz zum bestätigten Commit verlagert der unbestätigte Commit die Reihenfolgekoordination vom transaktionsabschließenden Knoten auf die Zielsysteme. Aufgrund der Nachteile des ersten Commit-Verfahrens gegenüber dem unbestätigten Commit-Ansatz, findet ersterer in dieser Arbeit keine weitere Betrachtung mehr.

Der unbestätigte Commit kann nach Erhalt des Tokens ebenfalls blockieren, aber nur sofern der Knoten noch nicht alle vorherigen Benachrichtigungen verarbeitet hat. Dies ist eindeutig

durch einen Vergleich der Commit-ID der zuletzt verarbeiteten Benachrichtigung und der im Token gespeicherten ID möglich. Das System gibt das Token erst nach der Verarbeitung aller Benachrichtigungen für die Verwendung (Commit eigener Transaktionen) frei, auch wenn es das Token bereits empfangen hat. Im letzteren Fall hält es das Token intern zurück, damit ein anstehender Commit erst nach der Konfliktprüfung mit allen vorherigen Commits früherer überlappender Transaktionen stattfindet, sofern die Transaktion nach der Prüfung weiterhin konfliktfrei ist. Der Protokollablauf ist in Abbildung 4.7 dargestellt.

Nachdem Peer N_2 seine Commit-Benachrichtigung verschickt hat, leitet er das Token entsprechend der zuvor von Peer N_1 an ihn weitergeleiteten Tokenanfrage an N_3 weiter. Weiterhin leitet er eine von N_1 getätigte Tokenanfrage an N_3 weiter. Zur Vereinfachung geht das Beispiel von einer Kollision der zu validierenden Transaktion auf Peer N_3 aus, so daß dieser keine Commit-Nachricht verschickt. Er gibt das Token stattdessen an den N_1 ab (entsprechend der an ihn weitergeleiteten Tokenanfrage). Aufgrund der Nebenläufigkeit trifft das Token vor der von N_2 versandten Commit-Nachricht bei N_1 ein, obwohl die Peers die Nachrichten zuvor in umgekehrter Reihenfolge versandt hatten. Würde N_1 nach Erhalt des Tokens sofort eine eigene Transaktion abschließen, die in Kollision mit der von N_2 abgeschlossenen Transaktion steht, würde N_1 dies aufgrund der verspätet eingetroffenen Commit-Nachricht nicht erkennen und die Konsistenz verletzen. Daher hält N_1 das Token solange zurück (rot gestrichelte Linie), bis er alle noch fehlenden Commit-Benachrichtigungen verarbeitet hat. Der Peer kann dies erkennen, indem er den Commit-Zähler im Token mit dem Zählerwert der letzten Benachrichtigung vergleicht. Damit ein Knoten im Fall einer verlorengegangenen Commit-Nachricht nicht endlos blockiert, muß er nach einer zuvor definierten Zeitüberschreitung von einem Fehler ausgehen (siehe Kapitel 4.6).

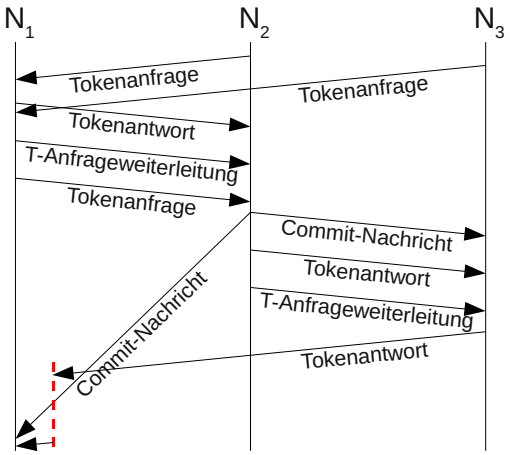


Abbildung 4.7: P2P-Commit mit unmittelbarer Tokenfreigabe und ohne Bestätigungen.

4.3.3 Verarbeitung von Commit-Benachrichtigungen

Die Commit-Benachrichtigungen dienen sowohl der Konflikterkennung gegen laufende Transaktionen auf einem Knoten als auch der Invalidierung der durch die Benachrichtigung referenzierten geschriebenen Objekte. Bei der First-Wins-Strategie muß ein Knoten beim Empfang einer Commit-Benachrichtigung die lokalen Objektreplikate invalidieren, da diese veraltet sind. Eine Ausnahme besteht nur in dem Fall, wenn die Commit-Nachricht und ein angefordertes Objektreplikat derselben Transaktion (oder ein Objektreplikat einer neueren Transaktion) auf einem Knoten in vertauschter Reihenfolge eintreffen und der Knoten dies beispielsweise

über den logischen Zeitstempel (Commit-ID) eindeutig feststellen kann. In diesem Fall darf er das Replikat nicht invalidieren. Eine laufende Transaktion verursacht einen Konflikt, wenn der Knoten ein von ihr gelesenes Objekt aufgrund der Commit-Benachrichtigung invalidieren muß. Die Transaktion hat in diesem Fall ein veraltetes Objekt gelesen. Darf der Knoten ein Replikat in dem geschilderten Ausnahmefall nicht invalidieren, besteht ebenso kein Konflikt bei der laufenden Transaktion, da sie nur die aktuelle Objektversion gelesen hat.

Die Reihenfolge, in welcher Knoten die Commit-Benachrichtigungen verarbeiten, ist nicht zwingend vorgeschrieben. Die Knoten müssen nur sicherstellen, daß ältere Commit-Benachrichtigungen keine aktuelleren Objekte invalidieren und folglich keine falschen Transaktionskonflikte provozieren. Ein Knoten erhält beispielsweise ein Replikat eines geschriebenen Objekts von einer festgeschriebenen Transaktion T_c und empfängt anschließend jedoch eine Commit-Benachrichtigung einer Transaktion $T_x < T_c$, welche ebenfalls dieses Objekt beschrieben hatte. In diesem Fall darf die Benachrichtigung der Transaktion T_x das Objektreplikat nicht invalidieren und ebenfalls eine eventuell laufende Transaktion nicht abbrechen, die dieses Objektreplikat gelesen hat. Bei der Invalidierung eines Objekts muß der Knoten gleichzeitig die in der Commit-Benachrichtigung mitgelieferte Versionsnummer speichern, da diese die minimale gültige Version des Objekts nach Verarbeitung der Commit-Benachrichtigung festlegt. Ein Knoten darf also nur solche Replikate dieses Objekts akzeptieren, welche dieselbe oder eine höhere Versionsnummer aufweisen.

Abhängigkeit von Objekten und Commits

Wie im Abschnitt des bestätigten Commit-Verfahrens diskutiert, ist für eine zuverlässige Erkennung der Abhängigkeit von Objektreplikaten und deren zugehörige Commit-Benachrichtigung eine ID notwendig, welche beide Daten miteinander verknüpft. Da beim bestätigten Commit-Verfahren jede Commit-Benachrichtigung bereits eindeutig über einen globalen logischen Zeitstempel identifizierbar ist, bietet es sich an, diesen ebenfalls zur Identifizierung der zugehörigen Objektreplikate zu verwenden. So entsteht eine eindeutige Beziehung zwischen einem Objektreplikat und einer Commit-Benachrichtigung. Jedes ausgelieferte Objektreplikat muß demnach die Commit-ID seiner erzeugenden Transaktion als Versionsnummer mitführen.

Aufgrund der Versionierung von Objektreplikaten ist die Reihenfolge der Verarbeitung von Commit-Benachrichtigungen für die Erhaltung der Datenkonsistenz nicht relevant, da ein Knoten anhand der Objektversionsnummer eindeutig feststellen kann, ob eine ältere Commit-Nachricht eine aktuellere Objektversion ersetzen würde. Die Verarbeitungsreihenfolge hängt aber auch von dem Synchronisierungsverfahren der Objektreplikate ab. Findet eine Objektsynchronisierung unter der Verwendung des Aktualisierungsverfahrens (siehe Kapitel 2.2.2) und differentiellen Objektaktualisierungen (siehe Kapitel 4.3.5) statt, erhält ein Knoten mit der Commit-Nachricht nur die Änderungen eines Objekts, basierend auf einer bestimmten vorherigen Objektversion. Hier ist eine Verarbeitung der Commit-Benachrichtigungen in beliebiger Reihenfolge nicht mehr möglich.

Abhängigkeit von Commit-Benachrichtigungen und Commits

Weiterhin muß ein Knoten, bevor er eigene Transaktionen abschließt, alle vorherigen Commit-Benachrichtigungen verarbeiten, um eventuelle Transaktionskonflikte zu erkennen. Dies ist nur relevant, falls ausstehende Commit-Benachrichtigungen Objekte geändert haben, die in der abzuschließenden Transaktion gelesen oder geschrieben wurden. Da ein Knoten diese im Vorfeld nicht erkennen kann, muß er zunächst alle Benachrichtigungen verarbeiten, bevor er selbst

Transaktionen abschließt. Eine Ausnahme ergibt sich, sofern die abzuschließende Transaktion und die ausstehenden Commit-Benachrichtigungen unterschiedliche Konsistenzdomänen betreffen (siehe Kapitel 6.4).

Auch hier ist eine Verarbeitung der Commit-Benachrichtigungen in der Reihenfolge der Commits sinnvoll, da ein Knoten anderenfalls koordinieren muß, welche Commit-Benachrichtigungen er noch nicht verarbeitet hat. Bei Verarbeitung in der Commit-Reihenfolge kann ein Knoten anhand der Commit-ID der zuletzt verarbeiteten Benachrichtigung erkennen, ob noch weitere nicht verarbeitete Benachrichtigungen ausstehen.

Kann ein Knoten Transaktionen unter der Anwendung lokaler Commits (siehe Kapitel 6.2.1) abschließen, so muß er nicht unbedingt sämtliche Commit-Benachrichtigungen abwarten. Solche Transaktionen dürfen, auch wenn sie zuvor veraltete Daten gelesen haben, erfolgreich abschließen, sofern zum Zeitpunkt des Commits dennoch ein gültiger Serialisierungsplan für die Transaktion möglich ist (siehe Kapitel 6.2.3).

4.3.4 Synchronisierung von Objektreplikaten

Neben dem Commit von Transaktionen müssen die einzelnen Knoten untereinander auch Objekte synchronisieren, da der transaktionale verteilte Speicher auf der Replikation von Objekten in den lokalen Speicher der einzelnen Knoten basiert (siehe Kapitel 2.1). Führt eine Transaktion eine Änderung an einem Objekt durch, so überführt sie die lokale Objektkopie in eine neue geänderte Version. Weiterhin besteht zwischen allen Versionen desselben Objekts eine totale Ordnung, da nur Commits Änderungen an Objekten hervorrufen können und auf diesen ebenfalls eine totale Ordnung besteht. Per Definition gilt für $T_x(O)$, daß Transaktion T mit der Commit-ID x eine neue Version des Objekts O erzeugt. Folglich gilt:

$$\forall T_x \, \forall T_y : a_v = T_x(O) \wedge b_w = T_y(O) \wedge y > x \Rightarrow w > v \tag{4.2}$$

Erzeugt eine Transaktion T_y eine neue Version b_w des Objekts O, so gilt für alle vorausgegangenen Transaktionen T_x und deren erzeugten Objektversionen a_v des Objekts O, daß deren erzeugten Versionsnummern v älter als die von der Transaktion T_y erzeugte Objektversionsnummer w sind. Der Umkehrschluß gilt gleichwohl, die Reihenfolge der Versionen eines Objekts entspricht der Reihenfolge der erzeugenden Transaktionen.

Bei der Synchronisierung von Objekten ist zu beachten, daß jede Objektversion auf einem bestimmten Commit basiert und somit eine zugehörige Commit-Benachrichtigung existiert. Da der Austausch von Objektreplikaten und der Empfang von Commit-Benachrichtigungen beim Invalidierungsverfahren zueinander nebenläufig erfolgt, müssen Empfängerknoten diese unabhängig von deren Empfangsreihenfolge synchronisieren. Ein Peer darf ein angefordertes Objektreplikat nicht ungeprüft übernehmen, da er für dieses zwischenzeitlich eine Commit-Benachrichtigung einer neueren Transaktion verarbeitet haben kann (siehe Abbildung 4.8). Mit der Commit-Benachrichtigung erhält ein Peer die minimale gültige Versionsnummer eines Objekts. Würde der Knoten das nunmehr veraltete Objektreplikat ungeprüft übernehmen, hätte eine ältere Objektversion die neuere ersetzt. Dabei ist es irrelevant, ob für die durch die Commit-Nachricht angekündigte Objektversion die zugehörigen Objektdaten im lokalen Speicher vorlagen. Akzeptiert der Knoten sein veraltetes angefordertes Objektreplikat als gültiges Objekt, würde die nächste Transaktion, welche dieses Objekt verwendet, trotz einer veralteten Objektversion erfolgreich abschließen. Empfängerknoten können diesen Fall erkennen, wenn Objektversionsnummern in Commit-Nachrichten trotz deren commit-id-basierten totalen Ordnung rückwärts laufen. Ein Knoten muß also darauf achten, daß keine ältere Objektversion eine neuere ersetzt.

Angenommen ein Knoten fordert ein aktuelles Objektreplikat an. Während es sich im Transit befindet, erfolgt eine Invalidierung des Objekts auf allen außer dem transaktionsabschließenden Knoten. Für das mittlerweile veraltete Objektreplikat, welches sich in der Übertragung zum Zielknoten befindet, erfolgt keine Invalidierung, da dies nur für Objektkopien im lokalen Speicher eines Knotens erfolgen kann. Ohne gesonderte Überprüfung der Versionsnummern würden in dem transaktionalen System zwei unterschiedliche als gültig angenommene Versionen von Objektreplikaten vorhanden sein. Kann der Knoten mit dem veralteten Objektreplikat seinerseits eine Transaktion durchsetzen (abschließen), in welcher er Änderungen auf dem Objekt durchführt, invalidiert er damit gleichzeitig alle Replikate der echt gültigen Objektversion. Somit hat der Knoten veraltete Daten im transaktionalen Speicher festgeschrieben und die Datenkonsistenz unwiderruflich verletzt. Gleiches gilt für einen Commit von andere Knoten, welche zuvor das veraltete Objekt mittels einer Objektanfrage bei sich replizieren und anschließend in einer Schreiboperation festschreiben. Die Verletzung der Datenkonsistenz hat dagegen keine Auswirkungen, sofern der nächste Commit, der dieses Objekt ändert, auf einem echt gültigen Objektreplikat basiert, da dieser Commit das veraltete aber als gültig angenommene Objektreplikat auf allen Knoten invalidiert.

Bei Verwendung des *Aktualisierungsverfahrens* (siehe Kapitel 2.2.2) ist dies unproblematisch, da die Commit-Nachricht und die geänderten Objektinhalte in einer gemeinsamen Nachricht zusammengefaßt sind und beim Empfänger gleichzeitig der Reihenfolgeerhaltung der Commit-Benachrichtigungen unterliegen. Weiterhin findet bei diesem Verfahren keine explizite Anforderung von Objektreplikaten statt. Beim *Invalidierungsverfahren* (siehe Kapitel 2.2.1) erfolgt nur eine Auslieferung von Commit-Nachrichten, welche die auf dem Zielknoten vorhandenen Objekte invalidieren.

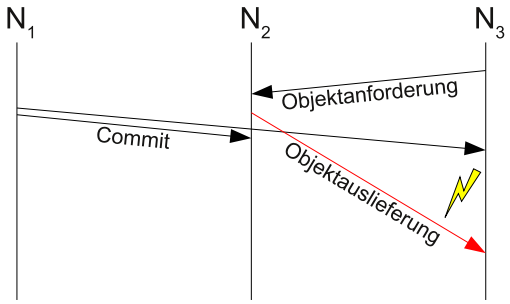


Abbildung 4.8: Durch Commit ungültig gewordenes Replikat während seiner Transitphase.

Empfängt ein Knoten zwischen Objektanforderung und Auslieferung des Replikats von einem weiteren Knoten eine Commit-Benachrichtigung, welche das angeforderte Objekt invalidiert, so sagt dies nichts darüber aus, daß das ausgelieferte Replikat mittlerweile veraltet ist. Es kommt vielmehr darauf an, ob der Knoten, der das Replikat versandt hat, die Commit-Benachrichtigung bereits verarbeitet hat und dennoch ein gültiges Replikat besitzt (siehe Abbildung 4.9).

4.3.5 Differentielle Objektsynchronisierung

Erfährt ein Objekt einer erfolgreich abgeschlossenen Transaktion zuvor einen Schreibzugriff, führt dies automatisch zu einer neuen Objektversion. Dabei ist es unerheblich, ob die Anwendung das Objekt vollständig oder nur Teile davon geändert hat. Da Objekte meistens nur teilweisen Änderungen während einer Transaktion unterliegen, ist es ausreichend, nur die Objektänderungen beziehungsweise Differenz (Diff) zweier Objektversionen zu übertragen. Ein

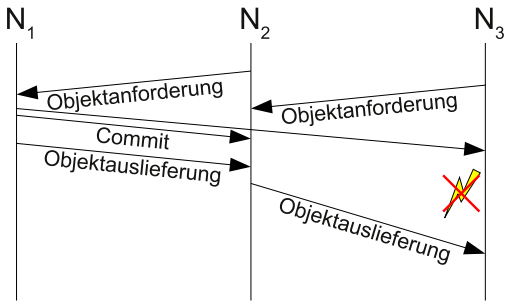


Abbildung 4.9: Gültiges Replikat im Transit bei nebenläufiger Invalidierung.

anfragender Knoten teilt in seiner Anfrage neben der gewünschten Objektversion auch die Versionsnummer des aktuell von ihm gespeicherten Objekts (Diff-Basis) mit. Die Übertragung von Diffs kann Netzwerkbandbreite sparen, was gerade für Knoten in Weitverkehrsnetzen, die oftmals über schmalbandige Verbindungen angebunden sind, wichtig ist.

Ein differentieller Objektaustausch ist beim Aktualisierungsverfahren einfach umzusetzen, vorausgesetzt die Aktualisierung findet fortwährend mit den Commit-Benachrichtigungen statt. Bei einem transaktionalen Commit fallen pro Objekt nur differentielle Objektdaten zwischen einer neuen und seiner vorherigen Objektversion an, die der transaktionsabschließende Knoten zusammen mit der Commit-Benachrichtigung verschickt. Weiterhin müssen die Knoten die Commit-Benachrichtigungen, wie bereits im Kapitel 4.3.3 beschrieben, in der Commit-Reihenfolge verarbeiten. Verarbeitet ein Knoten Commit-Benachrichtigungen, obwohl noch ältere fehlen, bleiben die aktuellen Objektreplikate unvollständig, da Teile eines Objekts veraltet sein können, was für einen Knoten ohne die Verarbeitung aller Benachrichtigungen nicht erkennbar ist. Verarbeitet ein Knoten dagegen eine ältere Commit-Benachrichtigung, obwohl er bereits eine neuere auf ein Objekt angewendet hat, kann er nicht erkennen, ob die in der Commit-Benachrichtigung mitgelieferten Teiländerungen mit den Teiländerungen einer neueren Objektversion überlappen. Bei Anwendung der älteren Commit-Benachrichtigung auf dieses Objekt, überschreibt das ältere Objekt den überlappenden Teil der neueren Objektversion mit veralteten Daten. Dies führt folglich zur Dateninkonsistenz. Aus diesem Grund ist in diesem Fall nur eine Verarbeitung der Commit-Benachrichtigungen in der Reihenfolge der Commits möglich.

Bei dem Invalidierungsverfahren ist die Synchronisierung mittels differentieller Objektdaten komplexer. Maßgeblich für die Anforderung von differentiellen Objektdaten ist die auf dem Knoten aktuell gespeicherte Objektversion, die als Basis dient. Auch wenn ein Objekt auf einem Knoten invalidiert ist, so ist es nur für Zugriffe durch die Applikation gesperrt, aber dennoch weiterhin existent und bezüglich seiner veralteten Versionsnummer ein gültiges Objekt. Es darf daher weiter als Basis für die Anforderung von Objektänderungen dienen. Bei diesem Verfahren fordern Knoten entweder einen Diff für eine Objektversion oder auch eine Menge von Objekt-Diffs an, falls sich ein Objekt seit der letzten Anforderung mehrfach geändert hat. Ein effizientes Verfahren zum Speichern von Diffs in verteilten Speichersystemen behandelt [86].

Falls ein Knoten ein längere Zeit nicht zugegriffenes aber durch andere Knoten mehrfach geändertes Objekt bei einem anderen Knoten anfragt (bevorzugt beim letzten Knoten der das Objekt geändert hat), kann dieser die Anfrage wegen der beschränkten Größe der Diff-Datenstruktur eventuell nicht aus seinen Diffs bedienen. Unter diesen Umständen kann der Knoten statt des Diffs das komplette Objekt verschicken, oder aber mehrere Knoten müssen die Anfrage gemeinsam beantworten. Für letzteren Fall kann der angefragte Knoten eine Teilanfrage an an-

dere Knoten weiterleiten, oder der anfragende Knoten fragt einen anderen Knoten falls sich die Antwort als unvollständig herausstellt. Diese Verfahren beeinträchtigen die Skalierbarkeit durch Verzögerungen negativ, da sich die einzelnen Latenzen der weitergeleiteten oder mehrfachen Anfragen akkumulieren, welche gerade in Weitverkehrsnetzen hoch sein können. Der Objektaustausch ist folglich ein Kompromiß zwischen Latenz und Netzwerkbandbreite. Alternativ ist eine parallele Anfrage nach Diffs bei mehreren Knoten denkbar, allerdings ist dem anfragenden Knoten im vorhinein nicht genau bekannt, welcher Knoten welche Diffs vorrätig hat. Daher kann der Knoten doppelte oder überlappende Diffs als Antwort erhalten, die er zusammenführen muß, noch ist garantiert, eine insgesamt vollständige Antwort zu erhalten. Ein Knoten kann aber probabilistisch durch Beobachtung der Commit-Benachrichtigungen die anzufragenden Knoten identifizieren.

Weiterhin führt der Versand des kompletten Objekts mit anschließendem Einfügen in die Diff-Datenstruktur auf dem Zielknoten zu einem grob granularen Diff, da dazwischenliegende Versionen fehlen. Ein Knoten kann eine Objektanfrage mit einem Diff dagegen nicht beantworten, wenn die zugrunde gelegte Basisversion eines Objekts selbst nicht in der Diff-Datenstruktur enthalten ist. Dies führt wiederum dazu, daß die Beantwortung der Anfrage mehrere Knoten erfordert oder wieder zum Versand des gesamten Objekts führt. Ein Knoten, der eine Objektanfrage beantwortet, muß daher individuell entscheiden, Diffs oder das komplette Objekt zu übertragen. Für das günstigste Übertragungsverfahren sind Zugriffsmuster auf einzelne Objekte, Anzahl beteiligter Knoten sowie Netzwerklatenz und Bandbreite der an der Anfrage beteiligten Knoten verantwortlich. Eine detaillierte Beleuchtung, welches Verfahren zum jeweiligen Zeitpunkt gegenüber dem anderen besser ist, ist nicht Bestandteil dieser Arbeit.

4.4 Tokenaustausch

Bei dem Austausch des Tokens zwischen den einzelnen Parteien lassen sich unterschiedliche Verfahren unterscheiden. Die hier betrachteten Verfahren müssen tolerant gegenüber Fehlern in verteilten Systemen sein, wie dies beispielsweise bei Tokenverlusten oder -duplikaten der Fall ist.

4.4.1 Zirkulierendes Token (Round Robin)

Bei der zirkulierenden Tokenweitergabe handelt es sich um ein implizites Verfahren, welches von den Teilnehmern keine gesonderte Anfrage nach dem Token erfordert. Alle Teilnehmer sind fest in einem logischen Ring angeordnet und haben je einen Vorgänger und Nachfolger, von dem sie das Token erhalten respektive weitergeben. Eine Rekonfiguration des Rings ist mit wenig Aufwand durchführbar, so daß sich dieses Verfahren auch für ein dynamisches Netz eignet. Bei einem Eintritt in den Ring meldet sich ein neuer Teilnehmer bei einem bereits integrierten Teilnehmer. Dieser teilt ihm daraufhin seinen Nachfolger als seinen bisherigen Nachfolger mit. Der angefragte Knoten verwendet seinerseits den neuen Knoten als Nachfolger. Der Austritt erfolgt analog, indem ein Knoten sich mit einem Austrittsgesuch an seinen Vorgänger wendet und gleichzeitig seinen bisherigen Nachfolger in dem Austrittsgesuch mitteilt[3]. Alternativ ließe sich der Ring auch als Liste im Token selbst transportieren.

Dieses Verfahren kann sich nachteilig auf die Skalierbarkeit auswirken, da dieses Verfahren nicht die Wünsche der Teilnehmer hinsichtlich von tokenrestriktiven Operationen berücksichtigt. Dies bedeutet, daß ein Teilnehmer das Token bei einer Zirkulation durch den Ring auch erhält, obwohl er dieses nicht benötigt. Die einzige Aufgabe besteht für den Teilnehmer nur

[3] Beim Ein- und Austritt können Vorgänger und Nachfolger gegeneinander getauscht werden.

in einer Weiterleitung an seinen Nachfolger. Neben einem unnötigen Netzwerkdatentransfer geht hier Zeit durch die sich akkumulierenden Latenzen verloren. Dies wirkt sich umso negativer auf die Skalierbarkeit aus, je mehr Teilnehmer der Ring umfaßt und je weniger Knoten das Token überhaupt benötigen. Das folgende Beispiel verdeutlicht den Nachteil dieser Vergabepraxis unter den zuvor genannten Bedingungen. Angenommen ein Knoten darf nur eine Transaktion pro Tokenerhalt ausführen, und in einem Netz von 100 Teilnehmern führen nur fünf Teilnehmer Transaktionen aus, dann sind ohne Berücksichtigung der Transaktionsausführungsdauer rund 95 Prozent der Weiterleitungszeit auf das Token überflüssig. Die Weitergabe des Tokens zwischen den fünf Teilnehmern nimmt im Mittel die zwanzigfache durchschnittliche Netzwerklatenz in Anspruch. Eine zirkulierende Tokenweitergabe wäre optimal, sofern jeder Knoten bei Erhalt des Tokens auf den Commit einer eigenen Transaktion wartet.

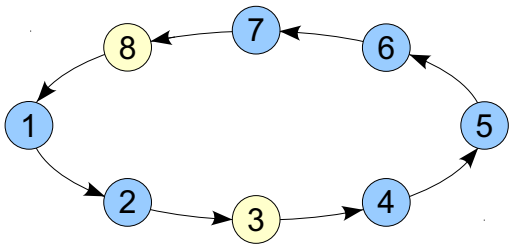


Abbildung 4.10: Tokenweitergabe im Ring-Verfahren.

Abbildung 4.10 zeigt die Tokenweitergabe in einem Ring. Angenommen, die in der Abbildung dargestellten Knoten sind am transaktionalen Speicher beteiligt, aber nur die beiden hervorgehobenen Peers *3* und *8* schließen derweilen Transaktionen ab. Zur Vereinfachung werden die Transaktionen als kollisionsfrei angenommen. Nachdem Peer *8* das Token abgegeben hat, muß Peer *3* die dreifache mittlere Netzwerklatenz auf das Token warten. Im umgekehrten Fall muß Knoten *8* die fünffache mittlere Netzwerklatenz abwarten.

4.4.2 Koordiniertes Token

Bei der koordinierten Tokenvergabe delegiert ein Koordinator den Tokenaustausch zwischen den einzelnen Parteien. Teilnehmer fordern das Token hierbei explizit beim Koordinator an, der ihnen das Token im weiteren Verlauf zuteilt. Hat der Tokeninhaber seine Operation ausgeführt, so gibt er es an den Koordinator zurück. Bei einer erneuten Anfrage verteilt er das Token weiter. Zwischenzeitlich auflaufende Anfragen puffert der Koordinator in einer Warteschlange zwischen. Im einfachsten Fall vergibt der Koordinator das Token immer an den nächsten anfragenden Teilnehmer aus der Warteschlange. Diese Art der Verteilung folgt dem *Windhundverfahren (engl. first-come, first-served)*. Bei dieser Vergabepraxis werden leistungsfähige Teilnehmer mit einer geringen Latenz unweigerlich gegenüber schwächeren Teilnehmern bevorzugt. Der Koordinator kann aber über Statistiken die Fairneß kontrollieren, indem er beispielsweise Teilnehmer, die das Token bisher weniger häufig angefragt hatten, bevorzugt.

In einer optimierten Variante integriert das Token eine Warteschlange, welche weitere Tokenanfragen transportiert. Über diese steuert der Koordinator, die direkte Weitergabe des Tokens an andere Knoten. Anstatt das Token nach der Freigabe an den Koordinator zurückzugeben gibt ein Knoten das Token an einen Knoten in der im Token integrierten Warteschlange weiter. Bevor der Koordinator das Token weggibt, verschiebt er eine eigens bestimmte Anzahl von Anfragen aus seiner Warteschlange in die des Tokens, sofern diese nicht leer ist. Der Koordinator darf die Reihenfolge der Anfragen in der Tokenwarteschlange festlegen, um Fairneß zu garantieren. Bevor ein Knoten das Token nach der Freigabe an den Koordinator zurückgibt

prüft er zunächst, ob die Warteschlange im Token weitere Anfragen erhält. Ist dies der Fall, entfernt er eine Anfrage aus der Warteschlange und leitet das Token an den durch die Anfrage identifizierten Knoten weiter. Ein Knoten gibt das Token erst wieder an den Koordinator zurück, wenn sich keine Einträge mehr in der Warteschlange befinden. Der Koordinator sollte die Anzahl der Einträge in der Tokenwarteschlange begrenzen. So kann er im Falle eines Tokenverlusts den Kommunikationsaufwand für die Invalidierung und Neugenerierung des Tokens minimieren. Er muß in diesem Fall nur die durch die Warteschlange referenzierten Knoten und den Knoten, der das Token von ihm zuerst erhalten hat, kontaktieren.

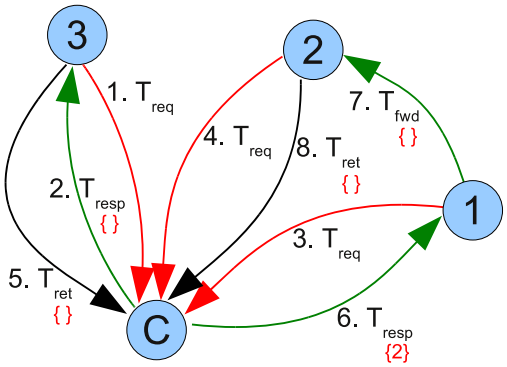


Abbildung 4.11: Koordiniertes Tokenverfahren mit integrierter Warteschlange.

Abbildung 4.11 zeigt die Tokenweitergabe mit Warteschlange anhand eines Beispiels. Die dort dargestellten Mengenangaben kennzeichnen die interne Warteschlange des Tokens während dessen Weitergabe. Knoten *3* schickt eine Anfrage an den Koordinator *C* (1). Dieser schickt das Token daraufhin an den anfragenden Knoten (2). Zwischenzeitlich schicken Knoten *1* und *2* ebenfalls Anfragen an den Koordinator (3 + 4). Diese laufen in der Warteschlange des Koordinators auf, da dieser zur Zeit nicht im Besitz des Tokens ist. Im folgenden Schritt (5) gibt der Knoten *3* das Token nach der Freigabe an den Koordinator zurück, da die tokeninterne Warteschlange keinen Eintrag enthält. Der Koordinator gibt das Token nun an den nächsten Knoten *1* aus der Warteschlange weiter (6), schiebt aber zuvor weitere Einträge aus der lokalen in die tokeninterne Warteschlange. Da die tokeninterne Warteschlange noch eine Anfrage von Knoten *2* enthält, gibt der augenblickliche Tokenhalter das Token nicht an den Koordinator zurück. Stattdessen entfernt er den ersten Eintrag (Anfrage von Knoten *2*) aus der Warteschlange des Tokens und leitet dieses entsprechend weiter. Knoten *2* gibt das Token mit der Freigabe an den Koordinator zurück, da die Warteschlange keine weiteren Einträge mehr enthält.

Das koordinierte Tokenverfahren bietet den Vorteil, daß sich ein Knoten für die Anforderung des Tokens nur an einen zentralen und zuvor bekannten Knoten wenden muß. Demnach entfällt eine Suche nach dem Token im Netzwerk. Nachteilig ist dagegen, daß der Koordinator bei vielen gleichzeitigen Tokenanfragen einer starken Belastung ausgesetzt ist und somit zum Flaschenhals im System werden kann. Ist der Koordinator nicht dediziert, also selbst an Operationen auf dem transaktionalen Speicher beteiligt, können sich Tokenverwaltung und die Anwendung, die auf dem transaktionalen Speicher arbeitet, gegenseitig ausbremsen. Weiterhin ist die Anwendung des tokenkoordinierenden Knotens aufgrund fehlender Netzkommunikation gegenüber anderen Knoten bei der Tokenanforderung im Vorteil. Daher ist es Aufgabe des Koordinators, das Token so zu verteilen, daß jeder Knoten seine Transaktionen gleichberechtigt abschließen kann (siehe Kapitel 6.5).

4.4.3 Peer-to-Peer-Token

Diese P2P-Strategie funktioniert ähnlich der koordinierten Vergabe, kommt jedoch ohne zentralen Koordinator aus. Teilnehmer fragen das Token beim aktuellen Tokenbesitzer statt bei dem Koordinator an. An dieser Stelle treten die klassischen Probleme eines hochdynamischen P2P-Netzwerkes hinsichtlich der Tokenauffindung in ähnlicher Weise auf, wie sie beim verteilten transaktionalen Speicher auch bei der Lokalisierung aktueller Objektkopien bestehen. Eine multicast-basierte Flutung des Netzes an alle Teilnehmer würde das Netzwerk übermäßig stark belasten. Zusätzlich muß diese Variante auch Token im Transit bei gleichzeitiger Vermeidung doppelter Anfragen und Vermeidung des Verlusts von Anfragen gerecht werden. Dies würde eine hohe Protokollkomplexität erfordern. Aus diesem Grund findet ein multicast-basiertes Protokoll an dieser Stelle keine weitere Beachtung.

Bei diesem Verfahren erfolgt die Tokenweitergabe nur auf Anfrage. Das Token verbleibt also so lange bei einem Knoten, bis dieses ein anderer Knoten abholt. Für einen anstehenden Commit schickt ein Knoten die Tokenanfrage an den ihm zuletzt bekannten Tokenhalter. Hierfür merkt sich ein Knoten den Empfänger des Tokens bei der Tokenweitergabe. Empfängt ein Knoten eine Tokenanfrage, obwohl er nicht mehr im Besitz des Tokens ist, leitet er diese wiederum an dem ihm zuletzt bekannten Tokenhalter weiter. Durch die Weiterleitung erreicht jede Anfrage nach endlich vielen Schritten den Tokenbesitzer.

Sofern der letzte Besitz des Tokens einige Zeit her ist, könnte das Token bereits mehrfach weitergegeben worden sein, was vermeintlich auch zu einer ebenso hohen Weitervermittlung führt. Zyklen in der Weiterleitungskette fallen automatisch heraus, da sich ein Knoten nur den letzten ihm bekannten Tokenbesitzer merken muß. Dies ist zunächst der Teilnehmer, an den er das Token zuletzt weitergegeben hat. Somit ergibt sich eine maximale Weiterleitungskette von maximal $n - 2$ Knoten. Eine weitere Reduzierung der Weiterleitungsstationen läßt sich durch Auswertung empfangener Informationen des P2P-Commit-Protokolls erreichen. Ein Teilnehmer geht bei einer empfangenen Commit-Nachricht davon aus, daß dieser Teilnehmer entsprechend den Protokollvereinbarungen auch das Token besitzt. Ein Teilnehmer aktualisiert in diesem Fall seinen Eintrag des letzten Tokenbesitzers mit dem Absender der Commit-Nachricht. In der Regel erreicht ein Teilnehmer so mittels weniger Weiterleitungsschritte mit seiner Tokenanfrage den aktuellen Besitzer, unabhängig vom Zeitpunkt seines letzten Tokenbesitzes. In der Praxis ergeben sich jedoch aufgrund der *Flüchtigkeit* des Tokens Wettlaufsituationen zwischen der Weiterleitung der Tokenanfrage und der Weitergabe des Tokens an Dritteilnehmer. Daher kann die Anzahl der Weiterleitungen auch auf mehr als die zuvor erwähnten $n - 2$ Schritte anwachsen, was aufgrund der Akkumulation der Netzlatenzen die Skalierbarkeit stark reduziert.

Unter hoher Last führen die Weiterleitungen der Tokenanfragen zu einer hohen Netzbelastung, da alle beim Tokenhalter aufgelaufenen Tokenanfragen (ausgenommen die des zukünftigen Tokenempfängers) dem Token bei seiner Weitergabe zum neuen Zielknoten folgen. Dieses Problem läßt sich reduzieren, indem das Token wie bei dem koordinierten Verfahren Tokenanfragen in einer integrierten Warteschlange transportiert. Jeder Tokenhalter ist verpflichtet, freie Plätze in der tokeninternen Warteschlange vor der Weitergabe des Tokens mit eventuell aufgelaufenen Anfragen aufzufüllen. Weiterhin haben die Anfragen in der Warteschlange Vorrang gegenüber denen beim Tokenhalter aufgelaufenen Anfragen.

Abbildung 4.12 zeigt beispielhaft die P2P-Tokenweitergabe mit der tokeninternen Weiterleitungswarteschlange (siehe Mengenangaben in der Abbildung). Angenommen Knoten *1* sei der aktuelle Tokenhalter. Knoten *2* fragt nach dem Token (1) und bekommt es in der darauffolgenden Antwortnachricht (2). Während Knoten *2* mit der Verarbeitung von Transaktionen beschäftigt ist, schicken die Knoten *3* und *4* Tokenanfragen (3 + 4), welcher der ihnen zuletzt bekannte Tokenhalter ist. Da Knoten *1* nicht mehr im Besitz des Tokens ist, leitet er die Tokenanfragen an dem ihm zuletzt bekannten Tokenhalter weiter (5). Nachdem Knoten *2* seine

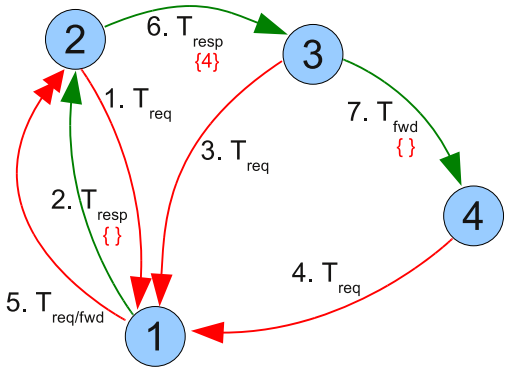


Abbildung 4.12: P2P-Tokenverfahren mit integrierter Warteschlange.

Transaktionsoperationen beendet und das Token nicht weiter benötigt, füllt er die tokeninterne Liste mit aufgelaufenen Anfragen bis auf die erste auf, welche bereits das Weiterleitungsziel des Tokens bestimmt. Anschließend leitet Knoten *2* das Token an den Knoten *3* weiter (6). Falls nicht alle aufgelaufenen Anfragen in der tokeninternen Warteschlange Platz finden, leitet ein Knoten diese nach der Abgabe des Tokens weiter. Nachdem Knoten *3* das Token freigegeben hat, leitet er es direkt an Knoten *4* weiter, da die tokeninterne Liste eine Anfrage von Knoten *4* enthält.

Unter der Annahme, daß die Tokenhaltedauer vernachlässigbar klein ist und Knoten das Token nach ihrem Commit unmittelbar an einen anderen Knoten weitergeben, ergibt sich für n zur gleichen Zeit anfragende Knoten mit einer durchschnittlichen Latenz T_l eine mittlere Tokenanforderungszeit von

$$T_{tok} = T_l + \frac{\left(\sum_{k=1}^{n} k\right) * T_l}{n} \qquad n, T_l \in \mathbb{N} \tag{4.3}$$

4.4.4 Peer-to-Peer-Token mit Tokenvorhersage

Die Warteschlange im P2P-Token verringert zwar die Netzbelastung, reduziert aber nicht die Anzahl der Weiterleitungen von Tokenanfragen und der damit verbundenen Wartezeit. Als Optimierung erlaubt die Verknüpfung unterschiedlicher Informationen eine probabilistische Vorhersage des Tokenbesitzers. Jeder Knoten merkt sich statt dem letzten Tokenhalter die Anfragen aus der internen Tokenwarteschlange, sofern diese nicht leer ist. Empfängt der Knoten eine Commit-Benachrichtigung, streicht er den Knoten aus der Liste, sofern vorhanden. Mittels einer zeitlichen Abschätzung kann der Knoten eigene Tokenanfragen beziehungsweise von anderen Knoten empfangene Anfragen an einen Knoten aus der Warteschlange weiterleiten, bei dem er das Token in naher Zukunft vermutet. Hierfür ist es notwendig, daß Knoten eingehende Anfragen nur weiterleiten, sofern sie selbst keine Anfrage nach dem Token gestellt haben. Anderenfalls müssen sie die Anfragen puffern und bedienen, nachdem sie das Token nach Ausführung ihrer eigenen Operationen freigegeben haben.

Es ist ausreichend, sich nur den letzten Eintrag der Warteschlange als zukünftigen neuen Tokenhalter zu merken, da der Verlauf der Tokenweitergabe bereits durch die Liste vorgegeben ist und nach dessen vorliegenden Informationen beim letzten Knoten der Liste endet. Um Einfluß auf die Fairneß bei der Tokenweitergabe zu nehmen, kann ein Knoten seine Anfrage dennoch an einen beliebigen Teilnehmer der Tokenliste schicken. Der empfangende Knoten entscheidet

aber, an welche Stelle der bereits bestehenden Anfragen er die neue einreiht. Hierbei ist zu beachten, daß weiter vorne in der Liste stehende Teilnehmer das Token eher als nachfolgende Teilnehmer erhalten. Die Anfrage an einen der vorderen Teilnehmer der Liste birgt das Risiko, daß der Knoten die Anfrage erst erhält, nachdem er das Token bereits weitergegeben hat. Demnach muß er die eingehende Anfrage dann weiterleiten. Gleiches gilt im Falle einer zwischenzeitlichen Umsortierung der tokeninternen Warteschlange, zum Beispiel aus Fairneßgründen. In diesem Fall geht ein Knoten, der seine Anfrage sendet beziehungsweise eine empfangene Anfrage weiterleitet von einer falschen Annahme aus und kann den aktuellen Tokenhalter möglicherweise verpassen. An dieser Stelle ist dann ebenfalls eine Weiterleitung der Tokenanfrage notwendig.

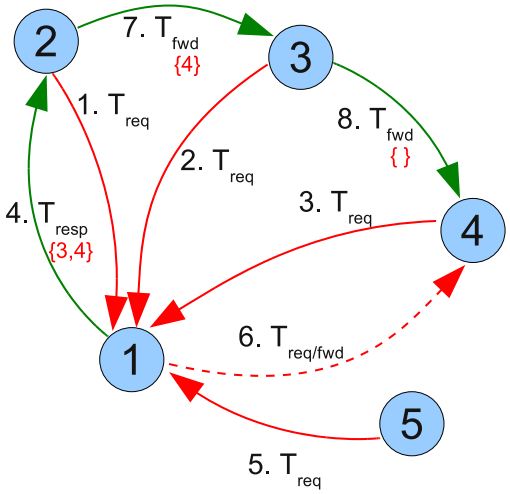


Abbildung 4.13: P2P-Tokenverfahren mit integrierter Warteschlange und Vorhersage.

