Diplomarbeit

Transaktionen für verteilte Wikis auf strukturierten Overlay-Netzwerken

Stefan Plantikow

02.04.2007

Betreuer/Gutachter:

Prof. Dr. Alexander Reinefeld

Prof. Dr. Miroslaw Malek

Ich, Stefan Plantikow, erkläre, dass ich die vorliegende Diplomarbeit selbtständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Berlin, den 02.04.2007

Inhaltsverzeichnis

1 Einleitung

Internetbasierte Informationssysteme sind heute ein integraler Bestandteil des alltäglichen Lebens. E-Commerce-Anwendungen, Online-Auskunftssysteme und soziale Netzwerke ersetzen zunehmend ihre traditionellen Pendants. Sie kennzeichnen sich durch den gleichzeitigen Zugriff vieler Nutzer auf eine gemeinsame Datenbasis.

Besonders interessant sind dabei jene Systeme, deren Mehrwert durch die Interaktion ihrer Benutzer miteinander entsteht. Typische Vertreter solcher gemeinschaftlich benutzter Informationssysteme sind Bookmark-Sharing-Systeme, Kontaktbörsen, Auktionssysteme – und Wikis.

Ein Wiki ist ein Content-Management-System, das Zugriffe nach dem Prinzip des geringsten Aufwands[1] erlaubt. Ein Wiki besteht aus mehreren Seiten, die in einem einfachen Textformat verfasst, und durch einen eindeutigen Titel (Name der Seite) bezeichnet werden. Steht ein solcher Name in einer Wikiseite, wird er bei der Darstellung automatisch durch einen Hyperlink ersetzt. Alle Seiten dürfen von jedem Benutzer der Systems gelesen und bearbeitet werden. Jede kleine Korrektur trägt dazu bei, Menge und Qualität des Inhalts zu verbessern. Das entstehende Wiki als Ganzes wird so zum Mehrwert für jeden einzelnen Benutzers.

Die Viralität und Nützlichkeit dieses Prinzips wird eindrucksvoll durch Projekte wie die Online-Enzyklopädie Wikipedia[2] belegt. Doch mit zunehmender Größe entstehen neue Probleme: Administrations- und Ressourcenaufwand nehmen zu, Lastverteilung erweist sich oft als schwierig. Jeder neue Nutzer erhöht zwar den Mehrwert des Systems, jedoch gleichzeitig auch seine Kosten. Wünschenswert wäre daher ein Ansatz, der nicht nur den Nutzen sondern auch die Lasten auf alle Nutzer verteilt.

Wie kann ein derartiger Ansatz aussehen und was muss er leisten, um den Anforderungen typischer Informationssysteme gerecht zu werden?

1.1 Informationssysteme mit verteilter Datenhaltung

Informationssysteme werden heute oft als Three-Tier-Webanwendungssysteme implementiert und setzen somit den Betrieb einiger weniger zentraler Server voraus. Dieser Ansatz vereinfacht Entwurf und Einsatz der Software, da auf bewährte Komponenten, wie Webframeworks und Datenbankmanagementsysteme zurückgegriffen werden kann.

Der klassische Three-Tier-Ansatz bietet jedoch keine Verteilung der Systemaufgaben. Der Einsatz eines verteilten Datenbankmanagementsystems ermöglicht zwar eine begrenzte Dezentralisierung, ist aber weit davon entfernt, eine tatsäch-

[1] Wiki bedeutet auf Hawaianisch „schnell“

[2] http://www.wikipedia.org

lich *faire Verteilung* der Datenmanagementaufgaben durch die Client-Rechner aller Systemnutzer zu realisieren.

Der Einsatz eines strukturierten Overlay-Netzwerks (structured overlay network, kurz: SON) ist eine interessante Alternative zu tradionellem Client-Server-Computing. Anstatt relevante Daten auf einem zentralen Server zu speichern, bilden alle Computer (Peers) gemeinsam ein virtuelles Netzwerk, das allen Knoten Datenspeicherungs- und Datenabfragefunktionen bietet (Abbildung 1.1). Durch die Teilnahme am Netzwerk wird einerseits das Recht erlangt, dieses zu benutze, und andererseits müssen eigene Ressourcen für dessen Betrieb eingebracht werden.

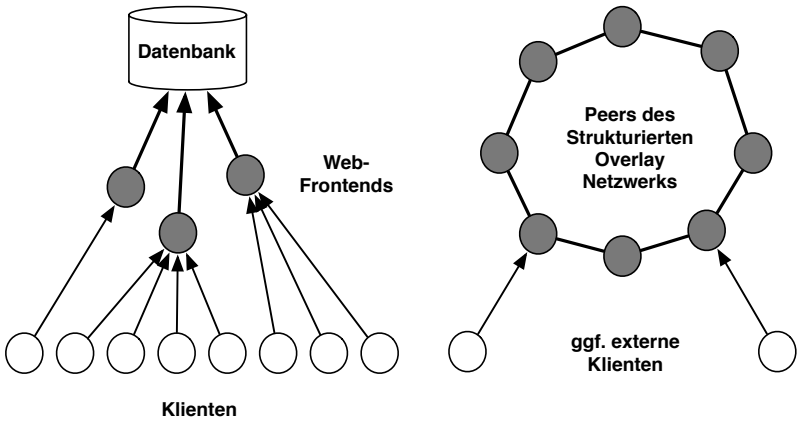

Abbildung 1.1: Architekturvergleich: Three-Tier Client-Server-System und Strukturiertes (Peer-To-Peer) Overlay-Netzwerk

Overlay-Netzwerke erscheinen somit als ideale Speicherstruktur für Informationssysteme, da sie auf natürliche Art und Weise gleichzeitig Lastverteilung, Speicherung und effizienten Zugriff realisieren. Doch ist dies genug?

1.2 Gemeinschaftliche Informationssysteme und Konsistenz

Jedes Informationssystem erfordert die Einhaltung spezifischer Sicherheitsinvarianten. Jede Operation muss ihre Einhaltung sicherstellen, wenn das korrekte Funktionieren des gesamten Systems garantiert werden soll. So darf beispielsweise bei einer Überweisung zwischen zwei Konten „kein Geld verloren gehen“. Wäre dies der Fall, würde die Invariante „Bei einer Überweisung innerhalb der Bank, ändert sich die Gesamtsumme des Geldes auf allen Konten nicht“ verletzt werden.

Als *gemeinschaftliches Informationssystem* wird in dieser Arbeit jede Form „sozialer Software“ (social software) bezeichnet, deren Mehrwert insbesondere durch die gemeinsame gegenseitige Verfügbarmachung aller gespeicherten Daten durch alle Nutzer des Systems entsteht und die keine besonderen Anforderungen an Sicherheit im Sinne von Vertraulichkeit und Geheimhaltung stellen. Da sich diese Arbeit auf derartige Systeme beschränkt, werden die Begriffe Anwendung und gemeinschaftliches Informationssystem synonym gebraucht.

1.2.1 Anwendungskonsistenz

Als *Anwendungsoperationen* wird im Folgenden eine endliche Abfolge von primitiven *Schritten* bezeichnet. Wo dies nicht zu Mehrdeutigkeit führt, werden Schritte auch gelegentlich als Basisoperationen bezeichnet und die Begriffe Anwendungsoperation und Transaktion synonym behandelt.

Im Folgenden wird *Anwendungskonsistenz* als die vollständige Einhaltung aller spezifizierten Invarianten einer Anwendung durch jede von ihr ausgeführte Anwendungsoperation verstanden. Dabei wird in dieser Arbeit Anwendungskonsistenz ausschließlich hinsichtlich der Interaktion zwischen Anwendungsklienten und Datenspeicher betrachtet.

Anwendungskonsistenz lässt sich durch mehrere Bedingungen sicherstellen. Zunächst ist zu fordern, dass die isolierte Ausführung der Schritte („Kontostand abfragen, Kontostand verändern“) einer Anwendungsoperation („Überweisung“) in richtiger Reihenfolge keine Invariante verletzt.

Auch bei gleichzeitiger Ausführung mehrerer Operationen muss die Konsistenz der Anwendung erhalten bleiben. Wenn dies nicht möglich ist, werden die den Konflikt auslösende Anwendungsoperationen abgebrochen. Auch andere Ursachen können einen derartigen Abbruch bewirken, wie beispielswiese der Absturz des Klienten. In jedem Fall dürfen alle Schritte einer Operation nur entweder vollständig und in der richtigen bzw. einer ergebnisäquivalenten Reihenfolge oder aber gar nicht ausgeführt werden, da ansonsten der Erhalt der Anwendungskonsistenz nicht garantiert werden kann. Auswirkungen von bereits vorläufig durchgeführten Schritten müssen zum Zeitpunkt des Abbruchs wieder aufgehoben werden. Außerdem ist es wünschenswert, dass das Ergebnis einer erfolgreich ausgeführten Anwendungsoperation permanent gespeichert wird und somit bei einem unerwarteten Ausfall eines Knotens des Systems nicht vernichtet oder verändert wird.

Nicht alle Anwendungen besitzen dieselben strenge Konsistenzanforderungen. Wie stark diese tatsächlich sein müssen wird genauer in Abschnitt 3.2 besprochen. Der Erhalt der Anwendungskonsistenz bei erfolgreicher, isolierter Ausführung jeder Operation liegt in der Verantwortung des Anwendungsentwicklers. Die Erfüllung der weiteren Bedingungen obligt jedoch dem Datenspeicher eines Informationssystems und allen Schnittstellen für den Zugriff auf diesen.

1.2.2 Transaktionen in relationalen Datenbanksystemen

Relationale Datenbanksysteme (RDBS) verwenden *Transaktionen* [27] als eine nützliche Abstraktion, die mehrere Schritte zusammenfasst und Konflikte bei gleichzeitigem Zugriff durch mehrere Transaktionen verhindert.

Der Transaktionsmechanismus eines Datenbankmanagementsystems garantiert für die von ihm ausgeführten Transaktionen vier Eigenschaften: *Atomizität*, *Isolation*, *Konsistenz* und *Dauerhaftigkeit* – dass sogenannte ACID[3]-Prinzip. Diese sind analog zu den Bedingungen des vorigen Abschnitts:

[3] engl. Atomicity, Isolation, Consistency, Durability

Atomizität garantiert, dass entweder alle oder aber kein Schritt der Transaktion ausgeführt werden. *Isolation* erzeugt die Illusion, dass jede Transaktion exklusiv auf den Daten operiert, d.h. gleichzeitig ablaufende Transaktionen werden so umgeformt, als ob sie nacheinander ausgeführt würden. *Konsistenz* bedeutet hier, dass durch die Transaktion die Struktur der Datenbank nicht beschädigt wird und Daten nur gemäss der Transaktion verändert werden. *Dauerhaftigkeit* verlangt, dass nach dem Abschluss einer Transaktion von ihr veränderten Daten permanent gespeichert sind. D.h. diese müssen auch nach einem „Absturz“ und Neustart des Datenbankservers oder -Klienten verfügbar sein.

Transaktionen sind ein geeignetes Mittel, um konsistente Anwendungsoperationen zu realisieren. Zusätzlich können sie benutzt werden, um mehrere Operationen atomar zusammenzufassen. Anwendungen können den Transaktionsmechanismus eines Datenbankmanagementsystems benutzen, um Anwendungskonsistenz zu garantieren.

1.2.3 Konsistenz in strukturierten Overlay-Netzwerken

Die verteilte Hash-Tabelle (Distributed Hash Table, kurz: DHT) ist die typische Datenstruktur, die mittels strukturierter Overlays (SONs) realisiert wird. Daten können unter einem Schlüssel hinterlegt bzw. mittels diesem abgefragt werden.

Die Begriffe SON und DHT werden oft synonym gebraucht. Es gibt jedoch auch Overlay-Netzwerke, die kein Hashing einsetzen [61]. DHTs sind somit spezielle SONs. In dieser Arbeit wird daher der allgemeinere Begriff des strukturierten Overlay-Netzwerks benutzt, außer wenn explizit auf klassische DHTs verwiesen werden soll.

Einige DHTs verfügen über eine primitive Suchfunktionalität. Zugriffsschutzmechanismen werden nur selten implementiert. Darüber hinaus wird häufig kein Mechanismus geboten, um Konsistenzgarantien für gleichzeitige Lese- und Schreibzugriffe durch mehrere Nutzer zu gewährleisten. Ohne Konsistenzgarantien des Datenspeichers ist es jedoch unmöglich Anwendungskonsistenz des Informationssystems zu gewährleisten.

Der Mangel an Konsistenzmechanismen ist ein Hinderungsgrund für die Entwicklung von verteilten, gemeinschaftlichen Informationssystemen, deren Datenhaltung mittels eines strukturierten Overlay-Netzwerks erfolgen soll. Dies gilt um so mehr, da die typische Implementierungen von Informationssystemen durch das Vorhandensein von Transaktionen relationaler Datenbankmanagementsysteme und deren Datenmodell geprägt sind. Die Erweiterung bestehender Overlay-Netzwerke um einen geeigneten Transaktionsmechanismus ist daher ein sinnvoller nächster Schritt.

1.3 Zielstellung und Gliederung

Es existiert nur wenig Literatur über generische Transaktionsverfahren für strukturierte Overlay-Netzwerke. MUTHITACHAROEN ET AL. [47] benutzen und implementieren eine Variante des Paxos-Algorithmus [39] um atomare Lese- und Schreiboperationen zu realisieren. Dies genügt prinzipiell für die Realisierung eines Transaktionsverfahrens, da damit das atomare Festschreiben von Trans-

aktionen mittels Commit-Records (Unterabschnitt 2.5.2) implementiert werden kann. Eine genaue Beschreibung und Implementation dieses Transaktionsverfahren wird jedoch in [47] nicht gegeben und statt dessen auf die Arbeit von LYNCH ET AL. [42] verwiesen.

MESAROS ET AL. [44] schlagen den Einsatz eines Sperrprotokolls vor, dass Verklemmungsvermeidung durch Transaktionsprioritäten erreicht. Die Arbeit geht jedoch von idealisierten Gegebenheiten des Overlay-Netzwerks aus: Keine Replikation und garantierte, verlässliche Kommunikation zwischen beliebigen Knoten. Beide Bedingungen sind für derzeit bekannten Implementationen von strukturierten Overlays nicht realistisch und somit die Entwicklung neuer Techniken erforderlich.

Das Ziel dieser Arbeit ist, die erforderlichen, grundsätzlichen Bedingungen für die Durchführung von Transaktionen in strukturierten Overlay-Netzwerken zu erarbeiten und darauf aufbauend ein konkretes, an diese Bedingungen angepasstes Transaktionsverfahren zu entwickeln. Die Aufgabe eines solchen Transaktionsverfahrens ist es, die Umsetzung von Anwendungsoperationen gemeinschaftlicher Informationssysteme und ihrer Konsistenzanforderungen zu ermöglichen.

Die Beschränkung auf gemeinschaftliche Informationssysteme erfolgt, da sie keine besonderen Sicherheitsanforderungen stellen. Sicherheit von SONs ist Gegenstand aktiver Forschung und wird in dieser Arbeit nicht behandelt. Außerdem profitieren große, gemeinschaftliche Informationssysteme besonders von der Verwendung eines Peer-to-Peer-Ansatzes. Während eine hohe Anzahl von Nutzern für zentralistische Systeme die Bereitstellung vieler Ressourcen erfordert, kehrt sich dieser Nachteil in dezentralen Systemen in einen Vorteil um: Je mehr Nutzer am System teilnehmen, desto mehr Ressourcen sind verfügbar.

Gliederung

Nach einer Beschreibung notwendiger Grundlagen in Kapitel 2 wird als Ergebnis dieser Arbeit in Kapitel 3

- eine Notation für Algorithmen entwickelt, die ein strukturiertes Overlay-Netzwerk als Datenspeicher verwenden,
- Anwendungsoperationen typischer, gemeinschaftlicher Informationssysteme und ihre Anforderungen an ein Transaktionsverfahren analysiert,
- ein neuer Ansatz für den Umgang mit Replikation und Knotenfluktuation (churn), das Zellenmodell, und ein darauf aufbauendes, für Overlay-Netzwerke geeignetes, optimistisches Transaktionsverfahren entworfen.

2 Grundlagen

In diesem Kapitel werden die nötigen Grundlagen für die Entwicklung eines Transaktionsverfahrens für SONs behandelt. Abschnitt 2.1 behandelt strukturierte Overlays am Beispiel von Chord und CAN. Zusätzlich werden häufig auftretende Fehler und geeignete Systemmodelle für SONs in Abschnitt 2.2 erläutert. Im Anschluß gibt Abschnitt 2.3 einen Überblick über klassische Transaktionsverfahren. Der Schwerpunkt liegt dabei auf Techniken für die Nebenläufigkeitskontrolle (Abschnitt 2.4) und verteiltem Transaktionsmanagement (Abschnitt 2.5).

2.1 Strukturierte Overlay-Netzwerke

Overlay-Netzwerke sind ein Lösungsansatz für die verteilte Speicherung von Daten in einem dezentralen Netzwerk aus gleichberechtigten Knoten (peer). Oberhalb eines vorhandenen Kommunikationsmediums, wie z.B. dem Internet, wird ein zusätzliches, virtuelles Overlay-Netzwerk gebildet. Knoten senden Nachrichten über das Overlay, um gespeicherte Information abzufragen oder zu verändern.

In unstrukturierten Overlays, wie z.B. Gnutella [43, 56], kennt jeder Knoten nur einige beliebige Nachbarn und kann bei der Weiterleitung von Nachrichten auf keine zusätzliche Routing-Information zurückgreifen. Empfange Nachrichten, für die ein Knoten nicht zuständig ist, werden an seine Nachbarn weitergeleitet (flooding). Das hat zur Folge, dass verschiedene Knoten unterschiedliche Ergebnisse für die selbe Anfrage erhalten können, je nachdem, welche Knoten die Anfrage jeweils erreicht hat. Zusätzlich wirkt sich Flooding negativ auf die Skalierbarkeit des Systems aus [57].

Strukturierte Overlay-Netzwerke (SONs) sollen die Vorteile unstrukturierter Overlays hinsichtlich Lastverteilung erhalten, ihre Probleme hinsichtlich Skalierbarkeit behandeln und gleichzeitig größere Funktionalität anbieten. Im Gegensatz zu unstrukturierten Systemen garantieren sie, dass alle zu einem bestimmten Zeitpunkt im System gespeicherten Daten auch prinzipiell[4] für alle daran teilnehmenden Knoten effizient erreichbar sind [21, 45, 31].

SONs realisieren einen Verzeichnisdienst (resource location service) für den Zugriff auf einzeln adressierbare Datenobjekte (Ressourcen). Der Verzeichnisdienst ermöglicht das Versenden von Nachrichten an den derzeit zuständigen Knoten der Zielressource. Dieser ist eindeutig festgelegt[4] und damit keine verschiedenen Ergebnisse für identische Anfragen möglich.

Strukturierte Overlay-Netzwerke unterscheiden sich insbesondere in zwei Punkten: Der Zuordnung von Ressourcen zu Knoten und der für die Nachrichtenzustellung eingesetzen Topologie.

[4] unter Vernachlässigung von Ausfällen und Kommunikationsintransitivität

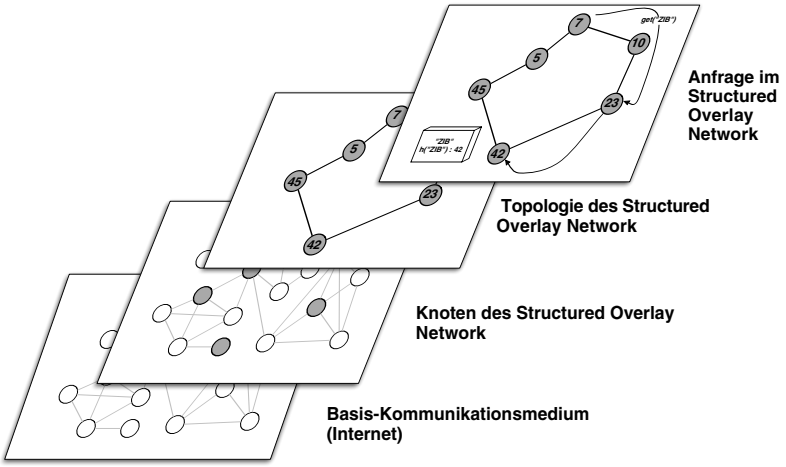

Abbildung 2.1: Architekturebenen in einem strukturierten Overlay-Netzwerk

Knoten und Ressourcen

Ressourcen und Knoten werden in einem strukturierten Overlay einander so zugeordnet, dass jeweils einzelne Knoten möglichst gleicher Last ausgesetzt sind. Jedem Knoten wird dazu eine global eindeutige Adresse gegeben. Dabei ist es wünschenswert, dass jede Adresse mit gleicher Wahrscheinlichkeit einem beliebigen neuen Knoten zugeordnet wird und somit alle vorhandenen Knoten gleichmäßig in den Adressraum verteilt werden.

Die Adressierung einzelner Ressourcen erfolgt anwendungsspezifischen durch einen Binärstring. Dieser Schlüssel wird meist mittels einer Hash-Funktion auf den Adressraum der Knoten projiziert, um alle Resourcen gleichmässig auf die vorhandenen Knoten zu verteilen. Anschließend wird je nach konkretem Verfahren die Ressource einem verfügbaren Knoten zugeordnet.

Das Hinzufügen und Entfernen von Knoten kann bewirken, dass Ressourcen neuen Knoten zugeordnet werden. Wenn der zuständige Knoten einer solchen Ressource nicht aus dem System entfernt wurde, entsteht unnötiger Kommunikationsaufwand. Dies wird durch den Einsatz von konsistentem Hashing [36] verhindert. Konsistentes Hashing garantiert, dass Ressourcen nur dann einem neuen Knoten zugeordnet werden, wenn ihr zuständiger Knoten tatsächlich ausgefallen ist. Insbesondere in Systemen mit hoher Knotenfluktuation (Churn) wird so die Anzahl der Zuständigkeitswechsel minimiert.

Routing und Topologie

Jeder Knoten verwaltet eine Routingtabelle, die Overlay-Adressen auf Adressen des zugrunde liegenden Kommunikationsmediums (z.B. Internet: IP-Adresse + Port) abbildet. Aufgrund hoher Knotenfluktuation und der Größe von strukturierte Overlays ist es nicht praktikabel, dass Knoten über eine vollständige Routingtabelle verfügen. Jeder Knoten verwaltet deshalb nur eine unvollständige

Tabelle und die Weiterleitung von Nachrichten muss u.U. über mehrere Zwischenschritte (hops) erfolgen. Overlay-Netzwerke unterscheiden sich darin, welche Routingtopologie sie verwenden, wie lokale Routingtabellen eines Knotens verwaltet und konsistent gehalten werden und wie die Nachrichtenweiterleitung im Detail erfolgt. Alle strukturierten Overlays folgen dabei dem Prinzip, bei jedem Schritt die Entfernung zum Zielknoten so zu reduzieren, dass die Anzahl der Routing-Hops möglichst logarithmisch ($O(\log n)$) bezüglich der Anzahl der Knoten n des Gesamtsystems ist.[5]

Anwendungen

Mittels des Ressourcen-Verzeichnisdienstes eines strukturierten Overlay-Netzwerks können verschiedene verteilte Datenstrukturen realisiert werden, von denen die Distributed Hash Table (DHT) wohl der typischste Vertreter ist. Ihre Basisoperationen put(*key*, *value*) und get(*key*) erlauben das Speichern und Abfragen von Daten zu einem Schlüssel über den zuständigen Knoten des Overlays. Doch auch andere Anwendungen sind möglich: Publish-Subscribe-Dienste [58], zusätzliche Indexstrukturen [52], Content Distribution [10], DNS [17] und Storage-Systeme[5].