In Abbildung 4.13 ist ein Beispiel der vorsagebasierten P2P-Tokenweitergabe dargestellt. Knoten *1* erhält als aktueller Tokenhalter zunächst drei Anfragen (1, 2, 3). Nachdem er das Token freigegeben hat, füllt er zunächst die Tokenwarteschlange mit den weiteren Anfragen der Knoten *3* und *4* und gibt das Token anschließend an den Knoten *2* weiter (4). Nachdem Knoten *1* das Token weitergegeben hat, empfängt er von Knoten *5* eine weitere Anfrage (5). Anstatt diese auch an Knoten *2* weiterzuleiten, schickt er diese auf direktem Weg an Knoten *4* weiter (6), da dieser Knoten nach seinen Informationen der letzte zukünftige Tokenhalter ist. Knoten *4* puffert die Anfrage anstatt sie weiterzuleiten, da er selbst eine Anfrage gestellt hat.

Dieses Verfahren beinhaltet jedoch einen Nachteil, der je nach Zeitverhalten der Anwendung eine Tokenweiterleitung auf Basis einer Vorhersage des Tokenhalters verhindert. Die lokale Kopie der Tokenwarteschlange nach der Tokenabgabe veraltet nach einiger Zeit beziehungsweise leert sich mit jeder empfangenen Commit-Benachrichtigung. Zudem erhält nur der Knoten die aktuelle Liste, der das Token gerade erhalten hat. Abhilfe schafft hier eine Aktualisierung der lokalen Kopie der Tokenliste auf den einzelnen Knoten. Ein transaktionsabschließender Knoten kann die aktuelle Tokenwarteschlange in der Commit-Benachrichtigung mitschicken. Dadurch hat jeder Knoten immer eine aktuelle Kopie der Tokenwarteschlange. Die aktuellste Version der Warteschlange aus den Commit-Benachrichtigungen und dem Token selbst bestimmt sich aus der höchsten Commit-ID. Bei gleicher Commit-ID zwischen einer Commit-Benachrichtigung und der Tokennachricht, ist die Warteschlange aus dem Token derjenigen aus der Commit-Benachrichtigung vorzuziehen, da ein Knoten das Token immer nach dem Commit freigibt, was eine Integration weiterer Anfragen nach dem Commit ermöglicht.

Abbildung 4.14 zeigt eine ähnliche Situation wie in Abbildung 4.13. Der Unterschied besteht

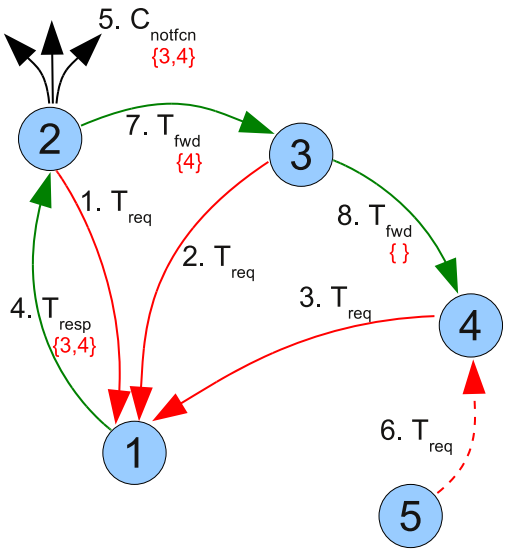


Abbildung 4.14: P2P-Tokenverfahren mit Warteschlangenaktualisierung durch Commit-Benachrichtigungen.

jedoch in dem Versenden der aktuellen Tokenwarteschlange in der Commit-Benachrichtigung. Bevor Knoten *2* das Token entsprechend der Warteschlange an den Knoten *3* weiterleitet (7), schickt er zunächst die aktuelle Warteschlange in der Commit-Benachrichtigung (5) aufgrund einer erfolgreich abgeschlossenen Transaktion an alle Knoten. Da alle Knoten nun mit jedem Commit die aktuelle Warteschlange erhalten, schickt Knoten *5* seine Tokenanfrage stattdessen an Knoten *4* statt an *1*, da ihm dieser Knoten als letzter zukünftiger Tokenhalter bekannt ist (6). Voraussetzung ist jedoch, daß Knoten *5* die Commit-Benachrichtigung vor dem Versand der eigenen Tokenanfrage erhält. Ansonsten wäre Knoten *1* der letzte bekannte Tokenhalter und würde die Anfrage erhalten.

Aufgrund von Nebenläufigkeit und ausbleibender Commit-Benachrichtigungen aufgrund konfliktbehafteter Transaktionen kann die lokale Kopie der Tokenwarteschlange dennoch vereinzelt veraltete Einträge enthalten, so daß versendete Tokenanfragen bei Knoten eintreffen, die selbst nicht mehr im Besitz des Tokens sind beziehungsweise nicht mehr auf dieses warten. Neue Commit-Benachrichtigungen ersetzen aber veraltete Warteschlangenkopien und somit auch veraltete Einträge. Die korrekte Tokenvorhersage trifft also nicht immer zu, kann die mehrfache Weiterleitung von Tokenanfragen aber vermindern.

4.4.5 Transaktionsabbruch nach Tokenanforderung

Die Anforderung des Tokens für den Commit der abzuschließenden Transaktion garantiert noch nicht, daß diese tatsächlich erfolgreich abschließen kann. Da die Komponenten eines verteilten Systems nebenläufig arbeiten, können in der Zeit von Transaktionsende und Erhalt des Tokens noch Schreibmengen (Commit-Benachrichtigungen) von entfernten Transaktionen eintreffen, die zur Invalidierung der Transaktion führen (siehe Abbildung 4.15[4]). Gleiches gilt, falls der Knoten das Token bereits erhalten hat, aber noch auf fehlende Commit-Benachrichtigungen und deren Schreibmengen und somit auch auf die interne Tokenfreigabe

[4] *BOT/EOT:* Transaktionsgrenzen, $r(x_n)/w(x_n)$*:* Lesen beziehungsweise schreiben der Version n des Objekts x.

warten muß (siehe Kapitel 4.3.2). Im Konfliktfall war die Anforderung des Tokens für den Commit der Transaktion nutzlos, läßt sich aufgrund der Nebenläufigkeit aber nicht voraussehen. Ein Knoten kann das Token unmittelbar vor Rücksetzung der Transaktion wieder freigeben, so daß andere Knoten ihrerseits mit dem Commit eigener Transaktionen fortfahren können. Alternativ kann der Knoten das Token auch behalten, damit würde die wiederholt ausgeführte Transaktion in jedem Fall abschließen, da zwischenzeitlich keine fremden Schreibmengen einen Konflikt verursachen können.

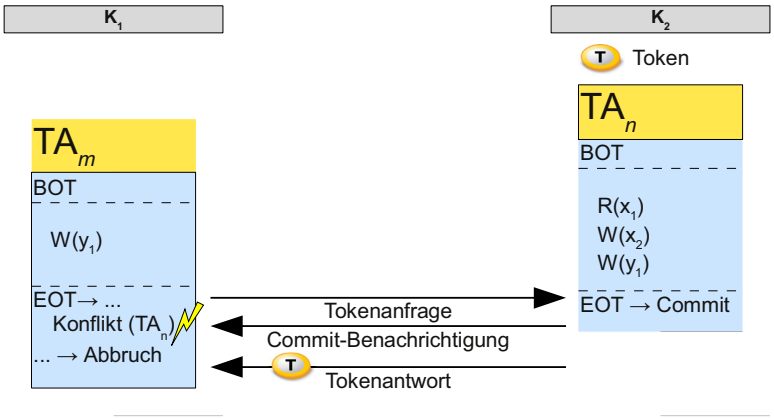


Abbildung 4.15: Transaktionsabbruch nach Anforderung des Tokens.

Ohne Betrachtung von Fairneß und den Einfluß auf die Konflikthäufigkeit von Transaktionen ist eine unmittelbare Tokenfreigabe immer dann sinnvoll, sofern die Ausführungsdauer der aufgrund eines Konflikts erneut auszuführenden Transaktion größer als die Netzwerklatenz (Round-Trip-Time (RTT)) ist. Hierbei ist zu beachten, daß erneut auszuführende Transaktionen einen zusätzlichen Zeitaufwand für die Neuanforderung konfliktverursachender Objekte über das Netzwerk haben. Die Tokenfreigabe ermöglicht es anderen Knoten, zwischenzeitlich ihre Transaktionen abzuschließen, sofern diese konfliktfrei sind. Anderenfalls ist es sinnvoller, das Token während der Wiederholung der abgebrochenen Transaktion nicht freizugeben, was nach dessen Abschluß auf jeden Fall zu einem erfolgreichen Commit führt, da keine konfliktverursachenden Commits anderer Knoten stattfinden können. Da eine Wiederholung einer Transaktion mindestens die Anforderung eines aktuellen Objektreplikats erfordert (konfliktverursachendes Objekt), ist die Ausführungsdauer der wiederholenden Transaktion in der Regel immer größer als die Netzwerklatenz. Demnach ist eine unmittelbare Tokenfreigabe in diesem Fall sinnvoll. Diese Verfahrensweise kann jedoch dazu führen, daß die erneut ausgeführte Transaktion immer wieder abbricht, da andere Knoten aufgrund der in der Regel kürzeren Ausführungsdauer ihrer Transktionen mit anschließender Tokenanforderung erneut Konflikte provozieren können. Dies kann schlimmstenfalls zum aushungern der transaktionsausführenden Anwendung führen. Das System muß dies im Rahmen der Fairneß vermeiden (siehe Kapitel 6.5).

4.5 Objektlokalisierung

Transaktionen lesen und schreiben Objekte des verteilten Speichers. Schließt ein Peer eine Transaktion ab, so bleibt die letzte Version der gelesenen Objekte weiterhin gültig. Hinsichtlich geschriebener Objekte ändert eine Transaktion den Inhalt der zuletzt gültigen Objektversion. Von einem Objekt entsteht also eine neue Version, welche die vorherige Version ungültig macht. Da von Objekten durch Lesezugriffe innerhalb von Transaktionen beliebig viele Objekt-

replikate (siehe Kapitel 2.1) auf Peers entstehen können, sind diese nach einem Schreibvorgang automatisch auch ungültig, da sie auf derselben vorherigen Objektversion basieren.

Nachdem ein Peer ein Objekt beschrieben hat, existiert vom diesem Zeitpunkt an nur ein einziges Replikat. Dies bedeutet, daß Objekte bei jedem Schreibzugriff zu dem jeweils schreibenden Knoten hinwandern. Möchte ein anderer Peer in einer nachfolgenden Transaktion dasselbe Objekt ebenfalls zugreifen, muß er zunächst den Peer finden, der die aktuelle beziehungsweise zuletzt geänderte Version besitzt.

Für eine gute Skalierbarkeit ist eine effiziente Suche nach Objekten im verteilten System notwendig. Die Schwierigkeit eines Peers, die aktuelle Objektversion im Netz zu lokalisieren, unterliegt einer ähnlichen Problematik wie der Suche nach dem Commit-Token. Der grundlegende Unterschied besteht jedoch darin, daß das Token aufgrund der Anfragen von vielen Knoten oft zwischen den Knoten wandert und somit nur kurzzeitig auf einem Knoten verbleibt. Objektreplikate verbleiben dagegen in der Regel länger auf einem Knoten, da nicht jede Transaktion alle Objekte verwendet und von den zugegriffenen auch nur einen Teil schreibend zugreift. Somit unterscheiden sich die Suche nach dem Token und Objektreplikaten aufgrund unterschiedlicher Zugriffsmuster. Peers empfangen Benachrichtigungen (Notifications), die Informationen darüber liefern, welche Objekte ein Peer in einer erfolgreich abgeschlossenen Transaktion beschrieben hat. Die Benachrichtigung dient dazu, eventuell auf dem Empfängerpeer existierende Objektreplikate zu invalidieren, da es sich bei denen nach dem Commit offensichtlich um eine veraltete Version handelt. Der Absender der Benachrichtigung identifiziert zugleich aber auch die Lage der aktuellen Objektversion, da diese beim zuletzt transaktionsabschließenden Knoten liegen muß. Ein Knoten kann so immer die aktuelle Objektversion abrufen. Lediglich aufgrund der Nebenläufigkeit kann es vorkommen, daß ein Knoten die aktuelle Objektversion bei einem Knoten anfordert aber dennoch nicht vorfindet. Dieser Fall tritt ein, wenn ein anderer Knoten gerade zum Zeitpunkt der Anfrage eine eigene Transaktion abschließt und so die Objektversion auf dem Peer, der die Objektanforderung empfängt, invalidiert. In ähnlicher Weise kann es vorkommen, daß der anfragende Knoten die Objektkopie gerade kurz vor der Objektinvalidierung erhält. Der anfragende Peer bekommt die Objektkopie zwar ausgeliefert, allerdings ist diese durch den zwischenzeitlichen Commit dennoch ungültig geworden.

Wenn ein Peer eine aktuelle Objektversion bei dem zum Zeitpunkt der Anfrage zuletzt transaktionsabschließenden Peer anfordert, erhält er im Optimalfall eine aktuelle Objektversion, die ihm zu einer eigenen erfolgreichen Transaktionsvalidierung verhilft. Im Gegensatz dazu erhält er von dem angefragten Knoten entweder keine oder aber eine veraltete Kopie. Ist die Kopie auf dem Empfängerknoten bereits invalidiert, bevor ihn die Objektanfrage erreicht, ist es sinnvoll, die Anfrage an den neuen aktuellen Objekthalter weiterzuleiten, um die Netzkommunikation und die damit verbundene Verzögerung zu minimieren. Erhält der Empfänger eine veraltete Kopie, kann er das erst dann bemerken, wenn er selbst die Commit-Benachrichtigung von dem zuletzt transaktionsabschließenden Peer erhält. Dies geschieht spätestens kurz bevor er das Token erhält. Ein Peer kann im Vorfeld nicht erkennen, daß eine erhaltene Objektversion während oder nach der Empfangsphase bereits veraltet ist. Dies läßt sich in nebenläufigen Systemen auf Basis von Objektreplikaten nicht verhindern.

4.6 Fehlertoleranz

Bei verteilten Anwendungen kann es neben dem erwünschten Verhalten auch zu Fehlersituationen kommen, die hauptsächlich durch Netzwerkfehler, Knotenabstürze und Abstürze von Anwendungsprozessen begründet sind. Eine verteilte Anwendung verhält sich fehlerhaft, sobald mindestens ein Knoten beziehungsweise dessen Netzwerkverbindungen von einem Fehler betroffen sind. Somit ist die Fehlerwahrscheinlichkeit einer verteilten Anwendung, die auf den

verteilten transaktionalen Speicher aufsetzt, proportional zu der Anzahl der beteiligten Knoten. Damit die Anwendungen trotz einer höheren Fehlerwahrscheinlichkeit zuverlässig arbeiten, müssen die verwendeten Protokolle tolerant gegenüber Fehlern sein. Dies kann zum einen durch eine für die Anwendung transparente Fehlerkompensation beziehungsweise Fehlerbehebung erfolgen. Treten Fehler auf, die das Protokoll selbst nicht behandeln kann, muß das System die Anwendung anhalten und neu starten. Eine Anwendung darf keinesfalls weiterlaufen, wenn Fehler das Anwendungsverhalten insofern beeinflussen, daß diese falsche Ergebnisse liefert.

4.6.1 Verlust von Commit-Nachrichten

Da Netzwerke generell nicht fehlerfrei sind, können Situationen entstehen, in denen Peers Commit-Nachrichten verpassen (z. B. wegen kurzzeitiger Netzunterbrechungen, Routingfehler oder Fehler im Overlay-Netzwerk). Eine fehlende Commit-Benachrichtigung würde bei einem anstehenden Commit auf dem jeweiligen Knoten zur Verklemmung führen. Ein Knoten muß daher ausgebliebene Commit-Benachrichtigungen von anderen Knoten nachfordern. Dies kann er beispielsweise mittels einer Zeitüberschreitung (Timeout) erkennen, falls er das Token für den Commit eigener Transaktionen oder bereits Commit-Benachrichtigungen mit einer höheren Commit-ID erhalten hat.

Ein Knoten stellt beispielsweise aufgrund der Commit-ID einer empfangenen Nachricht (Commit-Nachricht oder Commit-Token) fest, daß er eine vorherige Commit-Nachricht noch nicht empfangen hat. Hat er die fehlende Nachricht nach einer Zeitüberschreitung noch immer nicht empfangen, so ist von einer verlorengegangenen Nachricht auszugehen. Der Knoten kann demnach keine eigenen Transaktionen festschreiben, da er für seine Objektreplikate nicht entscheiden kann, ob diese weiterhin gültig sind. Damit der Knoten weiterarbeiten kann, muß er nicht erhaltene Commit-Nachrichten von anderen Knoten nachfordern (siehe Kapitel 4.6.2). Hat er die nachgeforderten Benachrichtigungen verarbeitet, kann er feststellen, welche der bei ihm gespeicherten Objektreplikate weiterhin gültig sind. Infolgedessen darf er eigene Transaktionen festschreiben, da er aufgrund des Gültigkeitsstatus seiner Objektreplikate feststellen kann, ob seine abzuschließende Transaktion konfliktfrei ist.

4.6.2 Transaction-History-Buffer

Damit Knoten Commit-Benachrichtigungen nachfordern können, integriert jeder Knoten einen *Transaction-History-Buffer (THB)*, welcher versendete Commit-Benachrichtigungen eine Zeit lang puffert. Ein THB dient der Fehlertoleranz des verteilten transaktionalen Speichers und speichert versendete und eventuell auch empfangene Commit-Benachrichtigungen. So können Knoten im Falle eines Fehlers Commit-Benachrichtigungen nachfordern und andere Knoten diese aus ihrem THB bedienen. Knoten können auch empfangene Commit-Benachrichtigungen für eine bessere Fehlertoleranz speichern und diese anderen Knoten auf Anfrage zur Verfügung stellen. Beruft sich ein Knoten auf eine Commit-Nachricht, die kein anderer Knoten mehr in seinem Puffer hat, so muß er sich auf anderem Wege auf den transaktionalen Speicher synchronisieren. Dies kann beispielsweise durch einen Neubeitritt in das transaktionale System erfolgen (siehe Kapitel 4.6.3).

Ein THB muß einer Größenbeschränkung unterliegen, da wegen der Menge von Commits ansonsten der Speicher eines Knotens vollaufen würde. Deswegen kann er nicht alle Benachrichtigungen aufnehmen und demzufolge nicht in jedem Fall nachgeforderte Commit-Benachrichtigungen bedienen. Ein Knoten kann über den Erhalt von Commit-Benachrichtigungen Annahmen treffen, ob selbst versandte oder empfangene Benachrichtigungen im System noch benötigt werden. Dies ist jedoch nicht zuverlässig, da ein Knoten nicht unbedingt

Kenntnis über alle anderen Knoten im System hat, insbesondere weil Knoten dynamisch bei- und austreten können.

4.6.3 Neusynchronisierung von Knoten im Fehlerfall

Erfordert ein Fehlerfall einen erneuten Beitritt eines Knotens in das System, da auf andere Weise keine Datenkonsistenz gewährleisten kann, kann ein Knoten auf folgende Weise verfahren. Ein Knoten kann alle nach seiner Ansicht gültigen Datenreplikate an einen oder mehrere andere Knoten seiner Wahl abgeben und diese anschließend bei sich invalidieren. Die anderen Knoten können anhand der lokal gespeicherten Versionsnummer der Objekte erkennen, welches der erhaltenen Datenreplikate gültig ist oder nicht. Gültige Replikate pflegen die Knoten lokal ein, ungültige verwerfen sie dagegen. Auf diese Weise kann ein Knoten ohne Datenverlust aufgrund eines Fehlers aus dem System austreten und erneut beitreten. Nach dem erneuten Beitritt besitzt der Knoten keine gültigen Objektreplikate und synchronisiert sich daher bei einem Objektzugriff in einer Transaktion automatisch mit den anderen Teilnehmern. Weiterhin muß sich der Knoten zu Beginn des Neubeitritts auf den aktuellen Stand des Commit-ID-Zählers synchronisieren, damit er nach dem Erhalt des Tokens für den Commit eigener Transaktionen nicht auf angeblich fehlende Commit-Benachrichtigungen wartet.

Auch wenn eine Resynchronisierung im Fehlerfall weiterhin die Datenkonsistenz des verteilten Speichers garantiert, folgt daraus nicht unmittelbar, daß die verteilte Anwendung fehlerfrei weiterlaufen kann. Die Anwendung kann nur in solchen Fehlerfällen weiterlaufen, in denen eine Neusynchronisierung zu keiner Inkonsistenz zwischen dem Zustand des gemeinsamen verteilten Speichers und dem Teilzustand des lokalen Speichers führt (siehe Kapitel 4.6.5). Muß sich ein Knoten erneut mit dem System synchronisieren, weil er Commit-Benachrichtigungen verpaßt hat, die kein anderer Knoten mehr in seinem THB vorliegen hat, so kann die Anwendung fehlerfrei weiterlaufen. Es tritt keine Inkonsistenz zwischen dem lokalen und verteilten Speicher auf, da die Resynchronisierung nur zum Abbruch gerade laufender Transaktionen führt. Das Anwendungsverhalten ist das gleiche, als wenn ein Knoten seine abzuschließende Transaktion wegen eines Konflikts abbrechen muß.

4.6.4 Knotenabsturz

Stürzt ein Knoten ab, kann es gleichzeitig zu einem Datenverlust des verteilten transaktionalen Speichers kommen. Hat der Knoten geänderte Objekte gehalten, die nicht auf anderen Knoten repliziert sind, sind Teile des gemeinsamen transaktionalen Speichers unwiederbringlich verloren. Das System muß infolgedessen entsprechend des *Fail-Stop*-Fehlermodells anhalten. Da das System nicht mehr weiterarbeiten kann, muß die gesamte verteilte Anwendung entweder neu starten oder beim zuletzt gespeicherten globalen Zustand wiederanlaufen. *Checkpointing*-Systeme können den globalen Zustand verteilter Anwendungen sichern, um diese im Fehlerfall vom letzten Sicherungspunkt wiederanlaufen zu lassen [34]. Insofern sind im Fall eines Fail-Stop-Fehlers nur die Ergebnisse vom letzten Sicherungspunkt mit gültiger Datenkonsistenz des verteilten Speichers bis zum Zeitpunkt des Fehlers verloren.

Sind die Objekte dagegen bereits auf anderen Knoten repliziert, tritt kein Datenverlust ein. Ein eventuell verlorengegangenes Token kann das System wiederherstellen (siehe Kapitel 4.6.6). Das System kann im Hinblick auf die Datenkonsistenz des transaktionalen Speichers weiterarbeiten und der abgestürzte Knoten erneut beitreten. Ob die verteilte Anwendung weiterarbeiten kann, hängt dagegen von dessen Implementierung ab (siehe Kapitel 4.6.5).

4.6.5 Inkonsistenz zwischen verteiltem und lokalen Systemzuständen

Kann der gemeinsame transaktionale Speicher trotz eines Knotenabsturzes weiterhin seine Konsistenz durch Replikation sicherstellen, so garantiert dies nicht automatisch die fehlerfreie Fortführung der verteilten Anwendung. Die Anwendung muß bei einem abgestürzten Knoten sicherstellen, daß die restlichen Knoten dessen Arbeit übernehmen. Die Knoten müssen trotz des fehlenden lokalen Kontexts des abgestürzten Knotens aus dem Kontext des gemeinsamen transaktionalen Speichers ableiten können, welche nicht abgeschlossenen Aufgaben der Knoten bearbeitet hatte, um diese erneut auszuführen zu können.

Ein ähnlicher Fall besteht, wenn ein abgestürzter Knoten erneut beitreten möchte. Nach dem Beitritt des Knotens fehlt diesem sein vorheriger lokaler Zustand. Nach dem Beitritt des Knotens besteht daher eine Inkonsistenz zwischen seinem lokalen und dem verteiltem Zustand des Systems. Der Knoten muß sich aus dem Zustand des gemeinsamen transaktionalen Speichers einen neuen lokalen Zustand ableiten können, um wieder an der verteilten Anwendung teilnehmen zu können. Ohne Ableitung eines gültigen lokalen Kontexts kann dies anderenfalls zu einem geänderten und eventuell nicht erwünschten Programmablauf führen. Das System kann dann nicht mehr garantieren, ob die Anwendung korrekt zu Ende läuft und fehlerfreie Ergebnisse liefert.

4.6.6 Tokenverlust

Stürzt ein Knoten ab oder ist er im Netzwerk nicht mehr erreichbar, so führt dies nicht nur zu einem eventuellen Datenverlust (falls der Knoten geänderte aber nicht replizierte Objekte besitzt), sondern hat auch Folgen für die Koordination von Commits. Wie in Kapitel 4.1.4 bereits erwähnt, dienen Replikate unter anderem auch der Fehlertoleranz und können einen Datenverlust im Fehlerfall möglicherweise vermeiden. Stürzt der tokenbesitzende Knoten ab oder verursacht ein Netzwerkfehler den Verlust des Tokens, kann das System aufgrund der fehlenden Synchronisierungsmöglichkeit nicht weiterarbeiten und würde zum Stillstand gelangen. Das System muß diese Situation zuverlässig feststellen, um diese Art von Fehler kompensieren zu können.

Ein Knoten kann einen Tokenverlust nur über den globalen Zustand aller tokenaustauschenden Knoten zuverlässig feststellen. Hierzu eignet sich beispielsweise der Leslie Lamports Algorithmus zur Aufzeichnung globaler Zustände in verteilten Systemen [22]. Ein Knoten muß hierfür alle anderen beteiligten Knoten erreichen. Anderenfalls kann er einen Tokenverlust nicht zuverlässig feststellen, sondern nur eine Annahme darüber treffen. Unter einer falschen Annahme eines Knotenabsturzes beziehungsweise während einer zeitweiligen Netzpartitionierung können daher Tokenduplikate entstehen (siehe Kapitel 4.6.7).

Jeder beliebige vom Tokenverlust betroffene Knoten darf ein neues Token generieren und in der ursprünglichen Art und Weise weiterverwenden. Beim koordinatorbasierten Tokenverfahren kann er sich gleichzeitig als Koordinator bekanntgeben. Allerdings könnten mehrere Knoten zu der gleichen Annahme gelangen, ein neues Token generieren zu müssen. Deshalb eignet es sich in dieser Situation eine Wahl eines Knotens zu veranlassen, der ein neues Token generiert und unter Einsatz des koordinierten Tokenverfahrens auch gleichzeitig die Koordinatorrolle für sich beansprucht. Eine Wahl kann beispielsweise unter Verwendung des *Bully*-Algorithmus von Garcia-Molina erfolgen [42]. Hierbei ist allerdings zu berücksichtigen, daß dieser nicht alle Situationen sicher auflösen kann. Sofern ein beitretender Knoten während der Wahl die ID des abgestürzten Knoten erhält, können sich zwei Knoten gleichzeitig als Koordinator ankündigen. Weiterhin kann ein Knoten den bestehenden Koordinator aufgrund einer Zeitüberschreitung als abgestürzt annehmen. Falls der neu gewählte und der alte Koordinator

daraufhin gleichzeitig eine Koordinatornachricht verschicken, können diese auf anderen Knoten in unterschiedlicher Reihenfolge eintreffen. Dies führt auf den Knoten wiederum zu einer unterschiedlichen Auffassung, welcher Knoten der aktuelle Koordinator ist, da die zuletzt empfangene Nachricht den neuen Koordinator festlegt [27].

Vor der Weitergabe eines neu generierten Tokens muß ein Knoten zudem sicherstellen, relevante im Token transportierte Metadaten (z. B. Commit-ID) zuvor wiederherzustellen. Die aktuelle Commit-ID ergibt sich aus der Benachrichtigung der zuletzt abgeschlossenen Transaktion. Da Commit-Benachrichtigungen während eines Knotenabsturzes möglicherweise nur einen Teil der Knoten im System erreichen, können sich die Knoten zuvor auf die höchste im System existierende Commit-ID austauschen und diese in dem neu generierten Token verwenden.

4.6.7 Tokenduplikate

Ein weitere Problematik ergibt sich bei einem duplizierten Token. Dieser Fall tritt ein, falls ein Knoten ein neues Token generiert, aber seine Annahme über ein verlorengegangenes Token nicht zutreffend war, weil beispielsweise der tokenbesitzende Knoten nur zeitweilig im Netzwerk nicht erreichbar war oder nur sehr langsam reagiert hat. Mehrfache Token können keine strenge Serialisierung von Transaktionen mehr gewährleisten, da jeder Tokenbesitzer berechtigt ist, seine Transaktion abzuschließen. Schließen mehrere Knoten gleichzeitig Transaktionen aufgrund mehrfach vorhandener Token ab, ist die Datenkonsistenz gewährleistet, solange die Transaktionen auf unterschiedliche Objekte und nicht veraltete Objektversionen zugreifen. Stehen überlappende Transaktionen, die ihren Commit über unterschiedliche Token abwickeln, dagegen miteinander in Konflikt, verletzt dies die Datenkonsistenz. Aufgrund der Nebenläufigkeit in verteilten Systemen können Knoten in diesem Fall nicht jederzeit eindeutig Transaktionskonflikte feststellen. Dies ist beispielsweise der Fall, sofern zwei Knoten die Commit-Benachrichtigung über ihren Commit gleichzeitig verschicken. Wenn die Knoten die Commit-Nachricht des jeweils anderen Knoten erhalten, sind die damit verbundenen Transaktionen nicht mehr rückabwickelbar, da sie den Konflikt erst nach dem Commit erkennen können. Daher darf nur ein gemeinsames Token im System existieren, und der tokengenerierende Knoten muß alle anderen Knoten informieren, daß das vorherige Token ungültig ist. Erst danach darf er das neue Token weitergeben.

Knoten können mehrere vorhandene Token im System anhand der mitgeführten Commit-ID erkennen. Erhält ein Knoten ein Token mit einer niedrigeren Commit-ID als der dem Knoten zuletzt bekannte Commit aufweist, ist von einem mehrfach existierenden Token auszugehen. Diese Erkennungsweise ist jedoch nicht zuverlässig. Damit Knoten ein veraltetes und neues Token exakt voneinander unterscheiden können, benötigt jedes neu generierte Token eine eindeutige ID. Andere Knoten, welche Commits über das für ungültig deklarierte Token erhalten, müssen diese ablehnen und dies dem sendenden Peer mitteilen. Der mit dem ungültigen Token abschließende Knoten muß seine lokale Transaktion also rückabwickeln. Wenn dies wie im Fall der First-Wins-Strategie aufgrund einer nicht erlaubten aber in seinem lokalen Kontext korrekt durchgeführten Validierung nicht mehr möglich ist, muß er gegebenenfalls alle seine mittlerweile inkonsistenten Daten verwerfen und dem System neu beitreten. Bei einem koordinierten Tokenaustausch existieren im Falle eines duplizierten Tokens ebenso zwei Koordinatoren. Der Koordinator des für ungültig deklarierten Tokens muß seine Rolle in diesem Fall aufgeben.

Ebenso kann ein Knoten auch im Falle einer Netzpartitionierung der falschen Annahme eines verlorengegangenen Tokens unterliegen. Bei einer Netzpartitionierung läßt sich ein dupliziertes Token dagegen nur sehr schwer aus dem System entfernen. Nachdem ein Knoten ein neues Token generiert hat, existieren mehrere unabhängig voneinander arbeitende Systeme

für die Dauer der Netzpartitionierung. Wenn sich die Teilnetze wieder miteinander verbinden, stehen im Gesamtsystem mehrere Token zur Verfügung und verhindern so eine strenge Commit-Serialisierung. Dies kann zur Dateninkonsistenz führen. Jedoch kann die Dateninkonsistenz auch schon während der Netzpartitionierung entstehen, da die einzelnen Teilsysteme für sich ein gültiges Commit-Token besitzen und weitere Transaktionen festschreiben können. Schließen die Teilsysteme Transaktionen ab, die Änderungen auf denselben Objekten durchführen, kommt es im Hinblick des Gesamtsystems zur Dateninkonsistenz. Diese läßt sich nicht mehr auflösen, da Knoten ihre bereits erfolgreich abgeschlossenen Transaktionen nicht mehr rückabwickeln können. Liegt bereits eine Dateninkonsistenz vor, oder läßt sich die Situation eines mehrfachen Tokens nicht fehlerfrei auflösen, muß das System gemäß des Fail-Stop-Fehlermodells anhalten und neu starten beziehungsweise zum letzten Checkpoint zurückkehren (siehe Kapitel 4.6.4).

4.7 Verwandte Arbeiten

4.7.1 DSTM2

DSTM2 (Dynamic Software Transactional Memory 2) [47, 48] ist ein objektbasierter STM (Software Transactional Memory) für Java und Multiprozessor-Architekturen mit einem gemeinsamen Speicher. DSTM2 unterstützt über eine Schnittstelle die Implementierung unterschiedlicher Synchronisierungsverfahren (Factories), wobei zwei Implementierungen bereits enthalten sind. Der STM benötigt keine Sperren, die Zugriffserkennung erfolgt stattdessen über einen *Contention Manager*, welcher Konflikte zwischen Transaktionen auflöst. Transaktionen sind wie in dieser Arbeit nicht in Größe, Laufzeit und Speicher beschränkt. DSTM2 hat aber im Vergleich zu dieser Arbeit eine andere Zielsetzung, da es nur für Multiprozessorsysteme und nicht für die Verwendung in Cluster- und Grid-Umgebungen konzipiert ist. DSTM2 benötigt deswegen auch keine verteilte Replikation und Transaktionsserialisierung.

4.7.2 DiSTM

Bei *DiSTM (Distributed Software Transactional Memory)* [57] handelt es sich um ein STM-Programmiergerüst für Clustersysteme, welches auf DSTM2 aufsetzt. Es implementiert drei unterschiedliche Verfahren für die Validierung von Transaktionen. Ein dezentrales Verfahren auf Basis von TCC [46] und zwei zentralisierte Ansätze mit Leases (siehe Kapitel 5.7.1).

Im Gegensatz zu TCC, welches auf cache-kohärente Architekturen ausgelegt ist, verwendet das auf TCC basierende Protokoll für die Serialisierung von Transaktionen ein Ticket-System. Knoten müssen vor einem Commit von einem Masterknoten ein Ticket mit einer globalen Serialisierungsnummer (logischer Zeitstempel) anfordern. Anschließend schickt der Knoten seine Lese- und Schreibmenge mittels eines Broadcasts an alle anderen Knoten, auf denen folglich eine Konfliktprüfung stattfindet. Befindet sich auf einem Knoten mindestens eine hinsichtlich des Zeitstempels ältere Transaktion, die konfliktbehaftet ist, muß die Transaktion, die ihre Lese- und Schreibmenge verschickt hat, abbrechen. Haben alle konfliktbehafteten Transaktionen eines Knotens einen größeren Zeitstempel, müssen diese dagegen abbrechen.

Das hier verwendete Protokoll generiert über viele Broadcast-Nachrichten und einer Aktualisierungskohärenz viel Netzwerkverkehr. Je nach den Zugriffsmustern der Transaktionen sind die Datenaktualisierungen auf den Arbeitsknoten teilweise überflüssig. Das P2P-Protokoll dieser Arbeit kommt aufgrund seiner First-Wins-Strategie mit weniger Netzwerkkommunikation aus, da es Schreibmengen von Transaktionen nur dann verschickt, wenn die Transaktionen erfolgreich abschließen. Zudem benötigt das P2P-Protokoll in dieser Arbeit für den Commit auch

keinen Masterknoten wie in DiSTM, da der Commit vollständig mittels P2P-Kommunikation erfolgt. Eine Ausnahme bildet hier das koordinierte Tokenverfahren, wobei der Tokenaustausch auch vollständig P2P-basiert erfolgen kann. DiSTM verfolgt einen Kompromiß, der eine niedrigere Netzlatenz gegen eine höheren Bandbreitenverbrauch eintauscht. Dies ist in reinen Clustersystemen mit Hochgeschwindigkeitsnetzen vertretbar, aber nicht wenn die Transaktionssynchronisierung auch in Weitverkehrsnetzen oder Netzwerken mit unterschiedlich hoher Bandbreite und Latenzen wie in Grid-Umgebungen stattfindet. Mittels weiterer Optimierungen wie lokale Commits (siehe Kapitel 6.2) erlaubt diese Arbeit, Transaktionen ohne Netzwerkkommunikation abzuschließen. Kaskadierte Transaktionen erlauben zudem, Transaktionen nebenläufig zu validieren und abzuschließen, so daß bei höheren Latenzen in Weitverkehrsnetzen kein Flaschenhals entsteht (siehe Kapitel 6.3).

4.7.3 Plurix

Plurix [92] ist ein verteiltes java-basiertes Betriebssystem für PC-Cluster als Single-System-Image (SSI). Daher erscheint ein Cluster gegenüber Anwendungen durch Hardwareabstraktion als ein großer virtueller Rechner. Plurix verwendet wie in dieser Arbeit einen verteilten transaktionalen Speicher als Objektsystem. Objekte sind hierbei aufgrund der Programmiersprache stets typsicher. Die Serialisierung von Transaktionen erfolgt in Plurix anhand eines Tokens. Zusammen mit einem 2-Phasen-Commit-Protokoll erfolgt so eine global geordnete Synchronisierung von Transaktionen im Netzwerk [104]. Das P2P-Protokoll in dieser Arbeit verwendet ebenfalls ein Token für die Transaktionsserialisierung.

Plurix und diese Arbeit verfolgen jedoch eine unterschiedliche Zielsetzung. Während ersteres ein von Beginn an typsicheres transaktionales Betriebssystem ist, ist das Ziel dieser Arbeit, einen verteilten transaktionalen Speicher in bestehende Linux-Betriebssysteme zu integrieren und verteilten Anwendungen zur Verfügung zu stellen. Da Linux-Systeme ursprünglich nicht auf verteilte Speichersysteme ausgelegt sind, lassen sich diese nicht ohne weiteres vollständig in einem gemeinsamen verteilten Speicher transferieren. Daher verwenden Anwendungen unter Linux nur einen Teil ihres Speichers als verteilten transaktionalen Speicher. Nachfolger des Plurix-Betriebssystems ist das System *RainbowOS* [59].

4.7.4 Cluster-STM

Bei *Cluster-STM* [18] handelt es sich um einen gemeinsamen verteilten transaktionalen Speicher für Cluster. Es verbindet softwareverteilte transaktionale Speicher und das Programmiermodell eines *Partitioned Global Address Space (PGAS)*. PGAS unterscheidet zwischen einem für jeden Cluster-Knoten lokalen Speicher und einem gemeinsamen Speicher, wobei jeder Knoten einen Teil seines lokalen Speichers zum globalen Speicher beisteuert. Alle Knoten können auf jeden Teil des gemeinsamen Speichers zugreifen und so mit anderen Knoten kommunizieren. Das PGAS-Modell wird von unterschiedlichen Programmiersprachen implementiert, unter anderem von *Unified Parallel C (UPC)* [33]. Diese stellen es indessen Anwendungen zur Verfügung, indem es Daten im globalen Speicher für Knoten übers Netzwerk beschafft, sofern diese nicht in der eigens beigesteuerten Partition des globalen Speichers liegen. Änderungen im globalen Speicher erfolgen analog.

Cluster-STM erweitert das Programmiermodell um Transaktionen und erlaubt Anwendungen so komplexere Operationen atomar auf dem verteilten Speicher durchzuführen und setzt Techniken ein, um möglichst die Lokalität auf den einzelnen Knoten auszunutzen und Metadaten aggregiert mit anderen Daten zu übertragen. So reduziert es kostspielige Datenübertragungen. Cluster-STM unterstützt unterschiedliche Transaktionsdesigns basierend auf Sperren sowie Aufzeichnen von Schreibänderungen und Schattenkopien. Das P2P-Protokoll arbeitet dagegen

nur mit optimistischer Synchronisierung und Schattenkopien. Zudem arbeitet dieser gemeinsame verteilte transaktionale Speicher mit lokal zwischengespeicherten Objektreplikaten, das ist bei Cluster-STM nicht der Fall. Dort ist das Zwischenspeichern transaktionaler Speicherinhalte Aufgabe der Anwendungen.

Vergleichbar mit dieser Arbeit ist daher nur die Transaktionssynchronisierung über Versionierung gelesener Objekte und Schreibsperren. Beim Commit erfolgt zunächst eine bestätigte Benachrichtigung aller anderen Knoten, ähnlich dem in Kapitel 4.3.1 beschriebenen Commit-Verfahren mit Bestätigungsnachrichten. Anschließend informiert der Knoten alle anderen über die Freigabe seiner Schreibsperren. Das P2P-Verfahren kommt dagegen mit weniger Netzkommunikation aus, da es keine Sperren anfordern muß und zudem über die tokenbasierte Synchronisierung der Commit-IDs auf Bestätigungsnachrichten verzichten kann. Details über Kohärenzprotokolle sind nicht genau dokumentiert, weswegen ein vollständiger Vergleich mit dem hier diskutierten Commit-Protokoll schwierig ist.

4.8 Zusammenfassung

Dieses Kapitel hat Konzepte aus software- und hardwaretransaktionalen Speichern für die einfache Entwicklung verteilter Anwendungen mit verteilten Systemen kombiniert. Der wesentliche Unterschied besteht beim gemeinsamen verteilten transaktionalen Speicher in der unterliegenden NoRMA-Speicherarchitektur (siehe Kapitel 2.1). Diese macht eine explizite Synchronisierung lokaler Speicher autonomer Rechner über das Netzwerk erforderlich, um den Prozessen verteilter Anwendungen die Illusion eines gemeinsamen transaktionalen Speichers zu geben. Dieses Kapitel hat Basistechnologien für die Synchronisierung replizierter Transaktionsobjekte evaluiert, wobei der Schwerpunkt auf einer effizienten Netzwerkkommunikation liegt, da diese im Vergleich zur Transaktionsausführung oftmals viel mehr Zeit in Anspruch nimmt.

Die Synchronisierung transaktionaler Speicherinhalte erfordert systemweit eine totale Ordnung der abzuschließenden Transaktionen, die ein Commit-Token gewährleistet. Die Erweiterung des Tokens auf einen globalen Zeitstempel als Commit-ID für die Transaktionen reduziert die Netzwerkkommunikation, da dieser Bestätigungsnachrichten für Commit-Benachrichtigungen entbehrlich macht und somit die Leistungsfähigkeit des verteilten transaktionalen Speichers verbessert.

Weiterhin hat das Kapitel für eine effiziente Serialisierung von Transaktionen zwei unterschiedliche Tokenverfahren untersucht. Beim P2P-Tokenverfahren reichen sich Knoten das Token untereinander direkt weiter. Hierbei kann eine erhöhte Netzkommunikation entstehen, falls Tokenanfragen dem Token bei dessen Weitergabe von einem zum nächsten Knoten folgen. Das Token kompensiert dies mittels einer internen Warteschlange, die Tokenanfragen aufnimmt. Das koordinierte Tokenverfahren sammelt Tokenanfragen in einer lokalen Warteschlange, hier entsteht keine erhöhte Netzkommunikation durch dem Token nachfolgende Anfragen, aber stattdessen wegen der Rückgabe des Tokens an den Koordinator. Hierzu integriert das koordinierte Tokenverfahren genauso wie der P2P-Ansatz eine interne Warteschlange, was den zentralisierten Ansatz um P2P-Funktionalität erweitert. Der Koordinator steuert über die Warteschlange die direkte Tokenweitergabe über mehrere Knoten, anstatt es nach der Freigabe von einem Knoten direkt zurückzufordern. Dies reduziert die Netzwerkkommunikation und Zustellungzeit des Tokens zwischen Knoten, was wiederum die Leistung des Systems steigert.

Bei Anwendung des P2P-Tokenverfahrens können gegebenenfalls mehr Tokenanfragen anfallen, als die interne Warteschlange wegen ihrer begrenzten Größe aufnehmen kann. Knoten müßten die überzähligen Anfragen dem Token bei dessen Weitergabe explizit hinterherschicken, was wiederum zu einer unnötigen Netzbelastung führt. Dieses Problem löst die ef-

fiziente Tokenvorhersage, bei der Knoten ihre Anfragen nach dem Token an Knoten schicken, die selbst auf das Token warten.