Multicast und Suche

Für einige Anwendungen genügt es nicht, Nachrichten nur an einzelne Ressourcen schicken zu können. Stattdessen kann es erforderlich sein, die selbe Nachricht an eine Gruppe von Ressourcen zu senden (Multicast). Hier können SONs ihre Stärken ausspielen. Die Struktur ihrer Routingtabellen eignet sich hervorragend für die dynamische Erzeugung von Multicast-Bäumen. Multicast-Senden ist somit neben dem normalen Senden (put, get der DHT) eine natürliche Operation eines strukturierten Overlay-Netzwerkes [25].

SCRIBE [58] implementiert einen *Publish-Subscribe*-Dienst, in dem Abonnenten (subscriber) automatisch über neu publizierte Nachrichten zu einem Thema informiert werden. Abonnenten senden eine Subscribe-Nachricht an den Knoten für das abonnierte Thema. Während des Routings derartiger Nachrichten wird automatisch ein lastverteilender Multicast-Baum erzeugt. Beim Publizieren einer neuen Nachricht für das Thema wird der Baum benutzt, um die Nachricht an alle Abonnenten effizient auszuliefern.

Die adressierten Ressourcen einer Nachricht sind nicht immer einzeln aufzählbar. Alternativ können sind stattdessen durch ein Intervall beschrieben werden. Eine so adressierte Nachricht soll an alle Ressourcen ausgeliefert werden, deren Schlüssel in diesem Intervall liegt. Derartige Nachrichten ermöglichen Bereichssuche (range queries), da sie ganze Bereiche des Schlüsselraums abdecken. Mögliche Techniken für ihre Realisierung sind Broadcasts [20], Prefix Hash Tries [52] oder die Verwendung eines strukturierten Overlay-Netzwerks, dass kein Hashing der Ressourcenschlüssel einsetzt [62, 61].

Als Nächstes werden zwei konkrete Overlay-Netzwerke vorgestellt, Chord und CAN. Dabei werden jeweils Topologie, Routing, Konstruktion und Stabilisierung

[5] Dies gilt auch für CAN, das zwar prinzipiell in $O(n^{1/d})$ routet, jedoch in $O(\log n)$, falls $d = \log_2 n/2$ gewählt wird [53].

der SONs beschrieben. Im Anschluss daran wird eine mögliche Erweiterung für die Implementation von Bereichssuche diskutiert.

2.1.1 Chord

Topologie

Chord [63] ist das wohl bekannteste strukturierte Overlay-Netzwerk. Chord benutzt konsistentes Hashing mittels der SHA-1-Funktion. Knotenadressen werden durch Anwendung der Hash-Funktion auf die Kommunikationsadresse des Knotens im zugrunde liegenden Kommunikationsmedium gebildet. Ressourcenschlüssel werden ebenfalls durch Anwendung dieser Funktion in den Adressraum projiziert. Alle Knoten bilden gemäß ihrer Adressen gemeinsam einen *Ring*. Ressourcenschlüssel werden genau dem Knoten zugeordnet, dessen Adresse am dichtesten auf die Adresse des Ressourcenschlüssel folgt bzw. mit dieser identisch ist. Dieser Knoten wird Nachfolger-Knoten des gegebenen Ressourcenschlüssels genannt (successor). Wenn die Ressource a mit Adresse (gehashtem Schlüssel) 4 dem Knoten mit der Adresse 7 zugeordnet wird, gibt es keinen weiteren Knoten k dessen Adresse $>= 4$ und < 7 ist.

Routing

Jeder Knoten des Rings kennt seinen direkten Nachfolger-Knoten. Bereits dies genügt theoretisch um den für eine Ressource verantwortlichen Schlüssel zu kontaktieren. Dies würde jedoch im ungünstigsten Fall eine vollständige Traversierung des Ringes erfordern! Deshalb verwaltet jeder Knoten zusätzlich eine sogenannte *Finger-Tabelle*. Ein Eintrag i der Finger-Tabelle eines Knotens k mit Adresse a_k im Ring verweist auf den Nachfolger-Knoten der Adresse $a_k + 2^i$ ($1 <= i <= m$, Anzahl der möglichen Schlüssel im Ring $= 2^m$). Diese Tabelle kann genutzt werden um die Distanz zum Zielschlüssel einer Anfrage in jedem Schritt zu halbieren. Gleichzeitig ist die Größe der Tabelle jedes Knotens hinreichend klein (logarithmisch), und damit das Verfahren praktisch einsetzbar.

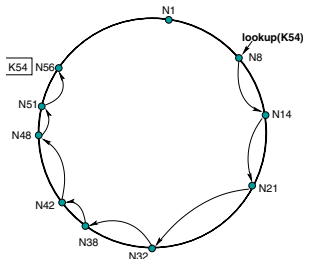

Abbildung 2.2: Routing in Chord (aus: [63])

Hinzufügen von Knoten und Stabilisierung

Das Hinzufügen eines Knoten zum System erfordert zunächst die Kenntnis eines beliebigen anderen Systemknotens. Dieser wird benutzt, um den Nachfolger des neuen Knotens zu bestimmen. Ausgehen von diesem Nachfolger werden dann all notwendigen Anfragen durchgeführt, um die Finger-Tabelle zu füllen. Doch wie erfahren vorherige Knoten im Ring von seiner Existenz?

Eine verwandte Frage stellt sich durch das plötzliche Ausscheiden von Knoten aus dem Ring. Immer wenn ein Knoten das System verlässt, müssen

die Routingtabellen anderer Knoten entsprechend korrigiert werden. Diese Korrektur leistet ein regelmässig ausgeführter Stabilisierungsalgorithmus.

Jeder Knoten verwaltet einen zusätzlichen Zeiger auf seinen Vorgänger (predecessor) im Ring. In regelmäßigen Abständen wird der Vorgänger des eigenen Nachfolgers bestimmt. Liegt dieser zwischen der eigenen Adresse und der des derzeitigen Nachfolgers, wird dieser Knoten der neue Nachfolger. Der neue Nachfolger wird über die Änderung informiert und ändert dann ggf. entsprechend seinen Vorgänger-Eintrag. Dieses einfache Verfahren genügt bereits, um die Ring-Struktur des Systems zu erhalten. Es berücksichtigt jedoch noch nicht die Korrektur der Finger-Tabellen, auf die in STOICA ET AL. [63] ausführlich eingegangen wird.

2.1.2 CAN

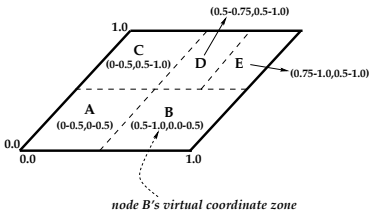

Abbildung 2.3: Routing in CAN (aus: [53])

CAN [53], das Content Addressable Network, verwendet einen mehrdimensionalen kartesischen Adressraum in der Form der Oberfläche eines d-Torus. Jeder Knoten des Systems verwaltet eine d-dimensionale rechteckige Zone des Torus. Ressourcen werden ihrem Schlüssel entsprechend einer Zone und dem für sie zuständigen Knoten zugeordnet. Jeder Knoten des Systems kennt seine unmittelbaren Nachbarn. Als Nachbar gilt dabei jeder Knoten, dessen Zone mit der eigenen in mindestens $d - 1$ Dimensionen direkt benachbart ist.

Diese Struktur ist bereits ausreichend, um unter Verwendung der jeweiligen Nachbarknoten Nachrichten zu einer beliebigen Zieladresse zu routen.

Routing

Das Routing erfolgt entlang eines Pfades vom Quell- zum Zielknoten. Dabei gibt es mehr als eine Möglichkeit, um zum Ziel zu routen. Zusätzliche Metriken, beispielsweise für die Schätzung der Kommunikationslatenz zwischen Knoten [16, 18], können verwendet werden, um jeweils den nächsten Hop zu bestimmen. Alternativ kann auch durch Ringsuche (ring search) die Routinglatenz minimiert werden. Ringsuche ist ein kontrolliertes Flooding entlang der mehrdimensionalen Ringstruktur des CAN, das heißt die gleichzeitige Benutzung alternative Pfade. Ringsuche wird auch verwendet, wenn eine Nachricht nicht zielführend weitergeleitet werden kann. Dies kann passieren, wenn aufgrund hoher Knotenfluktuation (Churn) die Routingtabelle unvollständig geworden ist.

Neuere Arbeiten [67, 7] erweitern das Routing von CAN, so dass nur $O(log\ n)$[6] Hops erforderlich sind. Dafür werden von jedem Knoten, analog

[6] In [67] ist e die optimale Basis für diesen Logarithmus unabhängig von der Anzahl der Dimensionen d

zu den Fingertabellen von Chord, zusätzliche Einträge in der Routingtabelle verwaltet.

Hinzufügen von Knoten

Hinzufügen erfordert die Kenntnis eines beliebigen Knoten des CANs. Ausgehend von diesem Knoten wird über den regulären Routingmechanismus eine Zone identifiziert, die neu aufgeteilt werden soll. Anschließend wird über den Knoten, der diese Zone verwaltet, eine Teilung (Split) der Zone durchgeführt und alle betroffenen Nachbarknoten über die Änderung informiert. Dabei werden die einzelnen Dimensionen einer Zone in einer festgelegten Reihenfolge geteilt, um eine später nötige Wiedervereinigung wegen eines Knotenausfalls zu ermöglichen.

Stabilisierung

Jeder Knoten sendet in regelmäßigen Abständen Informationen (Heartbeats) an seine Nachbarn. Diese aktualisieren ihre Routingtabellen entsprechend. Scheidet jedoch ein Knoten aus dem CAN aus, wird es notwendig, die von ihm verwaltete Zone neu zu vergeben. Nachbarn erkennen das Ausscheiden eines Knotens durch das Ausbleiben der Heartbeats. Nach einer gewissen Zeit, die proportional zur Größe der eigenen Zone gewählt wird, sendet ein Nachbar, der einen Ausfall bemerkt hat, eine *Takeover*-Nachricht an alle ehemaligen Nachbarn des ausgefallenen Knotens. Diese antworten entweder mit einer Bestätigung oder mit einer Takeover-Nachricht, falls die eigene Zone kleiner ist. Sobald ein Nachbar von allen anderen Knoten eine Bestätigung empfangen hat, übernimmt er die Zone des ausgefallenen Knotens. Dadurch wird genau derjenige Nachbar ausgewählt, der die kleinste Zone verwaltet. Dies soll verhindern, dass ein einzelner Knoten für einen überproportional großen Bereich des Adressraums zuständig ist.

Es kann vorkommen, dass zu viele benachbarte Knoten gleichzeitig ausfallen und eine konsistente Stabilisierung durch das obige Verfahren nicht möglich ist. Dann muss mittels Ringsuche (ring search) nach erreichbaren, neuen Nachbarn gesucht werden.

2.2 Fehler- und Systemmodelle strukturierter Overlay-Netzwerke

2.2.1 Systemmodelle verteilter Systeme

Bei der Betrachtung verteilter Systeme ist die Wahl des zugrunde liegenden Fehler- und Systemmodells von entscheidender Bedeutung. Ein Modell legt fest, welche Fehler im formalen Modell betrachtet und durch das verteilte System behandelt werden. Zusätzlich bestimmt es, welche Probleme im betrachteten System algorithmisch überhaupt gelöst werden können. Es existiert beispielsweise keine Lösung für Konsens-Probleme im Fail-Silent-Modell [22].

Nach GUERRAOUI, RODRIGUES [29] besteht ein Systemmodell eines verteilten Systems aus der Kombination einer Prozessabstraktion, einer Verbindungsabstraktion für das Netzwerk, gewissen zeitlichen Annahmen und ggf. einer Fehlererkennungsabstraktion (Abbildung 2.4). Jede dieser Abstraktionen garantiert die Einhaltung bestimmter Sicherheits- und Fortschrittseigenschaften (safety und liveness properties) während der Ausführung *einer Instanz* eines verteilten Algorithmus. Dieser Unterabschnitt gibt einen Überblick über die Abstraktionen und folgt dabei [29].

Prozesse und Prozessfehler

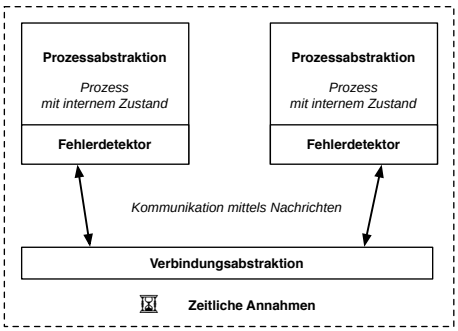

Abbildung 2.4: Systemmodellabstraktionen

Jede Prozessabstraktion beschreibt sowohl Ausführung und Kommunikation der beteiligten Prozesse, als auch mögliche Fehler (Prozessfehlermodell). Dabei wird von einem ereignisbasierten Ansatz ausgegangen. Prozesse kommunizieren durch den Austausch von Nachrichten über die gegebene Verbindungsabstraktion miteinander. Das Ausliefern einer zuvor empfangenen Nachricht löst eine atomare Aktion aus, die neue Nachrichten versendet oder den Prozesszustand ändert. Alle empfangenen Nachrichten werden gleich fair behandelt. Insbesondere wird keine Nachricht unendlich lange zurück gestellt.

Mögliche Fehlerklassen für Prozesse sind: Dauerhafter Ausfall von Prozessen (crash), Auslassen einzelner Nachrichten (omission), Ausfall von Prozessen mit späterer Wiederherstellung (crash-recovery) und byzantinisches Fehlverhalten. Prozesse werden als korrekt bezeichnet, solange sie den verteilten Algorithmus korrekt ausführen, das heißt nicht abgestürzt sind bzw. kein byzantinisches Fehlverhalten aufweisen.

Zeitliche Annahmen

In dieser Arbeit werden ausschließlich asynchrone Systeme betrachtet, das heißt, es wird keine zeitliche Obergrenze für die Dauer der Ausführung von Aktionen und das Versenden von Nachrichten angenommmen. Fortschritt und Korrektheit von Algorithmen für asynchrone Systeme sind unabhängig vom tatsächlichen Zeitverhalten eines echten Systems.

Verbindungen (links)

Verbindungsabstraktionen beschreiben die beim Nachrichtenversand geltenden Garantien, sowie die Topologie der Verbindungen zwischen den Systemprozessen. Folgend wird davon ausgegangen, dass alle Prozesse durch direkte Punkt-zu-Punkt-Verbindungen miteinander kommunizieren können, Nachrichten wäh-

rend der Übertragung nicht verändert werden und keine Einschleusung externer Nachrichten in das System erfolgt. Verbindungsabstraktionen unterscheiden sich hinsichtlich des Umgangs mit Verlust und Duplikation von Nachrichten.

Sogenannte „perfekte“ Verbindungen (perfect links) garantieren die Zustellung von Nachrichten ohne Duplikation, falls beide Prozesse korrekt sind. Gelegentlich wird zusätzlich die Zustellung in FIFO-Reihenfolge gefordert. „Sture“ Verbindungen (stubborn links) garantieren hingegen nur, dass, wenn eine Nachricht unendlich oft gesendet wird, diese auch unendlich oft empfangen wird, falls kein Kommunikationspartner abstürzt.

Fehlererkennung

Viele verteilte Algorithmen erfordern einen *Fehlerdetektor*, der Vorhersagen über die Korrektheit anderer Prozesse des Systems trifft. Fehlerdetektoren werden durch zwei Eigenschaften klassifiziert: Vollständigkeit (completeness) und Exaktheit (accuracy). Vollständigkeit beschreibt, ob und wann fehlerhafte Prozesse durch den Fehlerdetektor eines anderen Prozesses erkannt werden. Exaktheit beschreibt, in welchem Ausmaß Fehlervermutungen richtig sind.

Ein perfekter Fehlerdetektor garantiert, dass jeder fehlerhafte Prozess irgendwann von jedem korrekten Prozess permanent als fehlerhaft vermutet wird, und dass ausschließlich fehlerhafte Prozess als fehlerhaft vermutet werden. Ein *letztendlich* (eventually) perfekter Fehlerdetektor schwächt die letzte Bedingung ab und verlangt, dass erst ab einem bestimmten, beliebig späten Zeitpunkt kein korrekter Prozess von einem anderen korrekten Prozess fälschlich als fehlerhaft vermutet wird. Letztendlich perfekte Fehlerdetektoren sind wichtig, da sie im Gegensatz zu perfekten Fehlerdetektoren realisierbar sind und genügen, um eine Vielzahl von verteilten Algorithmen zu implementieren.

Klassische Systemmodelle verteilter Systeme

Die Kombination von Modellabstraktionen führt zu konkreten Systemmodellen [29]:

Fail-Stop Prozesse stürzen ab und werden nicht wiederhergestellt. Verbindungen sind perfekt. Jeder Prozess verfügt über einen perfekten Fehlerdetektor. Das Fail-Stop-Modell bietet somit idealisierte Bedingungen für den Entwurf verteilter Algorithmen.

Fail-Silent Prozesse stürzen ab und werden nicht wiederhergestellt. Verbindungen sind perfekt. Prozesse verfügen über keine Möglichkeit zur Fehlererkennung.

Fail-Noisy Prozesse stürzen ab und werden nicht wiederhergestellt. Verbindungen sind perfekt. Jeder Prozess verfügt über einen letztendlich perfekten Fehlerdetektor. Das Fail-Noisy-Modell bietet realistische Bedingungen für den Entwurf verteilter Algorithmen.

Fail-Recovery Prozesse stürzen ab, werden jedoch wiederhergestellt. Verbindungen sind stur. Optional verfügt jeder Prozess über einen letztendlich perfekten Fehlerdetektor.

Byzantinisch Eine begrenzte Anzahl von Prozessen zeigt byzantinisches Fehlverhalten. Verbindungen sind perfekt. Optional verfügt jeder Prozess über einen Fehlerdetektor.

2.2.2 Systemmodelle für strukturierte Overlay-Netzwerke

Es gibt eine Reihe von Systemmodellen, die für strukturierte Overlay-Netzwerke geeignet sind. Fail-Stop/Fail-Noisy beschreibt einfache Knotenausfälle, das Spam-Modell betrachtet Knoten, die zwar das Protokoll korrekt ausführen, jedoch fehlerhafte Daten erzeugen, und byzantinische Fehlermodelle erlauben beliebiges Kommunikationsverhalten fehlerhafter Knoten [48].

In Systemmodellen für strukturierte Overlay-Netzwerke muss zusätzlich unterschieden werden, welche Knoten fehlerhaft sein können. Im randomisierten Fehlermodell werden einzelne Knoten mit einer gewissen Wahrscheinlichkeit zufällig fehlerhaft. Im nicht-randomisierten Fehlermodell wählt ein Angreifer unter Verwendung von Wissen über den Zustand des Gesamtsystems eine Menge von Knoten aus, die dann fehlerhaft werden. Verwandt hiermit ist die *Sybil Attack*, bei der ein Angreifer die Knoten nicht auswählt, sondern einfach beliebig viele, eigene, fehlerhafte Knoten zum System hinzufügt, bis dieses zusammenbricht bzw. von ihm übernommen wird.

Techniken für den Umgang mit byzantinischen Fehlern werden von CASTRO ET AL. [9] besprochen. Es werden kryptographische Zertifikate verwendet, um die Identität von Knoten zu verifizieren. Die Vergabe von Zertifikation erfolgt entweder durch einen externen, kostenpflichtigen Dienst oder erfordert das Lösen von Rechenaufgaben. Dadurch werden die Kosten für die Erzeugung vieler Zertifikate erhöht und somit Sybil Attacks verhindert bzw. zumindest verteuert. Ist die Identität von Knoten sichergestellt, können byznatinische Fehler durch redundantes Routing, die Erkennung und Vermeidug fehlerhafter Routen und die digitale Signierung von Daten vermieden werden. Letzteres ist auch ein möglicher Ansatz für den Umgang mit Spam.

In dieser Arbeit wird jedoch auf die Betrachtung derartiger Fehler verzichtet, da der Schwerpunkt auf der Entwicklung eines geeigneten, verteilten Transaktionsverfahrens für die Bedingungen strukturierter Overlay-Netzwerke liegt. Im Folgenden wird deshalb von einem klassischen Fail-Noisy-Systemmodell ausgegangen. Knoten sind prinzipiell gutartig, können jedoch ausfallen. Ausfälle werden duch andere Knoten in endlicher Zeit durch die Verwendung ihres lokalen Fehlerdetektors erkannt.

2.2.3 Fehler des Kommunikationsmediums

Jenseits ihrer Topologie und der verwendeten Algorithmen entstehen in der Praxis einige Probleme während des Einsatzes strukturierter Overlay-Netzwerke, die ihre Benutzung erschweren. Es wird meist davon ausgegangen, dass jeweils zwei Knoten eines Overlay-Netzwerks miteinander direkt kommunizieren können. Dies ist jedoch in der Praxis durchaus nicht der Fall. So kann es vorkommen, dass sowohl die Knoten A und B als auch die Knoten B und C direkt miteinander kommunizieren können, jedoch nicht A und C. Die Knoten A und C vermuten sich dann gegenseitig fälschlich als ausgefallen (Intransitivitätsproblem). FREED-

MAN ET AL. geben an, dass zu einem beliebigen Zeitpunkt mit einer Wahrscheinlichkeit von 26 % zwei beliebige Knoten nicht direkt miteinander kommunizieren können [24]. Im Folgenden werden verschiedene Teilaspekte und mögliche Lösungsansätze dieses Problems in Anlehung an [24] beschrieben.

Unsichtbare Knoten

Ein Knoten ist unsichtbar, wenn ein anderer Knoten *indirekt* von seiner Existenz erfährt, aber nicht mit ihm direkt kommunizieren kann. Ein einfache Lösung ist, nur Knoten in die eigene Routingtabelle aufzunehmen, mit denen direkt kommuniziert werden kann. Unsichtbare Knoten werden hingegen unter Verwendung des SON kontaktiert. Zusätzlich gibt es Techniken, um den negativen Einfluss von Knotenunsichtbarkeit auf das Routing zu reduzieren (nach [24]):

- Minimierung von Timeouts durch den Einsatz von Latenz-Schätzungen (z.B. mittels Netzwerkkoordinaten [16]),
- Paralleles Senden von Nachrichten,
- Caching nicht direkt erreichbarer Zieladressen.