Andere Systeme wie DiSTM und Plurix verwenden entweder kein oder kein so effizientes Tokenaustauschverfahren, wie diese Arbeit diskutiert hat. In DiSTM muß jede abzuschließende Transaktion explizit ein Ticket von einem Masterknoten anfordern. Plurix tauscht das Token direkt auf Anforderung zwischen Knoten aus. Die in diesem Kapitel vorgestellten Tokenverfahren verursachen im Vergleich aufgrund der internen Warteschlange und der Tokenvorhersage weniger Netzwerkkommunikation, da Knoten Anfragen nach dem Token nur selten explizit an andere Knoten weiterleiten müssen. Zudem darf ein Knoten im Vergleich zum Ticketsystem pro ausgetauschtem Token auch mehr als eine Transaktion abschließen.

5 Commit-Protokolle für hierarchisch strukturierte Overlay-Netze

Kapitel 4 hat den Commit von Transaktionen in einem *vollvermaschten* unstrukturierten P2P-Netzwerk behandelt. In dieser Netzstruktur kommunizieren alle Peers untereinander direkt, unabhängig von ihrer geographischen Lage, Netzgüte und gemeinsamen semantischen Interessen. Die unstrukturierte Kommunikation kann dazu führen, daß das Commit-Protokoll und die Suche nach Objektreplikaten bei vielen Knoten nicht mehr gut skalieren. Dies erfolgt bei steigender Knotenanzahl zum einen durch ein stark ansteigendes Nachrichtenaufkommen durch das Commit-Protokoll. Weiterhin verursacht eine hohe Anzahl von Knoten eine höhere Verzögerung durch die Tokenweitergabe. Dies ist auch der Fall, wenn zu synchronisierende Transaktionen disjunkte Objektmengen (Lese- und Schreibmengen) aufweisen und deshalb keinen Konflikt verursachen können. Die Leistungsfähigkeit des Commit-Protokolls hängt daher maßgeblich von der Anzahl beteiligter Knoten, der Netzbandbreite und der Netzwerklatenz zwischen den Knoten ab. Durch eine gegenseitige Abstimmung von Protokoll und Netzstruktur lassen sich Synergieeffekte erzielen, die zu einer Leistungssteigerung und einer besseren Skalierbarkeit führen.

Die physische Struktur eines Netzwerks ist beispielsweise durch die geographische Lage der Knoten, Netzwerkverbindungen und Vermittlungspunkte (Switches und Router) fest bestimmt. Die Kommunikationswege verteilter Anwendungen decken sich zumeist nicht unbedingt mit denen des physischen Netzwerks. So spielt je nach Anwendung die geographische Lage der Knoten zueinander eine untergeordnete Rolle. Für Anwendungen stehen oftmals semantische Aspekte im Vordergrund. Diese können beispielsweise gemeinsame Interessen wie der häufige Zugriff auf gemeinsame Speicherbereiche umfassen. Ebenso können sich Knoten auch über die Bereitstellung oder Übernahme bestimmter Funktionen auszeichnen. Da die Netzkommunikation einen wesentlichen zeitlichen Kostenfaktor verteilter Systeme darstellt, gilt es diese möglichst zu minimieren und sofern es für die Anwendung möglich ist, Lokalität auszunutzen. Daher kann die Strukturierung eines Netzwerks auch logisch auf den Anforderungen einer Anwendung erfolgen. Bei der logischen Strukturierung ist es sinnvoll, neben den Anforderungen von Anwendungen auch die physische Netzstruktur mit zu berücksichtigen, da diese die grundlegenden Kommunikationseigenschaften wie Bandbreite und Latenz vorgibt. Eine logische Strukturierung des Netzwerks über der physischen Struktur führt zu einem Overlay-Netzwerk.

Damit sich der Einsatz eines Overlay-Netzwerks lohnt, muß für die Anwendungen ein Zusatznutzen entstehen. Der Aufbau eines Overlay-Netzes verursacht in erster Hinsicht Nachteile, da es einen zusätzlichen Mehraufwand verursacht und die physischen Limitierungen bezüglich Latenz, Bandbreite und Routingstationen nicht außer Kraft setzen kann. Zusammenspiel von Anwendungsverhalten und Overlay-Netzwerk muß den Aufwand, der durch das Overlay-Netzwerk und dessen Verwaltung entsteht, kompensieren. Damit die Anwendungen dieses Ziel erreichen können, müssen sie weiterzuleitende Informationen zwischenspeichern oder mit eigenen Daten verknüpfen, um daraus neue Erkenntnisse ableiten zu können. Ferner müssen sie Informationen replizieren können, um Kommunikationswege für den Austausch von Informationen zu minimieren und die Verteilung beziehungsweise Routing von Nachrichten anhand anwendungssemantischer Aspekte definieren. Zum Beispiel können Knoten Informationen nur auf bestimmte Regionen des Overlay-Netzes verteilen.

5.1 Peer-to-Peer-Netzarchitekturen

Die Abgrenzung aktueller P2P-Systeme erfordert eine etwas differenziertere Betrachtung des *Peer-to-Peer*-Begriffs, da dieser auch schon vor deren Entwicklung für jegliche Dienste mit integrierter Client-/Serverfunktionalität gebräuchlich war, ohne explizit Bezug auf die Netzstruktur zu nehmen. Dienste wie die Dateifreigabe des *Windows-Netzwerks* basieren auf P2P-Kommunikation, da die Rechner Dateien anbieten (Server) und ebenfalls auf fremde Dateien zugreifen (Client) können und in lokalen Netzwerken aufgrund von Broadcast-Kommunikation zudem auch ohne zentralen Koordinator auskommen. Ebenso könnten auch Mail-Transfer-Agents (MTA) als Peers eines P2P-Systems betrachtet werden, da sie als Server E-Mails von Benutzern entgegennehmen (Server), anschließend aber selbst als Client fungieren, um die E-Mails an einen anderen MTA weiterzuleiten. Das gleiche gilt für Usenet-Server, die Nachrichten an die Nutzer verteilen und entgegennehmen, sich aber zusätzlich auch untereinander durch den Abgleich von Nachrichten synchronisieren.

Heutige P2P-Systeme definieren sich neben der reinen Client-/Server-Funktionalität auch über die Selbstorganisation, dynamischen Bei- und Austritt aus dem Netzwerk, Suchdienste für Ressourcen und Routing von Nachrichten. Größere Verbreitung und Popularität haben P2P-Systeme durch *Napster* [16], einer Musiktauschbörse für das Internet gefunden. Mit dem Beginn dessen Verbreitung haben sich P2P-Systeme weiterentwickelt und lassen sich aktuell in zwei Klassen einteilen: strukturierte und unstrukturierte P2P-Systeme. Weiterhin lassen sich die unstrukturierten Netze in zentralisierte, dezentralisierte oder hybride Netze unterteilen. Die Peers eines P2P-Netzwerks bilden mit ihren untereinander bestehenden Verbindungen zusammen ein Overlay-Netzwerk (siehe Kapitel 5.1.1). Napster selbst basiert auf einem unstrukturierten zentralisierten Ansatz. Ein zentraler Server im Internet beantwortet Suchanfragen nach Musikdateien, indem er anfragenden Peers mitteilt, an welchem Ort (fremder Peer) eine Musikdatei hinterlegt ist. Weiterhin melden die Peers eine Liste mit eigenen Musikdateien an den zentralen Suchserver. Die Peers tauschen die Musikdateien dagegen direkt untereinander aus, ohne auf den zentralen Server angewiesen zu sein [91].

5.1.1 Overlay-Netzwerk

Ein Overlay-Netzwerk bezeichnet eine logische Netzwerkstruktur, die auf einer physischen Netzstruktur aufsetzt. Das logische Netzwerk besteht aus allen Kommunikationsbeziehungen (Verbindungen) zwischen den Teilnehmern, welches sich zumeist von der physischen Struktur unterscheidet. Teilnehmer übernehmen in dem logischen Netzwerk genauso Routingfunktionalitäten wie (dedizierte) Router in physischen Netzen. Die logische Netzstruktur basiert überwiegend auf semantischen Eigenschaften einer Anwendung, während die physische Netzstruktur von geografischen Gegebenheiten abhängt. Beispielsweise können Peers mit ähnlichen Interessen eine direkte logische Kommunikationsbeziehung unterhalten und sich gegebenenfalls auch gruppieren, während die Peers geografisch weit voneinander entfernt liegen und das physischen Netz Nachrichten mehrfach routet.

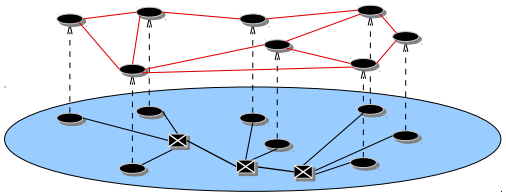


Abbildung 5.1: Overlay-Netzwerk.

Abbildung 5.1 zeigt beispielhaft das Zusammenspiel von einem physischen und einem logischen Netzwerk. Die Kreise bezeichnen die einzelnen Peers des Netzwerk, die über physische Netzwerkverbindungen miteinander verbunden sind (schwarze Kanten). Die Quadrate stellen Router des physischen Netzwerkes dar. Die rot hervorgehobenen Kanten sind direkte Kommunikationsbeziehungen zwischen den Knoten und formen (anhand semantischer Parameter) das logische Netzwerk. Die direkten Kommunikationsbeziehungen im Overlay-Netzwerk erfolgen also über Routing und Verbindungen des physischen Netzwerks.

5.1.2 Unstrukturierte Peer-to-Peer-Netzwerke

Unstrukturierte P2P-Netzwerke sind nicht nach semantischen Anwendungskriterien aufgebaut. Die Peers besitzen kein Lokationsbewußtsein und sind daher beliebig im Netzwerk angeordnet und miteinander verbunden. Jeder Peer unterhält Verbindungen zu einer bestimmten Anzahl von Nachbarpeers. Kann ein Peer andere Peers aufgrund fehlender Nachbarschaftsbeziehungen nicht direkt erreichen, so leiten Nachbarn die Nachrichten ihrerseits an ihre Nachbarn weiter, welche die Nachrichten wiederum auf dieselbe Art weiterleiten, bis die Nachrichten ihr Ziel erreichen. Bei dieser Kommunikationsart vervielfachen Peers weiterzuleitende Nachrichten, was letztendlich zu einer Flutung des gesamten Netzwerks führt. Da alle Peers, die nicht Empfänger einer Nachricht sind, die Nachricht weiterleiten, kommt es ohne weitere Einschränkungen der Kommunikation zu einer immer weiteren Vervielfachung derselben Nachricht. Diese wiederum belasten das Netzwerk sehr stark. Daher muß die Weiterleitung nach endlich vielen Schritten abbrechen. Hierzu eignet sich ein vergleichbarer Ansatz wie beim Internetprotokoll (IP) [79].

IP ist ein routingfähiges Netzwerkprotokoll, welches die gesamte Internetkommunikation abwickelt. Router verknüpfen dabei unterschiedliche Netzsegmente miteinander. Ist ein Paket an ein fremdes Netz adressiert, so leitet es ein für das Zielnetz zuständiger Router weiter. Diese Prozedur wiederholt sich so lange, bis das Paket sein Ziel erreicht. Pakete erreichen im Regelfall ihren Empfänger, aber im Falle von Routingfehlern (zum Beispiel Schleifen) können Pakete endlos weitergeleitet werden und so dauerhaft Ressourcen beanspruchen. Zur Vermeidung solcher Probleme enthält der Header eines jeden IP-Pakets einen Zähler, der die maximal erlaubten Routingstationen angibt (Time-to-Live, TTL). Durchläuft ein Paket einen Router (Hop), so dekrementiert der Router den Zähler vor der Weiterleitung. Ist der TTL-Wert null, so verwirft er das Paket anstatt es weiterzuleiten. Der TTL-Wert ist beim Nachrichtenversand so zu wählen, daß ein Paket sein Ziel praktisch erreichen kann.

Zur Begrenzung der Nachrichtenweiterleitung durch Flutung verwenden P2P-Systeme (zum Beispiel *Gnutella* [1]) einen TTL-Zähler in vergleichbarer Weise. Dies verhindert eine Überlastung und endlose Weiterleitung von Paketen. In diesem Fall ist es aber ebenso möglich, daß eine Nachricht seinen Empfänger nicht erreicht, falls der initiale TTL-Wert zu niedrig ist. Gegenüber IP ist diese Problematik in unstrukturierten P2P-Netzen häufiger anzutreffen, da aufgrund der beliebigen Anordnung von Peers im Netzwerk auch kein strukturiertes Routing stattfindet und Routen über eine Vielzahl von Hops verlaufen können. Dies gilt vor allem dann, wenn der Verbindungsgraph ausgeartet ist. Ein ausgearteter Verbindungsgraph führt dazu, daß die Kommunikation zwischen manchen Peers zu einem Routing über viele andere Peers führt. Daher kann dieser Ansatz nicht garantieren, daß eine Nachricht immer seinen Empfänger erreicht, obwohl dieser im Netz präsent ist. Abbildung 5.2 zeigt eine ausgeartete Verbindungstopologie, in der sowohl kurze als auch lange Verbindungswege existieren. Eine Kommunikation der beiden Knoten *1* und *2* involviert viele Peers, um deren Nachrichten weiterzuleiten, während ein Nachrichtenaustausch zwischen Knoten *3* und *4* nur einen weiteren Peer erfordert. Ist das Maß für die maximal erlaubten Weiterleitungen von Nachrichten geringer als durch die Verbindung zwischen Knoten *1* und *2* spezifiziert, können sie nicht miteinander kommunizieren, da ein zwischenliegender Peer die Nachricht vor der endgültigen Auslieferung verwirft.

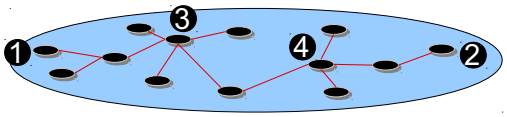


Abbildung 5.2: Unstrukturiertes P2P-Overlay.

Zentralisierte Peer-to-Peer-Systeme

Zentralisierte P2P-Netzwerke zeichnen sich durch eine zentrale Instanz aus, die einen Index bezüglich der Suche und Anfragen von Daten verwaltet. Ein Indexserver verwaltet hierzu einen Schlüsselindex, welcher für eine Anfrage das Ziel als Antwort zurückliefert. Am Beispiel von Napster würde ein Peer eine Anfrage nach einer bestimmten Datei an den Indexserver richten. Dieser antwortet daraufhin, daß die Datei bei Peer *P* zu finden ist, vorausgesetzt, die Ressource ist überhaupt im P2P-System vorhanden. Anschließend fordert der Peer die Datei bei *P* an. Nachteil dieses Verfahrens ist, daß ein einzelner Indexserver wie beim Client/Server-Modell (siehe Kapitel 4.1.1) bei einem Ausfall das gesamte P2P-Netz lahmlegt, da keine Suchanfragen mehr möglich sind. Daher sollten die Indexserver in diesem System redundant ausgelegt sein.

Reine Peer-to-Peer-Systeme (dezentralisiert)

Reine P2P-Systeme kommen dagegen ohne zentralen Indexserver aus, sind also dezentral angelegt. Jeder Knoten verwaltet seinen eigenen Index über die von ihm angebotenen Daten. Wie bereits in Kapitel 5.1.2 erwähnt, erfolgt die Nachrichtenvermittlung in unstrukturierten P2P-Systemen durch Flutung. Bei Suchanfragen erfolgt bei diesem Ansatz ebenso eine Netzflutung. Erhält ein Peer eine Suchanfrage, schaut er zunächst in seinem lokalen Index nach, ob er die angeforderten Daten zur Verfügung stellen kann. Ist dies der Fall, so schickt er eine Antwort an den anfragenden Peer zurück. Er braucht die Anfrage dann nicht mehr an andere Knoten weiterzuleiten. Kann er die Anfrage allerdings nicht bedienen, so leitet er die Anfrage an alle ihm bekannten Nachbarn weiter. Hierbei kann es aufgrund der Flutung vorkommen, daß mehrere Peers die gesuchten Daten besitzen und der anfragende Knoten die gewünschte Antwort möglicherweise mehrfach erhält. Hier findet nicht nur eine Netzbelastung durch die eventuelle Anfragevervielfachung statt, sondern auch durch ein unter Umständen mehrfaches beantworten derselben Anfrage. Ein zu Napster vergleichbarer Dienst *Gnutella* zum Austauschen von Dateien setzt auf eine dezentralisierte P2P-Kommunikation [91].

Hybride Peer-to-Peer-Systeme

Hybride P2P-Systeme sind eine Kombination des zentralen und dezentralen Ansatzes. Ausgewählte Peers fungieren als Vermittler in dem System und werden als Superpeers bezeichnet. Die üblichen Peers kommunizieren dagegen nur mit einem ihnen während der Beitrittsphase zugeordneten Superpeer. Weiterhin führen die Superpeers einen Index der ihnen untergeordneten Peers. Bei einer Anfrage wendet sich ein Peer an seinen Superpeer. Dieser prüft zunächst, ob ein Peer in derselben Gruppe die Anfrage beantworten kann. Ist dies der Fall, sendet er eine Antwort an den anfragenden Peer zurück. Beide Peers können die angefragten Daten anschließend direkt austauschen. Ansonsten flutet der Superpeer die Anfrage durch das Netzwerk zu allen anderen Superpeers. Diese können beantworten, ob die Daten in ihrer Gruppe vorhanden sind und die Anfrage entweder verwerfen oder eine Antwort zurückliefern. Der Datenaustausch erfolgt dann in gleicher Weise wie beim zentralisierten Ansatz. Ein Beispiel eines hybriden P2P-Systems ist *Fasttrack* [63].

Hybride Systeme verteilen die Last, die ein Indexserver im zentralisierten Ansatz trägt, auf mehrere Rechner. So vermindern diese Systeme eine mögliche Überlastung einzelner Indexserver. Sucht ein Peer Daten, ist ebenso keine Flutung des gesamten Netzwerks wie beim dezentralisierten Ansatz notwendig. Stattdessen findet die Flutung nur auf Ebene der Superpeers statt. Dennoch besitzt das hybride P2P-System weiterhin die Nachteile beider P2P-Netzwerke. Zum einen findet zwischen den Superpeers weiterhin eine Flutung des Netzwerkes statt, was sich aufgrund von Caching auch in mehrfachen Antworten auf eine Anfrage äußern kann. Der Ausfall eines Superpeers koppelt die ihm zugewiesenen Clients teilweise vom Netzwerk ab. Zwar können diese Peers weiterhin Daten austauschen, jedoch keine Suchanfragen mehr stellen beziehungsweise durch ihren Superpeer beantworten lassen.

5.1.3 Strukturierte Peer-to-Peer-Netzwerke

Im Unterschied zu den unstrukturierten P2P-Netzwerken können sich die Peers in den strukturierten Netzwerken selbständig organisieren. Die Peers sammeln dazu Informationen über verschiedene Parameter im Netzwerk und bilden damit ein Lokationsbewußtsein. Anhand dieser Parameter und fest definierter Kriterien bilden die Peers eine Overlay-Struktur. Parameter können beispielsweise Netzwerklatenz und -bandbreite, Systemleistung oder aber auch semantische Parameter einer Anwendung sein. Zum Beispiel können Peers mit gleichen semantischen Interessen oder mit einer geringen Latenz zueinander eine gemeinsame Gruppe im Overlay-Netzwerk bilden. Abbildung 5.3 zeigt beispielhaft eine Gruppierung von Peers (Wolken) in einem strukturierten Overlay-Netzwerk anhand semantischer Parameter. Ändern sich Parameter während der Laufzeit, so kann sich das Overlay-Netz dynamisch rekonfigurieren.

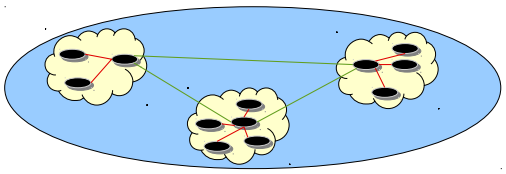


Abbildung 5.3: Strukturiertes P2P-Overlay.

Ein weiterer Aspekt ist die strukturierte Ablage von Daten beziehungsweise dessen Indexierung. Dies erlaubt eine effiziente Suche, ohne daß eine Flutung wie bei unstrukturierten P2P-Systemen notwendig ist. Die Datenindexierung erfolgt zumeist über verteilte Hashtabellen, *engl. Distributed Hash Tables (DHT)*. Bei DHTs handelt es sich um dezentrale Datenstrukturen zum effizienten Auffinden von Daten in verteilten Systemen. Jeder Peer verwaltet einen Teil des Schlüsselraums. Ein Schlüssel ist ein eindeutiger Index auf ein Datum, welcher sich nach einem vorgegebenen Algorithmus berechnen läßt. Der Hintergrund ist eine schnelle Auffindung der gesuchten Daten (möglichst in logarithmischer Komplexität), eine gleichmäßige Verteilung der Suchanfragen auf alle Knoten und eine möglichst gleichmäßige Verteilung der Hashtabelle auf die Peers. Bekannte Vertreter von DHTs sind Chord [97], Pastry [89], Tapestry [107] and CAN [80]. Eine nähere Beleuchtung von DHTs zur Objektsuche findet sich in Kapitel 5.6.

5.2 Ultrapeer-Commit

Neben dem in Kapitel 4.2 besprochenem P2P-Protokoll definiert das Ultrapeer-Verfahren ein weiteres Commit-Protokoll [74]. Im Unterschied zu dem vollständig verteilten Verfahren stützt

es sich auf einen koordinatorbasierten Ansatz zur Transaktionsvalidierung. Peers, welche eine Transaktion abschließen möchten, senden eine Commit-Anfrage an einen Koordinator. Dieser validiert die Transaktion daraufhin und schickt entsprechend des Validierungsergebnisses eine Antwort zurück. Der Peer schreibt seine Transaktion daraufhin fest oder bricht sie ab. Der Ultrapeer (UP) kann entweder dediziert sein oder auch als gegenüber den anderen Peers gleichwertiger Peer selbst Transaktionen ausführen. Falls der UP selbst als Teilnehmer fungiert, unterliegt seine Transaktionsvalidierung selbst keiner Netzkommunikation, da er die Validierung lokal bei sich selbst durchführt. Lediglich im Falle eines erfolgreichen Commits erzeugen die Commit-Benachrichtigungen Netzkommunikation. Aufgrund dessen kann der UP einen höheren Transaktionsdurchsatz erzeugen, da andere Knoten in der Validierungsphase einer Verzögerung verursachenden Netzwerkkommunikation unterliegen. Daher ist er ohne Fairneßkontrolle gegenüber den anderen Peers bevorteilt. Da alle Commit-Anfragen bei einem Koordinator auflaufen, unterliegen sie dessen lokaler Zeit. Er selbst bestimmt die Serialisierungsreihenfolge aller eingehenden Commits. Das UP-Protokoll kommt daher ohne Serialisierungstoken aus, weswegen eine Suche oder Weiterleitung desselbigen nicht notwendig ist. Bei der *First-Wins*-Strategie verarbeitet der UP die Commit-Anfragen entsprechend der Reihenfolge seines Empfangs. Der UP kann aber einfach von der First-Wins-Strategie abweichen und Commit-Anfragen bevorzugt behandeln, indem er diese in der Warteschlange umsortiert.

Die Commits erfolgen bei diesem Protokoll zunächst in gleicher Weise wie bei dem P2P-Protokoll aus Kapitel 4.2. Der UP verschickt die Schreibmenge einer Commit-Anfrage an alle beteiligten außer dem anfragenden Knoten, die ihrerseits dann eine Vorwärtsvalidierung gegen lokal laufende Transaktionen vornehmen und diese im Konfliktfall abbrechen. Genauso wie beim optimierten P2P-Protokoll brauchen die Knoten den Empfang von Commit-Nachrichten nicht zu bestätigen. Während das P2P-Protokoll mithilfe des Tokens eine strenge Transaktionsserialisierung im Netzwerk garantiert, können sich beim UP-Protokoll Commit-Anfragen und Commit-Nachrichten anderer Transaktionen zeitlich überschneiden.

Nach der Beendigung einer Transaktion verschicken Knoten unmittelbar im Anschluß ihre Validierungsanfrage an den UP als Koordinator. Mehrere Anfragen unterschiedlicher Knoten treffen unabhängig voneinander auf dem UP ein. Nun kann es durch die fehlende Serialisierung zum Zeitpunkt des Versandes der Commit-Anfragen eine Situationen entstehen, in der Knoten eine Commit-Anfrage an den UP verschickt haben, aber anschließend ein Transaktionsabbruch durch die Commit-Benachrichtigung einer fremden Transaktion erfolgt. Dies ist beispielsweise der Fall, wenn ein Knoten eine Commit-Anfrage an den UP schickt, dieser aber noch eine andere Anfrage in der Warteschlange hat, die er zuvor abarbeitet. Im angenommenen Fall, die vorherige Transaktion sei konfliktfrei, führt dies zu einer Commit-Benachrichtigung an alle Knoten, die ihrerseits eine Vorwärtsvalidierung ausführen. Zeigt die Vorwärtsvalidierung auf dem Knoten, der seine Anfrage bereits versandt hat, einen Konflikt auf, so muß er seine Transaktion entsprechend der First-Wins-Strategie abbrechen (siehe Abbildung 5.4).

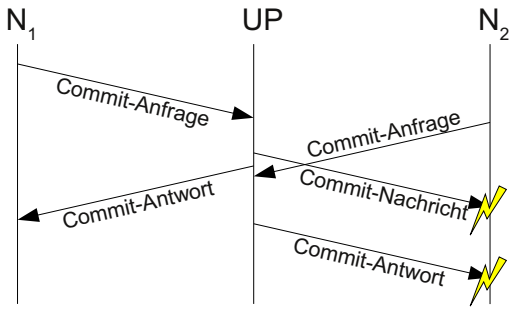


Abbildung 5.4: Überschneidung von Commit-Anfrage und Commit-Benachrichtigung.

Das P2P-Protokoll verhindert solche Situationen mithilfe des Tokens. Der Knoten mit der abzubrechenden Transaktion kann seine bereits verschickte Commit-Anfrage nicht mehr zurückziehen. Der UP muß also sicherstellen, die Schreibmengen solcher Commit-Anfragen nicht ungeprüft an andere Knoten zwecks dort auszuführender Vorwärtsvalidierung weiterzuleiten. Anderenfalls kann die Weiterleitung der Schreibmenge der abzubrechenden Transaktion die Datenkonsistenz verletzen und schlimmstenfalls nicht replizierte Objekte zerstören. Der UP muß demnach auch selbst Transaktionen auf Basis veralteter Objektreplikate zuverlässig erkennen können. Eine Validierung auf den Zielknoten ist unter Berücksichtigung der First-Wins-Strategie daher nicht ausreichend.

5.2.1 Commit-Serialisierung und zuverlässige Konflikterkennung

Damit eine sichere Konflikterkennung beim UP-Protokoll möglich ist, muß das Protokoll neben der Vorwärtsvalidierung auf den Zielknoten weitere Mechanismen integrieren, damit es zu keiner Konsistenzverletzung durch den Commit veralteter Transaktionen kommt. Allgemein hat ein Knoten Kenntnis über seine eigenen laufenden Transaktionen, aber nicht über nebenläufige noch nicht abgeschlossene Transaktionen anderer Knoten. Da eine zusätzliche Koordination möglicher konfliktverursachender Transaktionen und zugegriffener Objekte über das Netzwerk zu zeitaufwendig ist (siehe Kapitel 4.2.2), ist eine skalierbare Validierung allein auf Basis einer Vorwärtsvalidierung von Transaktionen nicht realisierbar. Insofern ist eine Erkennung solcher Transaktionen anhand der Commit-Anfragen auf dem UP anzustreben. Der UP ist im Vergleich zu den anderen Knoten der einzige Knoten, bei dem die Verarbeitung von Commit-Anfragen und Bekanntmachung erfolgreicher Commits eine totale Ordnung bilden. Dem UP ist es so möglich, jede Commit-Anfrage mit vorherigen erfolgreich abgeschlossenen Transaktionen zu vergleichen. Daher kommt neben der Serialisierung von Transaktionen auf dem UP nur eine weitere Konflikterkennung über eine Rückwärtsvalidierung in Betracht.

Vergleicht man den UP-basierten Ansatz mit der tokenbasierten Lösung des P2P-Protokolls, liegt zunächst eine Commit-Serialisierung auf Basis von Commit-IDs nahe. Mittels dieses Verfahrens kann der UP erkennen, ob eine abzuschließende Transaktion auf aktuellen Versionen von Objektreplikaten basiert und somit konfliktfrei ist.

5.2.2 Rückwärtsvalidierung mittels Commit-IDs

Bei diesem Ansatz vergibt der UP für jede erfolgreiche Validierung eine neue Commit-ID in aufsteigender Reihenfolge (logischer Zeitstempel) und schickt diese in der Commit-Benachrichtigung zusammen mit der Schreibmenge an die Peers. Beabsichtigt ein Peer eine eigene Transaktion abzuschließen, so schickt er seinerseits die ID der zuletzt verarbeiteten Commit-Benachrichtigung in der Commit-Anfrage mit. An der ID kann der UP erkennen, ob eine zu validierende Transaktion auf aktuellen Daten basiert. Dies ist genau dann der Fall, wenn eine Anfrage die Commit-ID der zuletzt durch den UP erfolgreich validierten Transaktion enthält. Der Umkehrschluß gilt dagegen nicht. Enthält eine Commit-Anfrage nicht die Commit-ID der zuletzt validierten Transaktion, folgt daraus nicht, daß die zu validierende Transaktion auf veralteten Daten basiert und somit einen Konflikt verursacht. In diesem Fall kommt es darauf an, ob die abzuschließende Transaktion Objekte gelesen hat, welche die Transaktionen der noch nicht verarbeiteten Commit-Benachrichtigungen geschrieben haben. Mittels der Commit-ID kann der UP einen Konflikt nicht eindeutig feststellen, sondern nur vermuten. Er müßte die Commit-Anfrage deshalb mit der Begründung noch nicht verarbeiteter Commit-Benachrichtigungen zurückweisen. Ein Peer müßte in diesem Fall warten, bis er die noch fehlenden Commit-Benachrichtigungen verarbeitet hat und seine Commit-Anfrage erneut an den

UP schicken, sofern die Vorwärtsvalidierung auf dem Peer zwischenzeitlich keinen Konflikt aufgezeigt hat.

Dieser Ansatz weist Ähnlichkeiten zur optimierten P2P-basierten Transaktionsserialisierung mittels eines gemeinsamen Tokens auf. Der UP weist Commit-Anfragen eventuell auch dann ab, wenn kein Konflikt vorliegt. Die Knoten müssen noch fehlende Commit-Benachrichtigungen verarbeiten. Dies kann gegebenenfalls zu einer kurzzeitigen Blockierungsphase führen, wenn die Benachrichtigungen verzögert eintreffen, wie dies auch beim P2P-Ansatz der Fall ist, falls Commit-Benachrichtigungen in vertauschter Reihenfolge eintreffen. Verhindern läßt sich dieser Umstand, indem der UP eine Historie mit den Schreibmengen über die erfolgreich validierten Transaktionen vorhält. Mittels dieser kann er dann zusätzlich eine Rückwärtsvalidierung ausführen, die eine genaue Konflikterkennung ermöglicht. Damit eine abzuschließende Transaktion T_c nach einer zuletzt verarbeiteten Commit-Benachrichtigung mit der ID v mit einer zuletzt auf dem UP erfolgreich validierten Transaktion mit der ID n als konfliktfrei gilt, muß folgende Abhängigkeit gelten, wobei T_x eine erfolgreich validierte Transaktion mit der ID x kennzeichnet:

$$\left(\bigcup_{x=v+1}^{n} WS_{T_x} \right) \cap RS_{T_c} = \emptyset \qquad c, n, v, x \in \mathbb{N} \tag{5.1}$$

Damit eine Rückwärtsvalidierung möglich ist, muß der UP alle von dem anfragenden Knoten noch nicht verarbeiteten Commit-Benachrichtigungen vorhalten. Dem UP ist die zuletzt verarbeitete Commit-Benachrichtigung eines Peers jedoch nicht bekannt. Als einziger Anhaltspunkt kann sich der UP von jedem Peer die ID der zuletzt erfolgreich abgeschlossenen Transaktion merken und über diese nicht mehr benötigte Einträge in der Commit-Historie erkennen und verwerfen. Das nachhaltige Vorhalten der Commit-Historien erfordert auf dem UP einen hohen Verwaltungsaufwand. Verwendet der UP dagegen eine Versionierung auf Objektebene, braucht er die Commit-Historien für folgende Commit-Anfragen nicht vorhalten.

5.2.3 Rückwärtsvalidierung mittels Objektversionierung

Bei der Rückwärtsvalidierung auf Objektebene besitzt anstatt der Transaktionen jedes einzelne transaktionale Objekt einen logischen Zeitstempel [60]. Jeder Knoten (ebenso der UP) speichert pro Objekt einen logischen Zeitstempel, dieser wird im Fall eines vom UP gewährten Commits aktualisiert. Bei einer Commit-Anfrage müssen Knoten für jedes Objekt den logischen Zeitstempel mitsenden. Anhand dieser kann der UP die Transaktionen auf Konflikte testen und feststellen, ob eine Transaktion veraltete Daten gelesen hat. Die Rückwärtsvalidierung über die logischen Objektzeitstempel garantiert die Datenkonsistenz, der UP muß sich daher auch keine Schreibmengen bereits abgeschlossener Transaktionen merken.

Der UP vergleicht die Versionsnummern aller in der Lesemenge der Transaktion befindlichen Objekte mit den Versionsnummern der entsprechenden validierten Objekte. Im Konfliktfall weist der UP Commit-Anfragen durch eine negative Rückmeldung an den anfragenden Knoten zurück. Als Hinweis enthält die Rückmeldung sowohl die Objekt-ID als auch die erforderlichen Versionsnummern der konfliktverursachenden Objekte. Für eine konfliktfreie Validierung einer abzuschließenden Transaktion T_c muß für die Objekte folgende Abhängigkeit gelten, wobei die Funktion $v()$ die Versionsnummer und $v_{max}()$ die letzte Versionsnummer eines Objekts beschreibt:

$$\forall x \in RS_{T_c}\, \exists y : x = y \wedge v(x) = v_{max}(y) \tag{5.2}$$

Insgesamt muß eine Commit-Anfrage also sowohl die Lese- als auch die Schreibmenge einer Transaktion enthalten. Die Lesemenge dient dem UP, eine Konflikterkennung über die Rückwärtsvalidierung durchzuführen, während die Schreibmenge der Weiterleitung auf den Empfängerknoten zwecks Vorwärtsvalidierung dient. Weiterhin bestimmen die Schreibmengenobjekte auf dem UP auch die zu aktualisierenden Objektversionsnummern im erfolgreichen Validierungsfall.

5.2.4 Kombinierte Vorwärts- und Rückwärtsvalidierung

Die Rückwärtsvalidierung auf dem UP läßt die Vorwärtsvalidierung überflüssig erscheinen zu lassen, da diese allein die Datenkonsistenz über die Objektversionsnummern garantiert. Die zusätzliche Vorwärtsvalidierung verbessert trotz der Netzkommunikation durch die Commit-Benachrichtigungen die Skalierbarkeit. Ohne die Benachrichtigungen muß jeder Knoten nach Beendigung seiner Transaktion zunächst eine Anfrage an den UP schicken, um eventuelle Konfliktinformationen zu erfahren. Dies führt aufgrund der synchronen Kommunikation zu einer kurzen Blockierungsphase. Zudem läßt die zusätzliche Vorwärtsvalidierung Peers Konflikte von Transaktionen bereits nebenläufig während dessen Laufzeit erkennen, während die Konflikterkennung bei der Rückwärtsvalidierung erst am Transaktionsende erfolgt. Peers können konfliktbehaftete Transaktionen so vorzeitig abbrechen, sofern sich die Transaktion in einem nicht kritischen Abschnitt befindet. Ein vorzeitiger Abbruch erhöht die Skalierbarkeit, da Peers Zeit sparen und Transaktionen unmittelbar wiederholen können, anstatt sie bis zum Ende durchlaufen zu lassen. Weiterhin verhindern Commit-Benachrichtigungen Konflikte, sofern sie trotz laufender Transaktionen Objektreplikate invalidieren, auf welche die Transaktionen erst nach der Invalidierung zugreifen. In diesem Fall fordern die Transaktionen vor dem ersten Zugriff automatisch die aktuelle Objektversion an. Aus diesen Gründen ist eine kombinierte Vorwärts- und Rückwärtsvalidierung für den UP-Commit sinnvoll.

5.2.5 Validierung von Nur-Lese-Transaktionen

Nur-Lese-Transaktionen brauchen in der Regel nicht die Commit-Phase zu durchlaufen, da sie keine Änderungen an dem transaktionalen Speicher durchführen. Diese Transaktionen müssen, auch wenn sie veraltete Objekte gelesen haben, nicht abbrechen. Allerdings ist es dann notwendig, daß das Commit-Protokoll Knoten über veraltete Objektreplikate informiert, da es bei bestimmten Programmierkonstrukten wie beispielsweise zählende Barrieren ansonsten zur Verklemmung kommen kann. Das UP-Protokoll verhindert dies durch seine Commit-Benachrichtigungen der kombinierten Vorwärts- und Rückwärtsvalidierung. Ohne Commit-Benachrichtigungen müßte ein Knoten bei Verwendung des UP-Protokolls dennoch eine Commit-Anfrage zwecks Rückwärtsvalidierung an den UP stellen (siehe auch Kapitel 6.1.1), obwohl die Transaktion keine Änderungen im transaktionalen Speicher vorgenommen hat. Denn nur so ist dann eine zuverlässige Identifizierung veralteter gelesener Objekte möglich. Beim P2P-Protokoll braucht ein Knoten in vergleichbarer Weise für Nur-Lese-Transaktionen kein Token anfordern, da seine Schreibmenge leer ist, wird aber gleichzeitig mittels eingehender Commit-Benachrichtigungen über veraltete Objektreplikate informiert. Validiert ein Knoten Nur-Lese-Transaktionen über den UP, braucht dieser ebenfalls keine Benachrichtigung an die anderen Knoten schicken, da die Schreibmenge der Transaktion leer ist. Die Vorwärtsvalidierung kann auch bei Nur-Lese-Transaktionen die Skalierbarkeit erhöhen, da Transaktionen dann keine blockierenden Commit-Anfragen an den UP stellen müssen.

Das Versenden einer Commit-Anfrage beziehungsweise Anwendung der Vorwärtsvalidierung ist beispielsweise bei verteilten Barrieren (Abbildung 5.5) notwendig. Dies läßt sich einfach an einem Beispiel einer primitiv implementierten Barriere im verteilten transaktionalen Speicher

erläutern. Ein an einer Barriere wartender Knoten liest periodisch eine Variable und wartet, bis diese einen bestimmten Wert annimmt. Nun ändert ein anderer Knoten die Variable auf einen Wert, welcher den ersten Knoten das Verlassen der Barriere ermöglicht. Da der transaktionale Speicher auf Objektreplikaten basiert, würde der erste Knoten wegen der ausbleibenden Vorwärts- und Rückwärtsvalidierung weiterhin auf einem veralteten Replikat arbeiten und folglich endlos in der Barriere verharren.

```
1 do {
2   BOT();
3   val = ta_barrier;
4   EOT();
5 } while (val != EXPECTED_VALUE);
```

Abbildung 5.5: Barriere im verteilten transaktionalen Speicher.

5.2.6 Commit-Protokoll

Der Protokollablauf ist in der Abbildung 5.6 dargestellt. Ein Knoten schickt eine Commit-Anfrage an den UP. Diese enthält die Lese- und Schreibmenge der Transaktion sowie die Objektversionsnummern (logische Zeitstempel). Die Anfrage landet zunächst in der Warteschlange von Commit-Anfragen. Bei der Verarbeitung der Commit-Anfrage führt der UP zunächst eine Rückwärtsvalidierung durch. Hierzu vergleicht er die Objektversionsnummern der Lesemenge mit den gespeicherten Versionsnummern der zuletzt festgeschriebenen Objekte. Ist mindestens eines der Objekte veraltet (kleinere Versionsnummer in der Lesemenge gegenüber der gespeicherten Versionsnummer) hat die Transaktion einen Konflikt verursacht. In diesem Fall schickt der UP eine negative Antwort auf die Commit-Anfrage zurück, welche die konfliktverursachenden Objekte inklusive der jeweils erforderlichen Objektversionsnummer, die für einen konfliktfreien Commit notwendig sind, zurück. Der anfragende Knoten kann nun die aktuelle Version der in der Antwortnachricht referenzierten Objekte anfordern und die Transaktion erneut starten, in der Annahme, die Transaktion im erneuten Anlauf erfolgreich abschließen zu können. Im Konfliktfall ist die Validierung auf dem UP beendet, und der UP schickt keine Commit-Benachrichtigung an die anderen Knoten. Tritt bei der Rückwärtsvalidierung dagegen kein Konflikt auf, gilt die Transaktion als erfolgreich validiert. Für jedes geschriebene Objekt vergibt der UP eine neue Versionsnummer und teilt diese in der Antwortnachricht mit. Gleichzeitig verschickt der UP eine Commit-Benachrichtigung an alle verbleibenden Knoten, welche den erfolgreichen Commit ankündigt. Die Benachrichtigung enthält die Schreibmenge der Transaktion inklusive der aktualisierten Versionsnummern. Die Validierung ist damit für den UP beendet.

Die Knoten, welche eine Commit-Nachricht empfangen, führen eine Vorwärtsvalidierung entsprechend des P2P-Protokolls durch. Für die Verarbeitung der Commit-Nachrichten auf den Zielknoten gelten dieselben Regelungen wie beim P2P-Protokoll. Entweder ein Knoten verarbeitet die Benachrichtigung entsprechend der globalen Commit-ID (sofern in der Nachricht mitgeliefert) oder anhand der Objektversionsnummern (siehe auch Kapitel 4.3.3). Die Objektversionsnummern kann ein Knoten zukünftig bei einer Objektanforderung als minimale Versionsnummer angeben, um bei einer Anforderung von einem anderen als den zuletzt transaktionsabschließenden Knoten keine veraltete Objektkopie zu erhalten.

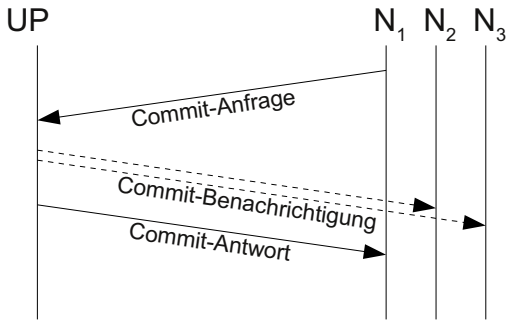


Abbildung 5.6: UP-Commit-Protokoll.

5.3 Hybrider Commit in strukturierten Overlay-Netzen

Die beiden Commit-Verfahren auf Basis eines P2P- und UP-Protokolls skalieren für eine begrenzte Anzahl von Knoten gut. Mit zunehmender Knotenanzahl sinkt die Skalierbarkeit, da sich beim P2P-Protokoll in erster Linie die Tokenlaufzeit verlängert, weil alle Teilnehmer um ein gemeinsames Token konkurrieren, und bei UP-Protokoll die Belastung des zentralen Koordinators zunimmt. Gerade in Grid-Umgebungen kommunizieren sehr viele Knoten miteinander über unterschiedliche Netzanbindungen und Entfernungen. Sowohl mehrere Token als auch Koordinatoren können die Skalierbarkeit verbessern. Die Wartezeit auf das Token verringert sich, wenn anfallende Tokenanfragen im Mittel gleichmäßig auf die Token verteilt beziehungsweise die Koordinatoren gleichmäßig belastet sind.

Die Verwendung mehrerer Token zur Erhöhung der Skalierbarkeit ist zulässig, aber nur insofern, wenn zuvor genau bekannt ist, daß durch unterschiedliche Token zu validierende überlappende Transaktionen keinen Konflikt auslösen können, sie also keine gemeinsamen Objekte zugreifen. Durch mehrere Token bestehen auch entsprechend viele logische Zeitstempel als Commit-Zähler, daß dieser global nicht mehr eindeutig ist. Die Validierung überlappender Transaktionen durch unterschiedliche Token kann deshalb dazu führen, daß Knoten veraltete Objektversionen nicht erkennen und mit diesen eigene Transaktionen abschließen und demzufolge die strenge Datenkonsistenz verletzen (siehe Abbildung 5.7).

Der Koordinator unter Anwendung des UP-Protokolls muß vermehrt gleichzeitige Commit-Anfragen verarbeiten. Dies führt zu einer höheren Systemauslastung (Prozessor- und Speicherauslastung). Ferner bewirken vermehrte Anfragen ebenso eine ansteigende Netzauslastung, da alle Anfragen auf einem Knoten auflaufen. Bei einer zu starken Auslastung wird der Koordinator zum Flaschenhals des Systems und ist demzufolge selbst für eine verminderte Skalierbarkeit ursächlich. Bei mehreren Koordinatoren treten die gleichen Probleme wie bei mehreren Token auf, daß unter Aufgabe der totalen Commit-Ordnung die Koordinatoren Konflikte eventuell nicht erkennen und deshalb Transaktionen mit veralteten Daten abschließen können (siehe Abbildung 5.8). Mittels Konsistenzdomänen (siehe Kapitel 6.4) kann der verteilte transaktionale Speicher dennoch mehrere Token beziehungsweise UPs einsetzen, um die Skalierbarkeit zu erhöhen.

Einerseits ist eine Kommunikation auf ein begrenztes Umfeld für eine gute Skalierbarkeit unerläßlich. Andererseits führt dies zu einem zusätzlichen Synchronisierungsaufwand, um eine mögliche Dateninkonsistenz zu vermeiden. Die Forderung, das Netzwerk in kleine Einheiten (Gruppen) zu unterteilen, trägt dem ersten Punkt Rechnung. Die Gruppen darüber hinaus als föderierte Einheiten zu bilden, daß diese sich untereinander synchronisieren, vermeidet die Dateninkonsistenz. Das Netzwerk erhält hierdurch eine logische Struktur (Overlay-Netzwerk),

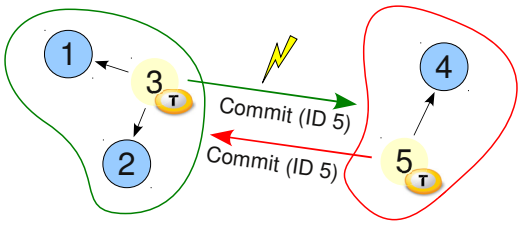


Abbildung 5.7: Unsynchronisierter Commit mit mehreren Token.

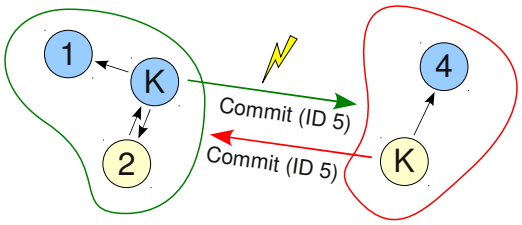


Abbildung 5.8: Unsynchronisierter Commit mit mehreren Koordinatoren.

die unabhängig der physischen Struktur ist (siehe Kapitel 5.1.3). Sind die Gruppen in ihrer Größe beschränkt und können die Kommunikation sofern möglich lokal begrenzen, ohne die Datenkonsistenz zu verletzen, kann das strukturierte Overlay-Netzwerk die Skalierbarkeit des verteilten transaktionalen Speichers weiter erhöhen. Das transaktionale Speichersystem muß genau feststellen, in welchen Fällen eine gruppenübergreifende Kommunikation für die Datenkonsistenz erforderlich ist (siehe Abbildung 5.3). Würden sich die Gruppen ihre Transaktionen weiterhin jederzeit global synchronisieren, würde das Overlay-Netzwerk gegenüber den Basis-Commit-Verfahren keinen Vorteil bringen, da der globale Synchronisierungsknoten der Flaschenhals wäre. Die Skalierbarkeit würde in Gegenteil weiter sinken, da das Overlay-Netzwerk einen zusätzlichen Mehraufwand gegenüber der direkten Kommunikation der Knoten untereinander aufweist. Die einzelnen Gruppen können Lokalität aber ähnlich dem lokalen Commit-Verfahren erzielen (siehe Kapitel 6.2.1 und 6.2.2).

5.3.1 Logische Knotengruppierung

Das Overlay-Netzwerkes bildet für das transaktionale Speichersystem eine geordnete Kommunikationsinfrastruktur. Ziel ist es, Commits von verschiedenen Knoten neben den Transaktionen selbst weitestgehend zu parallelisieren, damit das System auch bei einer großen Anzahl von Knoten skalierbar bleibt. Ebenso soll das Netzwerk die Netzkommunikation minimieren, indem es einen Kompromiß zwischen optimaler Wegewahl und dem Zwischenspeichern von transaktionssensitiven Informationen bei der Weiterleitung von Nachrichten zwischen den Knoten erreicht. Hierzu gehört ebenso eine optimierte Objektverwaltung und -suche, indem Knoten Objektanfragen an einen Knoten auch anhand zwischengespeicherter Informationen zum Ziel weiterleiten. Das gleiche gilt für Commit-Benachrichtigungen. Das Overlay-Netz muß entscheiden, welche Knoten im Falle von Commits zu benachrichtigen sind.

Knoten verfolgen in verteilten Anwendungen je nach Rollenverteilung beziehungsweise anwendungssemantischen Aspekten ähnliche oder unterschiedliche Interessen. Dies äußert sich wiederum in ihrem Zugriffsverhalten auf den gemeinsamen Speicher. Dies führt dazu, daß einige Knoten häufig auf gemeinsame Bereiche des verteilten transaktionalen Speichers zugreifen.

Andere Knoten wiederum können auf Speicherbereiche zugreifen, die teilweise oder vollständig zueinander disjunkt sind. Die Häufigkeit, mit denen Knoten gleichzeitig nicht disjunkte Speicherzugriffe durchführen, bestimmt primär die Konfliktwahrscheinlichkeit und folglich auch, wie oft die Knoten Objektreplikate miteinander synchronisieren müssen. Daher ist es sinnvoll, die Knoten zunächst nach ihrem Zugriffsmuster und der damit verbundenen transaktionalen Konfliktwahrscheinlichkeit zueinander zu klassifizieren und in gemeinsame Gruppen einzuteilen. Hierdurch minimiert sich die gruppenübergreifende Netzkommunikation. Ein Beispiel hierfür ist die interessenbasierte Kommunikation *Area of Interest Management* der verteilten virtuellen Welt (engl. Massively Multiuser Virtual Environment (MMVE)) *Wissenheim* [95].

Overlay-Parametrisierung

Für die Strukturierung des Overlay-Netzwerkes spielen sowohl anwendungssemantische Eigenschaften als auch die Eigenschaften des unterliegenden physischen Netzwerkes eine wichtige Rolle. Aufgrund der Zugriffsmuster der einzelnen Knoten ergeben sich folgende anwendungstypischen Parameter, welche für die Strukturierung des Overlay-Netzes zu berücksichtigen sind

- Häufigkeit des Objektaustauschs zwischen Peers (normiert auf Transaktionsdurchsatz)
- Durchschnittliche Abbruchrate von Transaktionen
- Mittlere Anzahl konfliktverursachender Objekte pro Transaktion

Die physischen Parameter sind dagegen zumeist fest durch das Kommunikationsnetzwerk bestimmt, welche unter anderem auf Weglänge, Netzwerkrouten und Kanalbandbreite beruhen. Wesentliche physische Parameter sind

- Netzwerklatenz
- Netzwerkbandbreite
- CPU-Auslastung
- Speicherauslastung

Die Parameter unterliegen aber dennoch einer gewissen Dynamik, so können andere Dienste und Datenübertragungen auf demselben physischen Netzwerk beispielsweise die Latenz und den Datendurchsatz des verteilten transaktionalen Speichers beeinflussen. Gleiches gilt auch für die Prozessor- und Speicherauslastung, wobei diese Parameter lokal je physischen Peer vorhanden sind, während sich die anderen beiden Parameter durch mehrere physische Peers bestimmen. Die physischen Parameter haben je nach Netzwerkkommunikation unterschiedliche Auswirkungen. Starke Auswirkungen auf das Anwendungsverhalten haben die Parameter insbesondere bei der synchronen Netzkommunikation, bei der ein Peer nach einer Anfrage entsprechend der Dauer der RTT auf eine Antwort wartet. Diese Art der Kommunikation kommt überwiegend bei Objekt-, Token- und Commit-Anfragen vor. CPU- und Speicherauslastung sind zumindest bei der Auswahl des Superpeers von wesentlicher Bedeutung, da seine Leistungsfähigkeit die gruppeninterne und -übergreifende Kommunikation beeinflußt.

Da einige Parameter dynamisch sind, sich während der Laufzeit also ändern können, muß das transaktionale Speichersystem diese ständig überwachen. Die Änderung der Parameter kann unter Umständen dazu führen, daß das Overlay-Netzwerks nicht mehr optimal strukturiert ist. In diesem Fall sollte sich das Overlay-Netzwerk selbständig restrukturieren, indem einzelne Knoten ihre Gruppenmitgliedschaft ändern beziehungsweise sich Gruppen auflösen oder neue

hinzukommen. Die dynamische Restrukturierung des Overlay-Netzwerkes wird detailliert in Kapitel 5.5 behandelt.

Damit das System die Netzwerklatenz zwischen zwei Knoten ermitteln kann, muß es diese während einer Datenübertragung einer bestehenden Verbindung messen. Zur Auswertung der Netzwerklatenzen zwischen allen Knoten ist daher wieder ein vollvermaschtes Netzwerk notwendig, was das Overlay-Netzwerk geradewegs vermeiden soll. Gemessene Netzwerklatenzen sind aufgrund von Jittern im Netzwerk und Verzögerungen durch die Kommunikationsendpunkte (zum Beispiel durch Prozessorauslastung oder Prozeßscheduling) selbst immer mit einem Fehler behaftet. Daher ist eine genaue Ermittlung der Latenz nicht möglich und auch nicht notwendig. Mittels Routing von Netzwerknachrichten und Kenntnis über die Netztopologie lassen sich Latenzen zwischen zwei Endpunkten approximativ ermitteln, ohne daß zwischen diesen Knoten eine physische Verbindung besteht. So kann das System ohne ein vollvermaschtes Netzwerk die Latenz zwischen allen Knoten ermitteln. Hierbei ist zu berücksichtigen, daß bei weniger Netzwerkverbindungen und Ermittlung der Netzlatenz über mehrere indirekte Endpunkte die Fehlerwahrscheinlichkeit steigt [29].

5.3.2 Overlay-Strukturierung

Wie bereits im Kapitel 5.1.1 erläutert, zeichnet sich ein Overlay-Netzwerk durch eine logische Strukturierung oberhalb eines physischen Netzwerkes aus. Das heißt, bestimmte Peers übernehmen auf der logischen Schicht die Routingfunktionalitäten, welche die (dedizierten) Router im physischen Netzwerk bewerkstelligen. Weiterhin sollen sich Knoten in Gruppen zusammenfinden und möglichst lokal kommunizieren.

Die Topologie eines Overlay-Netzes kann unterschiedliche Formen annehmen (beispielsweise Stern- oder Baumstruktur). Für die Parallelisierung von transaktionalen Commits mithilfe föderierter Gruppen eignet sich nur eine hierarchische Overlay-Topologie. Eine Gruppe muß selbständig entscheiden, ob sie den Commit einer Transaktion vollständig innerhalb der Gruppe abwickeln kann oder eine Interaktion mit anderen Knoten außerhalb der Gruppe benötigt. Hierfür muß der Gruppe bekannt sein, welche Objekte nur innerhalb und welche außerhalb der Gruppe repliziert sind. Da die Synchronisierung dieser Information unter allen Gruppenteilnehmern, vor allem wegen der Nebenläufigkeit von Commits und Objektaustausche, zu aufwendig ist, ist es sinnvoll, daß ein ausgezeichneter Gruppenteilnehmer diese Information verwaltet. Der gesamte Kommunikation zwischen einer Gruppe und dem restlichen Netzwerk erfolgt über diesen ausgezeichneten Knoten (Superpeer), der ausgetauschte Objekt- und Commit-Nachrichten protokolliert. Der Superpeer übernimmt daher im Overlay-Netzwerk die Routingfunktionalität für seine Gruppe.

5.3.3 Topologiemodelle

Die einfachste Topologie besteht aus einer zweistufigen hierarchischen Strukturierung. Auf der untersten Ebene befinden sich die föderierten Gruppen. Deren Superpeers befinden sich währenddessen ebenso auf der übergeordneten Hierarchieebene. Sie gehören demnach zwei übereinanderliegenden Hierarchieebenen an und steuern aufgrund ihrer Routingfunktionalität die Kommunikation zwischen diesen (siehe Abbildung 5.9).

Je nach Anwendungsfall können Knoten aber auch gemeinsame Interessen mit anderen Knoten haben, die nicht in ihrer Gruppe liegen. Demnach ließen sich diese Gruppen zusammenführen, und deren Knoten würden sich in einer großen Gruppe befinden (siehe Abbildung 5.10). Dies kann jedoch ineffizient sein, falls die Gruppe zu groß wird und deshalb die Skalierbarkeit sinkt.

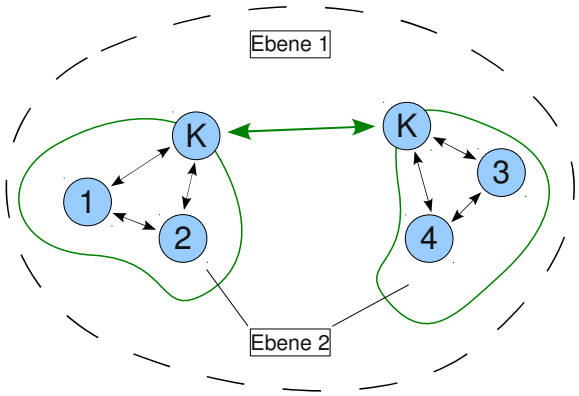


Abbildung 5.9: Zweistufige Overlay-Topologie.

In einem solchen Fall, kann die Overlay-Topologie statt der Vereinigung der Gruppen auch auf ein mehrstufiges System ausgeweitet werden (siehe Abbildung 5.11).

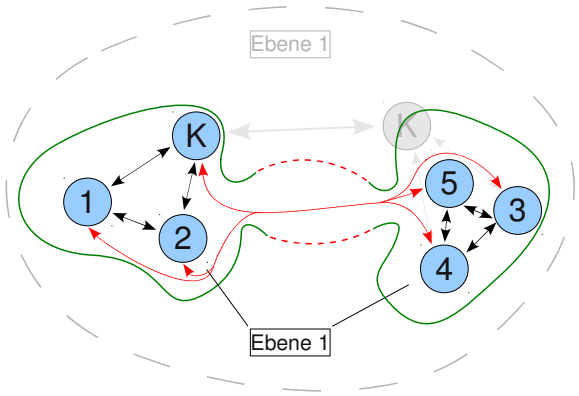


Abbildung 5.10: Verschmelzung zweier Gruppen im Overlay-Netzwerk mit Herabstufung eines Koordinators zum normalen Knoten (5).

Es gilt jedoch an dieser Stelle zu beachten, daß der mehrstufige Ansatz mit einem zusätzlichen Mehraufwand einhergeht, da ein gruppenübergreifender Commit nun eventuell über mehrere Superpeers läuft. Dieses Szenario tritt jedoch nur ein, falls ein gruppenlokaler Commit (siehe Kapitel 6.2.2) nicht möglich ist. Der Vorteil einer mehrstufigen Organisation liegt darin, daß wenn ein lokaler Commit in einer Gruppe auf der untersten Hierarchieebene nicht möglich ist, dieser dennoch in der übergeordneten Gruppe auf der darüberliegenden Hierarchieebene dennoch möglich sein kann. So muß ein Commit nicht im gesamten Overlay-Netzwerk bekannt gemacht werden wie beim zweistufigen Ansatz.

Es können ebenso Anwendungsfälle auftreten, in denen sich die gemeinsamen Interessen der Knoten zueinander nur auf einzelnen Speicherbereiche beziehen. Während sich ein Knoten für einen Speicherblock A in einer Gruppe mit weiteren Knoten zusammenfindet, kann für einen weiteren Speicherblock B die Mitgliedschaft in einer anderen Gruppe von Knoten vorteilhaft sein. Dies läßt sich lösen, indem man die Speicherblöcke unterschiedlichen Konsistenzdomänen zuordnet (siehe Kapitel 6.4).

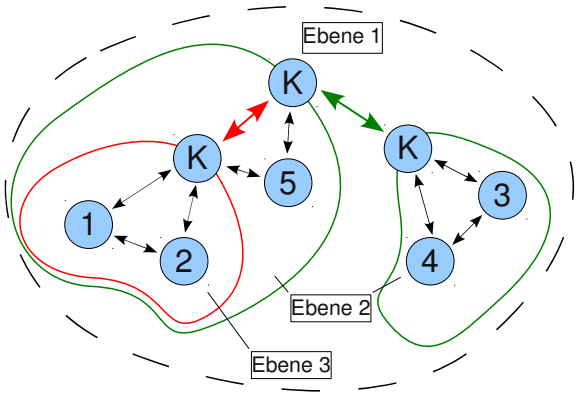


Abbildung 5.11: Mehrstufige Overlay-Topologie.

5.3.4 Integration der Commit-Protokolle in die Overlay-Topologie

Die P2P- und UP-Commit-Protokolle für sich betrachtet verwenden kein strukturiertes Overlay-Netzwerk und erfordern daher eine Abbildung auf die einzelnen Ebenen und Gruppen des Overlay-Netzwerkes, da jede Gruppe Commits zunächst nur innerhalb ihrer Gruppe durchführen und nur falls notwendig sukzessive auf übergeordnete Ebenen expandieren. Aufgrund des Superpeers bestehen Einschränkungen, die Protokolle effizient mit einem strukturiertem Overlay-Netzwerk zu verknüpfen, damit ein gruppenbasierter Commit möglich ist.

Peer-to-Peer-Protokoll

Das P2P-Protokoll hat wegen seiner Kommunikationsinfrastruktur einen stetig wechselnden Knoten, der den Commit von Transaktionen steuert. Dieser Knoten ist in den meisten Fällen nicht der Superpeer, welcher für die Kommunikation zwischen seiner Gruppe und dem restlichen Overlay-Netzwerk zuständig ist. Der eine Transaktion abschließende Knoten kann deshalb nicht mit der übergeordneten Gruppe kommunizieren, da dies im strukturierten Overlay-Netzwerk allein dem Superpeer vorbehalten ist. Demnach müßte der Superpeer einer Gruppe zu dem Knoten wechseln, der gerade eine Transaktion abschließt. Dies erfordert einen hohen Protokoll- und Kommunikationsaufwand, da der aktuelle Superpeer seine Zustandsinformationen über replizierte Objekte auf den neuen Superpeer übertragen muß, bevor dieser über den Commit einer Transaktion entscheiden kann (siehe auch Kapitel 6.2.2).

Da der Superpeer auch Mitglied der übergeordneten Gruppe ist, müßte er ebenso seine Mitgliedschaft wechseln und den Mitgliedern der übergeordneten Gruppe jederzeit bekannt sein. Alternativ könnte der Superpeer (abgesehen von Rekonfigurationen des Overlay-Netzes) statisch bleiben, dann ist keine aufwendige Kommunikation für den Mitgliedswechsel der übergeordneten Gruppe notwendig. Weiterhin bestehen bleibt aber die Notwendigkeit, daß der Knoten, der eine Transaktion abschließt, vom Superpeer seiner Gruppe vor dem Commit Informationen einholen muß, ob von ihm geänderte Objekte der abzuschließenden Transaktion außerhalb seiner Gruppe repliziert sind. Das P2P-Protokoll würde an dieser Stelle die Komplexität des Systems massiv steigern und einen erheblichen Mehraufwand verursachen, der dem Commit einer einzelnen Transaktion nicht angemessen ist. Daher eignet sich das P2P-Protokoll nicht für den Commit innerhalb von Gruppen.

Auf der höchste Ebene des Overlay-Netzwerkes läßt sich das P2P-Protokoll dagegen problemlos einsetzen. Der Unterschied besteht darin, daß die höchste Ebenen keine übergeordnete

Gruppe besitzt. Somit benötigt diese Ebene keinen Peer, der die Kommunikation mit einer übergeordneten Gruppe steuert und demnach Mitglied zweier Gruppen sein müßte. Daher können die statischen Superpeers (beim UP-Protokoll der Fall) der einzelnen Gruppen untereinander direkt mittels des P2P-Protokolls kommunizieren.

Ein direkter Vergleich zeigt mehrere Nachteile des reinen P2P-Protokolls im Gegensatz zur Verwendung desselbigen im Overlay-Netzwerk auf. Das reine P2P-Protokoll muß wegen der vollvermaschten Topologie zum einen viele Netzwerkverbindungen verwalten. Zudem muß ein Knoten bei einem Commit an jeden einzelnen Knoten eine Commit-Benachrichtigung schicken, was zu einer hohen Netzbelastung führt. Außerdem ist das Serialisierungstoken bei vielen Knoten ein Flaschenhals. In Verbindung mit dem Overlay-Netzwerk bestehen diese Nachteile nicht, da nur wenige Superpeers das Token untereinander austauschen müssen und die vollvermaschte Topologie aufgrund der wenigen Superpeers überschaubar ist. Der allgemeine Nachteil des Overlay-Netzwerks ist, daß wegen der indirekten Kommunikation zweier Knoten aufgrund des Nachrichtenroutings oftmals eine höhere Latenz besteht.

Ultrapeer-Protokoll

Das UP-Protokoll hat gegenüber dem P2P-Protokoll einen festen Transaktionskoordinator. Da dessen Koordinator sämtliche Informationen für einen Commit innerhalb seiner Gruppe auf sich vereinigt, ist es sinnvoll, den Superpeer einer Gruppe als Vermittler zur übergeordneten Gruppe zusammen mit dem Koordinator der Gruppe auf demselben Knoten zu vereinigen. Prinzipiell können der Koordinator und der Superpeer auch auf zueinander disjunkten Knoten existieren, was für eine bessere Lastverteilung spricht. Allerdings treten an dieser Stelle dieselben Probleme wie beim P2P-Protokoll auf. Der Koordinator hat keine Informationen über die ausgetauschten Objektreplikate zwischen der Gruppe und dem restlichen Overlay-Netzwerk, da dies dem Superpeer vorbehalten ist. Dies würde eine weitere Indirektion beim Commit verursachen. Dann müßte der Superpeer die Commit-Nachricht selbständig nochmals daraufhin untersuchen, ob er diese verwerfen kann, wenn der Commit den Kriterien eines gruppenlokalen Commit entspricht. Somit ist es nicht sinnvoll, den Koordinator und Superpeer auf zwei unterschiedliche Knoten aufzuteilen. Aufgrund des festen Superpeers (Koordinators) ist ebenso nicht notwendig, die übergeordnete Gruppe über eine wechselnde Mitgliedschaft des vermittelnden Knotens der untergeordneten Gruppen mitzuteilen.

Das UP-Protokoll eignet sich ebenso für die oberste Ebene, auf der auch das P2P-Protokoll einsetzbar ist. Statt sich die Gruppen über das P2P-Protokoll synchronisieren zu lassen, vereinigen sich die Gruppen in einer neuen übergeordneten Gruppe mit einem neuen Koordinator. Der Koordinator benötigt keine Superpeerfunktionalität beziehungsweise nutzt diese nicht, da keine übergeordnete Gruppe existiert (siehe Abbildung 5.12).

Das reine UP-Protokoll hat genauso wie das reine P2P-Protokoll mehrere Nachteile gegenüber seiner Verwendung im Overlay-Netzwerk. Für ersteres ist ein einzelner UP (Superpeer) bei vielen Knoten einer hohen Belastung aufgrund der Transaktionsvalidierung ausgesetzt. Ebenso führt die Verteilung von Commit-Benachrichtigungen an alle Knoten zu einer hohen Netzbelastung. In Verbindung mit dem Overlay-Netzwerk ist die Last besser verteilt, da jeder UP nur für eine kleine Gruppe von Knoten zuständig ist.

5.4 Nachrichtenrouting

Da die gruppenübergreifende Kommunikation in einem strukturierten Overlay-Netzwerk indirekt über die Superpeers stattfindet, müssen diese Routingfunktionalitäten übernehmen. Das Routing erfolgt analog zu dem in IP-basierten Netzwerken. Die Gruppen entsprechen einem

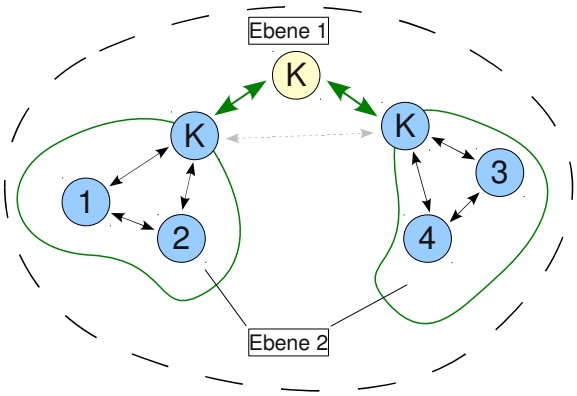


Abbildung 5.12: UP- und P2P-Protokoll auf oberster Ebene des Overlay-Netzwerkes.

IP-Subnetz und die Superpeers den zugehörigen Gateways in andere Subnetze respektive Gruppen im Overlay-Netz. Bei Anwendung des P2P-Protokolls besteht der Unterschied darin, daß die Superpeers nicht zwei Gruppen angehören. In Analogie zur IP-Kommunikation entspricht dies einer Kommunikation zwischen zwei Subnetzen nur über ihre Gateways. Zu unterscheiden ist das Routing, welches auf der untersten nachrichtenverarbeitenden Ebene stattfindet (Netzschicht), von der Nachrichtenweiterleitung auf Anwendungsebene. Das Routing auf der Netzschicht findet anhand der Ziel-ID einer Nachricht statt, während eine Weiterleitung von Nachrichten auf Anwendungsebene anhand anwendungssemantischer Kriterien erfolgt, beispielsweise falls ein Knoten eine Anfrage nicht beantworten kann.

Nachrichtenrouting in Overlay-Netzen ist gegenüber IP-Routing komplexer. Der IP-Adreßraum ist größtenteils geographisch aufgeteilt, damit die Routingtabellen eines Routers klein bleiben [83]. Somit ist es möglich, mit einem Routingeintrag ein gesamtes Netz oder Teilnetz zu adressieren. Die Aufteilung ist möglich, da die Rechner zumeist einen statischen Standort besitzen. Die Knoten des Overlay-Netzwerkes sind aufgrund ihrer wechselnden Interessen und damit verbundenen möglichen Netzwerk-Restrukturierungen (siehe Kapitel 5.5) dynamisch und können so mit der Zeit ihre Gruppenmitgliedschaft ändern. Daher läßt sich von der ID eines Knotens nicht auf dessen Ort schließen. Die Routingtabellen enthalten deshalb nur Einträge einzelner Knoten (vergleichbar mit Host-Einträgen in Routingtabellen von IP-Netzwerken), da die Knoten-IDs keinem Schema folgen, über welche sich ihre Gruppe respektive Ort bestimmen läßt. Bei vielen Knoten können die Routingtabellen der Superpeers daher stark anwachsen, wenngleich sich eine Suche in den Routingtabellen mittels Hashtabellen (HT) mit einer Komplexität von *O(1)* realisieren läßt. Zudem müssen die einzelnen Peers einer Gruppe ebenfalls eine Tabelle über ihre anderen Gruppenteilnehmer führen, damit sie entscheiden können, welche Nachrichten über den der Gruppe zugehörigen Superpeer weiterzuleiten sind. Dies ist notwendig, da sich die Gruppenmitglieder nicht aus den Routingtabellen für Knoten außerhalb der eigenen Gruppe ableiten lassen, da diese aufgrund des dynamischen Beitritts und Verlassens von Teilnehmern nicht unbedingt vollständig sind.

Das Routing von Nachrichten erfolgt in dem Overlay-Netzwerk symmetrisch, da der Graph zum einen zyklenfrei ist und ein Superpeer neue Routen schon während einer Nachrichtenanfrage lernt. Demzufolge ist den weiterleitenden Knoten die Route vor dem Versand der Antwort bereits bekannt (siehe Abbildung 5.13). Knoten *1* stellt beispielsweise eine synchrone Netzwerkanfrage an Knoten *5*, welche die Superpeers (Koordinatoren) der Gruppe entsprechend zum Ziel weiterleiten. Die Antwort an Knoten *1* gelangt auf dem gleichen Weg zurück.

Das Routing erfolgt lediglich dann asymmetrisch, wenn ein Knoten eine an ihn selbst gerich-

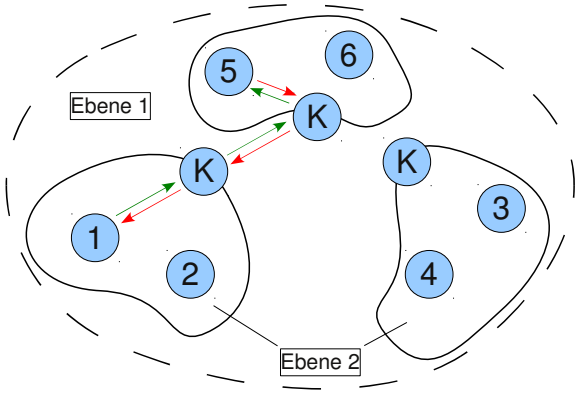


Abbildung 5.13: Symmetrisches Routing von Nachrichten im Overlay-Netzwerk.

tete Anfrage auf Applikationsebene an einen anderen Knoten weiterleitet. Dann erfolgt der Versand der Antwort nicht über den auf Anwendungsebene weiterleitenden Knoten (siehe Abbildung 5.14). Knoten *1* schickt beispielsweise wie beim symmetrischen Routing eine Anfrage an Knoten *5*. Dieser leitet diese auf Anwendungsebene (Forwarding) an den Knoten *3* weiter. Dies kann geschehen, falls ein Knoten eine Anfrage nicht beantworten kann. Die Antwort von Knoten *3* an Knoten *1* erfolgt in diesem Fall auf einer zur Anfrage asymmetrischen Route. Ist ein Weiterleitungsziel für eine Nachricht nicht bekannt und das Ziel kein Knoten der jeweiligen Gruppe eines Superpeers, so leitet ein Superpeer die Nachricht an alle von ihm direkt erreichbaren Superpeers weiter (Multicast). Superpeers, die den Nachrichtenempfänger nicht in ihrer oder ihnen untergeordneten Gruppe haben, verwerfen die Nachricht (siehe Abbildung 5.15).

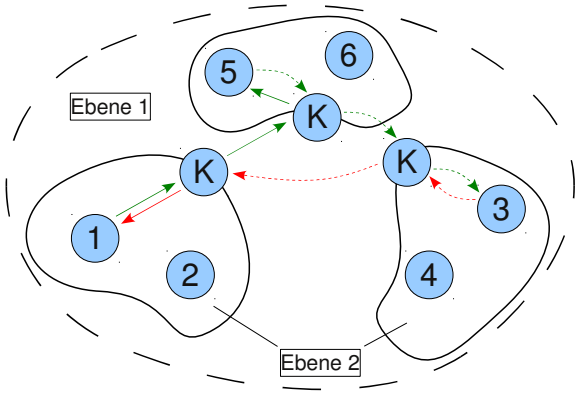


Abbildung 5.14: Asymmetrisches Routing von Nachrichten im Overlay-Netzwerk.

5.4.1 Overlay-Multicast

Neben einer Vervielfachung von Nachrichten infolge des Routings im Overlay-Netz ist dies auch durch den Nachrichtenaustausch des transaktionalen verteilten Speichers selbst der Fall. Bei der Kommunikation müssen Knoten dieselbe Nachricht gelegentlich an mehrere Knoten schicken. Dies kommt vor allem bei Commit-Benachrichtigungen von Transaktionen vor, um

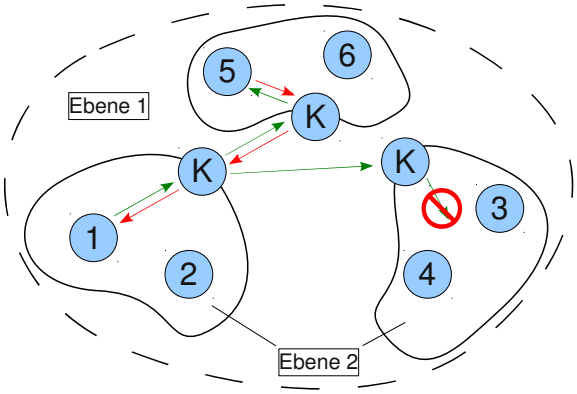


Abbildung 5.15: Multicast-Routing von Nachrichten im Overlay-Netzwerk.

andere Knoten über die geänderten Objekte zu informieren. Da IP-Multicast nur sehr eingeschränkt zur Verfügung steht und das physische Netzwerk aufgrund des Overlay-Netzwerkes verborgen bleibt, kann eine Multicast-Kommunikation auch auf Applikationsebene *(engl. Application Layer Multicast (ALM))* erfolgen [24].

Die zusätzliche Schicht des Overlay-Netzes oberhalb der physischen Kommunikationsschicht verursacht Indirektionen, die zu einer höheren Latenz, Netzbelastung und Auslastung der Knoten führen (siehe Abbildung 5.1). Das ist darin begründet, daß Knoten im Overlay-Netzwerk oftmals mehrere logische Verbindungen unterhalten, aber nur eine physische Anbindung besitzen. Auf der physischen Netzebene können Router kürzeste Wege zwischen Endpunkten im Netzwerk garantieren, im Overlay-Netzwerk sind sie dagegen transparent. Knoten übernehmen die Routingfunktionalität im Overlay-Netzwerk. Diese sind zumeist Rechner, die physische Endpunkte im physischen Netzwerk bilden. Daher wandern dieselben Nachrichten beim Routing mehrfach über dieselbe physische Netzverbindung. Weiterhin bestehen im Overlay-Netzwerk die gleichen Probleme wie bei der Nachbildung eines Multicast durch den Versand von Unicast-Nachrichten im IP-Netzwerk (siehe Kapitel 4.1.3).

Der Multicast auf Applikationsebene führt zu einer verbesserten Skalierung im Overlay-Netz, da er die Replizierung von Nachrichten über dieselbe logische Verbindung vermeidet (siehe Abbildung 5.16). Die routenden Knoten müssen hierzu genauso wie IP-Multicast-Router die Verteilung und Replizierung von Nachrichten im Overlay-Netzwerk steuern. Dies senkt die Netzbelastung und Auslastung der Knoten im Overlay-Netzwerk. Die routenden Knoten entscheiden selbst, auf welche logische Netzverbindung eine Nachricht weiterzuleiten ist. Da die Superpeers in dem Overlay-Netzwerk die Routingfunktionalität übernehmen, ist es ebenfalls möglich, den Multicast wie die Weiterleitung von Unicast-Nachrichten anhand semantischer Aspekte zu steuern. Beispielsweise kann ein Superpeer mittels Informationen von superpeerlokalen Commits entscheiden, ob er eine Commit-Benachrichtigung an alle Gruppen ausliefern muß oder dies teilweise unterlassen kann. Zwischen den Superpeers selbst findet bei einer zweistufigen Overlay-Hierarchie kein Routing statt, da alle Superpeers derselben Hierarchieebene angehören und wegen ihrer beschränkten Anzahl untereinander direkt kommunizieren (vollvermaschte Netzebene) können. Multicast-Kommunikation kann vor allem bei Netzanbindungen mit niedriger Bandbreite (beispielsweise in Weitverkehrsnetzen) Vorteile bringen. Banerjee et al. haben gezeigt, daß ein Multicast auf Applikationsebene für eine große Anzahl von Teilnehmern mit niedriger Bandbreite gut skalieren kann [12].

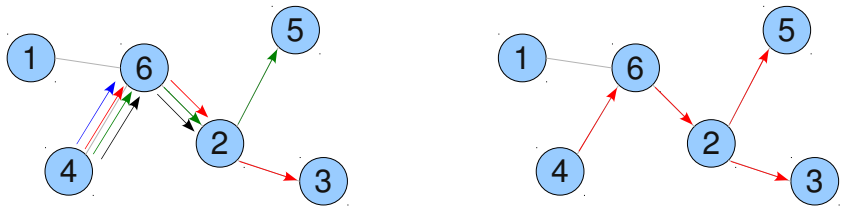


Abbildung 5.16: Unicast- und Multicast-Kommunikation im Overlay-Netzwerk.

5.5 Dynamische Overlay-Restrukturierung

Die Struktur eines Overlay-Netzwerks basiert überwiegend auf dynamischen Parametern, die bereits Kapitel 5.3.1 behandelt hat. Die Parameter können sich während der Laufzeit ändern, daß das Overlay-Netzwerk nicht mehr optimal strukturiert ist. Beispielsweise können sich auf der Infrastrukturseite die Netzlatenzen oder auf Applikationsseite die Interessen einzelner Knoten ändern. Je nach Änderung der für den Strukturaufbau herangezogenen Parameter kann eine Rekonfiguration notwendig sein, um eine gute Skalierbarkeit weiterhin zu gewährleisten. Folgende Restrukturierungsereignisse können auftreten

- Gruppenübergreifender Knotenwechsel
- Wechsel des Superpeers innerhalb einer Gruppe
- Auflösung oder Erstellung von Gruppen

Andere Ereignisse können sich aus mehreren Ereignissen zusammensetzen, wie beispielsweise der Wechsel eines Superpeers als Nicht-Superpeer in eine andere Gruppe. In diesem Fall gibt der Superpeer zunächst seine Aufgaben an einen anderen Knoten innerhalb derselben Gruppe ab. Es findet also ein Superperwechsel statt. Danach wechselt der ehemalige Superpeer als gewöhnlicher Knoten in eine andere Gruppe.

5.5.1 Gruppenübergreifender Knotenwechsel

Beim Wechsel der Gruppenmitgliedschaft eines Knotens muß sich dieser aus seiner alten Gruppe ab- und in der neuen anmelden. Dies bedeutet, der wechselnde Knoten muß alle Gruppenteilnehmer über seinen Wechsel informieren. Weiterhin müssen die Superpeers ihre bestehenden Routen für diesen Knoten aktualisieren, damit Nachrichten von anderen Knoten den wechselnden Knoten innerhalb des Overlay-Netzwerkes weiterhin erreichen. Da eine atomare Aktualisierung aller Routen während des laufenden Betriebs nicht möglich ist, muß der Superpeer der verlassenen Gruppe Nachrichten an den gewechselten Knoten eine Zeit lang weiterleiten. So kann die Aktualisierung aller bestehenden Routen nebenläufig erfolgen.

Bei einem Knotenwechsel im Overlay-Netzwerk ist zu berücksichtigen, daß sich damit gleichzeitig die Replizierung von Objekten ändern kann. Einem Superpeer ist nur bekannt, welche Objektreplikate in der eigenen Gruppe existieren, aber nicht auf welchen Knoten, da Knoten in derselben Gruppe untereinander direkt kommunizieren und Objekte austauschen dürfen. Bei einem Knotenwechsel wandern gleichzeitig die auf dem wechselnden Knoten gespeicherten Objektreplikate in die neue Gruppe. Daher muß der Knoten seinem aktuellen Superpeer mitteilen, welche Objekte mit dem Wechsel außerhalb der Gruppe repliziert werden. Dies ist erforderlich, damit Commit-Benachrichtigungen weiterhin alle Knoten erreichen, die Objektreplikate der zugehörigen Transaktion besitzen und der Superpeer den Commit nicht fälschlicherweise als gruppenlokalen Commit auffasst.

5.5.2 Superpeerwechsel

Bei einem Wechsel des Superpeers einer Gruppe muß dieser seine exklusiv vorhandenen Metadaten (außerhalb der Gruppe replizierte Objekte und Routingtabellen) an den neuen Superpeer abgeben. Zusätzlich müssen die Superpeers ihren Wechsel neben der eigenen auch in der übergeordneten Gruppe ankündigen, da ein Superpeer zu dieser als Vermittler auftritt.

Ein Superpeerwechsel läuft im einzelnen folgendermaßen ab. Ein Knoten meldet beim derzeitigen Superpeer seiner Gruppe einen Wechsel an, dieser kann diesem zustimmen oder ablehnen. Falls der alte Superpeer den Wechsel genehmigt, nimmt der zukünftige Superpeer eine Wartezustand ein. Währenddessen informiert der gegenwärtige Superpeer alle seine Gruppenteilnehmer und von ihm direkt erreichbaren Superpeers über den Wechsel, daß dieser fortan alle an den Superpeer gerichteten Nachrichten erhält. Anschließend überträgt der Superpeer die relevanten Metadaten an den anderen Knoten. Nach Übertragung der Metadaten nimmt der bestehende Superpeer die Rolle eines gewöhnlichen Knotens ein. Der zukünftige Superpeer verlässt dagegen nach Erhalt der Metadaten den Wartezustand und arbeitet nun als Superpeer. Nach einem Superpeerwechsel muß das Overlay-Netzwerk eine definierte Sperrzeit einhalten, in der kein weiterer Superpeerwechsel möglich ist. Dies ist notwendig, um Schwingungen im System durch häufige Superpeerwechsel in kurzer Zeit zu vermeiden.

5.5.3 Gruppenverwaltung

Als weiteres Restrukturierungselement kann das Overlay-Netz neue Gruppen bilden oder auch bestehende Gruppen zu einer gemeinsamen zusammenführen. Die Bildung neuer Gruppen erfolgt analog zum Gruppenwechsel von Knoten. Statt in eine neue Gruppe zu wechseln, erstellt der Knoten eine neue Gruppe und erhebt sich selbst in den Superpeerstatus durch Mitteilung an die anderen Superpeers. Weitere Knoten können anschließend über den gruppenübergreifenden Knotenwechsel beitreten. Die Zusammenführung von Gruppen erfolgt in umgekehrter Reihenfolge. Zunächst wechseln alle Knoten einer bestehenden Gruppe in andere Gruppen, bis nur noch der Superpeer vorhanden ist. Anschließend gibt er seinen Superpeerstatus auf und teilt dies den anderen Superpeers mit. Gleichzeitig wechselt er ebenfalls in eine bestehende Gruppe. Bei der Restrukturierung von Gruppen muß das System analog zu dem Superpeerwechsel eine Sperrzeit einhalten, um Schwingungen im System zu vermeiden.

5.6 Objektlokalisierung im Overlay-Netzwerk

Kapitel 4.5 hat bereits die Problematik der effizienten Suche nach aktuellen Versionen von Objekten in einem verteilten transaktionalen Speicher aufgegriffen. Strukturierte Filesharing-Systeme verwenden überwiegend DHTs für eine effiziente Suche von Objekten in einem P2P-Netzwerk. In Filesharing-Systemen sind zu suchende Objekte Dateien oder Dateifragmente. Nachdem ein Knoten eine Datei zum System hinzugefügt hat, lesen andere Knoten und erzeugen bei sich gleichzeitig ein Replikat, welches sie wiederum anderen Knoten zur Verfügung stellen. Weil Peers Objekte nur lesen und selbst nicht ändern, brauchen Knoten Objekte im Netz nicht zu invalidieren. Objektreplikate verschwinden aus dem Netz nur, sofern sich ein Peer aus dem Netz abmeldet. Deshalb müssen die Knoten ihre DHTs nur selten aktualisieren, da die Objekte zumeist statisch bleiben.