Routing-Schleifen

In der Chord-Variante i3 können durch Intransitivität Endlosschleifen während des Routings auftreten [24].

Angenommen in einem Routingschritt ist der Zielknoten R einer Nachricht der (eigentliche) Nachfolger des routenden Knotens P im Ring. Da mit diesem jedoch nicht direkt kommuniziert werden kann, wird er übersprungen. Im nächsten Routingschritt bemerkt der routende Knoten Q, dass er für die Nachricht nicht zuständig ist. Q sendet die Nachricht bei i3 nun nicht an seinen Vorgänger sondern an seinen Nachfolger in der anderen Ringhälfte. Von dort wird die Nachricht erneut zu P weitergeleitet und dadurch eine Routing-Schleife gebildet.

Um diese zu verhindern genügt es, während des Routings eine totale Ordnung über allen Knoten zu definieren. A sendet eine Nachricht mit dem Schlüssel k nur dann an B weiter, wenn $|B - k| < |A - k|$. Dabei ist „$-$“ die Subtraktion $\bmod n$. Eine Nachricht wird in jedem Schritt dichter zum Zielknoten geroutet, bis sie diesen erreicht hat.

Fehlerhafte Antwortpfade

Die meisten Nachrichten erfordern eine Antwort an den Sender (z.B. die Daten eines `get`-Requests einer DHT). Dafür ist es günstig direkt, statt über das SON, zu antworten. Ist das nicht möglich, muss die Antwort zurück über das SON gesendet werden. Dies ist jedoch zeitaufwändig. Einfacher ist es, einen beliebigen anderen Knoten als Relay-Station zu verwenden. FORD ET AL. [23] diskutieren diese Vorgehensweise genauer für die Internet-Kommunikation zwischen Knoten, die sich hinter einer Firewall mit Network Address Translation

(NAT) befinden und schlagen den Einsatz von UDP- und TCP-Hole-Punching-Techniken vor.

2.2.4 Konsistentes Routen und Knotenfluktuation (Churn)

Knotenfluktuation und Replikation

In jedem strukturierten Overlay-Netzwerk werden ständig alte Knoten entfernt und neue Knoten hinzugefügt. Diese Knotenfluktuation (churn) ist problematisch, da sie zu Datenverlust und damit aus Sicht der Anwendung zu fehlerhaften Reaktionen auf empfangene Nachrichten führen kann. Knotenfluktuation kann teilweise durch den Einsatz von Replikation behandeln werden (z.B. [47, 26]). MUTHITACHAROEN ET AL. [47] schlagen die Verwendung von replizierten Zustandsmaschinen (replicated state machines) vor, um Daten atomar lesen und schreiben zu können.

Konsistentes und zuverlässiges Routen

Konsistentes Routen [8] bedeutet, dass Knoten des Overlays Nachrichten nur dann an die Anwendungsschicht ausliefern, wenn sie tatsächlich für die Zieladresse der Nachricht verantwortlich sind. Durch Knotenfluktuation werden Routingtabellen einzelner Knoten fehlerhaft. Dies wird frühesten während der nächsten periodische Stabilisierung korrigiert und kann davor zu inkonsistentem Routing führen.

Zuverlässiges Routen (dependable routing) erfordert zusätzlich die garantierte Auslieferung der Nachricht durch den zuständigen Knoten [8]. Nachrichten dürfen nicht verlorengehen. Das kann durch die Verwendung von Bestätigungsnachrichten in jedem Routing-Schritt erreicht werden. Bei Ausbleiben der Bestätigung wird die Nachricht über eine alternative Route erneut gesendet. Zielknoten müssen zusätzlich Nachrichtenduplikate herausfiltern.

CASTRO ET AL. [8] geben für MSPastry einen Algorithmus an, der zuverlässiges Routen realisiert. Das Verfahren basiert auf der Annahme, dass jeder Knoten mindestens einen korrekten Knoten in seiner Routingtabelle (Pastrys Leaf-Set) hat und korrekte Knoten *nie* fälschlich als ausgefallen vermutet werden. Dies scheint unrealistisch.

GHODSI [25] beschreibt für Chord eine Situationen, in der zwei Knoten des Systems gleichzeitig davon ausgehen, dass sie für eine bestimmte Ressource zuständig sind und keine Möglichkeit besitzen, um diese Situation zu erkennen. Diese Lookup-Inkonsistenz (inconsistent lookup) hat weitreichende Implikationen für die konsistente Datenhaltung mittels SONs (Abschnitt 3.3).

Sowohl GHODSI als auch JOHNSON [35] beschreiben eine mögliche Lösung. Während des regulären Hinzufügens bzw. Entfernens von Knoten werden alle betroffenen Routingtabellen durch die Verwendung eines atomaren Festschreibprotokolls (atomic commit) konsistent gehalten. Entweder wird ein Knoten korrekt hinzugefügt bzw. entfernt oder aber Routingtabellen werden bezüglich dieses Knotens nicht geändert. Beide Arbeiten bieten jedoch keine Lösung für Lookup-Inkonsistenz, wenn Knoten das Overlay nicht ordnungsgemäß verlassen.

GHODSI [25] beweist, *dass es in einem Overlay mit Ringstruktur nicht möglich ist, gleichzeitig Lookup-Konsistenz, Verfügbarkeit*[7] *und Tolerierbarkeit von Netzwerkpartitionen zu ermöglichen*. Es ist nach [25] ungeklärt, ob Lookup-Konsistenz und Verfügbarkeit in Overlay-Netzwerken mit Ringstruktur möglich ist, die keine Partitionierung aufweisen. GHODSI [25] hält dies jedoch für unwahrscheinlich, da fehlerhafte Fehlerdetektoren für einen beliebig langen Zeitraum eine künstliche Netzwerkpartition bewirken können.

CHEN ET AL. [15] untersuchen Routingkonsistenz aus dem Blickwinkel von Systemen für Gruppenkommunikation. Die Arbeit definiert schwache Lookup-Konsistenz (weak key-based routing, kurz: W-KBR). Diese darf optional fehlschlagen, muss jedoch den Sender darüber informieren und garantiert Konsistenz und Fortschritt (progress), falls das strukturierte Overlay sich stabilisiert. CHEN ET AL. geben an, dass W-KBR im voll-asynchronen Modell nicht implementiert werden kann, da es äquivalent zum Ω-Fehlerdetektor ist [12, 22]. Sie empfehlen daher die Untersuchung minimal nötiger Synchronizitätsannahmen als ein sinnvolles, zukünftiges Forschungsziel.

2.3 Transaktionsverfahren im Überblick

Dieser Abschnitt beschreibt bekannte klassische Transaktionsverfahren, wie sie üblicherweise in relationalen Datenbankmanagementsystemen eingesetzt werden. Das Thema Transaktionsverfahren ist sehr umfangreich und eine vollständige und detaillierte Darstellung hier nicht möglich. Deshalb wird lediglich ein Überblick gegeben, der Schwerpunkt liegt dabei auf Nebenläufigkeitskontrolle und Verteilung. Weiterführendes Material enthält das Buch von WEIKUM und VOSSEN [66], das auch die wesentliche Grundlage für Inhalt und Notation der folgenden Abschnitte zu Transaktionsverfahren bildet.

ACID

Die vorgestellten Techniken realisieren das in der Einleitung bereits erwähnte ACID-Prinzip für alle gespeicherten Inhalte des Systems. Es beinhaltet:

Atomizität (atomicity)**:** Alle Operationen einer Transaktion werden entweder vollständig oder gar nicht ausgeführt.

Isolation: Die Transaktion wird so ausgeführt, als ob sie „allein“ auf den Daten operierern würde. Das heißt für jede Transaktion sind die durch gleichzeitig ablaufende Transaktionen durchgeführten Veränderungen unsichtbar.

Konsistenz (consistency)**:** Durch die Ausführung einer Transaktion ist es nicht möglich, einen inkonsistenten Zustand des Datenspeichers zu erzeugen.

Dauerhaftigkeit (durability)**:** Alle durch eine erfolgreich ausgeführte Transaktion vorgenommenen Änderungen werden *dauerhaft* gespeichert, das heißt in nicht-flüchtigem, permanentem Speicher. Dauerhaftigkeit muss gewährleistet sein, bevor der Initiator der Transaktion über die erfolgreiche Ausführung informiert wird.

[7] Verfügbarkeit wird hier schwach als (beliebig verzögerbare) garantierte Auslieferung der Antwort auf eine Nachricht verstanden, die an einen nicht ausfallenden Knoten gesendet wurde.

Aspekte eines Transaktionsverfahrens

Jedes Transaktionsverfahren lässt sich in zwei Aspekte gliedern, deren Realisierung dann insgesamt die Einhaltung der ACID-Garantien gewährleistet.

Nebenläufigkeitskontrolle (concurreny control) garantiert die isolierte Ausführung gleichzeitig ablaufender Transaktionen und ist damit der Kern jedes Transaktionsverfahrens. Nebenläufigkeitskontrolle wird durch den sog. Scheduler (*engl.* Planer) realisiert. Ein Scheduler bestimmt für jede eingehende, auszuführende Basisoperation, ob sie sofort durch den Datenmanager auszuführen ist, ob sie abzulehnen und damit die Transaktion abzubrechen ist oder aber ob ihre Ausführung bis zu einem späteren Zeitpunkt (maximal bis zum Festschreiben oder aber Abbrechen der Transaktion) zu verzögern ist.

Recovery-Komponente behandelt den Abbruch (abort) bzw. das finale Festschreiben (commit) von Transaktionen und garantiert damit Atomizität und Dauerhaftigkeit. Teilaufgaben einer Recovery-Komponente sind die Behandlung von Transaktionsabbrüchen (transaction recovery), atomares Festschreiben (atomic commit) und Absturzbehandlung (crash recovery), das heißt die Wiederherstellung des Systemzustands nach einem Absturz. Dabei wird verlangt, dass festgeschriebene Daten durch einen Absturz nicht gelöscht oder beschädigt werden.

Konsistenz ist eine Sicherheitseigenschaft, die durch beide Teilaspekte eines Transaktionsverfahrens gemeinsam gewährleistet werden muss.

Systemmodell und Verteilung

In den folgenden Abschnitten wird von einem Client/Server-Modell (Abbildung 1.1) ausgegangen. Alle Transaktionen werden auf einem zentralen Server ausgeführt. Das bedeutet, dass alle gespeicherten Daten (unpartitioniert) durch den Server verwaltet werden. Erst in Abschnitt 2.5 wird der Umgang mit partitionierten Daten beschrieben.

Klienten senden über das Netzwerk Operationen an den Server, die dort innerhalb einer Transaktion für den Klienten ausgeführt werden. Nach einem Absturz des Servers, wird dieser in endlicher Zeit widerhergestellt (crash-recovery), Klienten jedoch nicht. Der Server erkennt den Ausfall eines Klienten durch seinen Fehlerdetektor und bricht alle noch nicht festgeschriebenen Transaktionen ab. Auch der Absturz des Servers führt zu einem Abbruch laufender Transaktionen. Falls ein Klient fälschlich als ausgefallen vermutet wurde, wird er nach einer Wiederherstellung der Verbingung über den Endzustand der Transaktion informiert.

Der Server verfügt über eine lokale Uhr für die Erzeugung von Zeitstempeln.