In einem gemeinsamen verteilten Speicher auf Basis von Objektreplikaten erfahren Objekte häufig Änderungen, was dazu führt, daß eine Änderung sämtliche Replikate invalidiert oder aktualisiert. Wie zuvor erläutert, wandern die Objekte bei Verwendung der Invalidierungsstrategie innerhalb des Systems. Bewegt sich ein Objekt zu einem anderen Peer muß der

für den DHT-Eintrag dieses Objekts zuständige Knoten den Eigentümer des Objekts aktualisieren. Bei vielen Objektbewegungen bedingt dies demnach entsprechend viele aufwendige DHT-Aktualisierungen. In Filesharing-Systemen können Peers DHT-Einträge selbst cachen und darauf zurückgreifen, anstatt bei jeder Suchanfrage immer den verantwortlichen Peer zu befragen beziehungsweise Anfragen anderer Knoten weiterzuleiten. Bei dem transaktionalen Speicher lohnt sich dies nicht, da die Cacheeinträge bei Schreibzugriffen schnell veralten. Deshalb eignen sich DHTs in Verbindung mit Caching nicht als primäre Suchstrategie für Objektreplikate in einen verteilten transaktionalen Speicher, zudem Knoten, die ein Replikat eines Objekts besitzen, den Verursacher des letzten Schreibzugriffs zwecks Invalidierung durch eine Commit-Benachrichtigung mitgeteilt bekommen. Die Verwendung von DHTs kann für die Suche in transaktionalen Speichersystemen als sekundäre Suchstrategie sinnvoll sein, sofern ein Knoten ein anzufordernes Objekt noch nie oder längere Zeit nicht zugegriffen hat, da ein Knoten durch Optimierungsstrategien wie *lokale* und *gruppenlokale Commits* (siehe Kapitel 6.2.1 und 6.2.2) eventuell keine Commit-Benachrichtigungen erhält, falls er kein Replikat besitzt. Vor der Suche mittels DHTs ist es allerdings sinnvoll, in strukturierten Overlay-Netzen zunächst einen übergeordneten Superpeer zu befragen, da diesem durch Caching von gerouteten Nachrichten wie Commit-Benachrichtigungen und ausgetauschten Objektreplikaten der aktuelle Eigentümer des gesuchten Objektreplikats eventuell bekannt ist. Strategien für die Replikation von Transaktionsobjekten erfolgt ausführlich in [81].

5.7 Verwandte Arbeiten

5.7.1 DiSTM

DiSTM [57] implementiert neben dem Commit-Verfahren auf Basis von Tickets (siehe Kapitel 4.7.2) noch zwei weitere Verfahren (*Serialization Lease* und *Multiple Leases*). Beim ersten der beiden Verfahren stellen Arbeitsknoten zunächst eine Anfrage an einen Masterknoten, um eine exklusive Commit-Berechtigung ihrer abzuschließenden Transaktion zu erhalten (Lease). Hierüber garantiert der Masterknoten eine strenge Transaktionsserialisierung. Nach Erhalten der Berechtigung darf der Arbeitsknoten seine Transaktion abschließen und schickt seine in der Transaktion geänderten Objekte an den Masterknoten. Dieser schickt sie wiederum an die anderen Arbeitsknoten des verteilten transaktionalen Speichers weiter, damit sie eine lokale Konfliktprüfung für eigene laufenden Transaktionen durchführen und Objektreplikate aktualisieren. Konfliktbehaftete Transaktionen brechen Knoten unmittelbar ab. Das Serialization-Lease-Verfahren arbeitet in vergleichbarer Weise, erlaubt aber mehr Nebenläufigkeit, weil der Masterknoten mehrere Berechtigungen zur gleichen Zeit vergeben kann. Im Vergleich zum ersten Verfahren muß der Masterknoten zusätzlich selbst eine Konfliktprüfung zwischen allen Transaktionen durchführen, für die er zur gleichen Zeit eine Berechtigung vergeben hat.

Die beiden Verfahren verursachen im Vergleich zum Ticket-Verfahren wesentlich weniger Broadcast-Nachrichten. Das Verfahren mit einem einzigen Lease blockiert Commit-Anfragen auf dem Masterknoten, was die Skalierbarkeit des Systems einschränkt. Dieses Problem umgeht der Masterknoten mit mehreren Leases. Durch die Nebenläufigkeit verkürzen sich die Blockierungsphasen, welche die Skalierbarkeit verbessert.

Beim UP-Protokoll stellt ein Knoten eine Validierungsanfrage an den UP und erhält gleich darauf das Validierungsergebnis. Dieser Vorgang ist mit der Erlangung der Commit-Berechtigung vergleichbar. Beim UP-Protokoll hat die Validierung allerdings bereits schon über die Rückwärtsvalidierung stattgefunden. Bei DiSTM kann ein Knoten das erst nach Erhalten des Leases feststellen. Andere Knoten können die Konfliktprüfung erst durchführen, wenn der Knoten, welcher seine Transaktion abschließt, die geänderten Daten an den Masterknoten schickt und dieser diese anschließend an die anderen Knoten weiterleitet. Beim UP-Protokoll können alle

Knoten nebenläufig über den UP validieren. Außerdem verschickt der UP für jeden erfolgreichen Commit eine Commit-Benachrichtigung an die andere Knoten, um Zeit zu sparen. Das UP-Protokoll verursacht bezüglich der Validierung eine geringere Netzbelastung, da zum einen weniger Kommunikationsschritte notwendig sind und Knoten nicht unnötig mit aktualisierten Objektreplikaten belastet werden, welche die Knoten nicht unbedingt verwenden.

5.7.2 Kademlia

Kademlia [69] implementiert eine verteilte Hashtabelle in einem P2P-System ähnlich wie Pastry und Tapestry. Jedes in der Hashtabelle abzulegende Datum besitzt einen Schlüssel aus einem Schlüsselraum, der sich aus der zugrundeliegenden Hashfunktion berechnet. Weiterhin besitzt jeder Peer eine eindeutige ID aus dem Schlüsselraum der Hashfunktion. Peers speichern Schlüssel, die nahe bei ihrer ID liegen, wobei mehrere Peers den gleichen Schlüssel replizieren. Somit sind Peers in der Lage, Nachbarschaftsknoten anhand eines Schlüssels ausfindig zu machen. Kademlia verwendet eine symmetrische Routingmetrik auf Basis einer XOR-Operation und ist damit gegenüber einer asymmetrischen Metrik in der Lage, Routinginformationen aus eingehenden (gerouteten) Nachrichten zu extrahieren. Die von Peers erhaltenen Anfragen decken sich daher mit einer Knotenverteilung entsprechend den Einträgen der eigenen Routingtabelle. Das Routing im strukturierten Overlay-Netzwerk dieser Arbeit angewandte Routing ist ähnlich zu Kademlia ebenfalls symmetrisch.

Da Änderungen in einer verteilten Hashtabelle gegenüber Suchanfragen relativ teuer sind, eignen sich verteilte Hashtabellen nicht für die Suche nach Objektreplikaten des verteilten Speichers. Bei Lesezugriffen auf den verteilten Speicher entstehen mitunter mehrere Replikate einer bestimmten Version desselben Objekts auf unterschiedlichen Knoten. Die Suche nach diesen Replikaten ist mittels der Tabellen einfach möglich. Schreibt eine Transaktion dagegen ein Objekt, entsteht von diesem eine neue Version, die zunächst nur auf dem transaktionsabschließenden Knoten vorliegt. Damit andere Knoten die neueste Objektversion finden können, muß das System zunächst den Eintrag in der verteilten Hashtabelle aktualisieren. Das gleiche gilt, wenn Replikate dieser Objektversion durch Lesezugriffe entstehen. Da sich viele Objekte im verteilten Speicher häufig ändern, zieht dies eine aufwendige Aktualisierung von verteilten Hashtabellen nach sich, was nicht vertretbar ist. Daher ist es effizienter Standortinformationen eines Objekts anhand der Netzwerkkommunikation des Commit-Protokolls herauszufiltern und Objektanfragen an diese Knoten zu stellen. Fluktuation von aktuellsten Objektversionen kann ein Knoten, der eine Objektanfrage erhält, mittels Weiterleitung von Anfragen anhand eigens zwischengespeicherten Standortinformationen begegnen.

5.7.3 Gnutella

Gnutella [91] ist ein Dienst zum direkten Austausch von Dateien in P2P-Netzwerken. Das Netzwerk entspricht einer dezentralen P2P-Netzwerkarchitektur. Jeder Knoten unterhält eine Verbindung zu mehreren Nachbarknoten, wobei jeder Knoten Anfragen von einem Nachbarknoten an seine anderen Nachbarknoten weiterleitet. So entsteht ein großer Netzwerkverbund von Rechnern. Da das Netzwerk keine Struktur besitzt, erfolgt die Suche nach Dateien über Anfragen an die Nachbarknoten, welche diese wiederum an ihre Nachbarknoten weiterleiten, sofern sie die gesuchte Datei nicht besitzen. Hierbei entsteht für jede Suchanfrage im Mittel eine sehr hohe Netzwerkkommunikation, da der Netzwerkverkehr mit der Anzahl an Routingstationen exponentiell zunimmt (Flutung des Netzwerks). Darüber hinaus ist die eingeschränkte Zuverlässigkeit nicht für ein verteilten transaktionalen Speicher geeignet, da Gnutella bei einer zu großen Distanz Dateien im Netzwerk eventuell nicht findet, da die TTL die Weiterleitung von Suchanfragen im Netzwerk beschränkt (siehe Kapitel 5.1.2).

Der overlay-basierte gemeinsame verteilte transaktionale Speicher verwendet hingegen ein strukturiertes Overlay-Netzwerk, welches sich an den Interessen der einzelnen Knoten orientiert, indem sich Knoten mit ähnlichen Zugriffsmustern in gemeinsamen Gruppen organisieren. Weiterhin unterhält das Netzwerk ein strukturiertes Routing von Netzwerknachrichten, wobei die Ausnutzung von Lokalität die Netzwerkkommunikation reduziert und so für eine bessere Skalierbarkeit sorgt. Das Overlay-Netzwerk kann sich zudem dynmaisch restrukturieren, wenn sich die von den Superpeers beobachteten Zugriffsmuster beziehungsweise die Interessen der Knoten ändern.

5.7.4 JuxMem

Bei Juxmem [8] handelt es sich um ein System von föderierten Clustern zur verteilten und persistenten Speicherung von Daten in Gridumgebungen. Ziel ist es, gemeinsame Daten im gesamten Grid schnell und ortstransparent zugreifbar zu machen. Juxmem verwendet für die Kommunikation im Netzwerk ein Overlay-Netzwerk, indem es Knoten zunächst zu virtuellen Clustern gruppiert. Knoten können anderen Knoten Speicher zur Verfügung stellen aber auch selbst anfordern. Allozierte Speicherblöcke repliziert Juxmen für eine bessere Verfügbarkeit automatisch und clusterübergreifend über mehrere Knoten, entsprechend den Anforderungen bei deren Allozierung. Knoten, die einen gemeinsamen Speicherblock replizieren, bilden eine gemeinsame Datengruppe. Aktualisierungen erfolgen dabei über Multicast-Kommunikation, und die Verwaltung und Suche freier Speicherblöcke erfolgt über eine DHT. Allozierte Speicheblöcke sind in Juxmem persistent, so daß sich Clients vom System abkoppeln können. Weitere Knoten können sich mit bereits allozierten Speicherblöcken verbinden und hierüber Daten austauschen.

Juxmen hat eine etwas andere Zielsetzung als diese Arbeit, da es sich bei Juxmen nicht um einen streng konsistenten verteilten transaktionalen Speicher handelt. Die Clients sind für die Einhaltung der Datenkonsistenz selbst verantwortlich. Hierfür bietet Juxmem Sperren an, jedoch ist die Entwicklung verteilter Anwendungen damit im allgemeinen fehleranfälliger als mit Transaktionen [88]. Der gemeinsame verteilte transaktionale Speicher dieser Arbeit dagegen bietet eine strenge Konsistenz unter optimistischer Synchronisierung, weswegen Sperren nicht notwendig sind. Ebenso bekommen Clients keine Mitteilung darüber, falls sich ein bereits zugegriffener Speicherblock ändert, da von ihnen gelesene Speicherblöcke nicht der Replikation dienen. Für die Verwaltung von Speicherblöcken verwendet Juxmem eine DHT, diese enthält jedoch nur Informationen über die Speicherverwaltung, weswegen Schreibzugriffe auf Speicherblöcke keine Änderungen in dieser implizieren.

Die Replikation von Speicherblöcken innerhalb einer Datengruppe erfolgt mittels einer Aktualisierungsstrategie. Der Nachteil ist diesbezüglich, daß jeder Schreibzugriff auf ein einzelnes Speicherobjekt eine Aktualisierung sämtlicher Replikate und damit eine hohe Netzwerkbelastung verursacht, auch wenn die restlichen Replikate keinen Zugriffen durch andere Knoten unterliegen. Der gemeinsame verteilte transaktionale Speicher kann neben der Aktualisierungsauch eine Invalidierungsstrategie oder Kombination aus beidem verwenden, um die Netzwerkbelastung zu reduzieren und Zugriffsgeschwindigkeit zu erhöhen (siehe Kapitel 2.2).

Die Knoten für die Replikation legt Juxmem während der Allokation eines Speicherblocks entsprechend den Anforderungen der Anwendung statisch fest. Der gemeinsame transaktionale Speicher folgt dagegen den Interessen der Anwendung, da er Transaktionsobjekte nur auf den Knoten repliziert, welche die Objekte nutzen. Weiterhin gruppiert das Overlay-Netzwerk Knoten mit gemeinsamen Interessen, um die Kommunikation durch die Synchronisierung von Transaktionsobjekten und Serialisierung von Transaktionen lokal zu begrenzen. So läßt sich eine gute Skalierbarkeit ohne statisch festgelegte Replikation erreichen.

5.8 Zusammenfassung

Dieses Kapitel hat zunächst ein alternatives Commit-Protokoll vorgestellt, welches einen Koordinator (UP) für die Transaktionsvalidierung verwendet. Aufgrund der koordinierten Validierung mit Rückwärtsvalidierung kann der UP eingehende Transaktionen nebenläufig validieren. Außerdem verbessert die Erweiterung um die Vorwärtsvalidierung als kombinierte Validierungsstrategie die Leistung, da Knoten regelmäßig über geänderte Objektinhalte informiert werden. Dies führt zu weniger Transaktionskollisionen und ermöglicht im Konfliktfall kürzere Transaktionslaufzeiten, da Knoten Transaktionskonflikte nicht ausschließlich mittels Commit-Anfragen an den UP erkennen. Da sowohl das P2P- als auch UP-Protokoll einzeln für eine hohe Anzahl von Knoten eine begrenzte Skalierbarkeit aufweisen, wurden diese zusammen mit einem strukturierten Overlay-Netzwerk zusammengeführt. Knoten mit gleichen Interessen finden sich in Gruppen zusammen, die eine UP-Validierung von Transaktionen durchführen. Der UP einer Gruppe agiert gleichzeitig als dessen Superpeer und koordiniert die Kommunikation zwischen dem Netzwerk und der eigenen Gruppe. Die Superpeers der einzelnen Gruppen kommunizieren untereinander direkt über das P2P-Protokoll, so daß hier eine Indirektion in der Kommunikation entfällt.

Das Overlay-Netzwerk führt in mehreren Punkten zu einer besseren Skalierbarkeit. Die Superpeers als Routingstationen leiten eingehende Netzwerknachrichten in Verbindung mit APL-Multicast – ähnlich wie in IP-Multicast-Netzwerken – bei Bedarf an mehrere Empfänger weiter. Damit reduzieren sie den Netzwerkverkehr, da die gleichen Netzwerk-Nachrichten nicht mehrfach über dieselbe logische Netzwerkverbindung laufen. Zum anderen können die Superpeers die Validierung von Transaktionen auf ihre Gruppe beschränken. So können mehrere Gruppen zur gleichen Zeit und voneinander unabhängig Transaktionen validieren, da sie in diesem Fall keine globale Transaktionsserialisierung benötigen. Dies erfolgt unter Verwendung von gruppenlokalen Commits, die als Optimierung in Kapitel 6 diskutiert werden.

Aufgrund der interessenbasierten Gruppierung von Knoten findet die Kommunikation überwiegend in den Gruppen statt, was zu einer kürzeren Validierungszeit und geringeren Latenzen beim Austausch von Nachrichten führt. Stellt das Overlay-Neztwerk eine Änderung der Zugriffsmuster oder Interessen einzelner Knoten fest, kann es sich darauf einstellen und umstrukturieren. Ein gemeinsamer verteilter transaktionaler Speicher auf Basis eines hierarchisch strukturierten Overlay-Netzwerkes ist bisher noch nicht Gegenstand der Forschung gewesen, so daß diese Arbeit die erste auf diesem Gebiet ist.

6 Techniken zur Maskierung der Commit-Latenzen

Kapitel 4 und 5 haben Commit-Protokolle für die Synchronisierung von Transaktionen in einem verteilten transaktionalen Speicher behandelt und zusammen mit einem Overlay-Netzwerk kombiniert. Der overlay-basierte mehrstufige Commit mit den beiden Commit-Protokollen auf Basis von P2P und UP ermöglicht gegenüber der einstufigen Synchronisierung eine bessere Lastverteilung und höhere Skalierbarkeit mit vielen Knoten. Dies setzt vom Overlay-Netz allerdings voraus, den Nachrichtenverlauf explizit steuern zu können und sofern möglich, regional zu beschränken (siehe Kapitel 6.2.2).

Ein ausschlaggebender Faktor, der die Leistungsfähigkeit netzwerkbasierter Kommunikationssysteme beschreibt, ist deren Latenz. Diese führt bei der Netzwerkkommunikation zu längeren Blockierungsphasen, was sich letztendlich in einem verminderten Transaktionsdurchsatz und verlangsamten Programmablauf äußert. Daher gilt es, die Latenzen zu maskieren oder sofern möglich, die Kommunikation zu vermeiden.

6.1 Multiversion-Objekte

Kapitel 3.2.2 hat bereits gezeigt, daß Knoten Transaktionen im Konfliktfall eventuell verzögert abbrechen müssen, da in Drittcode wie Bibliotheken und Systemfunktionen Operationsergebnisse auf nicht transaktionalem Speicher bei einer Transaktionsrücksetzung erhalten bleiben und so beispielsweise zu einer Verklemmung führen können. Diese Problematik läßt sich mit einem verzögerten Transaktionsabbruch oder Auslassen des Transaktionsabbruchs im Fall von Nur-Lese-Transaktionen (siehe Kapitel 6.1.1) umgehen. Diese Vorgehensweise ist allerdings riskant, da sie zu einem unerwünschten Programmverhalten (bis hin zum Programmabsturz) führen kann. Dieser Fall kann eintreten, wenn solche Transaktionen auf mehreren Objekten operieren, deren Änderungen mindestens zwei unterschiedlichen Transaktionen entstammen, obwohl diese die Objekte nur atomar geändert haben. Damit verletzt die laufende Transaktion die Konsistenzeigenschaft von Transaktionen und arbeitet auf einem inkonsistenten Zwischenzustand des transaktionalen Speichers.

Angenommen eine laufende Transaktion auf Knoten N_1 liest ein Objekt x_n aus einer bereits abgeschlossenen Transaktion mit der Commit-Nummer n eines Knotens N_2. Nun schließt der Knoten N_2 eine Transaktion mit der Commit-Nummer $n + 1$ ab, in der er mittels Schreiboperationen die Objektversionen x_{n+1} und y_{n+1} erzeugt. Greift der Knoten N_1 nun auf Objekt y zu, fordert er automatisch die Objektversion y_{n+1} an, da er wegen des Commits durch Knoten N_2 kein aktuelles Replikat besitzt. Somit verarbeitet N_1 Objekte aus zwei unterschiedlichen Transaktionen. Schließt die laufende Transaktion erfolgreich ab oder ändert eventuell den gegenwärtigen lokalen Teilzustand der Applikation, kann dies dennoch zu einem unerwünschten Applikationsverhalten oder Programmabsturz führen (siehe Abbildung 6.1).

Diese Situation läßt sich vermeiden, indem Knoten von dem bisherigen Grundsatz hinsichtlich der Verarbeitung von Commit-Benachrichtigungen abweichen dürfen. Generell ist es nicht erlaubt, daß ältere Objektversionen und Commit-Benachrichtigungen neuere ersetzen, um die

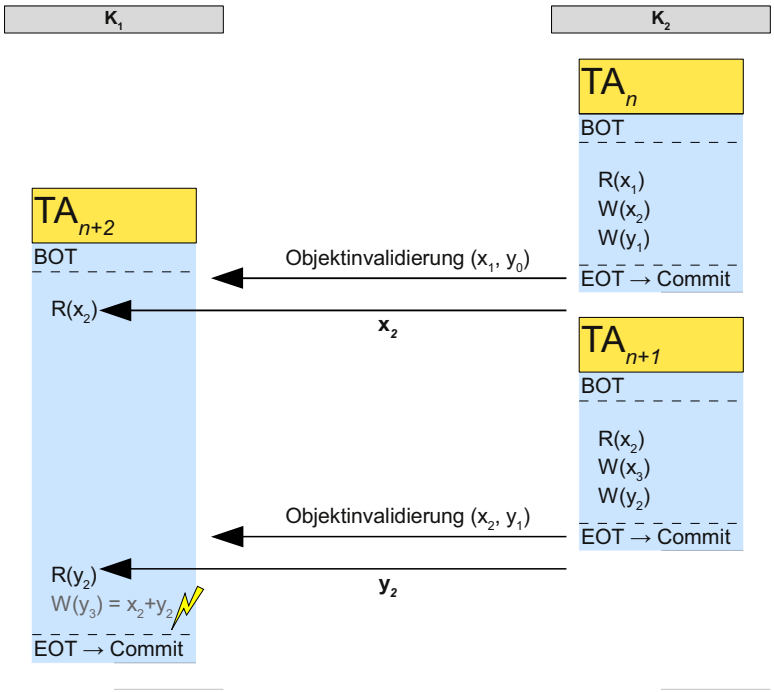


Abbildung 6.1: Objektanforderung aus unterschiedlichen Transaktionen.

Datenkonsistenz beim Commit nachfolgender Transaktionen nicht zu verletzen. Bei Nur-Lese-Transaktionen und verzögert abzubrechenden Transaktionen ist dies dennoch erlaubt, da diese Transaktionen keine Änderungen auf dem transaktionalen Speicher durchführen und folglich auch nicht die Datenkonsistenz verletzen können. Steht eine laufende Transaktion aufgrund einer eingehenden Commit-Benachrichtigung in Konflikt, definiert dessen Commit-ID eine obere Grenze bezüglich abgeschlossener Transaktionen, von denen die laufende Transaktion Daten (Objektreplikate) lesen darf. Eine laufende Transaktion darf deshalb keine Objektreplikate anfordern, die einer nachfolgenden als der konfliktverursachenden Transaktion entstammen, sondern muß auf eine ältere Version der angeforderten Objektreplikate zurückgreifen. Daher müssen die Transaktionen gegebenenfalls erst die Verarbeitung der zum Objektreplikat zugehörigen Commit-Benachrichtigung abwarten, falls Objektreplikat und Commit-Benachrichtigung in vertauschter Reihenfolge eintreffen. Verzögerte Transaktionsabbrüche und Nur-Lese-Transaktionen erfordern daher, daß das verteilte transaktionale Speichersystem mehrere Versionen desselben Objekts vorhält. Es muß erzeugte Objektversionen bereits zurückliegender Commits in einem vorgegebenen Zeitrahmen vorhalten, zu denen laufende aber noch nicht abgeschlossene Transaktionen einen Bezug (durch mindestens ein zugegriffenes Objekt) haben. Sind angeforderte Objektversionen im System nicht mehr verfügbar, muß die Transaktion sofort abbrechen – dies gilt auch für Nur-Lese-Transaktionen (siehe Kapitel 6.1.1). Falls sich die Transaktion in einem kritischen Abschnitt befindet (zum Beispiel in einer Systemfunktion in Verbindung mit der verzögerten Transaktionsabbruchstrategie), darf sie nicht unmittelbar abbrechen, hier greift der Checkpointing-Mechanismus [34] und setzt die Anwendung dann zum letzten Checkpoint zurück.

Ein Replikatmanager hält hierfür auf jedem Knoten unterschiedliche Versionen eines Objekts in einem Zwischenspeicher (Cache) vor. Fordert eine Transaktion ein Replikat eines Objekts an, versucht der Replikatmanager zunächst eine Version aus dem Zwischenspeicher zurückzuliefern. Hat der Replikatmanager kein Replikat des angeforderten Objekts oder keine für diese

Transaktion gültige Version zwischengespeichert, fordert er ein Replikat von einem anderen Knoten an und liefert es an die laufende Transaktion aus. Ebenso speichert er das Replikat in seinem Zwischenspeicher, um dieses auf Anforderung an andere Knoten ausliefern zu können. Eine detaillierte Betrachtung einer Replikatverwaltung durch einen Replikationsmanager findet sich in [81].

Die lokale Speicherung mehrerer Versionen eines Objekts kann auf den einzelnen Knoten zu einem hohen Speicherverbrauch führen. Da sich aufeinanderfolgende Objektversionen oftmals nur partiell voneinander unterscheiden, verbraucht die dort enthaltene Redundanz unnötig viel Speicher. Speichert man nur die Unterschiede (Diffs) zweier aufeinanderfolgender Objektversionen, spart dies Speicher. Da die zuletzt geänderte Version eines Objekts vollständig im Speicher vorhanden ist, bietet es sich an, die partiellen Änderungen als Rückwärtsdiffs (Diffs, die durch Anwendung zur Vorgängerversion eines Objekts führen) zu speichern.

6.1.1 Nur-Lese-Transaktionen

Bisher hat die Arbeit nur Transaktionen behandelt, die sowohl Objekte gelesen als auch geschrieben haben. Jedoch können genauso Transaktionen auftreten, die nur gelesene Objekte beinhalten. Derartige Transaktionen rufen also keine Änderungen im transaktionalen Speicher hervor und können daher auch nicht die Konsistenz des gemeinsamen verteilten transaktionalen Speichers verletzen respektive Transaktionsabbrüche auf anderen Knoten verursachen. Daher brauchen diese Transaktionen ihren Commit anderen Knoten auch nicht durch eine Commit-Benachrichtigung mitteilen, zudem die dort enthaltene Schreibmenge leer wäre. Die Auswirkungen dieser Transaktionen auf den gemeinsamen verteilten transaktionalen Speicher entspricht dem Verhalten, als hätte die Transaktion niemals stattgefunden. Dennoch können diese Art von Transaktionen mit anderen Transaktionen in Konflikt stehen, falls eine laufende Nur-Lese-Transaktion bereits Objekte gelesen hat, für die zwischenzeitlich eine andere Transaktion Änderungen propagiert. Insofern hat die Nur-Lese-Transaktion veraltete Daten gelesen und müßte daher abbrechen. Aufgrund der fortwährend garantierten Konsistenz dürfen solche Transaktionen dennoch zu Ende laufen.

Nur-Lese-Transaktionen ermöglichen verteilten Anwendungen eine bessere Skalierbarkeit. Sofern die innerhalb der Transaktion gelesenen Daten keinen Anspruch auf starke Konsistenz erheben, sprich die Verarbeitung nicht aktueller Daten ist aus anwendungsspezifischer Sicht unkritisch, vermindert ein Abbruch mit anschließender Neuausführung der Transaktion unnötig die Skalierbarkeit der Anwendung. Aus diesem Grund bietet es sich an, Transaktionen in diesem Fall trotz Lesekonflikten nicht abzubrechen. In Fällen, in denen eine Anwendung starke Konsistenz fordert, muß das System Nur-Lese-Transaktionen im Konfliktfall dennoch abbrechen. Trotz der ausbleibenden Änderungen am transaktionalen Speicher und der nicht versendeten Commit-Benachrichtigung, muß der transaktionsabschließende Knoten das Token anfordern. Ansonsten können eventuelle Konflikte aufgrund noch nicht verarbeiteter oder im Transit befindlicher Commit-Benachrichtigungen von anderen Knoten unentdeckt bleiben.

6.2 Lokaler Commit von Transaktionen

6.2.1 Knotenlokaler Commit

Bisher muß ein Knoten bei Verwendung des P2P-Protokolls für jeden Commit zunächst das Commit-Token anfordern, um anschließend den erfolgreichen Abschluß seiner Transaktion im Netz veröffentlichen zu können. Dies verursacht gerade in Weitverkehrsnetzen relativ lange

Wartezeiten durch Tokenanfragen. Wartezeiten entstehen auch, falls ein Knoten noch auf fehlende Commit-Benachrichtigungen für die korrekte Validierungsreihenfolge warten muß (siehe Kapitel 4.3.3). Beim UP-Protokoll muß ein Knoten dagegen auf die Beantwortung seiner Commit-Anfrage warten.

Bei einer genaueren Betrachtung des Commit-Protokolls wird ersichtlich, daß nicht bei jedem Commit eine Kommunikation über das Netzwerk notwendig ist. Eine Commit-Nachricht führt auf entfernten Knoten zum Abbruch konfliktverursachender Transaktionen und unter Verwendung des Invalidierungsverfahrens zur Invalidierung von Änderungen betroffener Objektreplikate. Das einzige gültige Replikat eines Objekts existiert dann nur noch auf dem Knoten, der das Objekt in einer Transaktion zuletzt geändert hat. Andere Knoten müssen erst wieder eine neue Kopie des gewünschten Objekts anfordern. Dies bedeutet, ein Knoten weiß nach der Änderung eines Objekts genau, ob weitere Replikate dieses Objekts bestehen oder nicht. Stellt ein Knoten bei einem bevorstehenden Commit seiner Transaktion fest, daß von den in der Transaktion geänderten Objekten keine Replikate außerhalb seines Kontexts vorhanden sind, so kann er die Transaktion auch lokal festschreiben. Er muß in diesem Fall dann weder das Commit-Token anfordern noch die Schreibmenge an alle anderen Knoten schicken. Der lokale Commit kann somit als Bestandteil vorheriger Transaktionen des Knotens angesehen werden, die auf diesen Daten gearbeitet haben.

Der Fall eines lokalen Commits tritt immer dann auf, wenn ein Knoten in einer Transaktion Objekte ändert, die er bereits in einer früheren Transaktion geschrieben hat und zwischen beiden Transaktionen kein anderer Knoten ein Replikat eines geänderten Objekts angefordert hat. Lokale Commits erhöhen damit weiter die Skalierbarkeit, da die zeitintensive Netzkommunikation entfällt. Dies fällt umso mehr ins Gewicht, je höher die Latenz im Kommunikationsnetzwerk ist.

6.2.2 Gruppenlokaler Commit

Das Prinzip des lokalen Commits eines Knotens läßt sich ebenso auf den Superpeer für gruppenlokale Commits übertragen. Während sich ein Knoten für jedes seiner Objektreplikate gemerkt hat, ob dieses ebenso auf anderen Knoten repliziert ist, verfährt der Superpeer in gleicher Weise. Er merkt sich, ob Objektreplikate auch außerhalb seiner Gruppe repliziert sind. Hat eine abzuschließende Transaktion nur Objekte geändert, die nicht außerhalb der Gruppe repliziert sind, kann der Commit innerhalb der Gruppe ohne Kommunikation zu Knoten, die sich außerhalb der Gruppe befinden, erfolgen.

Fordert ein externer Knoten ein Objektreplikat von einem Gruppenteilnehmer an, so läuft die Kommunikation über den Superpeer der Gruppe als nachrichtenvermittelnder Knoten. Folglich vermerkt er für das jeweilige Objekt, daß dieses außerhalb der Gruppe repliziert ist. Im Falle eines erfolgreichen Transaktionsabschlusses von einem Gruppenknoten löscht der Superpeer die Vermerke für alle durch die Transaktion geänderten Objekte. War für eines der Objekte ein externer Replikationsvermerk verzeichnet, leitet der Superpeer die auftretende Commit-Nachricht für die Gruppenteilnehmer auch an das gesamte Overlay-Netzwerk weiter, damit die externen Knoten ihre Replikate der in der Transaktion geänderten Objekte invalidieren. Da sowohl die ein- als auch ausgehende Kommunikation über Superpeers (Routing) läuft, kommunizieren Superpeers mit anderen Gruppen nur über deren Superpeers. Eine Ausnahme besteht, wenn der Superpeer gleichzeitig ein normaler Teilnehmer einer übergeordneten Gruppe ist, wie das bei einer mehrstufigen hierarchischen Overlay-Topologie der Fall ist. Die Superpeers können anhand der ausgetauschten Daten immer feststellen, welche Objekte außerhalb ihrer Gruppe repliziert sind.

Ein wichtiger Unterschied besteht zu knotenlokalen Commits. Wenn ein externer Knoten ein nicht außerhalb der Gruppe existierendes Objektreplikat anfordert, kann er aufgrund der Ne-

benläufigkeit innerhalb der Gruppe eine veraltete Version ausgeliefert bekommen. Es ist Aufgabe des Superpeers der jeweiligen Gruppe, dies zu unterbinden, indem er jedes Objektreplikat, welches seine Gruppe verläßt, zuvor auf Gültigkeit überprüft.

Gegeben sei folgendes Szenario. Das Transaktionsobjekt x liegt in der aktuellen Version *5* vor und ist nicht außerhalb der Gruppe repliziert, dafür aber auf mehreren Knoten innerhalb der Gruppe. Nun fordert ein externer Knoten dieses Replikat an. Der Superpeer leitet die Anfrage an einen der Gruppenknoten weiter. Zwischenzeitlich schließt ein anderer Knoten der Gruppe eine Transaktion – mit Schreibzugriff auf Objekt x – ab. Der Superpeer ordnet dem geschriebenen Objekt bei der Validierung der Transaktion daraufhin die neue Versionsnummer *6* zu. Bevor die anderen Knoten der Gruppe die Invalidierungsnachricht erhalten, schickt der Knoten, welcher die externe Objektanfrage erhalten hat, noch die veraltete Objektversion *5* über den Superpeer (Nachrichtenvermittler) an den externen anfragenden Knoten. Erhält der anfragende Knoten die veraltete Objektversion *5*, würden dieser und das restliche Overlay-Netzwerk diese als gültig annehmen, da die Transaktion in der Gruppe nur lokal abgeschlossen hat, weil das Objekt zu dieser Zeit nirgendwo außerhalb der Gruppe repliziert war. Schließt der externe Knoten nun seinerseits eine Transaktion mit der veralteten Objektversion ab, kommt es zur Dateninkonsistenz, da der Superpeer seiner Gruppe weiterhin von der Gültigkeit der Objektversion ausgeht.

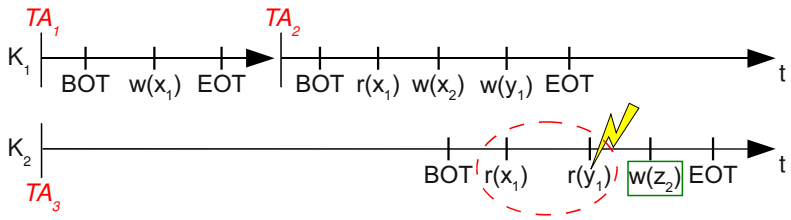
Da der Superpeer die Kommunikation zwischen einer Gruppe und dem restlichen Overlay-Netzwerk kontrolliert, muß er vor der Weiterleitung von Objektreplikaten aus seiner Gruppe deren Versionsnummer mit der des letzten Commits vergleichen. Ist das auszuliefernde Objekt veraltet, weil zwischenzeitlich ein anderer Knoten der Gruppe in einer erfolgreich validierten Transaktion dieses Objekt geschrieben hat, darf der Superpeer dieses nicht ausliefern. In diesem Fall muß der Superpeer eine aktuelle Objektversion von einem Gruppenteilnehmer nachfordern, um die Anfrage beantworten zu können, oder den externen Knoten informieren, eine neue Anfrage zu stellen. Trifft die Validierungsanfrage ein, nachdem der Superpeer das Objekt bereits ausgeliefert hat, kann die Transaktion nicht mehr lokal sondern muß global abschließen und eine Commit-Benachrichtigung an das Overlay-Netzwerk verschicken.

6.2.3 Lokaler Commit mit veralteten Objektreplikaten

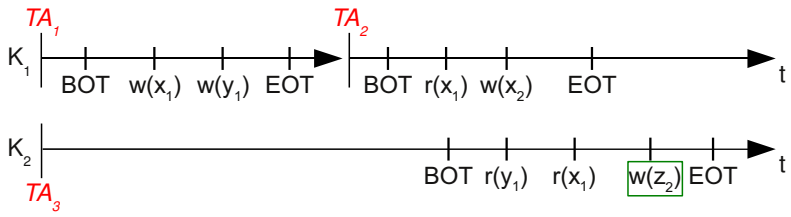
Wie in Kapitel 2.5 beschrieben, dürfen Transaktionen keine veralteten Daten gelesen haben, damit sie erfolgreich abschließen können, da sie ansonsten nicht konfliktfrei sind. Gelesene und geschriebene Transaktionsobjekte, die nicht außerhalb eines Knotens repliziert sind, braucht das System für eine Konfliktprüfung nicht heranzuziehen. Zum einen können diese Objekte nicht veraltet sein, da sie aus einer früheren Transaktion desselben Knotens stammen müssen, und zum anderen kann ein anderer Knoten diese nicht gelesen haben. Deshalb muß die Validierung nur die Objekte in die Konfliktprüfung einbeziehen, welche die abzuschließende Transaktion ausschließlich lesend zugegriffen hat.

Die Konflikterkennungsregeln (siehe Kapitel 2.5) sind hinreichend für einen erfolgreichen Transaktionsabschluß. Hat eine Transaktion veraltete Daten gelesen, darf sie dennoch erfolgreich abschließen, falls die gelesenen Objekte einem konsistenten Zustand entsprechen, sprich die Konsistenzeigenschaft nicht verletzt ist. Abbildung 6.2 zeigt zwei Beispiele, in denen eine lokal abzuschließende Transaktion veraltete Daten gelesen hat. Ein Knoten K_1 führt nacheinander die Transaktionen TA_1 und TA_2 aus, während ein weiterer Knoten K_2 eine weitere Transaktion TA_3 nebenläufig ausführt. Der lokale Commit soll einzig für Transaktion TA_3 stattfinden, da nur das geschriebene Transaktionsobjekt z_1 (grüne Umrandung) nicht auf anderen Knoten repliziert ist.

Die gelesenen Objekte x_1 und y_1 der Transaktion TA_3 in Abbildung 6.2a repräsentieren einen ungültigen Zustand des verteilten transaktionalen Speichers, da die Objektreplikate aus zwei



(a) Inkonsistenter gelesener Zustand



(b) konsistenter gelesener Zustand

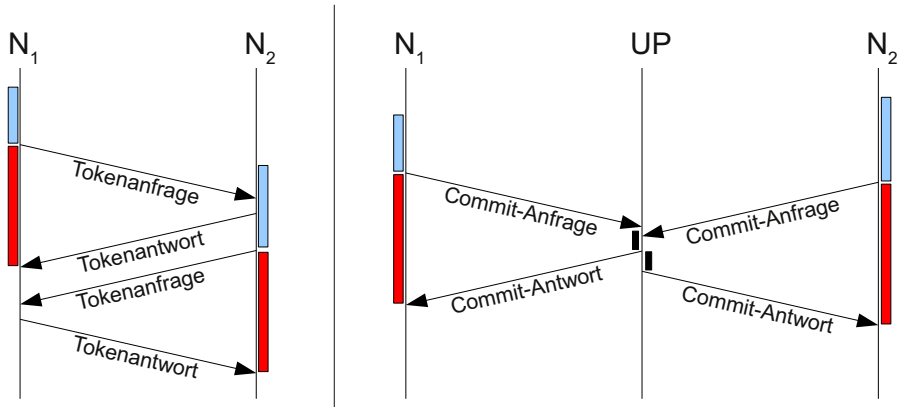
Abbildung 6.2: Lokaler Commit von Transaktionen mit veralteten gelesenen Transaktionsobjekten.

unterschiedlichen Transaktionen entstammen, diese aber beide in der letzten Transaktion TA_2 des Knotens K_1 atomar geschrieben wurden. Die gelesenen Transaktionsobjekte der Transaktion TA_3 in Abbildung 6.2b repräsentieren dagegen einen konsistenten Zustand, da beide Objekte zwar zwei unterschiedlichen Transaktionen entstammen, aber in diesen nicht zusammen atomar geändert wurden.

6.3 Kaskadierte Transaktionen

Der Commit von Transaktionen im verteilten System kann aufgrund der Serialisierung und der Netzwerklatenz den Transaktionsdurchsatz mindern. Beim P2P-Protokoll ist hierfür maßgeblich der Tokenaustausch für die Transaktionsserialisierung verantwortlich, beim UP-Protokoll dagegen die Verarbeitungsgeschwindigkeit der in der Warteschlange eintreffenden Commit-Anfragen. Beim hybriden Commit im Overlay-Netzwerk sind hierfür beide Serialisierungstechniken ursächlich. Ein transaktionsabschließender Knoten blockiert nach dem Eintritt in die Commit-Phase so lange, bis er entweder das Token oder eine positive beziehungsweise negative Commit-Antwort erhalten hat. In dieser Zeit kann er in demselben Thread keine weiteren Transaktionen ausführen. Abbildung 6.3 verdeutlicht dies anhand des Protokollablaufs für das P2P- und UP-Commit-Protokoll in einem Sequenzdiagramm. Die blau gefärbten Balken in der Abbildung kennzeichnen den zeitlichen Transaktionsverlauf, die rot gefärbten dagegen die zeitliche Verzögerung, die beim Commit auftritt. Die schwarzen Balken in Abbildung 6.3b kennzeichnen die Verzögerung durch die Validierung auf dem UP-Knoten, die bei vielen gleichzeitigen Commit-Anfragen die Verzögerung auf den Clients zusätzlich erhöhen kann.

Die Verlagerung des Commit-Vorgangs in einen separaten Anwendungsthread ermöglicht es dem jeweiligen Thread, direkt mit dem weiteren Programmablauf fortzufahren, anstatt während der Commit-Phase zu blockieren. Vorausgesetzt, der nebenläufig abzuschließenden Transaktion schließt sich direkt eine nachfolgende Transaktion an (siehe Kapitel 6.3.2). Ist der im Hintergrund laufende Commit-Vorgang erfolgreich, das heißt die Transaktion konfliktfrei, hat der Thread die Wartezeit sinnvoll genutzt. Zeigt sich während des Commit-Vorgangs ein Konflikt, der die Transaktion zum Abbruch zwingt, so muß der betroffene Thread die Transaktion entweder erneut ausführen oder ohne Neuausführung endgültig abbrechen (siehe Kapitel



(a) P2P-Commit *(b)* UP-Commit

Abbildung 6.3: Verzögerungen beim Commit von Transaktionen.

3). Bei einem nebenläufigen Commit entsteht immer ein Gewinn, da das transaktionale Speichersystem die Wartezeit sinnvoll nutzen kann. Deshalb kann die Kaskadierung (Verkettung) von Transaktionen den Leistungsdurchsatz des verteilten transaktionalen Speichers verbessern. Dieser wirkt sich umso mehr aus, je länger die Commit-Phase einer Transaktion dauert und je geringer die Konfliktrate ist.