2.3.1 Nebenläufigkeitskontrolle und Fehler im Seitenmodell

Das klassische Modell für die Beschreibung transaktionaler Nebenläufigkeit ist das Seitenmodell (page model) nach GRAY [27]. Dieses beschreibt mehrere,

gleichzeitig ablaufende Transaktionen als eine Menge von geordneten Basisoperationen der jeweiligen Transaktionen über elementaren Datenobjekten (Seiten). Jede Basisoperation wird dabei mit einer eindeutigen Kennung für ihre jeweilige Transaktion indiziert. Mögliche Basisoperationen sind:

$r_i(x)$	Leseoperation auf x durch Transaktion i
$w_i(x)$	Schreiboperation auf x durch Transaktion i
c_i	Festschreiben (*engl.* Commit) der Transaktion i
a_i	Abbruch (*engl.* Abort) der Transaktion i

Darauf aufbauend werden die Begriffe *Historie* und *Ablaufplan* (schedule) definiert: Sei $T = \{t_1, t_2, \ldots, t_n\}$ eine Menge von Transaktionen $t_i = (op_i, <_i)$, jeweils beschrieben durch eine Menge von Lese- und Schreiboperationen op_i und einer darüber definierten partiellen Ordnung (Reihenfolge) $<_i$.

Eine Historie bezüglich T ist dann ein Tupel $s = (op_s, <_s)$ bestehend aus

- op_s, einer Menge von Basisoperationen, die alle Lese- und Schreiboperationen op_i für jedes $t_i \in T$ sowie zusätzlich genau eine Abort- oder aber Commit-Operation pro Transaktion beinhaltet,
- $>_s$, einer partiellen Ordnung über der gemeinsamen Operationsmenge op_s, die
  - alle partiellen Ordnungen $>_i$ der Transaktionen $t_i \in T$ beinhaltet,
  - für je zwei Operationen aus verschiedenen Transaktionen, die das selbe Objekt bearbeiten, eine Reihenfolge festlegt, falls mindestens eine der Operationen eine Schreiboperation ist,
  - die finalen Abbruch bzw. Festschreib-Operationen zu den anderen Operationen ihrer Transaktion so in Reihenfolge setzt, dass sie stets als letztes ausgeführt werden.

Prefixe einer Historie werden als Ablaufpläne bezeichnet.

Historien beinhalten nach dieser Definition eine ausgezeichnete Terminierungsoperation für jede Transaktionen, alle Basisoperationen der einzelnen Transaktionen unter Beibehaltung ihrer Reihenfolge und ordnen in Konflikt stehende Operationen der Transaktionen untereinander.

Fehlertypen

Durch die nebenläufige Ausführung mehrerer Transaktionen im Seiten-Modell ohne Nebenläufigkeitskontrolle werden die ACID-Garantien verletzt. Solche Fehler werden wie folgt klassifiziert:

Verlorene Aktualisierung (lost update): Wenn zwei Transaktionen das selbe Objekt lesen und danach modifizieren wollen, kann es passieren, dass die Änderungen einer Transaktion durch die zweite Transaktion überschrieben werden ohne diese gelesen zu haben („Verlorengegangene Einzahlung“).

Inkonsistentes Lesen (inconsistent read): tritt auf, wenn eine lesende Transaktion ihr Ergebnis auf Werten berechnet, die danach durch eine parallel

laufende, noch nicht abgeschlossene, zweite Transaktion verändert werden, und dieses somit falsch ist („Nicht berücksichtigte Einzahlung bei der Zinsberechnung“).

Ungültiges „schmutziges“ Lesen (dirty read): Wenn eine Transaktion nach der Modifizierung von Daten abgebrochen wird, die *zuvor* von einer anderen Transaktion gelesen wurde, ist die Auslösung eines Abbruches dieser Transaktion erforderlich. Arbeitet diese Transaktion hingegen mit den modifizierten, jedoch durch den Abbruch ungültigen Werten weiter, entstehen fehlerhafte Ergebnisse („Falscher Kontostand“).

Ein korrektes Transaktionsverfahrens muss das Auftreten dieser Fehler verhindern. Dafür ist es nötig, dass keine fehlerhaften Ablaufpläne erzeugt werden. Die ausschließliche Erzeugung korrekter Ablaufpläne für die einzelnen Schritte auszuführender Transaktionen ist Aufgabe des Schedulers.

2.3.2 Serialisierbarkeit von Ablaufplänen

Bevor einzelne Verfahren für die Nebenläufigkeitskontrolle diskutiert werden können, ist noch zu klären, was ein „korrekter“ Ablaufplan ist.

Für jedes Korrektheitskriterium für Ablaufpläne ist zu fordern, dass a) es überhaupt durch einen Plan erfüllbar ist, b) für einen gegebenen Plan effizient entscheidbar ist, ob er das Kritierum erfüllt und c) diese Kriterium von möglichst vielen Ablaufplänen erfüllt wird. Der letzte Punkt zielt dabei auf eine Maximierung der Nebenläufigkeit ab. Je mehr Möglichkeiten ein Scheduler bei der Abarbeitung von Transaktionen hat, um so wahrscheinlicher ist eine nebenläufige Ausführung von Operationen.

Geht man davon aus, dass die isolierte Ausführung einzelner Transaktionen die Daten konsistent und korrekt verändert, folgt daraus durch Induktion, dass derartige *serielle Ablaufpläne* aus nacheinander ausgeführten Transaktionen ebenfalls korrekt sind. Solche seriellen Ablaufpläne beinhalten jedoch keine nebenläufig ausführbaren Operationen und sind damit nicht effizient.

Ablaufpläne sind *sichtäquivalent*, wenn sie sowohl bezüglich der Liest-Von-Beziehungen nicht abgebrochener Transaktionen, als auch bezüglich des Ergebnisses aller finalen Schreiboperationen identisch sind. Ein Plan ist *sichtserialisierbar*, falls er sichtäquivalent zu einem beliebigen seriellen Plan ist. Sichtserialisierbarkeit ist ausreichend streng, um das Auftreten der oben beschriebenen Fehlertypen zu verhindern.

Von einem Scheduler ist daher zu fordern, dass er für beliebige eingehende Sequenzen von auszuführenden Basisoperationen stets sichtserialsierbare Schedules erzeugt.

Sichtserialisierbarkeit von Ablaufplänen kann nicht effizient entschieden werden. Daher werden in der Praxis noch stärkere Serialisierbarkeitskriterien verwendet, für die eine effiziente Entscheidung möglich ist. Besonders wichtig ist dabei Konfliktserialsierbarkeit.

Konfliktserialisierbarkeit

Konfliktäquivalenz definiert zunächst eine Konfliktrelation. Zwei Operationen eines Ablaufplans sind in Konflikt, wenn sie auf das selbe Datenobjekt zugreifen und eine der Operationen eine Schreiboperation ist. Konfliktäquivalent zweier Ablaufpläne aus gleichen Operationen ist gegeben, wenn ihre Konfliktrelationen identisch sind. *Konfliktserialisierbarkeit* eines Ablaufplanes entspricht wiederum der Existenz eines konfliktäquivalenten seriellen Ablaufplanes. Es kann gezeigt werden, dass jeder konfliktserialisierbare Plan auch sichtserialisierbar ist und somit Konfliktserialisierbarkeit ein hinreichendes Kriterium für die die korrekte Ausführung transaktionaler Ablaufpläne ist.

Für einen konkrete Scheduler ist zu zeigen, dass er ausschließlich Ablaufpläne erzeugt, die das gewählte Serialisierbarkeitskriterium erfüllen. Oft können jedoch umgekehrt entsprechende Algorithmen direkt aus dem verwendeten Kriterium abgleitet werden. So ist beispielsweise Konfliktserialisierbarkeit die Grundlage von Zwei-Phasen-Sperrprotokollen.

2.4 Verfahren für die Nebenläufigkeitskontrolle

Grundsätzlich können alle Verfahren für die Nebenläufigkeitskontrolle[8] in zwei Klassen eingeteilt werden:

Pessimistische Verfahren garantieren die Serialisierbarkeit einzelner Operationen zum *frühstmöglichen* Zeitpunkt. Wenn eine Unteroperation der Transaktion erfolgreich ausgeführt wurde, kann diese Operation in der Zukunft nicht mehr das Scheitern der Transaktion verursachen. Alle klassischen Verfahren (2-PL, MVCC, SGT, TO) sind pessimistische Verfahren.

Optimistische Verfahren verzögern die Überprüfung der Serialisierbarkeit (Validierung) der Operationen einer Transaktion bis zum Festschreiben. Bis dahin arbeitet die Transaktion auf einer lokalen Kopie der Daten. Die Validierung kann jedoch scheitern. Dies führt dann entweder zum Abbruch oder aber zu einer erneuten Ausführung der Transaktion. Optimistische Verfahren begünstigen die Nebenläufigkeit und vermeiden langes Blockieren. Der optimistische Ansatz ist neuer und wird außerhalb des Datenbankbereichs auch bei der Implementierung von Hauptspeichertransaktionen (State Transactional Memory) eingesetzt [34, 32].

Zusätzlich wird auch orthogonal unterschieden, ob ein Verfahren Sperren (locks) einsetzt oder nicht.

Häufig können sowohl Transaktionen, als auch bearbeitete Daten so partitioniert werden, dass sich für einzelne Datenbereiche bzw. bestimme Transaktionen unterschiedliche Verfahren für die Nebenläufigkeitskontrolle als günstig erweisen. Die Kombination mehrerer Verfahren wird als hybride Nebenläufigkeitskontrolle bezeichnet.

[8] Dieser Abschnitt basiert ebenfalls auf [66].

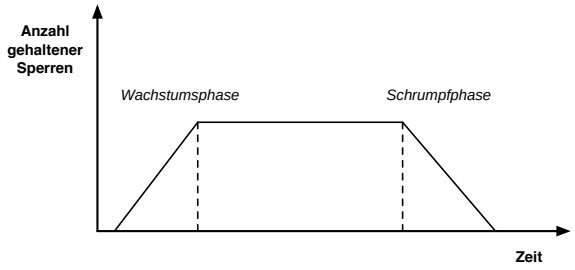

Abbildung 2.5: Phasen von 2PL

2.4.1 Zwei-Phasen-Sperrprotokolle (2PL)

In Sperrprotokollen wird jedem Datenobjekt eine Lese- und eine Schreibsperre (lock) zugeordnet. Transaktionen *müssen* diese erwerben, bevor sie die entsprechendem Operationen auf einem Objekt ausführen dürfen. Die Lesesperre kann von mehreren Transaktionen gleichzeitig erworben werden. Zu einem Zeitpunkt darf aber nur eine Transaktion eine Schreibsperre und keine andere Transaktion eine Lesesperre besitzen. Wenn eine Transaktion eine für ihr Voranschreiten erforderliche Sperre nicht erwerben kann, so wartet sie, bis diese verfügbar ist. Nach Abschluss der auszuführenden Basisoperationen werden dann die gehaltenen Sperren durch die Transaktion wieder freigegeben. Dies geschieht spätestens zum Terminierungszeitpunkt der Transaktion.

Das Zwei-Phasen-Sperrprokoll (2-Phase-Locking, kurz: 2PL) ist das älteste und meistbenutzte Verfahren für die Nebenläufigkeitskontrolle. 2PL ist ein pessimistisches Verfahren, das konfliktserialisierbare Ablaufpläne erzeugt. Die folgende Darstellung beschränkt sich auf klassische Zwei-Phasen-Sperrprotokolle. Klassisches 2PL erfordert, dass nach dem ersten Aufgeben einer Sperre keine neuen Sperren durch die Transaktion erworben werden. Die Transaktion wird in zwei Phasen abgearbeitet (Abbildung 2.5), einer Wachstumsphase, in der benötigte Sperren erworben werden, und einer Schrumpfphase, in der diese nach und nach wieder aufgehoben werden. 2PL stellt jedoch keine weiteren Anforderungen bezüglich der Ausführungsreihenfolge von Basisoperationen auf den jeweils gesperrten Objekte.

Zwei-Phasen-Sperrprotokolle haben den Vorteil, dass sie leicht verständlich und einfach zu implementieren sind. Es existieren mehrerer Varianten, die das Verfahren erweitern, z.B. um abgestufte Sperrgranularität oder altruistisches Sperren.[9] Eine häufig anzutreffende Variante, konservatives 2PL (C2PL) verlangt, dass zu Beginn der Transaktion *alle* erforderlichen Sperren erworben werden müssen. Dafür ist jedoch zum Startzeitpunkt der Transaktion die Kenntnis der zu bearbeitenden Objekte notwendig! Starkes 2PL (SS2PL) verlangt, dass alle Sperren bis zum Ende der Transaktion gehalten werden. Striktes 2PL (S2PL) beschränkt dies auf Schreibsperren.

[9] Die Möglichkeit, dem Scheduler dass Ende der Bearbeitung einzelner Objekte vor dem Ende der Transaktion zu signalisieren

2.4.2 Sperrprotokolle und Verklemmungen (deadlock)

Das gegenseitige Warten auf Sperren, die von einer anderen Transaktion gehalten werden, führt zu Verklemmungen (deadlock). Verklemmungen bewirken, dass keine der beteiligten Transaktionen beendet werden kann und entspricht der Existenz eines Kreises im Wartegraphen (waits-for-graph, kurz: WFG) der aktiven Transaktionen. Verfahren für die Behandlung von Verklemmungen werden danach unterschieden, zu welchem Zeitpunkt sie die entstandene Verklemmung behandeln.

Verklemmungsauflösung (deadlock resolution): Verklemmung werden erst während der Abarbeitung von Transaktionen als Zyklus im Wartegraphen erkannt und dann aufgelöst. Dafür muss entschieden werden, welche Transaktionen abgebrochen werden sollen (Opferauswahlstrategie). Je nach Strategie werden bestimmte Transaktionen immer wieder abgebrochen. Für diese ist damit kein erfolgreiches Voranschreiten mehr möglich (livelock, starvation). Dies kann durch den zusätzlichen Einsatz von Heuristiken, die die genutzte Strategie modifizieren, verhindert werden.

Verklemmungsvermeidung (deadlock prevention): Es wird vorab das Entstehen von Verklemmungen verhindert. Potentielle Verklemmungen werden bereits zum Startzeitpunkt der Transaktion erkannt und durch Abbruch oder Verzögerung der beteiligten Transaktionen behandelt. Dafür ist die Verwendung eines konservativen 2PL notwendig, da ohne eine vollständige, apriori Kenntnis der zu bearbeitenden Objekte Verklemmungsvermeidung nicht möglich ist. Auch bei Verklemmungsvermeidung existieren unterschiedliche Strategien für die Auflösung von Konflikten.

Außerdem können auch Zeitschranken eingesetzt werden, um Verklemmungen zu behandeln. Die Wahl sinnvoller Wartezeiten ist aber eine schwierige Aufgabe.

2.4.3 Multiversionierung

Die bisher vorgestellten Verfahren betrachten die einzelnen Datenobjekte als singuläre Einheiten und regeln, wie Transaktionen Lese- und Schreibzugriffe auf diesen ausführen dürfen. *Multiversion Concurrency Control* (MVCC) [54, 55] betrachtet statt dessen den Raum aller Versionen eines Datenobjektes.[10]

Jeder Transaktion wird ihr Startzeitpunkt als eindeutiger Zeitstempel oder eine forlaufende Transaktions-Id zugeordnet. Diese geordneten Zeitstempel werden auch für die Bezeichnung von Versionen verwendet.

Schreibt eine Transaktion ein Datenobjekt, wird stets eine neue Version dieses Datenobjektes mit dem Zeitstempel der schreibenden Transaktion erzeugt. Einmal erzeugte Versionen können nicht nachträglich verändert werden. Alle Versionen eines Datenobjektes werden gemäß ihres Zeitstempels geordnet. Leseoperation wird durch MVCC stets die jüngste Version zugeordnet, deren Zeitstempel kleiner als der Zeitstempel der lesenden Transaktion ist.

[10] Die formale Korrektheit des Verfahrens erfordert eine Erweiterung der Sichtserialisierbarkeit für Multiversionierung und wurde erstmals in [3] gezeigt.

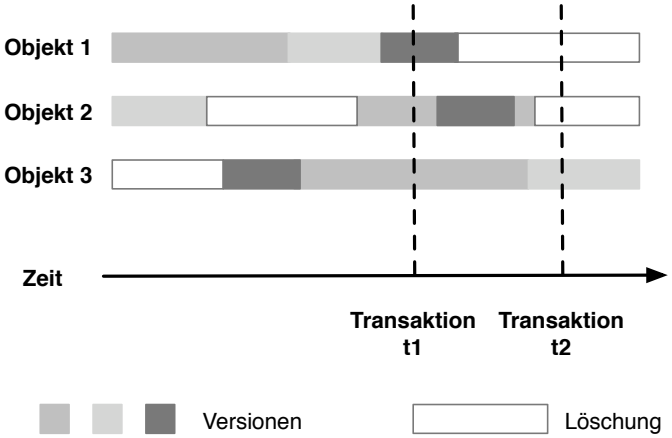

Abbildung 2.6: Versionsraum des MVCC-Verfahrens: MVCC bestimmt für jedes Datenobjekt, welche Version zu welchem Zeitpunkt gültig ist

Stellt man sich den Versionsraum der Datenobjekte als vorab festgelegt und kontinuierlich über die Zeit vor, das heißt durch das Ausführen von Transaktionen werden Versionen „aufgedeckt“, wird ersichtlich, dass MVCC für jede lesende Transaktion einen Querschnitt der gerade gültigen Daten bereithält und somit automatisch Isolation gewährleistet (Abbildung 2.6).

Damit dies funktioniert, dürfen einmal gelesene Daten nicht nachträglich ungültig werden, das heißt durch neuere Versionen überschrieben werden. Dafür wird zusätzlich jeder Version ein zweiter Zeitstempel zugeordnet. Dieser beinhaltet den größten Zeitstempel aller Transaktionen, die diese Version gelesen haben. Versucht nun eine andere Transaktion eine neue Version zu erzeugen, deren Zeitstempel zwischen den Erzeugungszeitpunkt und dem größten Lesezeitpunkt einer bereits existierenden Version fallen würde, wird sie abgebrochen. Das ist notwendig, da eine Ausführung des Schreibzugriffes die zuvor gelesene Version invalideren würde. Vorherige, lesende Transaktionen hätten nicht die aktuellste, ihrem Schnappschuss zugehörende Versionen erhalten. Anders formuliert: Das Lesen einer Version fixiert den kontinuierlichen Versionsraum des gelesenen Datenobjektes zwischen dem Erzeugungszeitpunkt der Version und dem Startzeitpunkt der lesenden Transaktion.

Multiversionierung erfordert zusätzlichen Speicherplatz für die verschiedenen Objektversionen. Alte Versionen eines Datenobjekts können jedoch gelöscht werden, wenn sie älter als alle aktiven Transaktionen sind und es eine jüngere Version des Objekts gibt, die der ältesten aktiven Transaktion zugeordnet werden kann. Dadurch wird für Multiversionierung kaum mehr Speicher benötigt, als für andere Nebenläufigkeitskontrollverfahren.

Optimierung

Der Abbruch von Transaktionen, deren Schreibzugriffe die Korrektkeit vorheriger Lesezugriffe anderer Transaktion invalidieren würde, ist einer der Nachteile von MVCC. Der Abbruch erzwingt die Aufhebung aller bisher ausgeführter Basisoperationen. Dies kann einen beträchtlichen Aufwand erfordern. Oft schließt

sich daran die erneute Ausführung der abgebrochenen Transaktion an. Wird ein kritisches Datum ständig von sehr vielen kurzen Transaktionen gelesen und zusätzlich gelegentlich von einer langlaufenden Transaktion als letzte Basisoperation geschrieben, kann es vorkommen, dass die schreibende Transaktion stets durch einen vorherigen Lesezugriff abgebrochen wird und „verhungert“ (von starvation).

Bereits in REED [54] werden deshalb *Schreibreservierungen* vorgeschlagen. Dies sind (flüchtige) Versionen eines Datums, die (noch) keinen Inhalt besitzen. Schreibreservierungen können bereits zum Beginn einer langlaufenden Transaktion erzeugt werden. Lesende Transaktionen, die versuchen eine Schreibreservierung zu lesen, werden blockiert, bis die schreibende Transaktion abgeschlossen wurde und ein Wert für die Version vorliegt. Die Blockierung des Systems durch Schreibreservierungen kann durch Timeouts verhindert werden. Schreibreservierungen sind somit zeitlich begrenzte Sperren für (flüchtige) Versionen. Sie ergänzen MVCC um einen Mechanismus, der das Verhungern von Transaktionen verhindert und können als früher Prototyp eines hybriden Verfahrens verstanden werden.

2.4.4 Optimistische Nebenläufigkeitskontrolle

Die Grundidee eines optimistischen Verfahrens ist es, davon auszugehen, dass während der Abarbeitung einer Transaktion keine Konflikte auftreten werden. Dies beruht auf der Beobachtung, dass gleichzeitig ablaufende Transaktionen oft auf völlig unterschiedlichen Daten arbeiten. Im Falle eines Konfliktes muss dann jedoch bereits durchgeführte Arbeit aufgehoben und betroffene Transaktionen abgebrochen und ggf. erneut gestartet werden.

Dementsprechend wird eine optimistische Transaktion in mehreren Phasen durchgeführt:

Lesephase Abarbeitung der Transaktion auf einer privaten Kopie der betroffenen Daten

Validierung Überprüfung, ob die durchgeführten Änderungen zu Konflikten mit anderen nebenläufig durchgeführten Transaktionen führen. Im Falle eines Konfliktes, werden betroffene Transaktionen abgebrochen, bis alle Konflikte gelöst sind.

Schreibphase Bei erfolgreicher Validierung werden abschliessend die durchgeführten Änderungen festgeschrieben.

Ferner wird zwischen zwei Arten von optimistischen Verfahren unterschieden:

Rückwärts validierende, optimistische Verfahren (BOCC) Führen die Konfliktüberprüfung in der Validierungsphase bezüglich bereits festgeschriebener, vergangener Transaktionen aus.

Vorwärts validierende, optimistische Verfahren (FOCC) Führen die Konfliktüberprüfung in der Validierungsphase bezüglich nebenläufig ablaufender, noch nicht festgeschriebener Transaktionen aus, die sich noch in ihrer Lesephase befinden.

Vorwärts validierende Verfahren erreichen einen höheren Grad an Nebenläufigkeit, da ihnen bei der Auswahl der im Konfliktfall abzubrechenden Transaktion mehr Möglichkeiten verfügbar sind. Rückwärts validierende Verfahren müssen hingegen stets diejenige Transaktion abbrechen, deren Validierung fehlschlug.

Einige optimistische Verfahren [64] erkennen auch bereits vor der Validierung, während der Lesephase, Konflikte zwischen nebenläufig ausgeführten Transaktionen. Dies führt zu einer weiteren Klassifizierung in Bezug auf das Abbruchverhalten optimistischer Nebenläufigkeitskontrollverfahren bei der *vorzeitigen* Entdeckung von Konflikten:

Optimistic-Kill Die Transaktion wird abgebrochen, sobald das Vorliegen eines Konfliktes entdeckt wird.

Optimistic-Die Die Transaktion wird vollständig ausgeführt, obwohl bereits das Vorliegen eines Konfliktes bekannt ist. Es erfolgt *kein* vorzeitiger Abbruch von Transaktionen.