6.3.1 Kaskadierter Transaktionsabbruch

Steht ein Transaktionsabbruch wegen eines Konflikts bevor, so hat der zugehörige Thread eventuell schon weitere Transaktionen ausgeführt, für die ebenfalls ein Commit anhängig ist, oder er befindet sich gerade in der Ausführung nachfolgender Transaktionen. In einem solchen Fall kommt es zu einem kaskadierten Transaktionsabbruch *(engl. cascading abort)*. Ein Thread muß demnach auch jene seiner nachfolgenden Transaktionen abbrechen, die mit mindestens einer übergeordneten abzubrechenden Transaktion in Beziehung stehen. Zwei Transaktionen stehen miteinander in Beziehung, sofern eine Transaktion geschriebene Transaktionsobjekte einer übergeordneten Transaktion gelesen hat. Dies ist deshalb der Fall, da nachfolgende Transaktionen bereits auf geänderte Transaktionsobjekte übergeordneter Transaktionen arbeiten. Demnach sind die Änderungen, auf die nachfolgende Transaktionen basieren, ungültig und führen zu einem verketteten Transaktionsabbruch.

Eine nachfolgende Transaktion muß ebenfalls abbrechen, wenn sie geschriebene Transaktionsobjekte einer übergeordneten abzubrechenden Transaktion statt zu lesen nur geschrieben hat. Dies liegt darin begründet, da ein Schreibzugriff auf ein Transaktionsobjekt aufgrund der transaktionalen Objektzugriffserkennung (siehe Kapitel 2.5) immer einen Lesezugriff impliziert.

Es ist zu beachten, daß die Beziehung zwischen Transaktionen auch threadübergreifend gilt. Das heißt, eine abzubrechende Transaktion kann auch nachfolgende Transaktionen zum Abbruch zwingen, die einem anderen Thread unterliegen, da alle Threads eines Prozesses auf demselben Speicher und somit auch denselben Transaktionsobjekten arbeiten. Weiterhin gilt für angeforderte Objektreplikate fremder Rechner, daß diese nur Replikate von erfolgreich abgeschlossenen Transaktionen erhalten dürfen, ansonsten würde dies der Isolationseigenschaft von Transaktionen widersprechen und kann die Datenkonsistenz verletzen. Deshalb muß die Objektverwaltung immer die älteste Schattenkopie ausliefern, sofern eine existiert.

Abbildung 6.4 stellt verkettete Transaktionen zweier Threads dar und zeigt, wie sich ein Abbruch einer Transaktion auch auf andere auswirkt. Die rote Linie (gestrichelte Linie im Falle

einer verzögerten Rücksetzung) zeigt dabei an, zu welcher Transaktion zurückgesetzt wird. Die erste Transaktion in Thread *1* muß wegen eines Konflikts abbrechen, daraufhin müssen auch die zweite und vierte Transaktion aufgrund veralteter gelesener Daten abbrechen. Die zweite und dritte Transaktion in dem zweiten Thread müssen auch abbrechen, da die dritte Transaktion von der zweiten abhängig ist und bei der zweiten Transaktion zudem eine Abhängigkeit zur ersten Transaktion aus dem ersten Thread vorliegt. Die dritte Transaktion des ersten Threads sowie die erste Transaktion des zweiten Threads sind konfliktfrei und müssen nicht abbrechen.

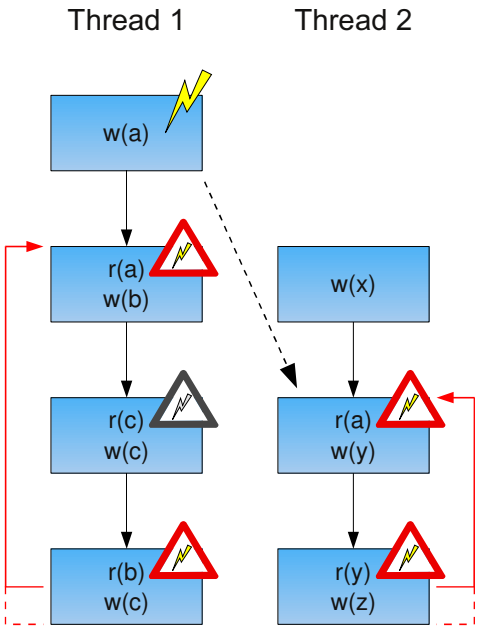


Abbildung 6.4: Kaskadierte Transaktionen: Konfliktbehaftete voneinander abhängige Transaktionen.

6.3.2 Transaktionsabhängigkeiten durch Programmfluß

Der Kontrollfluß (Programmablauf) eines Threads gibt die Verarbeitungsreihenfolge der in dem Thread ausgeführten Transaktionen vor. Daraus ergeben sich Besonderheiten bei der Transaktionsrücksetzung. Entsprechend der Abhängigkeitsbeziehungen zwischen Transaktionen eines Threads kann eine Transaktion wegen eines Konflikts zum Abbruch gezwungen sein, obwohl nachfolgende Transaktionen weiterhin konfliktfrei sind. Die nachfolgenden Transaktionen müßten demnach nicht abbrechen. Allerdings stehen alle Transaktionen eines Threads über dessen Kontrollfluß miteinander in Beziehung. Die Rücksetzung einer Transaktion impliziert automatisch eine Neuausführung aller nachfolgenden Transaktionen desselben Threads, da der Thread den damit verbundenen Programmcode ebenfalls erneut ausführt[1] (siehe Abbildung 6.4). Setzt man nachfolgende konfliktfreie Transaktionen nicht zurück, verursacht dies eine mehrfache Ausführung, die dem Kontrollfluß des Threads widerspricht. Somit sieht ein Thread im Abbruchfall alle nachfolgenden Transaktionen ebenfalls als konfliktbehaftet an und macht dessen Modifikationen am transaktionalen Speicher rückgängig.

Bezüglich threadübergreifender Transaktionen gilt für einen Abbruch weiterhin die objektbasierte Abhängigkeitsbeziehung, da jeder Thread seinem eigenen Kontrollfluß unterliegt.

[1] Ein eventueller Nichtdeterminismus durch nicht transaktionale Parameter im Kontrollfluß eines Threads ist an dieser Stelle nicht berücksichtigt.

6.3.3 Nichttransaktionalisierbarer Programmcode

Verkettete Transaktionen eines Threads sind bezüglich des Kontrollflusses nicht direkt aufeinanderfolgend. Das heißt, Threads führen zwischen zwei Transaktionen eventuell auch nicht transaktionalen Programmcode aus. Bei einer verketteten Transaktionsrücksetzung würde der Thread diesen erneut ausführen. Dies ist unproblematisch, wenn sämtliche Operationen zwischen den Transaktionen idempotent sind oder sich nur auf den Threadstack beziehen. Anderenfalls, wenn der Thread beispielsweise auf statische Variablen oder Felder zugreift, kann dies den Programmablauf verändern und zu Fehlern (zum Beispiel falsche Ergebnisse oder Programmabsturz) führen. Aus diesem Grund findet eine nebenläufige Validierungs- und Commit-Phase von Transaktionen nur dann statt, wenn zwischen zwei Transaktionen kein nicht transaktionaler Programmcode steht, die Transaktionen also im Kontrollfluß direkt aufeinanderfolgend sind. Anderenfalls blockiert die Validierungsphase wie bei nicht kaskadierten Transaktionen.

Zur Erkennung, ob ein ein Thread zwischen zwei Transaktionen auch nicht transaktionalen Programmcode ausführt, bietet es sich für eine Implementierung an, eine weitere Funktion (zum Beispiel `nextTA()`) einzuführen. Diese vereint die Funktionsaufrufe `EOT()` der abzuschließenden und `BOT()` der nachfolgenden Transaktion miteinander. Der neu eingeführte Funktionsaufruf führt die Validierungs- und Commit-Phase der abzuschließenden Transaktion im Hintergrund aus und startet unmittelbar die neue Transaktion, so daß keine Blockierungsphase entsteht. Folgt einer Transaktion nicht transaktionaler Prgrammcode, schließen Transaktionen wie bei nicht kaskadierten Transaktionen mit einem blockierenden `EOT()` ab. So ist gewährleistet, daß Threads wegen zurückgesetzter verketteter Transaktionen keinen nicht transaktionalen Programmcode mehrfach ausführen.

6.3.4 Commit von kaskadierten Transaktionen

Damit kaskadierte Transaktionen ohne Blockierung validieren und erfolgreich abschließen können, sind je nach zugrundeliegendem Commit-Protokoll Anpassungen notwendig. Beim P2P-Protokoll blockiert die Anfrage nach dem Commit-Token. In dieser Zeit können weitere Transaktionen in der Warteschlange der abzuschließenden Transaktionen auflaufen. Hat ein Knoten das Token erhalten, kann er für alle Transaktionen nicht blockierend die Commit-Nachrichten verschicken und anschließend das Token wieder freigeben. Das P2P-Protokoll verlangt demnach keine Modifikation.

Unter Anwendung des UP-Protokolls schicken Knoten ihre Commit-Anfragen blockierend an den UP. Der Thread, welcher die Validierungs- und Commit-Phase übernimmt, schickt für abzuschließende Transaktionen unmittelbar eine Commit-Anfrage an den Koordinator. Die Blockierungsphase entsteht hier ausgehend von der Anfrage bis zum Erhalt der Antwort. Die blockierende Kommunikation ist ineffizient, da während der Blockierungsphase weitere Transaktionen aus demselben oder weiteren Anwendungsthreads auflaufen können. Demnach würden sich diese weiter aufstauen, sofern die Commit-Anfrage mehr Zeit als die Transaktion selbst in Anspruch nimmt, was bei einer hohen Netzwerklatenz häufig der Fall ist. Deshalb ist es angebracht, die Commit-Anfragen ebenfalls nicht blockierend abzuwickeln. Hierbei kann es vorkommen, daß ein Knoten eine Transaktion abbrechen muß, aber für Folgetransaktionen bereits Commit-Anfragen verschickt hat, obwohl er diese aufgrund der Verkettung ebenso abbrechen muß. Abbildung 6.5 verdeutlicht dies beispielhaft.

Knoten N_1 schickt für jede verkettete Transaktion eine Commit-Anfrage an den UP und erhält von diesem für jede Anfrage eine Antwort. Der erste Commit ist erfolgreich, während die Antwort des zweiten Commits einen Konflikt aufzeigt und somit wegen der Rückwärtsvalidierung eine Transaktionsrücksetzung erzwingt. Bis der Knoten die Antwort über die Transak-

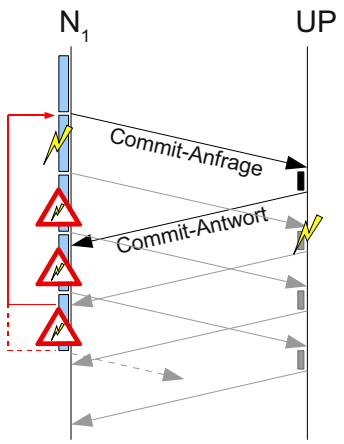


Abbildung 6.5: Nichtblockierender verketteter UP-Commit.

tionsrücksetzung erhält, hat er jedoch bereits zwei weitere Transaktionen verarbeitet und für diese Commit-Anfragen an den UP gestellt. Dessen Antworten kann der Knoten deshalb ignorieren. Der Abbruch der laufenden Transaktion mit nachfolgender erneuter Ausführung der ersten konfliktbehafteten Transaktion (roter Pfeil) erfolgt je nach Abbruchstrategie entweder unmittelbar oder verzögert nach Beendigung der gerade laufenden Transaktion (rot gestrichelte Linie).

Ein Knoten muß dem UP mitteilen, welche Transaktionen miteinander verkettet sind, damit er die Objektversionsnummern der nachfolgenden Transaktionen korrekt zuordnen kann, da dessen Commit-Anfragen noch nicht die aktualisierten Objektversionsnummern der vorherigen Transaktion enthalten dürfen. Diese erhält ein Knoten erst zusammen mit der Commit-Antwort.

6.4 Konsistenzdomänen

Konsistenzdomänen dienen einer nebenläufigen Validierung transaktionaler Speicherblöcke, ohne dabei die Datenkonsistenz zu verletzen. Ziel ist es, die Transaktionsvalidierung für eine höhere Leistungsfähigkeit des transaktionalen Speichersystems zu optimieren. Hierzu bilden die Konsistenzdomänen einen Teil der Applikationssemantik auf den Validierungsprozeß ab. Gibt die Semantik einer Anwendung vor, daß sie bestimmte unterschiedliche transaktionale Speicherblöcke nicht gleichzeitig atomar zugreift – Zugriffe auf die Speicherbereiche finden also immer in disjunkten Transaktionen statt – kann sie die Validierungs- und Commit-Phase dieser Transaktionen parallelisieren. Diesbezüglich alloziert die Anwendung die transaktionalen Speicherblöcke in unterschiedlichen Konsistenzdomänen, wobei jeder Speicherblock genau einer Konsistenzdomäne zugeordnet ist.

Vergleichbar ist dieser Ansatz mit Multithreading-Anwendungen mit mehreren disjunkt zugegriffenen Datenstrukturen, in der je zwei Threads immer auf dieselbe Datenstruktur zugreifen. An dieser Stelle wäre es ineffizient, eine globale Sperre für die Synchronisierung der Threads zu verwenden, was genau einer Konsistenzdomäne entsprechen würde. Stattdessen hat jede Datenstruktur eine eigene Sperre, was der Allozierung der transaktionalen Speicherblöcke in unterschiedlichen Konsistenzdomänen entspricht. Aus Sicht des Commit-Protokolls erhält jede Konsistenzdomäne unter Anwendung des P2P-Protokolls ein eigenes Token. Beim

UP-Protokoll ist jede Konsistenzdomäne einem dedizierten UP zugewiesen. In einem Overlay-Netzwerk (siehe Kapitel 5) kann analog zu mehreren Token und UPs eine Gruppierung von Knoten auch pro Konsistenzdomäne erfolgen. So können Knoten für jede Konsistenzdomäne Mitglied in einer anderen Gruppe sein. Auch können Knoten für bestimmte Gruppen als Superpeer agieren, während sie in einer anderen Gruppe einen gewöhnlichen Gruppenteilnehmer repräsentieren.

Konsistenzdomänen eignen sich beispielsweise für netzwerkbasierte Spiele, in denen viele Spieler zusammentreffen (engl. Multiplayer Online Game (MOG)). So können Konsistenzdomänen voneinander unabhängige Zonen des Spiels hinsichtlich der Serialisierung transaktionaler Speicherzugriffe voneinander isolieren [54, 51].

6.4.1 Konsistenzdomänenübergreifende Transaktionen

Würde ein Knoten innerhalb einer Transaktion auf transaktionale Speicherblöcke mehrerer Konsistenzdomänen zugreifen, so müßte er im Falle des P2P-Protokolls beim Commit die Token aller betroffenen Konsistenzdomänen anfordern. Der daraus resultierende Nachteil ist eine mögliche Verklemmung. Diese kann auftreten, wenn mehrere Knoten gleichzeitig dieselben Token in unterschiedlicher Reihenfolge akquirieren wollen, so daß keiner der Knoten alle Token auf sich vereinigen kann. Fordert jeder Knoten die Token seriell in einer fest vorgegebenen Reihenfolge (zum Beispiel nach der Token-ID) an, so entsteht keine Verklemmung, allerdings dauert die serielle Tokenanforderung länger, als wenn dieser Vorgang parallel erfolgt.

Für das UP-Protokoll gilt analog, daß ein Knoten die entsprechenden UPs kontaktieren muß. Dabei kann ein Commit nur erfolgen, wenn alle UPs gleichzeitig einen erfolgreichen Commit anzeigen. Hierzu ist eine Koordination der UPs untereinander notwendig. Das Zeitverhalten ist ähnlich wie beim P2P-Protokoll, da der zeitliche Aufwand wegen der zusätzlichen Koordination ansteigt. Ein Knoten kann vor einem Commit die Anzahl betroffener Konsistenzdomänen feststellen, so daß dieser nicht zwangsläufig mehrere Token anfordern beziehungsweise UPs kontaktieren muß.

Der Speicherzugriff über mehrere Konsistenzdomänen innerhalb einer Transaktion ist nicht zu empfehlen, da dessen Zweck genau der disjunkte transaktionale Speicherzugriff ist und an dieser Stelle der Vorteil der parallelen Validierung und des parallelen Commits verlorengehen. Zum anderen steigert dies die Komplexität und Fehleranfälligkeit des Systems wegen der Anforderung mehrerer Token respektive Kontaktierung mehrerer UPs. Ebenso führt diese Vorgehensweise im Vergleich zur Anforderung der Speicherblöcke in einer gemeinsamen Konsistenzdomäne zu einem unnötigen Mehraufwand in der Kommunikation.

6.5 Fairneß

Knoten in einem verteilten System haben unterschiedliche Eigenschaften wie Netzbandbreite, Latenz, Prozessorauslastung sowie weitere hardwarespezifische Parameter. Diese beeinflussen das Leistungsverhalten des transaktionalen Speichers und der damit verbundenen Fairneß im Gesamtsystem. Kann ein Knoten seine Transaktionen aufgrund von Konflikten nur nach mehrfacher wiederholter Ausführung erfolgreich abschließen, so verlangsamt dies den Programmablauf, da der betroffene Applikationsthread durch die wiederholte Transaktionsausführung längere Zeit in demselben Programmcodeabschnitt verharrt. Dies verzögert den Thread dahingehend, den darauffolgenden Programmcode auszuführen. Bricht dieselbe Transaktion immer wieder ab, so kommt der Programmablauf zum Stillstand. Der Prozeß verhungert, da der Transaktionscode die weitere Abarbeitung des nachfolgenden Programmcodes dauerhaft verhindert.

In einem fairen System sollen Peers unabhängig von ihren Eigenschaften einen nahezu gleichen Transaktionsdurchsatz bei einer ähnlichen mittleren Transaktionslaufzeit erreichen können. Das transaktionale System muß daher die Commits und die Abbrüche von Transaktionen überwachen. Ist ein Knoten beispielsweise aufgrund vieler Transaktionsabbrüche gegenüber anderen Knoten hinsichtlich des Transaktionsdurchsatzes und dem daraus resultierenden verlangsamten Programmfortschritt benachteiligt, so muß das System ihn bei folgenden Transaktionen bevorzugen. So kann er seine Transaktionen im Konfliktfall mit höherer Wahrscheinlichkeit gegenüber anderen Knoten durchsetzen.

Damit das System Fairneß garantieren kann, muß die Leistung eines jeden Knotens hinsichtlich Transaktionen miteinander vergleichbar sein. Hierzu ist ein definiertes numerisches Maß (Leistungsindex) notwendig, welcher beispielsweise als Grundbaustein aus dem durchschnittlichen Transaktionsdurchsatz besteht. Dieser bestimmt sich aus der Anzahl der erfolgreich abgeschlossenen Transaktionen über einen Zeitraum. Es ist zu berücksichtigen, daß der Zeitraum nur eine Akkumulation der Ausführungszeiten von Transaktionen umfaßt, da Programmcode, der außerhalb von Transaktionen ausgeführt wird, keinen Beitrag zum gemeinsamen verteilten transaktionalen Speicher liefert. Weiterhin darf das System den durchschnittlichen Transaktionsdurchsatz nicht über die gesamte Programmlaufzeit berechnen, da der Einfluß eines einzelnen für die Durchschnittsberechnung herangezogenen Meßwertes mit zunehmender Anzahl von Meßwerten sinkt. Die Folge ist, daß sich das Gesamtmaß mit zunehmender Anzahl erfolgreich abgeschlossener Transaktionen nur noch sehr langsam ändert. Dies hat insbesondere den Nachteil, daß ein Vergleich mit neu zum System beitretenden Knoten schwer möglich ist, da sich deren Leistungsindex im Gegensatz sehr zügig ändert. Die Berechnung des durchschnittlichen Transaktionsdurchsatzes eines Knotens muß deswegen auf eine maximale Anzahl von Transaktionen beschränkt sein, zum Beispiel auf die letzten 100 Transaktionen. Für einen genaueren Leistungsindex gilt es weiterhin, lokale Commits, Objekt- und eventuell Tokenanforderungszeiten und Eigenschaften der Computerhardware zu berücksichtigen, die aber an dieser Stelle nicht weiter diskutiert werden.

Trotz des Leistungsindexes kann der Programmablauf eines Knotens ins Stocken geraten, falls dieselbe Transaktion wegen eines Konflikts mehrfach hintereinander abbrechen muß. Wenn eine Transaktion im Abbruchfall automatisch erneut startet, sollte die maximale Anzahl ihrer Abbrüche begrenzt sein und die Transaktion bei Überschreitung dieses Wertes sich gegenüber anderen Transaktionen bevorzugt durchsetzen, um erfolgreich abzuschließen. Hierfür zeigen die folgenden Unterkapitel in Bezug auf das gegebene Beispiel einige Lösungsansätze.

Das folgende Beispiel zeigt, daß ein Transaktionsabbruch zu einer Kettenreaktion führen kann, welche dieselbe Transaktion immer wieder abbrechen läßt. Angenommen zwei Knoten A und B führen fortlaufend die gleiche Transaktion aus und möchten zur selben Zeit ihre Transaktion abschließen. Beide Transaktionen stehen miteinander in Konflikt. Knoten A besitzt anfangs das Token, daher setzt sich seine Transaktion gegenüber der von B durch. Nachfolgende Transaktionen kann A schneller als B abschließen, da der Knoten keine Objektreplikate anfordern muß. Im P2P-Tokenverfahren muß er zudem auch kein Token anfordern. Knoten B muß nach der Rückabwicklung der fehlgeschlagenen Transaktion zunächst die gültigen Versionen der konfliktverursachenden Objekte anfordern. Dies führt wegen der Netzkommunikation zu einer verlängerten Transaktionslaufzeit gegenüber der von A. Die Transaktion auf B erfährt während ihrer Ausführung jederzeit einen Abbruch durch die kürzer dauernde Transaktion auf dem Knoten A. Knoten B würde daher ohne Gegenmaßnahmen verhungern.

6.5.1 Commit-Sperrzeit

Das P2P-Protokoll mit der First-Wins-Strategie führt keine Abstimmung bezüglich der im Konfliktfall abzubrechenden Transaktionen durch. Knoten B kann im Zuge des obigen Bei-

spiels auf folgende Weise verfahren, um Knoten A zu mehr Fairneß zu veranlassen. B muß A mitteilen, daß er bei der Transaktionsausführung aufgrund des wiederholten Abbruchs benachteiligt ist. Dies kann er entweder explizit oder bei Verwendung des Invalidierungsverfahrens auch implizit, wenn er aktuelle Objektreplikate anfordert, erledigen.

Knoten A legt daraufhin eine Sperrzeit – ähnlich wie die Kollisionsauflösung von Datenpaketen in Ethernet-Netzwerken [50] – für den Commit seiner Transaktion ein. In dieser Zeit kann B aktuelle Objektreplikate anfordern und seine Transaktion erfolgreich abschließen, da A seine Transaktion zwischenzeitlich nicht abbricht. Alternativ läßt sich die Sperrzeit auch auf wiederholt angeforderte Objektreplikate einschränken, so kann A dennoch Transaktionen abschließen, welche diese Objekte nicht verwenden beziehungsweise nicht schreibend darauf zugreifen.

6.5.2 Tokenfreigabe

Ein Knoten kann bei Verwendung des P2P-Protokolls einen wiederkehrenden Transaktionsabbruch verhindern, indem er nach mehrfachen wiederholten Abbrüchen das Token nach der Validierungsphase nicht freigibt und über die Zeit der Transaktionswiederholung behält. So verhindert der Knoten, daß andere Knoten seine Transaktion abbrechen, da nur der Tokenhalter Transaktionen abschließen darf.

Dieses Verfahren eignet sich nicht als primäres Fairneßverfahren, um wiederholt ausgeführte Transaktionen gegenüber anderen durchzusetzen. Dieses Verfahren ist selbst unfair gegenüber anderen Knoten. Es verhindert zwar, daß fremde Transaktionen eigene Transaktionen wiederholt abbrechen, aber ebenso hindert es andere Knoten daran, Transaktionen abzuschließen, die keinen Konflikt verursachen.

6.5.3 Umsortierung von Tokenanfragen

Eine weitere Möglichkeit für mehr Fairneß besteht darin, Tokenanfragen umzusortieren. Dies kann einerseits auf dem Koordinator erfolgen, wenn das System das koordinierte Tokenverfahren verwendet. Wahlweise können Knoten sowohl beim koordinierten als auch P2P-basierten Tokenverfahren (siehe Kapitel 4.4) die im Token integrierte Warteschlange von Anfragen umsortieren. Die Warteschlange im Token bietet zudem den Vorteil, daß ein Knoten, der das Token erhalten hat, aber aufgrund eines Konflikts abbrechen muß, sich vor der Tokenfreigabe erneut in die Warteschlange eintragen kann. Hierzu fügt er sich in eine Position der Liste ein und gibt das Token frei. Der Knoten kann nun seine Transaktion wiederholen, bevor er das Token erneut erhält. Zwischenzeitlich können aber auch andere Knoten ihrerseits Transaktionen abschließen.

Bevor der Knoten das Token freigibt, muß er sicherstellen, daß er in der Liste vor seinem Eintrag nur Anfragen von Knoten einsortiert, die in letzter Zeit nicht zum Abbruch seiner zu wiederholenden Transaktion geführt haben. Da der Knoten nicht voraussehen kann, welche Transaktionen andere Knoten zukünftig abschließen, sortiert der Knoten die Einträge in der Warteschlange spekulativ, um eine geringe Abbruchwahrscheinlichkeit zu erreichen. Beim koordinierten Tokenverfahren ohne Warteschlange im Token erfolgt die Umsortierung analog, indem ein Knoten seinen Wunsch auf Umsortierung explizit in seiner Tokenanfrage an den Koordinator mitteilt.

6.5.4 Umsortierung und Priorisierung von Commit-Anfragen

Beim UP-Protokoll laufen die Commit-Anfragen in der Warteschlange des UPs auf. Der UP kann die Einträge in der Warteschlange umsortieren, damit möglichst wenige Knoten ihre Transaktionen wegen Konflikten abbrechen müssen. Der UP kann die Einträge allerdings nur dann umsortieren, wenn Anfragen schneller in der Warteschlange auflaufen, als er bezüglich einer Validierung abarbeiten kann. Dies wäre beispielsweise bei vielen gleichzeitigen Anfragen der Fall und der UP (auch aufgrund der eigenen Anwendung) stark ausgelastet ist. Besser ist es dagegen, bestimmte Commit-Anfragen zu bevorzugen und andere zu verzögern.

Ein Knoten teilt dem UP in seiner Commit-Anfrage zusätzlich seine Strategie für Transaktionsabbrüche (automatische Neuausführung oder endgültiger Abbruch) mit. Stellt der UP nach einer (wiederholten) Transaktionsausführung in der Validierungsphase einen Konflikt fest, führt das wegen der Rückwärtsvalidierung zum Abbruch der anfragenden Transaktion. Der UP kann für die ursächlichen Objekte eine Sperrzeit verfügen, so daß die abgebrochene Transaktion nach ihrer Wiederholung erfolgreich abschließen kann. Transaktionen anderer Knoten, welche die gesperrten Objekte schreibend zugreifen, müssen zunächst die Sperrzeit abwarten und dürfen anschließend in die Validierungsphase eintreten. Alle anderen Transaktionen sind von der Sperrzeit nicht betroffen und dürfen unmittelbar in die Validierungphase eintreten.

6.6 Verwandte Arbeiten

Multiversion-Objekte [26] finden in Datenbanksystemen Verwendung und erhöhen dort die Nebenläufigkeit von Objektzugriffen [14], falls Transaktionen konkurrierend auf dieselben Objekte zugreifen. Transaktionen können Zugriffe so blockierungsfrei ausführen, und im Fall von Schreibzugriffen ist weiterhin die Datenkonsistenz gewährleistet. In dieser Arbeit dienen Multiversion-Objekte primär der konsistenten Sicht auf den transaktionalen Speicher, da bereits eine zwischenzeitliche Verletzung der Konsistenzeigenschaft von Transaktionen in Verbindung mit der verzögerten Abbruchstrategie zum Programmabsturz oder unerwünschtem Programmverhalten führen kann. Das liegt darin begründet, daß eine inkonsistente Sicht ungültige Programmzustände hervorrufen kann und ungültige Zeiger Daten im lokalen oder transaktionalen Speicher überschreiben können.

Kaskadierte Transaktionsabbrüche sind vor allen aus dem Datenbankbereich bekannt und können dort auftreten, wenn Transaktionen bereits von anderen noch nicht abgeschlossenen Transaktionen geschriebene Daten (Zwischenwerte) lesen [61]. Datenbanksysteme erlauben Transaktionen das Lesen von Zwischenwerten anderer Transaktionen, um die Nebenläufigkeit zu erhöhen. Transaktionen brauchen beim Zugriff auf Objekte nicht auf andere Transaktionen warten, die diese Objekte bereits verwenden. Dieser Optimierung kann zu kaskadierten Transaktionsabbrüchen führen. Bricht eine Transaktion, von denen andere Transaktionen geschriebene Zwischenwerte gelesen haben, ab, so müssen die anderen Transaktionen ebenfalls abbrechen, da die gelesenen Zwischenwerte ungültig sind. In dieser Arbeit treten kaskadierte Transaktionsabbrüche in anderer Form auf, da dies überwiegend Transaktionen desselben Threads betrifft, die aufgrund des Programmflusses aber nur seriell auftreten. Im Kontext dieser Arbeit verlaufen Transaktionen und die Validierungs- und Commit-Phase vorhergehender Transaktionen eines Threads nebenläufig. Der gemeinsame verteilte transaktionale Speicher maskiert hiermit Verzögerungen, die durch die Netzwerkkommunikation und -latenzen entstehen, des zugrundeliegenden Commit-Protokolls.

Konsistenzdomänen werden häufig in verteilten Computerspielen wie MMVEs verwendet, um Synchronisierungsaufwand und -häufigkeit zwischen den dort agierenden Individuen zu reduzieren [54]. Eine Synchronisierung von Individuen ist in der Regel nur notwendig, wenn die-

se miteinander interagieren oder Aktionen eines Avatars den Szenenausschnitt eines anderen Avatars betreffen. Für die Isolierung hinsichtlich der Synchronisierung befinden sich Objekte und Individuen in unterschiedlichen Konsistenzdomänen. Konsistenzdomänen eignen sich beispielsweise dazu, um voneinander unabhängige Szenen eines Spiels oder disjunkte Sichten von Avataren – beispielsweise, wenn sich Avatare in einer virtuellen Welt innerhalb und außerhalb eines Gebäudes befinden – zu isolieren. In dieser Arbeit finden Konsistenzdomänen für die isolierte und nebenläufige Synchronisierung von Transaktionen Anwendung. Damit vermeidet der gemeinsame verteilte transaktionale Speicher die unnötige globale Synchronisierung transaktionaler Speicherinhalte, auf die aus Transaktionssicht nur disjunkte Zugriffe erfolgen.

6.7 Zusammenfassung

Dieses Kapitel hat unterschiedliche Optimierungen des transaktionalen Speichersystems diskutiert. Transaktionen dürfen während ihrer Ausführung einen inkonsistenten Zustand aufweisen, solange die Zustände am Transaktionsanfang und -ende konsistent sind. Dies ist jedoch für replikatbasierte transaktionale Speicher nicht ausreichend, da sie neben Skalaren auch Zeiger – zum Beispiel verzeigerte Datenstrukturen wie Listen, Bäume und programmiersprachliche Objekte – enthalten können. Diese können wegen des ungültigen Zustands falsche Speicherbereiche zugreifen und schlimmstenfalls überschreiben. Diese Arbeit löst das Problem mit Multiversion-Objekten. Falls Transaktionen Objektreplikate anfordern, und neue Replikate eine inkonsistente Sicht hervorrufen würden, liefert dieser Mechanismus veraltete Objektreplikate aus, um die konsistente Sicht während der Laufzeit von Transaktionen zu gewährleisten. Im Datenbankkontext dienen Multiversion-Objekte keiner durchgängig konsistenten Sicht, sondern erzielen dort mehr Nebenläufigkeit für eine verbesserte Leistung.

Knotenlokale Commits analysieren zur Laufzeit den Austausch von Transaktionsobjekten und steuern damit für jede abzuschließende Transaktion selbständig, ob eine knotenübergreifende Validierung und globale Serialisierung notwendig ist. Die Zeit für die Validierung und damit verbundene Netzkommunikation spart der validierende Knoten ein und kann ohne Zeitverlust mit seinem Programmablauf fortfahren. Gruppenlokale Commits übertragen dieses Konzept im Overlay-Netzwerk (siehe Kapitel 5) auf die Gruppierung von Knoten. Analog analysieren die Superpeers den gruppenübergreifenden Austausch von Transaktionsobjekten und bestimmen auf Basis dieser Informationen, ob der Commit von Transaktionen innerhalb seiner Gruppe ablaufen kann. Genauso wie knotenlokale Commits benötigen gruppenlokale ebenfalls keine globale Serialisierung und erlauben demnach nebenläufige Commits in unterschiedlichen Gruppen. Nebenbei reduzieren sie die globale Netzkommunikation.

Die Kaskadierung von Transaktionen ist – vor allem in Weitverkehrsnetzen mit üblicherweise hohen Latenzen – für die Skalierbarkeit des verteilten transaktionalen Speichers wichtig, da sie die auftretenden Blockierungsphasen durch die Netzkommunikation und Validierungszeit maskieren. Der Vorteil liegt darin, daß Knoten in der Blockierungsphase mithilfe der Kaskadierung bereits weitere Transaktionen ausführen können, was Zeit spart. Auch wenn der Konflikt von nur einer Transaktion im Mittel zum Abbruch mehrerer nachfolgender führt, entsteht immer ein Gewinn, da die abzubrechenden kaskadierten Transaktionen ohne diese Technik zeitlich in die Blockierungsphase fallen und nur ein Teil aller ausgeführten Transaktionen von einer Rücksetzung betroffen ist. Im Vergleich zu dieser Arbeit tritt die Kaskadierung in Datenbanksystemen in einer etwas anderen Form auf. Transaktionen ist es dort gestattet, bereits Zwischenwerte anderer Transaktionen zu lesen, obwohl diese noch nicht abgeschlossen sind. Jedoch dient die Kaskadierung in beiden Systemen der Leistungsverbesserung durch Erhöhung der Nebenläufigkeit.

7 Evaluation

Im Rahmen des XtreemOS-Projekts wurde der Dienst *OSS* entwickelt, welcher die in dieser Dissertation behandelten Konzepte eines gemeinsamen verteilten transaktionalen Speichers umsetzt. OSS stellt diesen Speicher Anwendungen über eine API zur Verfügung und wurde für das Betriebssystem Linux als dynamische Bibliothek in der Programmiersprache *C* entwickelt. Die hier behandelten Testanwendungen für OSS gliedern sich in reale Anwendungen und Mikrobenchmarks und sind überwiegend in derselben Programmiersprache wie OSS geschrieben. Eine Ausnahme bildet die virtuelle Welt *Wissenheim Worlds*.

Mittels realer Anwendungen soll die Verwendbarkeit des gemeinsamen verteilten transaktionalen Speichers unter realen Umgebungsbedingungen untersucht werden. Hierbei entstehen beim Zugriff auf den transaktionalen Speicher aufgrund von Nichtdeterminismen nicht vorhersagbare Zugriffsmuster. Diese Tests sollen zeigen, daß der gemeinsame verteilte transaktionale Speicher auch mit solchen Zugriffsmustern gut verwendbar ist und skaliert.

Mikrobenchmarks simulieren Zugriffsmuster, um zum einen das Verhalten von OSS und dessen Commit-Protokolle bei unterschiedlich starken Konflikraten zu testen. Zum anderen erfolgt mit diesen Tests die Evaluation der unterschiedlichen OSS-Konfigurationen – beispielsweise lokale Commits und kaskadierte Transaktionen – und deren Auswirkungen auf das Gesamtverhalten des gemeinsamen verteilten transaktionalen Speichers.

7.1 Fallstudie Verteilter Raytracer

Für die Evaluation wurde OSS zusammen mit einem Raytracer untersucht, der am MIT (Massachusetts Institute of Technology) entwickelt wurde. Als Ausführungsplattform unterstützt der Raytracer nur Einprozessorsysteme, weshalb er für die Verwendung als verteilte Anwendung in Verbindung mit OSS angepaßt wurde. Die Berechnung eines Bildes erfolgt durch verfolgen reflektierter Lichtstrahlen auf eine Bildebene (Pixelraster des Bildschirms), welche Objekte einer vorgegebenen zu berechnenden Bildszene reflektieren. Die Berechnung des Bildes erfolgt pixelweise disjunkt.

Ziel dieses Tests ist es, die Berechnung eines Bildes durch Bündelung von Rechnerkapazitäten zu beschleunigen. Hierfür wird das zu berechnende Bild entsprechend der Anzahl beteiligter Knoten in mehrere Abschnitte unterteilt. Jeder Knoten berechnet einen Bildabschnitt. Die Szeneninformationen – Objekte, deren Reflektionseigenschaften und Relation zueinander – für die Berechnung eines Bildes liegen im gemeinsamen verteilten transaktionalen Speicher, genauso wie das zu berechnende Bild. Durch die Überlagerung von Abschnittsgröße und Objektgranularität kann es beim Zugriff auf den gemeinsamen verteilten transaktionalen Speicher zu Kollisionen kommen, die das System automatisch auflöst. Für die bestmögliche Leistung sind die auf die Knoten aufgeteilten zu berechnenden Bildabschnitte an Transaktionsobjekten (Speicherseitengrenzen) ausgerichtet, was einen kollisionsfreien Zugriff auf die Bildebene im transaktionalen Speicher erlaubt.

Die Testumgebung besteht aus 128 Knoten der Grid-Plattform *Grid 5000* [21]. Jeder Knoten besitzt einen Prozessor mit 4 Kernen (Intel Xeon X3440 mit 2,53 GHz) und 16 GB Hauptspei-

cher. Alle Knoten sind über ein geswitchtes Gigabit-Ethernet-Netzwerk miteinander verbunden.

7.1.1 Verteilte Bildberechnung

Der folgende Test berechnet ein Bild mit einer Auflösung von 2048 x 2048 Pixeln. Das zu berechnende Bild unterteilt sich in gleichgroße Bildabschnitte, wobei die Abschnitte an den Grenzen von Transaktionsobjekten (Speicherseitengrenzen) ausgerichtet sind (siehe Abbildung 7.1). Jedes Pixel belegt im Speicher 4 Bytes. Ein Masterknoten stellt die Szeneninformationen im verteilten Speicher zur Verfügung. Die Arbeitsknoten, die das Bild berechnen, lesen zunächst nebenläufig die Szeneninformationen aus dem verteilten Speicher und beginnen nachfolgend mit der Berechnung ihres Bildabschnitts. Ist ein Knoten mit seiner Berechnung fertig, signalisiert er dies dem Masterknoten. Sind alle Bildabschnitte berechnet, liest der Masterknoten das fertig berechnete Bild aus dem verteilten Speicher aus.

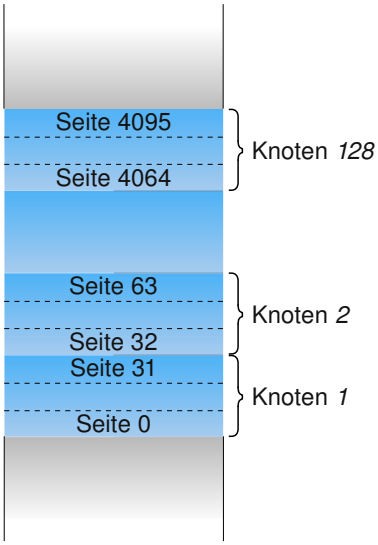


Abbildung 7.1: Speicherlayout des zu berechnenden Bildes für 128 Knoten mit Ausrichtung der Bildbereiche an Speicherseitengrenzen.

Abbildung 7.2 zeigt die Berechnungsdauer des Bildes in Abhängigkeit von der Anzahl der Arbeitsknoten. Die gemessene Zeit umfaßt die Berechnung der Bildabschnitte und das anschließende Auslesen des berechneten Bildes. Bei Verwendung des UP-Protokolls arbeitet der Masterknoten als UP. Er übernimmt nur die Validierung von Transaktionen und ist keine bildberechnende Komponente der verteilten Anwendung. Eine Bevorzugung des Masterknotens gegenüber den Arbeitsknoten aufgrund seiner gleichzeitigen Rolle als UP ist auszuschließen, da der Masterknoten während der Arbeit der Arbeitsknoten mit niedriger Priorität auf das fertig berechnete Bild wartet. Die Abbildung zeigt, daß die Berechnungszeit des Bildes mit der Anzahl der Knoten sinkt.

Bewertung

Jeder Knoten berechnet einen disjunkten Bildbereich. Da die zu berechnenden Abschnitte zudem an den Transaktionsobjektgrenzen ausgerichtet sind, treten zwischen den Arbeitsknoten keine überlappenden Zugriffe auf dieselben Transaktionsobjekte auf. Demzufolge verursacht

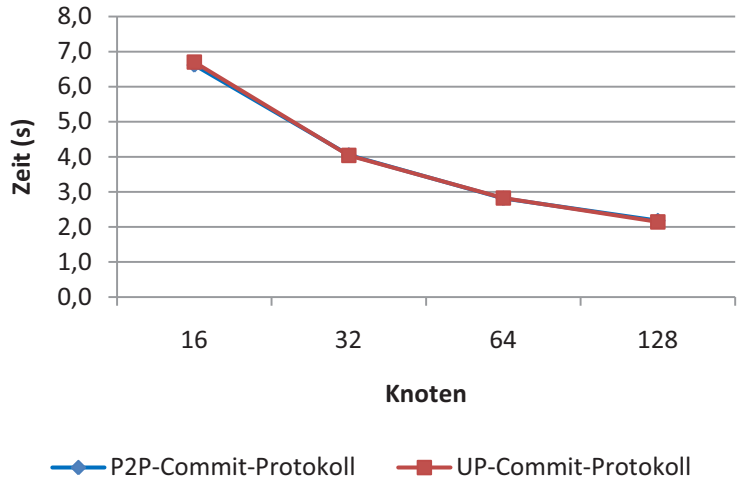


Abbildung 7.2: Berechnungsdauer einer Bildszene mit einer unterschiedlichen Anzahl von Rechnern.

die Berechnung auch keine Konflikte beim Zugriff auf den transaktionalen Speicher. Die Ergebnisse in Abbildung 7.2 zeigen, daß die Anwendung auch mit vielen Knoten sehr gut und fast linear skaliert. Die nicht vollständig lineare Skalierung ist auf den zeitlichen Mehraufwand des Commits und den seriell angeforderten Speicherseiten beim erstmaligen Zugriff zurückzuführen. Weiterhin ermittelt der Raytracer nur die Gesamtzeitdauer der Berechnung. Die Berechnungsdauer gleichgroßer Bildabschnitte weist aber aufgrund der unterschiedlichen Komplexität der Bildabschnitte eine verschieden lange Berechnungsdauer auf.

Die Implementierung des Raytracers läßt sich weiter optimieren, indem jeder Knoten statt eines großen Bildabschnitts in einer einzigen Transaktion mehrere Transaktionen mit der Berechnung kleinerer Bildabschnitte ausführt. So ergibt sich eine gleichmäßigere Verteilung der Berechnungszeit zwischen den Knoten, da jeder Knoten unterschiedlich viele Bildabschnitte in Abhängigkeit deren Komplexität berechnen kann. Greifen bei kleineren Bildabschnitten mehrere Transaktionen eines Knotens auf dieselbe Speicherseite zu, können knotenlokale Commits unnötige Netzwerkkommunikation beim Commit vermeiden und die Leistung weiter steigern. Bei dem derzeitigen Zugriffsmuster würden knotenlokale Commits dagegen keine Leistungssteigerung bewirken, da ein Knoten nicht mehrere Transaktionen auf derselben Speicherseite ausführt.

Die Tokenanforderungen beim P2P-Protokoll und Validierungsanfragen beim UP-Protokoll zeigen beide ein ähnliches Zeitverhalten, da sich die Kurven der beiden Commit-Protokolle in Abbildung 7.2 vollständig überlappen. Jeder Knoten berechnet seinen Bildabschnitt in einer Transaktion, so daß aufgrund der unterschiedlichen Berechnungsdauer nur wenige gleichzeitige Tokenanfragen entstehen und die Transaktionsanzahl insgesamt gering, die Ausführungsdauer einzelner Transaktionen dagegen aber hoch ist.

7.2 Fallstudie Wissenheim Worlds

Wissenheim Worlds ist eine virtuelle Welt für Unterhaltung und Lehre, die an der Heinrich-Heine-Universität Düsseldorf in Kooperation mit der Universität Ulm entwickelt wurde. Wissenheim [96] basiert auf der Programmiersprache Java, wurde aber für die Verwendung mit OSS mithilfe des *Small Java Compilers (SJC)* [41] unter Linux in nativen Code für die IA-32-Systemarchitektur übersetzt. Ziel ist es, die Verwendbarkeit des transaktionalen Speichers in

Verbindung mit verteilten virtuellen Welten zu untersuchen.

In der virtuellen Welt Wissenheim liegt der Szenengraph, der die Welt und die Relation aller Objekte zueinander beschreibt, im gemeinsamen verteilten transaktionalen Speicher. Jeder Knoten hat die gleiche Sicht auf den Szenengraph. Änderungen, die ein Knoten auf dem Szenengraph durchführt, werden durch die Synchronisierung beim Zugriff automatisch auf anderen Knoten sichtbar. Die Operationen auf den Szenengraph gliedern sich in fünf unterschiedliche Phasen (Animation und Physik, Extraktion und Renderung der Szene und Eingaben, die eine Änderung des Szenengraphs bewirken), die in einer Endlosschleife ablaufen. Jede dieser Phasen ist in einer Transaktion eingekapselt. Tests haben gezeigt, daß eine Umsetzung mit strenger Konsistenz für alle Phasen – wobei die Physik- und Animationsphase wegen der identischen Berechnung nur ein einzelner Knoten durchführt – nicht gut skaliert. In diesem Fall treten viele Konflikte aufgrund veralteter Daten auf, die eine erneute Ausführung von Transaktionen verursachen.

In Wissenheim ist beim Rendern einer Szene eine Transaktionsrücksetzung im Konfliktfall nicht zwingend erforderlich. Die Extraktions- und Renderphasen benötigen beispielsweise keine strenge transaktionale Konsistenz. Auch wenn während der Extraktions- und Renderphasen zwischenzeitlich Konflikte auftreten, so ist dies für Spieler kaum sichtbar, zudem gerenderte Bilder nachfolgender Schleifendurchläufe wieder auf aktuellen Daten basieren. Aus diesem Grund wurde die strenge transaktionale Konsistenz für einige Phasen abgeschwächt und die Physik- und Animationsphase auf jedem Knoten lokal ausgeführt. Alle Phasen mit Ausnahme der Eingabephase setzen ihre Transaktionen im Konfliktfall nicht zurück, sondern fahren mit der Programmausführung fort und überspringen die Commit-Phase. Schreibzugriffe auf den Szenengraph erfolgen in der Animations- und Physikphase sowie explizit durch die Spieler in der Eingabephase. Der Commit einer Transaktion und folglich die Anforderung des Tokens finden nur bei einer Änderung des Bewegungsvektors eines Avatars statt. Zudem entsteht hierbei kein Schreibkonflikt, da alle Avatare aufgrund der Speicherallozierung auf disjunkten Transaktionsobjekten liegen.

Die Synchronisierung des verteilten transaktionalen Speichers erfolgt über das P2P-Commit-Protokoll, wobei lokale Commits Transaktionen nur dann global synchronisieren, falls von einem Knoten geschriebene Objekte auf anderen Knoten repliziert sind. Kaskadierte Transaktionen finden bei diesem Test keine Anwendung. Die Testumgebung besteht aus einem AMD-Opteron-Cluster mit 8 Knoten, wobei jeder Knoten aus zwei Prozessoren mit jeweils 1,8 GHz und 2 GB Hauptspeicher (ccNUMA-Speicherarchitektur) besteht. Alle Knoten sind über ein geswitchtes Gigabit-Ethernet-Netzwerk miteinander verbunden.

7.2.1 Bewegungssimulation von Avataren

Neben dem Aufwand für die Transaktionsverwaltung erfordert die Synchronisierung des Szenengraphs im gemeinsamen verteilten transaktionalen Speicher den Austausch von Objektreplikaten und die Synchronisierung von Transaktionen, die folglich auch den Austausch des Tokens zwischen den Knoten impliziert. Der folgende Test simuliert ein Spielszenario mit einer unterschiedlichen Anzahl von Spielern, entsprechend der Anzahl der verwendeten Clusterknoten. Für die Simulation führt jeder Avatar alle 500 ms eine Bewegungsänderung durch. Mit steigender Anzahl von Knoten nehmen zeitlich gesehen auch die Modifikationen des Szenengraphs zu, wobei anzumerken ist, daß sich nur die Bewegungsvektoren der Avatare ändern, die Struktur des Szenengraphs dagegen nicht. Demnach entstehen bei der Bewegung der Avatare keine Schreibkonflikte unterschiedlicher Knoten auf dieselben Transaktionsobjekte.

Abbildung 7.3 zeigt die mittlere Bildaktualisierungsrate aller Knoten in Abhängigkeit zur Knotenanzahl. Als Meßgröße dient die Anzahl der gerenderten Bilder pro Sekunde (engl. frames

per second (fps)). Für die Messungen ist diese Größe im Leerlauf auf 200 fps normiert[1]. Die zweite Kurve zeigt die Standardabweichung der Bildaktualisierungsrate. Das Diagramm zeigt bei bis zu vier Knoten eine leicht abfallende Bildaktualisierungsrate, die ab acht Knoten etwas ausgeprägter ist. Die Standardabweichung nimmt beginnend von einem zu acht Knoten nur moderat zu.

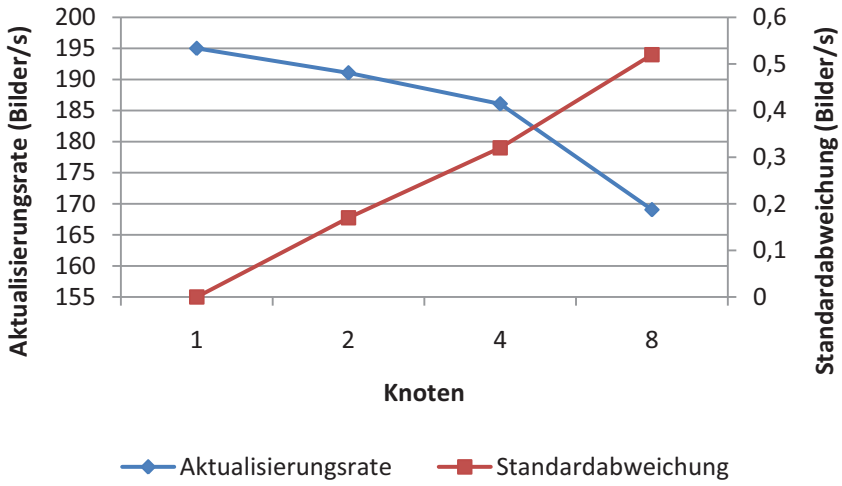


Abbildung 7.3: Bildaktualisierungsrate bei Avatarbewegung.

Bei Verwendung eines einzelnen Knotens entspricht die Bildaktualisierungsrate nahezu dem Maximalwert, da hier keine Synchronisierung des transaktionalen Speichers notwendig ist. Die Verminderung um wenige Bilder pro Sekunde hinsichtlich des normierten Ausgangswertes ist der Transaktionsverwaltung und Objektzugriffserkennung geschuldet. Ab zwei Knoten ist eine Synchronisierung der Objektreplikate im transaktionalen Speicher und die Serialisierung von Transaktionen über das Commit-Token notwendig. Die Anzahl auszutauschender Objektreplikate und zu serialisierende Transaktionen steigen mit der Anzahl der Knoten, zudem wächst der Szenengraph mit mehreren Knoten durch die zusätzlichen Avatare ebenso.

Abbildung 7.4 zeigt das simulierte Szenario, erweitert um weitere explizite Handlungen der Avatare, welche zusätzliche Modifikationen des Szenengraphs bewirken. Jeder Avatar nimmt fortlaufend ein Objekt aus der Szene auf und legt es wieder zurück. Diese Handlung läuft bezüglich der Szenengraphmodifikation kollisionsfrei ab, da sich die Handlung eines jeden Avatars auf ein eigenständiges Objekt bezieht. In diesem Szenario nimmt die Bildaktualisierungsrate bereits ab vier Knoten etwas stärker ab, liegt aber dennoch noch weit über der minimalen Rate, die für einen flüssigen Spielverlauf notwendig ist. Die Standardabweichung ändert sich gegenüber dem ersten Szenario dagegen nur wenig.

Die gemessenen Zeiten (RTT) für die Anforderung eines einzelnen Objektreplikats und des Commit-Tokens beider Szenarien sind in Abbildung 7.5 dargestellt. Die Seitenanforderungszeit bleibt unabhängig von der Anzahl verwendeter Knoten nahezu konstant. Dies liegt darin begründet, daß Knoten beim verteilten transaktionalen Speicher lediglich Replikate von Objekten nebenläufig anfordern. Im Vergleich steigt die Anforderungszeit des Tokens mit zunehmender Knotenanzahl an, da sich alle Knoten ein gemeinsames Token teilen (siehe Kapitel 4.4).

[1] Die Animations- und Physikphase laufen je 30 mal pro Sekunde, die anderen Phasen entsprechend der Bildaktualisierungsrate.

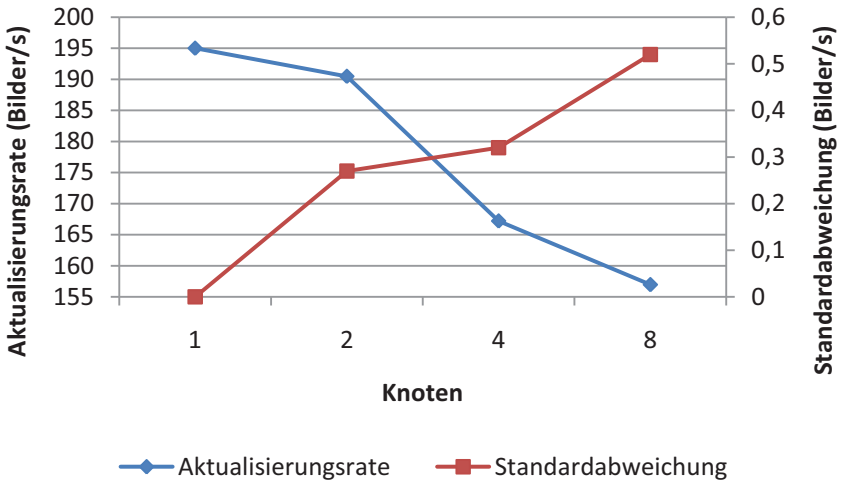


Abbildung 7.4: Bildaktualisierungsrate bei Avatarbewegung und gleichzeitiger Szenengraphmodifikation.

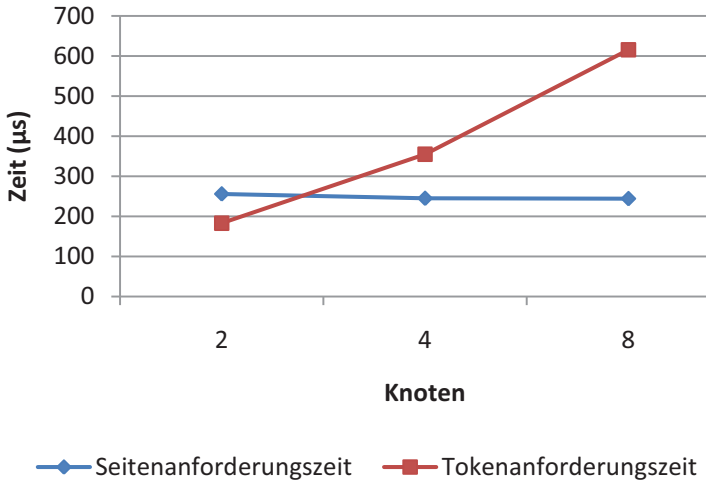


Abbildung 7.5: Seiten- und Tokenanforderungszeiten.

Bewertung

Da Transaktionen den Austausch von Objekten und des Commit-Tokens bedingen, sind die gemessenen Bildaktualisierungsraten in den Diagrammen 7.3 und 7.4 immer in Abhängigkeit der in Abbildung 7.5 dargestellten durchschnittlichen Anforderungszeit für Objektreplikate und Token zu betrachten. Die Tokenwartezeit steigt bei zunehmender Knotenanzahl an, allerdings verursacht nur die Transaktion zur Änderung der Avatarposition einen nicht lokalen Commit und folglich die Anforderung des Tokens (sofern es auf dem Knoten nicht bereits präsent ist). Deswegen bleibt die Anzahl der Tokenanforderungen mit 1–2 Tokenanforderungen/s bei unterschiedlicher Knotenanzahl konstant und ist demnach für den Einfluß auf die Bildaktualisierungsrate vernachlässigbar.

Bezüglich des Objektaustauschs bleibt die Anforderungszeit bei steigender Knotenanzahl wegen der nebenläufigen Bearbeitung von Objektanfragen mehrerer Knoten konstant. Allerdings steigt die Anzahl ausgetauschter Objekte mit zunehmender Anzahl von Avataren, da jeder Avatar seine Position in der Szene ändert und demzufolge die zugehörige Transaktion abschließt,

welche die Objektkopien auf den anderen Knoten invalidiert. Die Anzahl ausgetauschter Objekte steigt linear mit der Anzahl von Avataren, ist aber wegen ihrer geringen Anzahl (circa 24 ausgetauschte Objekte/s bei 8 Avataren) für das Gesamtzeitverhalten ebenfalls vernachlässigbar.

Bei zunehmender Anzahl von Avataren und Szenenobjekten wächst der Szenengraph an, um die zusätzlichen Objekte (ein Avatar besteht aus mehreren Objekten) zu verwalten. Für den zeitlichen Aufwand, der sich in der Reduzierung der Bildaktualisierungsrate niederschlägt, ist zum großen Teil die Zugriffsverwaltung der Speicherseiten und deren Rechteverwaltung in OSS ursächlich. Die Animations-, Physik- und Extraktionsphase durchlaufen bei jedem Aufruf jeweils den gesamten Szenengraph, welcher bei 8 Avataren ungefähr 60 Transaktionsobjekte (Speicherseiten) umfaßt. Die Renderphase durchläuft dagegen nur einen Teil des Szenengraphs.

Die Objektzugriffserkennung in OSS verursacht bei einer Lesezugriffsverletzung Kosten von rund 12 µs, bei einer Schreibzugriffsverletzung rund 16 µs. Die Traversierung des gesamten Szenengraphs in einigen Phasen akkumuliert diese Zeiten in Abhängigkeit der Anzahl von Transaktionsobjekten. Weil der Durchlauf der einzelnen Phasen zudem teilweise an die Bildaktualisierungsrate gekoppelt ist und auch die Zugriffsverletzungen auf Transaktionsobjekte bei steigender Knotenanzahl zunehmen, verbringt OSS viel Zeit mit der Zugriffsverwaltung, so daß die Bildaktualisierungsrate vermehrt sinkt. Der Aufruf des Unix-Signalhandlers selbst verursacht auf dem Cluster nur Kosten von circa 2 µs. Eine Optimierung der Zugriffsverwaltung oder eine durch Compiler unterstützte Zugriffserkennung können diese Zeiten verringern, so daß die Bildaktualisierungsrate nicht so stark abfällt. Die Standardabweichung bleibt für beide Testszenarien mit unter 0,6 Bildern/s vernachlässigbar klein, was für MMVEs wichtig ist (siehe hierzu Kapitel 7.2.2).

7.2.2 Latenzauswirkung

Der letzte Test zeigt den Einfluß der Netzwerklatenz auf die Bildaktualisierungsrate. Als Testaufbau dient hier ein Spielszenario mit nur zwei Avataren, um weitere Einflüsse auf die Aktualisierungsrate zu minimieren. Abbildung 7.6 zeigt die Bildaktualisierungsrate in Abhängigkeit von Netzwerklatenzen (RTT) zwischen 10 und 60 ms. Wie erwartet, nimmt die Bildaktualisierungsrate bei zunehmender Netzwerklatenz ab. Dies ist auch bei MMVEs zu beobachten, die nicht auf einem verteilten transaktionalen Speicher basieren, da die Netzwerklatenz generell den Nachrichtenaustausch verzögert. Die Aktualisierungsrate nimmt bei zunehmender Netzwerklatenz fast linear ab. Die Standardabweichung nimmt dagegen unerwartet stark zu, was nicht mit der Netzwerklatenz selbst begründbar ist.

Bewertung

Der negative Einfluß der Netzwerklatenz bei MMVEs läßt sich meistens nicht vollständig maskieren. Entweder sinkt auch dort die Bildaktualisierungsrate, weil die Anwendung blockierend auf notwendige Spielzustandsinformationen wartet. Oder die Anwendung maskiert die Netzwerklatenz durch Schätzung der Zustandsänderung. Eine bewährte Technik hierfür ist *Dead Reckoning* [78]. Dead Reckoning aber auch kaskadierte Transaktionen, welche die Netzwerklatenz ebenfalls maskieren, machen sich aber visuell beim Rendern von Szenen bemerkbar, entweder aufgrund kaskadierter Transaktionsabbrüche oder durch die Korrektur eines falsch geschätzten weiteren Spielverlaufs.

Ein stärkerer Einflußfaktor auf das Rendering (vergleichbar mit Audio- und Videostreaming) ist die Standardabweichung. Diese führt bei MMVEs in der Regel zu Schwankungen der Bild-

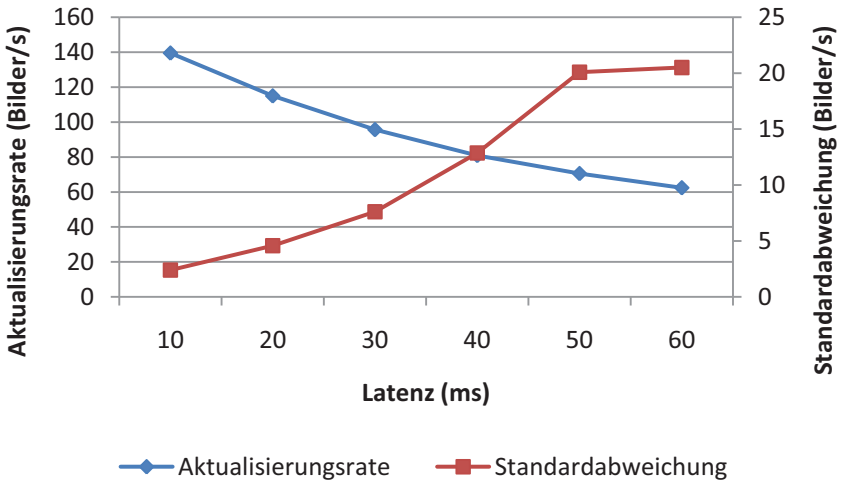


Abbildung 7.6: Einfluß der Netzwerklatenz auf die Bildaktualisierungsrate bei 2 Knoten.

wiederholrate, was sich für Spieler in störenden Bildrucklern äußern kann. Daher ist eine niedrige Standardabweichung bei MMVEs wichtiger als eine konstant geringere Bildaktualisierungsrate, solange diese für einen flüssigen Spielablauf nicht einen unteren Grenzwert unterschreitet.

Der Test hat insgesamt gezeigt, daß Wissenheim mit OSS gut skaliert. Wie bereits erwähnt, sind hierzu aber Optimierungen notwendig, welche die Konsistenz einzelner Anwendungsphasen abschwächen. Ausschließlich mit strenger Konsistenz kann Wissenheim in Verbindung mit OSS und vielen Spielern nicht beliebig gut skalieren, da verteilte MMVEs bereits bei rein nachrichtenbasierter Kommunikation – vor allem wegen der Netzwerklatenz – viele Optimierungen und schwächere Konsistenzmodelle einsetzen müssen, um skalierbar zu bleiben. Diese Einschränkungen kann der gemeinsame verteilte transaktionale Speicher nicht umgehen, da er intern selbst nachrichtenbasiert arbeitet und für Anwendungen nur von dieser Kommunikation abstrahiert. Wird OSS dagegen für die Kommunikation eines Server-Clusters eingesetzt, sind die Latenzen durchweg gering, und die Synchronisierung der Server kann weiter über die strenge transaktionale Konsistenz erfolgen.

7.3 Mikrobenchmarks

Die Mikrobenchmarks simulieren unterschiedliche Zugriffsmuster, um das Verhalten des verteilten transaktionalen Speichers unter stark konfliktbehafteten Zugriffen zu ermitteln. Ebenso werden Zugriffsmuster simuliert, um die Leistungsfähigkeit des mehrstufig overlay-basierten Commit-Protokolls zu evaluieren. Weiterhin soll der Test kaskadierter Transaktionen zeigen, ob sich mit dieser Technik die Leistungsfähigkeit des transaktionalen Speichers durch Maskierung der Netzwerklatenzen verbessern läßt.

Die Testumgebung besteht aus 129 Knoten der Grid-Plattform *Grid 5000* [21]. Jeder Knoten besitzt einen Prozessor mit 4 Kernen (Intel Xeon X3440 mit 2,53 GHz) und 16 GB Hauptspeicher. Alle Knoten sind über ein geswitchtes Gigabit-Ethernet-Netzwerk miteinander verbunden. Die Tests wurden jeweils mit 16, 32, 64 und 128 Knoten durchgeführt. Gegenüber dem P2P-Protokoll verwenden die synthetischen Tests beim UP-Protokoll einen zusätzlichen Knoten für den UP, welcher aber dediziert ist. In den Diagrammen sind nur die an der verteilten Anwendung beteiligten Knoten dargestellt, weshalb der UP hier als zusätzlicher Knoten nicht in Erscheinung tritt.

7.3.1 Konkurrierender Datenzugriff

Dieser Test zeigt das Verhalten des transaktionalen Speichers bei permanent konfliktbehafteten Speicherzugriffen. Alle Knoten inkrementieren gleichzeitig eine gemeinsame Variable im verteilten transaktionalen Speicher. Jeder Knoten führt dabei 3.000 Inkrementierungen durch. Um der Simulation realer Anwendungen zu entsprechen, wird jede Transaktion mit einer zufälligen Zeitspanne zwischen 0 und 2 ms beaufschlagt, so daß die Transaktionen eine unterschiedliche Ausführungsdauer besitzen. Ohne den Zeitaufschlag wären die Transaktionen zu kurz, und der überwiegende Aufwand würde auf die Kommunikation und Transaktionsverwaltung entfallen. Abbildung 7.7 zeigt den gesamten Transaktionsdurchsatz für das P2P- und UP-Protokoll in Abhängigkeit von der Knotenanzahl. Der Transaktionsdurchsatz ist ein Maß für die Leistungsfähigkeit des gemeinsamen verteilten transaktionalen Speichers. Beim UP-Protokoll bleibt der gesamte Transaktionsdurchsatz bis zu 64 Knoten stabil, bricht aber bei 128 Knoten ein. Beim P2P-Protokoll bricht der gesamte Transaktionsdurchsatz bereits bei 32 Knoten stark ein. Zu berücksichtigen ist, daß der Transaktionsdurchsatz pro Knoten bei beiden Commit-Protokollen bei zunehmender Knotenanzahl sinkt.

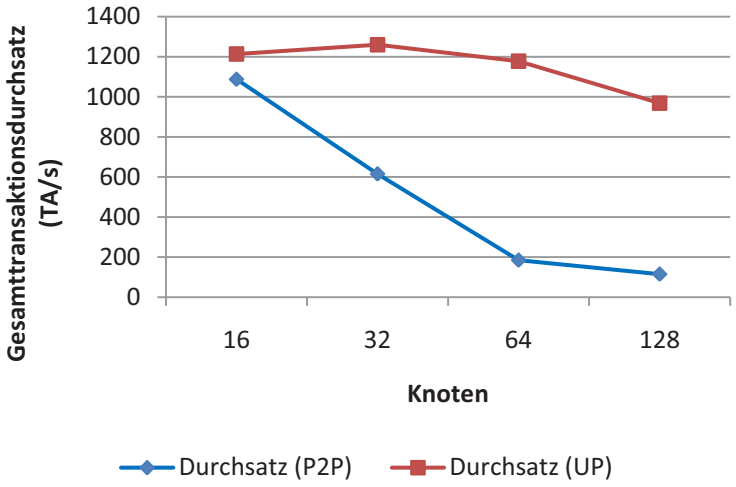


Abbildung 7.7: Gesamttransaktionsdurchsatz bei Inkrementierung einer gemeinsamen Variable.

Die Diagramme 7.8 und 7.9 zeigen die Seitenanforderungszeiten für das P2P- und UP-Protokoll beziehungsweise die Tokenanforderungszeit unter Verwendung des P2P-Protokolls. Die Seitenanforderungszeiten bleiben bei P2P-Protokoll unabhängig von der Knotenanzahl konstant mit einer vernachlässigbar kleinen Standardabweichung (1–3 TA/s auf einem Knoten). Beim UP-Protokoll steigt die Seitenanforderungszeit und dessen Standardabweichung mit zunehmender Knotenanzahl an. Gleiches zeigt auch die Tokenanforderungszeit beim P2P-Protokoll.

Bewertung

Beim UP-Protokoll bleibt der Durchsatz bis zu 64 Knoten nahezu konstant. Dies bedeutet, daß sich trotz der höheren Konfliktwahrscheinlichkeit keine nennenswerte Verschlechterung ergibt. Die höhere Konfliktwahrscheinlichkeit, die sich mit zunehmender Anzahl von Knoten, die auf der gemeinsamen Variablen arbeiten, zwangsläufig ergibt, können die Knoten durch ihre mehrfach parallele Ausführung kompensieren. Bei 128 Knoten nimmt die Konfliktwahrscheinlichkeit stärker zu, als die Knoten durch ihre parallele Berechnung kompensieren kön-

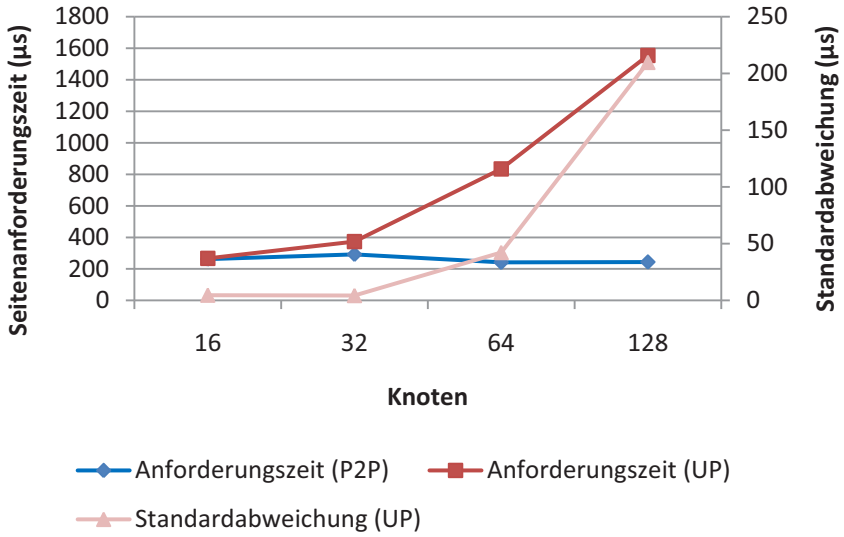


Abbildung 7.8: Seitenanforderungszeiten bei Inkrementierung einer gemeinsamen Variable.

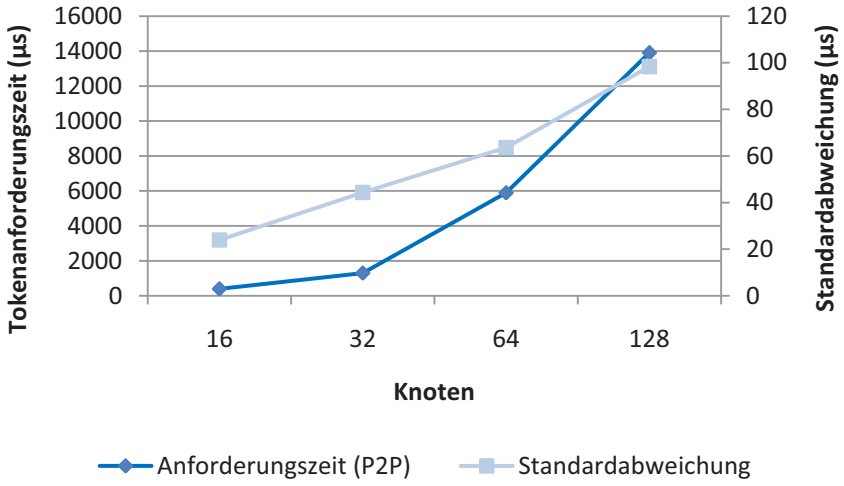


Abbildung 7.9: Tokenanforderungszeit bei Inkrementierung einer gemeinsamen Variable.

nen. Deswegen ist eine Verringerung des Gesamttransaktionsdurchsatzes zu beobachten. Beim P2P-Protokoll bricht der gesamte Transaktionsdurchsatz bereits bei 32 Knoten stark ein. Dies liegt zum einen darin begründet, daß die Knoten beim Commit von Transaktionen auf das Token warten müssen. Zudem können Transaktionen während der Tokenanforderung abgebrochen werden, so daß das Token zum Zeitpunkt des Abbruchs vergeblich an den Knoten ausgeliefert wird. Dies erhöht wiederum die Tokenanforderungszeiten auf anderen Knoten. Abbildung 7.9 zeigt, daß die Anforderungszeit für das Token mit zunehmender Knotenanzahl stark zunimmt, welche neben der Konfliktwahrscheinlichkeit hauptsächlich für den verminderten Transaktionsdurchsatz beim P2P-Protokoll verantwortlich ist.

Die langen Wartezeiten bezüglich des Tokens können nicht allein auf dessen Austausch zurückgeführt werden (siehe Kapitel 4.4.3). An dieser Stelle ist zu berücksichtigen, daß ein Knoten bei Erhalt des Tokens vor dem Commit der eigenen Transaktion auf die Verarbeitung noch fehlender Commit-Benachrichtigungen von früheren Transaktionen warten muß. Dieser Prozeß nimmt umso mehr Zeit in Anspruch, je mehr Knoten Transaktionen abschließen und dies-

bezüglich Commit-Benachrichtigungen verschicken. Weiterhin ist die benötigte Zeitdauer für das Versenden der Commit-Benachrichtigungen ebenso von der Anzahl der Knoten abhängig, da die derzeitige Implementierung keinen Multicast gestattet.

Die Seitenanforderungszeit (siehe Abbildung 7.8) bleibt beim P2P-Protokoll bis zu 128 Knoten konstant. Dies beruht darauf, daß Knoten die Anforderung von Objektreplikaten für viele parallel eintreffende Anfragen im Gegensatz zum Token gleichzeitig beantworten können. Die Zeitdauer für die Anforderung von Objektreplikaten würde nur dann ansteigen, wenn das Netzwerk oder die Knoten überlastet sind, beispielsweise durch volle Ausnutzung der Netzwerkbandbreite, des Prozessors oder falls sich eingehende Anfragen aufgrund der Verarbeitungsgeschwindigkeit in der Warteschlange für eingehende Netzanfragen aufstauen. Dies ist beim UP-Protokoll der Fall, weshalb die Seitenanfragezeit und die Schwankung derselbigen (Standardabweichung) bei mehreren Knoten zunimmt. Dies liegt daran, daß Knoten bei Verwendung des UP-Protokolls wegen der nicht vorhandenen Tokenanforderung keinen langen Blockierungszeiten unterliegen. Folglich führt dies im Vergleich zum P2P-Protokoll zu einer höheren Geschwindigkeit bei der Validierung und Ausführung von Transaktionen (inklusive wiederholter Ausführung konfliktbehafteter Transaktionen) und impliziert einen gesteigerten Nachrichtenaustausch über das Netzwerk. Dadurch kommt es in Abhängigkeit zu der Anzahl von Knoten zu einer verzögerten Bearbeitung eingehender Objektanfragen, die sich in einer längeren Seitenanforderungszeit niederschlägt.

7.3.2 Gruppenlokale Commits im Overlay-Netzwerk

Dieser Test evaluiert das overlay-basierte Commit-Protokoll mittels eines Zugriffsmusters, indem mehrere Knoten eine gemeinsame Variable inkrementieren. Jeweils vier Knoten arbeiten auf einer gemeinsamen Variable. Alle Knoten, die eine gemeinsame Variable inkrementieren, sind unter Verwendung des overlay-basierten Commit-Protokolls in einer Gruppe zusammengefaßt. Die Synchronisierung in der Gruppe erfolgt über das UP-Protokoll, wobei einer der Gruppenknoten den Superpeer stellt, welcher die Synchronisierung mit anderen Superpeers über das P2P-Commit-Protokoll durchführt. Gruppenlokale Commits entscheiden dynamisch, ob abgeschlossene Transaktionen global im gesamten Overlay-Netzwerk oder nur in der Gruppe synchronisiert werden müssen. Zum Vergleich wird dieser Test ohne einem strukturierten Overlay-Netzwerk mit dem P2P- und UP-Protokoll durchgeführt, wobei beim UP-Protokoll wieder ein dedizierter Knoten als UP zum Einsatz kommt. Abbildung 7.10 zeigt den Gesamttransaktionsdurchsatz in Abhängigkeit der verwendeten Knoten.

Die Meßergebnisse zeigen, daß der gesamte Transaktionsdurchsatz beim P2P-Protokoll von 16 zu 128 Knoten sinkt und sich halbiert. Beim UP-Protokoll steigt der Transaktionsdurchsatz an, kann aber nur leicht zulegen. Das overlay-basierte Commit-Protokoll verzeichnet dagegen einen stark ansteigenden Transaktionsdurchsatz, der linear mit der Anzahl der Knoten ansteigt.

Bewertung

Beim overlay-basierten mehrstufigen Commit-Protokoll zeigt sich beim verwendeten Zugriffsmuster wie erwartet, daß der Transaktionsdurchsatz linear mit der Anzahl der Knoten steigt. Mit zunehmender Knotenanzahl treten dem Overlay-Netzwerk weitere Knoten in neuen Gruppen bei. Somit ändert sich die Konfliktwahrscheinlichkeit bezüglich der einzelnen zu inkrementierenden Variablen nicht, da jederzeit nur vier Knoten auf einer Variablen arbeiten. Normalerweise würde der Transaktionsdurchsatz nicht linear mit der Anzahl der Knoten steigen, da die Superpeers Commits von Transaktionen global über das P2P-Protokoll synchronisieren müßten. Das Token würde an dieser Stelle den Transaktionsdurchsatz vermindern. Der gruppenlokale Commit-Mechanismus erkennt jedoch dynamisch daß die Transaktionsobjekte

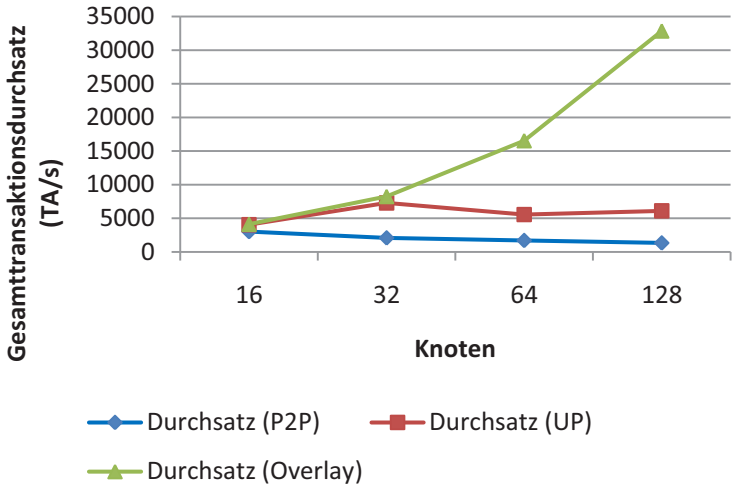


Abbildung 7.10: Gesamttransaktionsdurchsatz bei Inkrementierung mehrerer Variablen.

(zu inkrementierende Variablen) nur innerhalb ihrer Gruppe zugegriffen werden, weshalb eine globale Synchronisierung über das P2P-Protokoll auf der ersten hierarchischen Ebene des Overlay-Netzwerks entfällt. Deshalb können alle Superpeers nebenläufig Transaktionen validieren, ohne daß sie miteinander kommunizieren müssen, was sich in dem linearen Verhalten bezüglich des Transaktionsdurchsatzes widerspiegelt.

Beim P2P-Protokoll erfolgt die Synchronisierung global zwischen allen Knoten. Zwar treten gegenüber dem konkurrierenden Datenzugriff (siehe Kapitel 7.3.1) wesentlich weniger Konflikte auf, aber die globale Synchronisierung aller Transaktionen zwischen den Knoten führt zu einer starken Verzögerung, so daß sich der Transaktionsdurchsatz nicht erhöhen kann, obwohl sich das Verhältnis der möglichen Transaktionskonflikte zur Anzahl der gesamten Transaktionen nicht verändert. Bei diesem Commit-Protokoll liegt der Flaschenhals an dem globalen Token, weshalb der gesamte Transaktionsdurchsatz sinkt. Beim UP-Protokoll erfolgt die Synchronisierung wie beim P2P-Protokoll global. Zunächst steigt der Gesamttransaktionsdurchsatz bis zu 32 Knoten an, bricht bei 64 Knoten ein, um anschließend bei 128 Knoten wieder leicht anzusteigen. Erwartet wäre beim UP-Protokoll, daß der Transaktionsdurchsatz mit zunehmender Knotenanzahl kontinuierlich ansteigt und dieser ab einer bestimmten Knotenanzahl wegen der maximalen Auslastung des UP in eine Sättigungsphase übergeht, da gegenüber dem P2P-Protokoll kein Commit-Token vorhanden ist und Commit-Anfragen nebenläufig beim UP eintreffen.

Ohne weitere Untersuchungen ist die Anomalie ab 64 Knoten nicht genau begründbar. Fakt ist jedoch, daß hier mehrere Faktoren zusammentreffen und möglicherweise gemeinsam für die Anomalie ursächlich sind. Mit zunehmender Knotenanzahl geht der UP in eine Sättigungsphase über, während gleichzeitig aber die Konfliktwahrscheinlichkeit steigt. Daraus kann sich ein unfaires Verhalten zwischen den Knoten ergeben, indem abgebrochene Transaktionen wiederholt abbrechen, da sie wegen der Wiederanforderung von Objektreplikaten eine längere Laufzeit aufweisen und sich deshalb nicht gegenüber anderen Transaktionen durchsetzen können.

Die Seiten- und Tokenanforderungszeiten entsprechen im wesentlichen denen des konkurrierenden Datenzugriffs (siehe Kapitel 7.3.1), weshalb deren ausführliche Diskussion an dieser Stelle ausbleibt. Der wesentliche Unterschied besteht darin, daß beim P2P-Protokoll die Tokenanforderungszeit und beim UP-Protokoll die Seitenanforderungszeit stärker ansteigen. Dies liegt darin begründet, daß die Knoten wegen des Zugriffsmusters ein anderes Konfliktverhalten aufweisen. Wegen der geringeren Konfliktwahrscheinlichkeit und geringeren Wahrschein-

lichkeit, daß Transaktionen vor Anforderung des Tokens abgebrochen werden, ist der konkurrierende Tokenzugriff stärker ausgeprägt. Bezüglich der Seitenanforderungszeit des UP-Protokolls ist der Austausch von Transaktionsobjekten wegen der geringeren Konfliktrate gegenüber dem stark konkurrierenden Datenzugriff ebenfalls geringer. Demzufolge ist die Auslastung der einzelnen Knoten wegen des höheren Transaktionsdurchsatzes ebenfalls höher und verursacht höhere Seitenanforderungszeiten.

7.3.3 Kaskadierte Transaktionen

Für die Evaluation der kaskadierten Transaktionen haben Knoten eine gemeinsame Variable inkrementiert, wobei jeder Knoten 300 Inkrementierungen respektive Transaktionen ausgeführt hat. Die Synchronisierung der Transaktionen erfolgte über das P2P-Protokoll, da der OSS-Prototyp die kaskadierten Transaktionen derzeit nur für dieses Protokoll implementiert. Die Warteschlangengröße für die kaskadierten Transaktionen beträgt 500 Einträge, so daß OSS 500 zu validierende Transaktionen zwischenspeichern kann, bevor die Beendigung weiterer Transaktionen (`EOT()`-Aufruf) blockiert. Im Gegensatz zu dem Test in Kapitel 7.3.1 unterliegen die Transaktionen in diesem Fall keiner zusätzlichen Zeitspanne hinsichtlich ihrer Ausführungsdauer. Die reduzierte Transaktionsanzahl und Ausführungsdauer liegen primär an der beschränkten Reservierungszeit von *Grid 5000*. Trotz dieser Einschränkungen zeigt die Abbildung 7.11, wie kaskadierte Transaktionen den Transaktionsdurchsatz entscheidend verbessern.

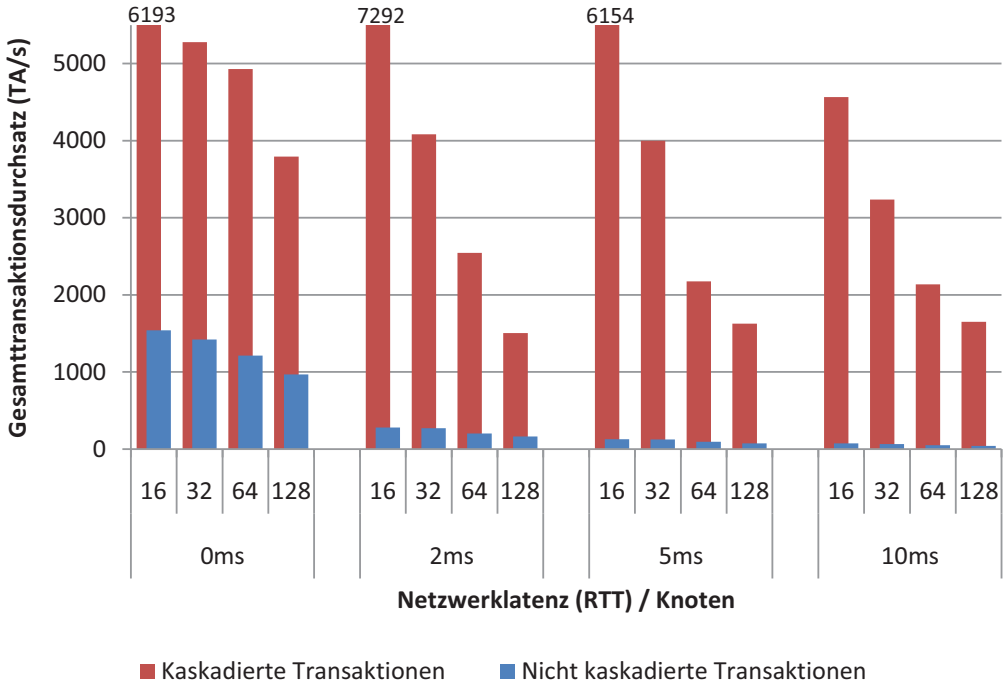


Abbildung 7.11: Gesamttransaktionsdurchsatz kaskadierter Transaktionen.

Die Abbildung stellt den Transaktionsdurchsatz zwischen kaskadierten und nicht kaskadierten Transaktionen für eine unterschiedliche Anzahl von Knoten und verschiedene Netzwerklatenzen (RTT) gegenüber. Gut zu sehen ist, daß sich der akkumulierte Transaktionsdurchsatz aller Knoten bei der Verwendung kaskadierter Transaktionen in sämtlichen Testszenarien von dem Transaktionsdurchsatz ohne kaskadierte Transaktionen absetzt. Der Transaktionsdurchsatz kaskadierter Transaktionen liegt um ein vielfaches höher. Ebenfalls gut zu erkennen ist,

daß sich das Durchsatzverhältnis zwischen kaskadierten und nicht kaskadierten Transaktionen mit zunehmender Netzwerklatenz erhöht, auch wenn der Gesamttransaktionsdurchsatz sinkt.

Bewertung

Die kaskadierten Transaktionen zeigen grundlegend einen verbesserten Transaktionsdurchsatz für das P2P-Protokoll. Zudem fällt der Transaktionsdurchsatz bei den kaskadierten Transaktionen mit zunehmendner Netzwerklatenz nicht so stark ab wie bei den nicht kaskadierten Transaktionen. Diese Indikatoren zeigen, daß die kaskadierten Transaktionen Verzögerungen über das Netzwerk, verursacht durch Netzwerklatenzen, maskieren können. Die kaskadierten Transaktionen können die Netzwerklatenz jedoch nicht vollständig maskieren. Dies hat folgende Gründe. Zum einen ist die Warteschlange auf 500 Einträge begrenzt. Ist die Warteschlange gefüllt, blockieren kaskadierte Transaktionen genauso wie nicht kaskadierten Transaktionen. In diesem Fall kann der Anwendungsthread keine weiteren Transaktionen ausführen, bis der Knoten wartende Transaktionen aus der Warteschlange entfernt hat, entweder durch einen Commit oder einen Transaktionsabbruch.

Die Warteschlangengröße wurde explizit für dieses Testszenario mit 500 Einträgen sehr großzügig bemessen, um zu zeigen, daß kaskadierte Transaktionen die Netzwerklatenz auch für viele kleine Transaktionen und ebenso mit vielen kaskadierten Abbrüchen gut maskieren können. Für reale Anwendungen ist diese Warteschlangengröße in der Praxis dagegen nicht sinnvoll. Transaktionen haben dort im allgemeinen eine höhere Ausführungsdauer, welche die Anzahl wartender Transaktionen bis zum Commit reduziert. Greifen die Transaktionen mehrere und unterschiedliche Transaktionsobjekte zu, steigt die Wahrscheinlichkeit für einen kaskadierten Abbruch und demnach auch eine Verschwendung von Ressourcen. Deshalb sollte die Anzahl kaskadierter Transaktionen pro Thread auf wenige Transaktionen begrenzt sein, die Warteschlange aber insgesamt soviele Einträge fassen können, daß Threads in Multithreading-Anwendungen sich nicht aufgrund einer überfüllten Warteschlange gegenseitig blockieren.

Weiterhin verursacht das Testszenario eine hohe Konfliktrate. Diese führt bei einem Transaktionsabbruch immer zu einem kaskadierten Abbruch aller in der Warteschlange wartenden Transaktionen. Die kaskadierten Transaktionen entschärfen aber den konkurrierenden Zugriff auf das Token, so daß die Knoten das Token zum einen weniger häufig untereinander austauschen müssen und zum anderen im Hintergrund weitere Transaktionen ausführen können.

Ein weiterer Anhaltspunkt für den sinkenden Transaktionsdurchsatz bei einer höheren Netzwerklatenz begründet sich durch die blockierenden Seitenanforderungen der Knoten untereinander, da der Transaktionsabbruch auf den Knoten aufgrund der Invalidierungsstrategie ebenso die Seiteninhalte invalidiert. Die Abbildung zeigt zudem zwei Anomalien. Der Transaktionsdurchsatz steigt bei 16 Knoten und einer Ehöhung der Netzwerklatenz von 0 ms auf 2 ms an. Gleiches tritt auch bei 128 Knoten und einer Erhöhung der Netzwerklatenz von 2 ms auf 10 ms auf. Dies läßt sich nicht allein auf die kaskadierten Transaktionen selbst zurückführen, sondern ist auf eine nicht ganz faire Transaktionsausführung hinsichtlich der Ausführungsdauer zurückzuführen. Ein weiterer Einflußfaktor ist möglicherweise auch ein verändertes Zeitverhalten bei den kaskadierten Transaktionsabbrüchen.

8 Zusammenfassung

8.1 Fazit

In dieser Arbeit wurde ein gemeinsamer verteilter transaktionaler Speicher konzipiert, welcher die Entwicklung verteilter Anwendungen vereinfacht, und mithilfe des Prototyps OSS evaluiert. Anwendungen kommunizieren über Speicherzugriffe, als wenn sie auf einem gemeinsamen physischen Speicher arbeiten würden, obwohl deren speicherbasierte Kommunikation lediglich auf der Synchronisierung replizierter Speicherinhalte disjunkter physischer Speicher beruht. Spekulative Transaktionen bündeln mehrere einzelne Operationen auf dem Speicher und führen diese atomar aus. Somit können keine verteilten Verklemmungen entstehen. Zugleich reduziert die Bündelung von Operationen die Netzwerkkommunikation hinsichtlich der Anzahl von Netzwerknachrichten und Metadaten. Bei synchroner Netzkommunikation treten ebenso weniger Blockierungsphasen auf. Die grundlegende Vereinfachung für Anwendungsentwickler besteht darin, daß beim datenzentrierten Ansatz Programmlogik und Netzwerkkommunikation strikt voneinander getrennt sind und die speicherbasierte Kommunikation vollständig von Mechanismen der Netzwerkkommunikation abstrahiert. Deshalb ist es für eine Anwendung auch nicht notwendig eigene Netzwerkprotokolle oder Nachrichtenformate zu entwickeln. Der Fokus der Anwendungsentwicklung liegt demnach nur auf der Programmlogik.

Für die transparente Synchronisierung von Objektreplikaten und Validierung von Transaktionen sorgen zwei Commit-Protokolle, wobei sich eines vollständig verteilt auf P2P-Techniken stützt und über eine Vorwärtsvaliderung Transaktionskonflikte auflöst. Die damit verbundene First-Wins-Strategie trägt erheblich zur Reduzierung der Netzwerkkommunikation und folglich zu kürzeren Validierungs- und Commit-Zeiten bei. Die Serialisierung erfolgt bei diesem Ansatz über ein vorhersagbares Commit-Token. Beim zweiten Protokoll vollzieht ein Koordinator die Transaktionsvalidierung über eine kombinierte Vorwärts- und Rückwärtsvalidierung, der gleichzeitig auch die Transaktionsserialisierung übernimmt. Beide Protokolle garantieren eine zuverlässige Synchronisierung des verteilten Speichers.

Neben der Entwicklung der vollständig verteilten und koordinatorbasierten Commit-Protokolle, wobei letzteres die Vorwärts- und Rückwärtsvalidierung kombiniert, sind viele neue Konzepte in die Entwicklung eingeflossen, welche die Effizienz und Leistungsfähigkeit des gemeinsamen verteilten transaktionalen Speichers entscheidend verbessern. Neu ist die tokeninterne Warteschlange, die über eine regelmäßige Verteilung mittels Commit-Benachrichtigungen an die Knoten eine Tokenvorhersage erlaubt. Dies reduziert vor allem die Netzwerkbelastung, da sich die Anzahl weitergeleiteter Tokenanfragen über mehrere Knoten vermindert. Neu ist auch der koordinierte Tokenansatz, der durch Integration des P2P-Verfahrens eine direkte Weitergabe des Tokens zwischen den Knoten erlaubt.

Neu ist die linux-basierte Unterstützung von ungebundenen Transaktionen. Dies bedeutet, daß die Objektmengen einer Transaktion keiner Größenbeschränkung unterliegen. Bei einigen anderen Systemen wie beispielsweise die Implementierung von *Deuce* [3] ist dies nicht der Fall. Ebenso neu sind die transparente Zugriffserkennung auf Transaktionsobjekte, so daß weder eine manuelle Objektregistrierung noch Compilermodifikationen notwendig sind, sowie die automatische und transparente Rücksetzung von Transaktionen im Konfliktfall.

Ein wesentlicher neuer Aspekt des gemeinsamen verteilten transaktionalen Speichers ist die Verknüpfung mit einem strukturierten Overlay-Netzwerk, wie es aus P2P-Dateitauschbörsen bekannt ist. Das Overlay-Netzwerk löst ein zentrales Problem der Skalierbarkeit des transaktionalen Speichers, da Transaktionen eine strenge Serialisierung benötigen, diese aber nur notwendig ist, wenn zwischen Transaktionen Abhängigkeiten durch gemeinsam zugegriffene Transaktionsobjekte bestehen. Daher begrenzt der transaktionale Speicher die Netzwerkkommunikation und damit auch die Tranaktionssynchronisierung regional oder vermeidet die Netzkommunikation vollständig. Auf unterster Ebene lösen dieses Problem lokale Commits auf den Knoten, indem Sie Transaktionen nur dann global synchronisieren, sofern die Gefahr einer möglichen Überlappung mit anderen Transaktionen besteht. Hierzu merken sich Knoten, ob geschriebene Objekte einer abzuschließenden Transaktion auch auf anderen Knoten repliziert sind. Ist dies nicht der Fall, können sie auf diese Weise nebenläufig und ohne Netzwerkkommunikation validieren und Transaktionen abschließen.

Das Overlay-Netzwerk teilt Knoten nach Interessen und Speicherzugriffsmustern gewichtet in Gruppen ein, um die globale Synchronisierung von Transaktion weiter zu reduzieren und so eine knotenübergreifende nebenläufige Validierung von Transaktionen zu ermöglichen. Mit dem Overlay-Netzwerk eingeführte gruppenlokale Commits übernehmen das Paradigma der knotenlokalen Commits und erlauben entsprechend auch Gruppen nebenläufig Transaktionen abzuschließen, wobei die Kommunikation nur auf die Gruppenteilnehmer entfällt. Analog zu den einzelnen Knoten kontrolliert der Superpeer, ob geschriebene Transaktionsobjekte außerhalb der eigenen Gruppe repliziert sind und entscheidet dahingehend, ob Transaktionen global oder gruppenlokal synchronisiert werden können. Die knoten- und gruppenlokalen Commits verbessern die Leistung des gemeinsamen verteilten transaktionalen Speichers, da in der Regel nicht alle Transaktionen untereinander Abhängigkeiten besitzen und deshalb Konflikte verursachen können.

Weiterhin bietet das Overlay-Netzwerk die Möglichkeit, sich dynamisch zu restrukturieren und seine Gruppen zu reorganisieren. So ist gewährleistet, daß Knoten mit gleichen Interessen sich auch bei wechselnden Zugriffsmustern in einer gemeinsamen Gruppe befinden und ihre Kommunikation größtenteils lokal abwickeln können. Ohne diese Reorganisation können sich Commits ansonsten vermehrt in einer globalen Transaktionssynchronisierung äußern. Weiterhin entlastet die Multicast-Kommunikation auf Applikationsebene das Netzwerk und spart so Bandbreite ein. Der nebenläufige Validierungs- und Commit-Vorgang einer Transaktion hinsichtlich direkt nachfolgender Transaktionen macht den gemeinsamen verteilten transaktionalen Speicher von der Netzwerklatenz unabhängig. Damit ist hoher Transaktionsdurchsatz auch in Netzwerken mit einer hohen Latenz, wie das in Weitverkehrsnetzen oder Grid-Umgebungen häufig der Fall ist, möglich.

Die Messungen haben gezeigt, daß der gemeinsame transaktionale Speicher die Entwicklung verteilter Anwendungen ermöglicht und effizient ist. Es hat sich ebenso gezeigt, daß eine geringe Konfliktrate in Verbindung mit der regional begrenzten Netzwerkkommunikation und der Latenzmaskierung auch bei vielen Knoten eine gute Skalierbarkeit aufweist. Die starke transaktionale Konsistenz eignet sich jedoch nicht für alle Anwendungsfälle wie beispielsweise zeitkritische Anwendungen mit einer hohen Konfliktrate. In diesen Anwendungsfällen sind schwächere Konsistenzmodelle oder wie bei Wissenheim eine Abschwächung der transaktionalen Konsistenz besser geeignet.

8.2 Ausblick

Die in dieser Arbeit entwickelten Konzepte und Strategien stellen einen Teil der Forschung über verteilte transaktionale Speicher dar. Da die Forschung auf diesem Gebiet komplex und umfangreich ist, erfordern verteilte transaktionale Speicher zukünftig weitere Untersuchungen.

Der Ausblick zeigt kurz einige weitere interessante Forschungsbereiche auf, die vor allem die Objektsynchronisierung und Netzwerkstrukturierung ansprechen.

Ein Replikatmanager verwaltet mehrere Versionen eines Objekts, wie dies beispielsweise Multiversion-Objekte (siehe Kapitel 6.1) fordern. Der Manager regelt den gesamten Objektaustausch eines Knotens und hat Kenntnis von allen auf einem Knoten vorhandenen Objektreplikaten. Weiterhin hält der Replikatmanager Objektreplikate in einem Zwischenspeicher vor. So ist möglich, das Aktualisierungsverfahren beim Objektaustausch auch zusammen mit der verzögerten Abbruchstrategie von Transaktionen zu kombinieren. Dies verhindert die Aktualisierung von Objektreplikaten während eines gleichzeitigen Zugriffs, was insbesondere bei Zeigern im verteilten Speicher kritisch ist (siehe Kapitel 2.7.2). Fragt ein Knoten ein Objektreplikat an, versucht der Replikatmanager, die Anfrage zunächst aus seinem Zwischenspeicher zu bedienen, bevor er die Anfrage an einen anderen Knoten weiterleitet. Genauso aktualisiert der verteilte transaktionale Speicher Transaktionsobjekte mittels empfangenen Objektreplikaten nur, wenn diese gerade von keiner Transaktion verwendet wird. Ebenso kann der Replikatmanager die in Kapitel 4.3.5 angesprochene differentielle Speicherung mehrerer Versionen von Objekten übernehmen, um Speicherplatz einzusparen.

Die Objektsynchronisierung sowohl über die Invalidierungs- als auch Aktualisierungsstrategie zu vollziehen, kann die Leistung des transaktionalen Speichers verbessern. Während erstere lokale Commits begünstigt, erlaubt die andere als aktive Objektreplikation neben der Replikation zur Fehlertoleranz auch Replikate zwischen Knoten proaktiv zu verteilen, welche aktuell dieselben Objekte miteinander austauschen. So brauchen Knoten nicht erst beim ersten Zugriff aktuelle Objektreplikate bei einem entfernten Knoten anfordern, sondern können direkt darauf zugreifen. Für die dynamische Entscheidung, welche der beiden Strategien zu einem Zeitpunkt für Objekte vorzuziehen ist, muß ein Monitor Objektzugriffe aufzeichnen und anhand dieser Informationen die Strategie auswählen.

Die Replikation von Transaktionsobjekten schützt bei einem Knotenausfall vor Datenverlust. Führen Knoten lokale Commits für Transaktionsobjekte aus, können sie diese nicht aktiv auf andere Knoten replizieren. Ein Knotenausfall würde demnach aufgrund der fehlenden proaktiven Replikation eher zu Datenverlust führen. Diesem Problem kann mit sogenannten Sicherheitsreplikaten begegnet werden. Knoten verteilen zugriffsgeschützte Replikate aus Fehlertoleranzgründen. Knoten, die diese Replikate empfangen, können diese aber nur dann zugreifen, wenn die Knoten im System einen Datenverlust festgestellt haben, den sie mit den Sicherheitsreplikaten kompensieren können. So ist es Knoten im fehlerfreien Betrieb trotz Replikation möglich, lokale Commits bei Transaktionen durchzuführen.

Erfolgt eine Invalidierung der abzuschließenden Transaktion erst, nachdem ein Knoten eine Anforderung nach dem Token verschickt hat, so erhält der Knoten das Token eventuell unnötigerweise. Verschickt ein Knoten in der Tokenanfrage auch seine Lesemenge mit, kann der tokenbesitzende Knoten anhand der Versionsnummern feststellen, ob die vom anfragenden Knoten abzuschließende Transaktion veraltete Daten gelesen hat und braucht das Token an diesen Knoten nicht auszuliefern. Zu berücksichtigen ist hier, daß mehrere nebenläufige abzuschließende Transaktionen eines Knotens dann jeweils eine eigene Tokenanfrage verschicken müssen. Hierbei ist zu untersuchen, inwiefern der zusätzliche Kommunikationsaufwand über das Netzwerk eine Verbesserung der Leistung des transaktionalen Speichers bewirkt.

Für die Overlay-Strukturierung hinsichtlich der Anzahl von Hierarchieebenen und der Gruppierung der Knoten sind sowohl physische als auch semantische Parameter (siehe Kapitel 5.3.1) von Interesse. Ziel ist es, die Netzwerkkommunikation anhand der Applikationssemantik überwiegend lokal auf einzelne Gruppen von Knoten zu beschränken. Ebenso sollen die Knoten beispielsweise auch eine niedrigere Netzwerklatenz zueinander haben. Zu untersuchen ist, wie sich aus einer intelligenten Verknüpfung der Parameter aus der physischen und semantischen Klasse ein (mehrstufiges) hierarchisch strukturiertes Overlay-Netzwerk aufbauen läßt,

was diesen Ansprüchen gerecht wird.

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ABI	Application Binary Interface
ACID	Atomicity, Consistency, Isolation, Durability (Transaktionseigenschaften)
ALM	Application Layer Multicast
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BOT	Begin of Transaction
ccNUMA	Cache Coherent Non Uniform Memory Access
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
DB	Datenbank
DBMS	Datenbankmanagementsystem
DBS	Datenbanksystem
DHT	Distributed Hash Table
DiSTM	Distributed Software Transactional Memory
DSM	Distributed Shared Memory
DSTM2	Dynamic Software Transactional Memory 2
DTM	Distributed Transactional Memory
E/A	Ein-/Ausgabe
EOT	End of Transaction
FIFO	First-In-First-Out
fps	frames per second
HPC	High Performance Computing
HT	Hashtabelle
HTC	High Throughput Computing

HTM	Hardware Transactional Memory
HTTP	Hypertext Transfer Protokoll
IP	Internetprotokoll
IPC	Inter Process Communication
KB	Kilobyte (2^{10} Byte)
MB	Megabyte (2^{10} Kilobyte)
MMU	Memory Management Unit
MMVE	Massively Multiuser Virtual Environment
MOG	Multiplayer Online Game
MP	Message Passing
MTA	Mail-Transfer-Agent
NoRMA	No Remote Memory Access
NUMA	Non Uniform Memory Access
OSS	Object Sharing Service
P2P	Peer-to-Peer
PGAS	Partitioned Global Address Space
RAM	Random Access Memory
RMI	Remote Method Invocation
RTT	Round Trip Time
SIGSEGV	Signal – Segmentation Violation
SIMD	Single Instruction Multiple Data
SJC	Small Java Compiler
SMP	Symmetric Multi-Processing
SMT	Simultaneous Multithreading
SPoF	Single Point of Failure
SSI	Single System Image
STM	Software Transactional Memory
TA	Transaktion
TCC	Transactional Memory Coherence and Consistency

TCP	Transmission Control Protocol
TGOS	Typed Grid Object Sharing
THB	Transaction History Buffer
TLB	Translation Lookaside Buffer
TTL	Time-to-Live
UCS	Universal Character Set
UDP	User Datagram Protocol
UMA	Uniform Memory Access
UP	Ultrapeer
UUID	Universal Unique Identifier
VMF	Virtual Memory Filter
WAN	Wide Area Network

Literaturverzeichnis

[1] ADAR, EYTAN und BERNARDO A. HUBERMAN: *Free Riding on Gnutella*. First Monday, 5(10), Oktober 2000.

[2] ADVE, SARITA V. und KOUROSH GHARACHORLOO: *Shared Memory Consistency Models: A Tutorial*. Computer, 29:66–76, Dezember 1996.

[3] AFEK, YEHUDA, ULRICH DREPPER, PASCAL FELBER, CHRISTOF FETZER, VINCENT GRAMOLI, MICHAEL HOHMUTH, ETIENNE RIVIERE, PER STENSTROM, OSMAN UNSAL, WALTHER MALDONADO MOREIRA, DERIN HARMANCI, PATRICK MARLIER, STEPHAN DIESTELHORST, MARTIN POHLACK, ADRIAN CRISTAL, IBRAHIM HUR, ALEKSANDAR DRAGOJEVIC, RACHID GUERRAOUI, MICHAL KAPALKA, SASA TOMIC, GUY KORLAND, NIR SHAVIT, MARTIN NOWACK und TORVALD RIEGEL: *The Velox Transactional Memory Stack*. IEEE Micro, 30:76–87, September 2010.

[4] AGUILERA, MARCOS K., ARIF MERCHANT, MEHUL SHAH, ALISTAIR VEITCH und CHRISTOS KARAMANOLIS: *Sinfonia: A New Paradigm for Building Scalable Distributed Systems*. In: *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, Seiten 159–174, New York, NY, USA, 2007. ACM.

[5] AMZA, CRISTIANA, ALAN COX, KARTHICK RAJAMANI und WILLY ZWAENEPOEL: *Tradeoffs Between False Sharing and Aggregation in Software Distributed Shared Memory*. In: *Proceedings of the sixth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '97, Seiten 90–99, New York, NY, USA, 1997. ACM.

[6] AMZA, CRISTIANA, ALAN L. COX, SANDHYA DWARKADAS, PETE KELEHER, HONGHUI LU, RAMAKRISHNAN RAJAMONY, WEIMIN YU und WILLY ZWAENEPOEL: *TreadMarks: Shared Memory Computing on Networks of Workstations*. Computer, 29:18–28, Februar 1996.

[7] ANANIAN, C. SCOTT, KRSTE ASANOVIĆ, BRADLEY C. KUSZMAUL, CHARLES E. LEISERSON und SEAN LIE: *Unbounded Transactional Memory*. In: *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, Seiten 316–327, Washington, DC, USA, 2005. IEEE Computer Society.

[8] ANTONIU, GABRIEL, LUC BOUGÉ und MATHIEU JAN: *JuxMem: An Adaptive Supportive Platform for Data Sharing on the Grid*. Scalable Computing: Practice and Experience, 6(3):45–55, September 2005.

[9] AT&T und SCO: *System V Application Binary Interface*, 4.1 Auflage, März 1997.

[10] AT&T und SCO: *System V Application Binary Interface – Intel386™Architecture Processor Supplement*, 4 Auflage, März 1997.

[11] BAL, HENRI E., RAOUL BHOEDJANG, RUTGER HOFMAN, CERIEL JACOBS, KOEN LANGENDOEN, TIM RÜHL und M. FRANS KAASHOEK: *Performance Evaluation of the Orca Shared Object System*. ACM Trans. Comput. Syst., 16:1–40, Februar 1998.

[12] BANERJEE, SUMAN, BOBBY BHATTACHARJEE und CHRISTOPHER KOMMAREDDY: *Scalable Application Layer Multicast*. In: *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, Seiten 205–217, New York, NY, USA, 2002. ACM.

[13] BENNETT, JOHN K., JOHN B. CARTER und WILLY ZWAENEPOEL: *Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence*. In: *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, PPOPP '90, Seiten 168–176, New York, NY, USA, 1990. ACM.

[14] BERNSTEIN, PHILIP A. und NATHAN GOODMAN: *Multiversion Concurrency Control – Theory and Algorithms*. ACM Trans. Database Syst., 8:465–483, Dezember 1983.

[15] BINDHAMMER, T., R. GÖCKELMANN, O. MARQUARDT, M. SCHÖTTNER, M. WENDE und P. SCHULTHESS: *Device Driver Programming in a Transactional DSM Operating System*. In: *Proceedings of the seventh Asia-Pacific conference on Computer systems architecture*, CRPIT '02, Seiten 65–71, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[16] BIRMAN, KENNETH P.: *Reliable Distributed Systems: Technologies, Web Services, and Applications*, Band 1. Springer-Verlag, 2005.

[17] BLUNDELL, COLIN, E. CHRISTOPHER LEWIS und MILO M. K. MARTIN: *Unrestricted Transactional Memory: Supporting I/O and System Calls within Transactions*. Technical Report CIS-06-09, Department of Computer and Information Science, University of Pennsylvania, April 2006.

[18] BOCCHINO, ROBERT L., VIKRAM S. ADVE und BRADFORD L. CHAMBERLAIN: *Software Transactional Memory for Large Scale Clusters*. In: *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, Seiten 247–258, New York, NY, USA, 2008. ACM.

[19] BOLOSKY, WILLIAM J. und MICHAEL L. SCOTT: *False Sharing and its Effect on Shared Memory Performance*. In: *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4*, Seiten 3–3, Berkeley, CA, USA, 1993. USENIX Association.

[20] BOVET, DANIEL P. und MARCO CESATI: *Understanding the Linux Kernel*, Band 3rd Edition. O'Reilly Media, Sebastopol, CA, USA, 2005.

[21] CAPPELLO, F., E. CARON, M. DAYDE, F. DESPREZ, Y. JEGOU, P. PRIMET, E. JEANNOT, S. LANTERI, J. LEDUC, N. MELAB, G. MORNET, R. NAMYST, B. QUETIER und O. RICHARD: *Grid'5000: A Large Scale and Highly Reconfigurable Grid Experimental Testbed*. In: *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, GRID '05, Seiten 99–106, Washington, DC, USA, 2005. IEEE Computer Society.

[22] CHANDY, K. MANI und LESLIE LAMPORT: *Distributed snapshots: determining global states of distributed systems*. ACM Trans. Comput. Syst., 3(1):63–75, 1985.

[23] CHAPMAN, MATTHEW und GERNOT HEISER: *vNUMA: A Virtual Shared-Memory Multiprocessor*. In: *Proceedings of the 2009 USENIX Annual Technical Conference*, USENIX'09, Seiten 349–362, Berkeley, CA, USA, Juni 2009. USENIX Association.

[24] CHU, YANG-HUA, SANJAY G. RAO, SRINIVASAN SESHAN und HUI ZHANG: *A Case for End System Multicast*. In: *Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '00, Seiten 1–12, New York, NY, USA, 2000. ACM.

[25] CHUANG, WEIHAW, SATISH NARAYANASAMY, GANESH VENKATESH, JACK SAMPSON, MICHAEL VAN BIESBROUCK, GILLES POKAM, OSVALDO COLAVIN und BRAD CALDER: *Unbounded Page-Based Transactional Memory*. In: *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, Seiten 347–358, New York, NY, USA, 2006. ACM.

[26] COUCEIRO, MARIA, PAOLO ROMANO, NUNO CARVALHO und LUÍS RODRIGUES: *D²STM: Dependable Distributed Software Transactional Memory*. In: *Proceedings of the 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*, PRDC '09, Seiten 307–313, Washington, DC, USA, 2009. IEEE Computer Society.

[27] COULOURIS, GEORGE, JEAN DOLLIMORE und TIM KINDBERG: *Distributed Systems: Concepts and Design (4th Edition) (International Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[28] CULLER, D.E., J.P. SINGH und A. GUPTA: *Parallel Computer Architecture: A Hardware/Software Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann Publishers, 1999.

[29] DABEK, FRANK, RUSS COX, FRANS KAASHOEK und ROBERT MORRIS: *Vivaldi: A Decentralized Network Coordinate System*. In: *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '04, Seiten 15–26, New York, NY, USA, 2004. ACM.

[30] DAMRON, PETER, ALEXANDRA FEDOROVA, YOSSI LEV, VICTOR LUCHANGCO, MARK MOIR und DANIEL NUSSBAUM: *Hybrid Transactional Memory*. In: *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, Seiten 336–346, New York, NY, USA, 2006. ACM.

[31] DIOT, CHRISTOPHE, BRIAN NEIL LEVINE, BRYAN LYLES, HASSAN KASSEM und DOUG BALENSIEFEN: *Deployment Issues for the IP Multicast Service and Architecture*. IEEE Network, 14:78–88, 2000.

[32] DREIER, BERND, MARKUS ZAHN und THEO UNGERER: *The Rthreads Distributed Shared Memory System*. In: *In Proc. 3rd Int. Conf. on Massively Parallel Computing Systems*, 1998.

[33] EL-GHAZAWI, TAREK, WILLIAM CARLSON, THOMAS STERLING und KATHERINE YELICK: *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, 2005.

[34] ELNOZAHY, MOOTAZ, LORENZO ALVISI, YI-MIN WANG und DAVID B. JOHNSON: *A Survey of Rollback-Recovery Protocols in Message-Passing Systems*. ACM Comput. Surv., 34(3):375–408, 2002.

[35] ESWARAN, K. P., J. N. GRAY, R. A. LORIE und I. L. TRAIGER: *The Notions of Consistency and Predicate Locks in a Database System*. Commun. ACM, 19:624–633, November 1976.

[36] FIELDING, R., ET AL.: *RFC2616 - Hypertext Transfer Protocol – HTTP/1.1*. URL: http://tools.ietf.org/search/rfc2616, Juni 1999.

[37] FOSTER, IAN und CARL KESSELMAN: *The Globus Project: A Status Report*. Future Gener. Comput. Syst., 15:607–621, Oktober 1999.

[38] FOTHERINGHAM, JOHN: *Dynamic Storage Allocation in the Atlas Computer, Including an Automatic use of a Backing Store*. Commun. ACM, 4:435–436, Oktober 1961.

[39] FREEH, VINCENT W. und GREGORY R. ANDREWS: *Dynamically Controlling False Sharing in Distributed Shared Memory*. International Symposium on High-Performance Distributed Computing, 0:403–414, 1996.

[40] FREEMAN, ERIC, KEN ARNOLD und SUSANNE HUPFER: *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley Longman Ltd., Essex, UK, 1 Auflage, 1999.

[41] FRENZ, STEFAN: *SJC — ein schlanker modularer Java Compiler für Forschung und Lehre*. URL: http://fam-frenz.de/stefan/slides2.pdf.

[42] GARCIA-MOLINA, HECTOR: *Elections in a Distributed Computing System*. IEEE Trans. Comput., 31(1):48–59, 1982.

[43] GELERNTER, DAVID: *Generative Communication in Linda*. ACM Transactions on Programming Languages and Systems (TOPLAS), 7:80–112, Januar 1985.

[44] HAERDER, THEO und ANDREAS REUTER: *Principles of Transaction-Oriented Database Recovery*. ACM Comput. Surv., 15:287–317, Dezember 1983.

[45] HAMMOND, LANCE, BRIAN D. CARLSTROM, VICKY WONG, BEN HERTZBERG, MIKE CHEN, CHRISTOS KOZYRAKIS und KUNLE OLUKOTUN: *Programming with Transactional Coherence and Consistency (TCC)*. In: *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XI, Seiten 1–13, New York, NY, USA, 2004. ACM.

[46] HAMMOND, LANCE, VICKY WONG, MIKE CHEN, BRIAN D. CARLSTROM, JOHN D. DAVIS, BEN HERTZBERG, MANOHAR K. PRABHU, HONGGO WIJAYA, CHRISTOS KOZYRAKIS und KUNLE OLUKOTUN: *Transactional Memory Coherence and Consistency*. In: *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, Seite 102, Washington, DC, USA, 2004. IEEE Computer Society.

[47] HERLIHY, MAURICE, VICTOR LUCHANGCO und MARK MOIR: *A Flexible Framework for Implementing Software Transactional Memory*. In: *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, Seiten 253–262, New York, NY, USA, 2006. ACM.

[48] HERLIHY, MAURICE, VICTOR LUCHANGCO, MARK MOIR und WILLIAM N. SCHERER III: *Software Transactional Memory for Dynamic-Sized Data Structures*. In: *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, Seiten 92–101, New York, NY, USA, 2003. ACM.

[49] HERLIHY, MAURICE und J. ELIOT B. MOSS: *Transactional Memory: Architectural Support for Lock-Free Data Structures*. In: *Proceedings of the 20th annual international symposium on Computer architecture*, ISCA '93, Seiten 289–300, New York, NY, USA, 1993. ACM.

[50] IEEE COMPUTER SOCIETY: *IEEE Std 802.3-2005 – Part 3: Carrier sense multiple access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications (Section One)*, Dezember 2008.

[51] IIMURA, TAKUJI, HIROAKI HAZEYAMA und YOUKI KADOBAYASHI: *Zoned Federation of Game Servers: a Peer-to-Peer Approach to Scalable Multi-player Online Games*. In: *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, NetGames '04, Seiten 116–120, New York, NY, USA, 2004. ACM.

[52] INTEL CORPORATION: *Intel 64 and IA-32 Architectures Software Developer's Manual – Basic Architecture*, November 2006.

[53] INTEL CORPORATION: *Intel 64 and IA-32 Architectures Software Developer's Manual – System Programming Guide*, November 2006.

[54] ITZEL, LAURA, VERENA TUTTLIES, GREGOR SCHIELE und CHRISTIAN BECKER: *Consistency Management for Interactive Peer-to-Peer-based Systems*. In: *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, SIMUTools '10, Seiten 1:1–1:8, ICST, Brussels, Belgium, Belgium, 2010. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[55] ITZKOVITZ, AYAL und ASSAF SCHUSTER: *MultiView and Millipage – fine-grain sharing in page-based DSMs*. In: *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, Seiten 215–228, Berkeley, CA, USA, 1999. USENIX Association.

[56] KNIGHT, TOM: *An Architecture for Mostly Functional Languages*. In: *Proceedings of the 1986 ACM conference on LISP and functional programming*, LFP '86, Seiten 105–112, New York, NY, USA, 1986. ACM.

[57] KOTSELIDIS, CHRISTOS, MOHAMMAD ANSARI, KIM JARVIS, MIKEL LUJÁN, CHRIS KIRKHAM und IAN WATSON: *DiSTM: A Software Transactional Memory Framework for Clusters*. In: *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, Seiten 51–58, Washington, DC, USA, 2008. IEEE Computer Society.

[58] KUNG, H. T. und JOHN T. ROBINSON: *On Optimistic Methods for Concurrency Control*. ACM Trans. Database Syst., 6:213–226, Juni 1981.

[59] KÄMMER, NICO, STEFFEN GERHOLD, TOBIAS BAEUERLE und PETER SCHULTHESS: *Transactional Distributed Memory Management for Cluster Operating Systems*. In: *Proceedings of the 32. International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2009.

[60] LAMPORT, LESLIE: *Time, clocks, and the ordering of events in a distributed system*. Commun. ACM, 21(7):558–565, 1978.

[61] LEWIS, PHILIP M., ARTHUR BERNSTEIN und MICHAEL KIFER: *Databases and Transaction Processing: An Application-Oriented Approach*. Addison-Wesley, 2002.

[62] LI, KAI: *IVY: A Shared Virtual Memory System for Parallel Computing*. In: *Proceedings of the 1988 International Conference on Parallel Processing*, ICPP '88, Seiten 94–101, University Park, Pennsylvania, USA, 1988. PSU.

[63] LIANG, JIAN, RAKESH KUMAR und KEITH W. ROSS: *The FastTrack overlay: A measurement study*. Computer Networks, 50(6):842–858, April 2006.

[64] LOVE, ROBERT: *Linux System Programming*, Band 1st Edition. O'Reilly Media, Sebastopol, CA, USA, 2007.

[65] LUK, WAI-SHING und TIEN-TSIN WONG: *Two New Quorum Based Algorithms for Distributed Mutual Exclusion*. In: *ICDCS '97: Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS '97)*, Seite 100, Washington, DC, USA, 1997. IEEE Computer Society.

[66] MAEKAWA, MAMORU: *A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems*. ACM Trans. Comput. Syst., 3(2):145–159, 1985.

[67] MATVEEV, ALEXANDER, ORI SHALEV und NIR SHAVIT: *Dynamic Identification of Transactional Memory Locations*. Unpublished Manuscript, Tel-Aviv University, 2007.

[68] MATZ, MICHAEL, JAN HUBIČKA, ANDREAS JAEGER und MARK MITCHELL: *System V Application Binary Interface – AMD64 Architecture Processor Supplement*, 0.99.5 Auflage, September 2010.

[69] MAYMOUNKOV, PETAR und DAVID MAZIÈRES: *Kademlia: A Peer-to-Peer Information System Based on the XOR Metric*. In: *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, Seiten 53–65, London, UK, 2002. Springer-Verlag.

[70] MICROSOFT CORP.: *Project codename „Velocity"*, 2008. White Paper.

[71] MOHR, BERND: *Introduction to Parallel Computing*. In: *Computational Nanoscience: Do It Yourself!*, Band 31 der Reihe *NIC*, Seiten 491–505, Jülich, Germany, Februar 2006. John von Neumann Institute for Computing.

[72] MOSBERGER, DAVID: *Memory consistency models*. SIGOPS Oper. Syst. Rev., 27:18–26, Januar 1993.

[73] MÜLLER, MARC-FLORIAN, KIM-THOMAS MÖLLER, MICHAEL SONNENFROH und MICHAEL SCHÖTTNER: *Transactional Data Sharing in Grids*. In: *PDCS 2008: International Conference on Parallel and Distributed Computing and Systems*, Calgary, Alberta, Canada, 2008. IASTED.

[74] MÜLLER, MARC-FLORIAN, KIM-THOMAS MÖLLER und MICHAEL SCHÖTTNER: *Commit Protocols for a Distributed Transactional Memory*. In: *Proceedings of the 11th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'10)*, 2010.

[75] MÜLLER, MARC-FLORIAN, KIM-THOMAS MÖLLER und MICHAEL SCHÖTTNER: *Efficient Commit Ordering of Speculative Transactions*. In: *Proceedings of the 2010 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'10)*, 2010.

[76] O'GORMAN, JOHN: *The Linux Process Manager*. Wiley, 2003.

[77] ORACLE CORP. URL: http://www.oracle.com/lang/de/products/middleware/coherence/index.html.

[78] PANTEL, LOTHAR und LARS C. WOLF: *On the Suitability of Dead Reckoning Schemes for Games*. In: *Proceedings of the 1st workshop on Network and system support for games*, NetGames '02, Seiten 79–84, New York, NY, USA, 2002. ACM.

[79] POSTEL, J.: *RFC791 - Internet Protocol*. URL: http://tools.ietf.org/search/rfc791, September 1981.

[80] RATNASAMY, SYLVIA, PAUL FRANCIS, MARK HANDLEY, RICHARD KARP und SCOTT SHENKER: *A scalable content-addressable network*. In: *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, Seiten 161–172, New York, NY, USA, 2001. ACM.

[81] REHMANN, KIM-THOMAS: *Laufende Dissertation zum Thema Objektreplikation für verteilte Speicher*. Universität Düsseldorf, 2011.

[82] REHMANN, KIM-THOMAS, MARC-FLORIAN MÜLLER und MICHAEL SCHÖTTNER: *Adaptive Conflict Unit Size for Distributed Optimistic Synchronization*. In: *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part I*, EuroPar'10, Seiten 547–559, Berlin, Heidelberg, 2010. Springer-Verlag.

[83] REKHTER, Y., ET AL.: *An Architecture for IP Address Allocation with CIDR*. URL: http://tools.ietf.org/search/rfc1518, September 1993.

[84] RICART, GLENN und ASHOK K. AGRAWALA: *An Optimal Algorithm for Mutual Exclusion in Computer Networks*. Commun. ACM, 24(1):9–17, 1981.

[85] ROBBINS, ARNOLD: *Linux Programming: User-Level Memory Management*. Computerworld, Mai 2004.

[86] ROGERS, MICHAEL, CHRISTOPHER DIAZ, RAPHAEL FINKEL, JAMES GRIFFIOEN und JAMES E. LUMPP JR.: *BTMD: Small, Fast Diffs for WAN-Based DSM*. In: *The Second International Workshop on Software Distributed Shared Memory (in conjunction with The International Conference of Supercomputing)*, Mai 2000.

[87] ROMBERG, MATHILDE: *The UNICORE Grid Infrastructure*. Sci. Program., 10:149–157, April 2002.

[88] ROSSBACH, CHRISTOPHER J., OWEN S. HOFMANN und EMMETT WITCHEL: *Is Transactional Programming Actually Easier?* SIGPLAN Not., 45:47–56, Januar 2010.

[89] ROWSTRON, ANTONY I. T. und PETER DRUSCHEL: *Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems*. In: *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Seiten 329–350, London, UK, 2001. Springer-Verlag.

[90] SAHA, BRATIN, ALI-REZA ADL-TABATABAI und QUINN JACOBSON: *Architectural Support for Software Transactional Memory*. In: *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, Seiten 185–196, Washington, DC, USA, 2006. IEEE Computer Society.

[91] SAROIU, STEFAN, KRISHNA P. GUMMADI und STEVEN D. GRIBBLE: *Measuring and analyzing the characteristics of Napster and Gnutella hosts*. Multimedia Syst., 9(2):170–184, 2003.

[92] SCHÖTTNER, MICHAEL, STEFAN TRAUB und PETER SCHULTHESS: *A transactional DSM Operating System in Java*. In: *In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applicattions, Las Vegas*, 1998.

[93] SHAVIT, NIR und DAN TOUITOU: *Software Transactional Memory*. In: *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, Seiten 204–213, New York, NY, USA, 1995. ACM.

[94] SONNENFROH, MICHAEL: *Ein datenzentriertes Programmiermodell für verteilte virtuelle Welten*. Doktorarbeit, Heinrich-Heine-Universität, Düsseldorf, Germany, 2010.

[95] SONNENFROH, MICHAEL, TOBIAS BÄUERLE, PETER SCHULTHESS und MICHAEL SCHÖTTNER: *Sharing In-Memory Game States*. In: *Proceedings of the 11th International Conference on Parallel and Distributed Computing, Applications, and Technologies (PDCAT)*, Los Alamitos, CA, USA, Dezember 2010. IEEE Computer Society.

[96] SONNENFROH, MICHAEL, MARC-FLORIAN MÜLLER, KIM-THOMAS MÖLLER und MICHAEL SCHÖTTNER: *Speculative Transactions for Distributed Interactive Applications*. In: *Proceedings of the 2010 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'10)*, 2010.

[97] STOICA, ION, ROBERT MORRIS, DAVID KARGER, M. FRANS KAASHOEK und HARI BALAKRISHNAN: *Chord: A scalable peer-to-peer lookup service for internet applications*. In: *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, Seiten 149–160, New York, NY, USA, 2001. ACM.

[98] TANNENBAUM, ANDREW: *Moderne Betriebssysteme*. Pearson Studium, 2003.

[99] TERRACOTTA INC.: *A Technical Introduction to Terracotta*, 2008. White Paper.

[100] TITOS, RUBÉN, MANUEL E. ACACIO und JOSÉ M. GARCÍA: *Speculation-based Conflict Resolution in Hardware Transactional Memory*. In: *IPDPS '09: Proc. 23rd International Parallel and Distributed Processing Symposium*, Mai 2009.

[101] VALLÉE, GEOFFROY, RENAUD LOTTIAUX, LOUIS RILLING, JEAN-YVES BERTHOU, IVAN DUTKA-MALHEN und CHRISTINE MORIN: *A Case for Single System Image Cluster Operating Systems: the Kerrighed Approach*. Parallel Processing Letters, 13(2), Juni 2003.

[102] VOGELS, WERNER: *Eventually Consistent*. Queue, 6:14–19, Oktober 2008.

[103] VOLOS, HARIS, ANDRES JAAN TACK, NEELAM GOYAL, MICHAEL M. SWIFT und ADAM WELC: *xCalls: Safe I/O in Memory Transactions*. In: *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, Seiten 247–260, New York, NY, USA, 2009. ACM.

[104] WENDE, MORITZ: *Kommunikationsmodell eines verteilten virtuellen Speichers*. Doktorarbeit, Universität Ulm, Ulm, Germany, 2003.

[105] WOLF, JÜRGEN: *Linux-UNIX-Programmierung*, Band 2. Auflage. Galileo Press, 2006.

[106] XTREEMOS CONSORTIUM: *XtreemOS: a Vision for a Grid Operating System*, 2008. White Paper.

[107] ZHAO, BEN Y., JOHN D. KUBIATOWICZ und ANTHONY D. JOSEPH: *Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing*. Technischer Bericht, Berkeley, CA, USA, 2001.

Stichwortverzeichnis

U

V

W

X

Z