Der Vorteil von Optimistic-Kill besteht darin, bereits frühzeitig den Neustart von Transaktionen zu veranlassen und dadurch ggf. den insgesamt erreichten Grad an Nebenläufigkeit zu verbessern. Andererseits kann Optimistisc-Kill auch permanentes Neustarten bewirken. Optimistic-Die ist sinnvoll, wenn eine konfligierende Transaktion mit hoher Wahrscheinlichkeit abgebrochen wird und somit zum Zeitpunkt der Validierung der Konflikt nicht mehr vorliegt. Optimistic-Die wird außerdem in hybriden, optimistischen Verfahren eingesetzt (nächster Unterabschnitt).

2.4.5 Hybride Verfahren

OCC und 2PL: Hybrides OCC (HOCC)

THOMASIAN [64] beschreibt ein hybrides optimistisches Transaktionsverfahren (folgend HOCC). Es wird für HOCC angenommen, dass sich Lese- und Schreibmenge während einer erneuten Ausführung einer Transaktion nicht ändern (acess invariance, *engl.* Zugriffsinvarianz. Dies ist für viele Anwendungsszenarien realistisch.

Das Verfahren ist ein Optimistic-Die-Verfahren. Bei Erreichen der Validierungsphase sind Lese- und Schreibmenge der betroffenen Transaktion bekannt und aufgrund von Zugriffsinvarianz auch konstant. Zu Beginn der Validierung werden alle notwendigen Sperren erworben, das heißt die Wachstumsphase eines SS2PL ausgeführt. Schlägt die Validierung fehl, wird die Transaktion neu gestartet ohne die erworbenen Sperren freizugeben. Bei erneutem Eintritt in die Validierungsphase werden nun jedoch keine neuen Sperren benötigt, da ja bereits alle Sperren zuvor erworben wurden. Die erfolgreiche Ausführung der Transaktion ist somit unter der Annahme von Zugriffsinvarianz garantiert. Falls eine Transaktion nicht zugriffsinvariant ist, kann jedoch eine weitere, erneute Ausführung der Transaktion notwendig sein.

Ein weiterer Vorteil ergibt sich aus der Verwendung eines starken Zwei-Phasen-Sperrprotokolls (SS2PL). Verklemmungen können nicht auftreten.

Read-Only Multiversioning (ROMV)

Read-Only Multiversioning (ROMV, [11, 46]) basiert auf der Beobachtung, dass in vielen Anwedungsszenarien alle auftretenden Transaktionen in zwei Gruppen eingeteilt werden können. Nur-Lesende und Lese-und-Schreib-Transaktionen, die jeweils andere Anforderungen an die Nebenläufigkeitskontrolle stellen. ROMV verwendet deshalb 2PL für schreibende und Multiversionierung für Nur-Lese-Transaktionen (sie müssen explizit als solche markiert werden). Die Integration beider Techniken erfolgt über die Zeitstempelvergabe und Versionserzeugung.

Schreibende Transaktionen verwenden ein Zwei-Phasen-Sperrprotokoll. Das Festschreiben der Transaktion erzeugt hingegen zusätzlich neue Versionen aller modifizierten Datenobjekte. Der Zeitstempel der so erzeugten Versionen entspricht dabei dem Festschreibzeitpunkt der schreibenden Transaktion.

Lesezugriffe durch Nur-Lese-Transaktionen werden so aufgelöst, dass stets die jüngste, bisher festgeschriebene Version gelesen wird (wie bei MVCC). Dabei wird der Startzeitpunkt der Nur-Lese-Transaktion als Referenzpunkt verwendet. Es wird implizit angenommen, dass nach dem Start einer Nur-Lese-Transaktion kein Festschreiben einer Schreibtransaktion mit früherem Zeitstempel mehr stattfindet, da sonst die Korrektheit verletzt wird. Dies spielt eine Rolle bei der verteilten Ausführung von ROMV.

2.4.6 Behandlung von Transaktionsabbrüchen

Die korrekte Behandlung von Transaktionsabbrüchen (transaction recovery) benötigt zusätzliche Unterstützung durch das Scheduling und ist eng mit der Aufgabe des Absturzbehandlung (crash recovery) verknüpft. Aufgabe der Absturzbehandlung ist es, Datenverlust und -inkonsistenz bei einem Ausfall des Datenbankservers zu verhindern. Da in den folgenden Kapiteln Ausfallsicherheit anderweitig gewährleistet wird, wird Absturzbehandlung in dieser Arbeit nicht näher besprochen. Auch die Theorie der Behandlung von Transaktionsabbrüchen kann an dieser Stelle nicht umfassend dargestellt werden. Der folgende Abschnitt gibt deshalb nur einen knappen Überblick über das Thema. Genaueres zu Transaktionsabbruch- und Absturzbehandlung enthält [66].

Transaktionsabbrüche im Seitenmodell

Die Grundidee für die Behandlung von Transaktionsabbrüchen besteht darin, einen äquivalenten Ablaufplan zu konstruieren, der keine Abbruchoperationen beinhaltet und diesen mit den normalen Methoden der Nebenläufigkeitskontrolle zu verarbeiten. Dafür wird zunächst jeder Leseschritt einer abgebrochenen Transaktion entfernt. Zusätzlich wird jede Abbruchoperation durch eine Menge von inversen Schreiboperationen und eine finale Festschreiboperation ersetzt. Dabei wird für jede Schreiboperation $w_i(x)$ der ursprünglichen Transaktion genau eine inverse Schreiboperation $w_i^{-1}(x)$ in umgekehrter Reihenfolge erzeugt, so dass die letzte Schreiboperation zuerst aufgehoben wird. Der so erzeugte Ablaufplan wird Expansion genannt.

Die Behandlung von Transaktionsabbrüchen erfordert die Einführung neuer Serialisierbarkeitskriterien. Erweiterte Konfliktserialisierbarkeit ist ein solches Kriterium. Ein Ablaufplan ist erweitert konfliktserialisierbar, wenn seine Expansion konfliktserialisierbar ist. Erweiterte Konfliktserialisierbarkeit ermöglicht die korrekte Behandlung von Transaktionsabbrüchen.

Die Konfliktserialisierbarkeit von Expansionen ist jedoch ein zu strenges Kriterium. Es gibt abbrechbare Ablaufpläne, deren Expansion nicht konfliktserialisierbar ist, da ihr Konfliktgraph (irrelevante) Zyklen zwischen mehreren abgebrochenen Transaktionen beinhaltet. Deshalb werden schwächere Kriterien, wie Logwiederherstellbarkeit [66], benutzt. Ein Ablaufplan ist logwiederherstellbar, wenn er

1. wiederherstellbar ist, das heißt eine Transaktion nur dann festgeschrieben wird, wenn alle Transaktionen, deren Daten sie gelesen hat, festgeschrieben wurden, und
2. für jeden Schreibkonflikt $w_i(x) < w_j(x)$ zwischen zwei verschiedenen Transaktion i und j des Ablaufplans, entweder
   a) i vor $w_j(x)$ abgebrochen wird (Konfliktirrelevanz),
   b) oder i vor j festgeschrieben wird, falls j festgeschrieben wird (Verzögertes Festschreiben),
   c) oder aber j vor i abgebrochen wird, falls i abgebrochen wird (Kaskadierendes Abbrechen).

Logwiederherstellbarkeit regelt die Reihenfolge der Terminierungsoperationen (commit und abort) und ermöglicht so, dass stets das isolierte Aufheben in Konflikt stehender Transaktionen möglich ist. Logwiederhestellbarkeit genügt, um die korrekte Behandlung von Transaktionsabbrüchen zu gewährleisten.

Für konkrete Verfahren muss gezeigt werden, dass abhängige Festschreiboperationen verzögert und kaskadierende Abbrüche zusammengefasst werden. Das entspricht der Definition von Logwiederherstellbarkeit.

Starkes 2PL (SS2PL)

Für die Abbruchbehandlung von SS2PL ist ein weiteres Kriterium wichtig, Rigorosität. Ein Plan ist rigoros, wenn er

1. strikt ist, das heißt kein Objekt gelesen oder geschrieben wird, bevor die Transaktion, die dieses Objekt zuletzt geschrieben hat, terminiert wurde, und
2. ein Objekt erst dann überschrieben wird, wenn alle Transaktionen, die es gelesen haben, terminiert wurden.

Es kann gezeigt werden, dass jeder rigorose und konfliktserialisierbare Ablaufplan logwiederherstellbar ist. Das starke Zwei-Phasen-Sperrprotokoll (SS2PL) erzeugt ausschließlich rigorose und konfliktserialisierbare Ablaufpläne. Somit ermöglicht SS2PL ohne Veränderungen die Behandlung von Transaktionsabbrüchen. Dies ist für das in Kapitel 3 entwickelte Transaktionsverfahren wichtg.

Multiversionierung und Transaktionsabbrüche

MVCC unterstützt direkt die Behandlung von Transaktionsabbrüchen. Schreiboperationen erzeugen zunächst nur „vorläufige“ Versionen, die mit einer eindeutigen Kennung für die schreibende Transaktion markiert sind. Lesende Transaktionen, die auf eine vorläufige Version zugreifen wollen, werden blockiert. Erst zum Festschreibzeitpunkt werden vorläufige Versionen in gültige Versionen umgewandelt. Ein Transaktionsabbruch führt zum Löschen der vorläufigen Versionen und einer erneuten Ausführung der blockierten Leseoperationen.

In gewisser Weise sind die so erzeugten Ablaufpläne rigoros. Striktheit entspricht dem blockierenden Lesen vorläufiger Versionen. Das Schreiben neuer Versionen mit einem Zeitstempel, der vor allen bekannten lesenden Transaktionen liegt, enspricht der zweiten Bedingung an rigorose Ablaufpläne.

2.5 Verteilte Transaktionsverwaltung

Bisher wurde davon ausgegangen, dass alle Daten auf einem zentralen Server gespeichert werden. In diesem Abschnitt wird beschrieben, wie Transaktionsverfahren angepasst werden müssen, wenn die Daten auf mehrere Server verteilt gespeichert werden (partitionierte Datenhaltung). Dabei werden ausschließlich homogene Systeme betrachtet. Das sind Systeme, in denen von allen Servern die selbe Datenbankmanagementsoftware eingesetzt wird.[11]

Die Verteilung ist aus Sicht der Anwendung transparent. Partitionierte Datenhaltung erfordert aber eine Anpassung der Nebenläufigkeitskontrolle. Die notwendigen Veränderungen werden in den folgenden Abschnitten jeweils für die einzelnen Verfahren beschrieben (Unterabschnitt 2.5.1). Zusätzlich muß die Recovery-Komponente um Unterstützung für verteiltes, atomares Festschreiben erweitert werden, da ansonsten die Einhaltung der Atomizität nicht gesichert werden kann. Entsprechende Verfahren werden in Unterabschnitt 2.5.2 erläutert.

Systemmodell

Das System besteht aus einer endlichen, fixierten Anzahl von Datenbankservern und beliebig vielen, unabhängigen Klienten.

Jeder Server verwaltet eine bestimmte Partition der Datenbank und besitzt einen lokalen Scheduler. Partitionen überlappen sich nicht. Für jedes gespeicherte Datum ist stets genau ein Server zuständig. Es gibt im System keine Replikation und Ausfallsicherheit muss auf der Ebene der einzelnen Server gewährleistet werden. Es wird keine zentrale Komponente für die Nebenläufigkeitskontrolle von Transaktionen eingesetzt. Das System ist vollständig dezentral.

[11] Vgl. heterogene Systeme ermöglichen die Ausführung verteilter Transaktionen unter Verwendung verschiedener Datenbankmanagementsysteme

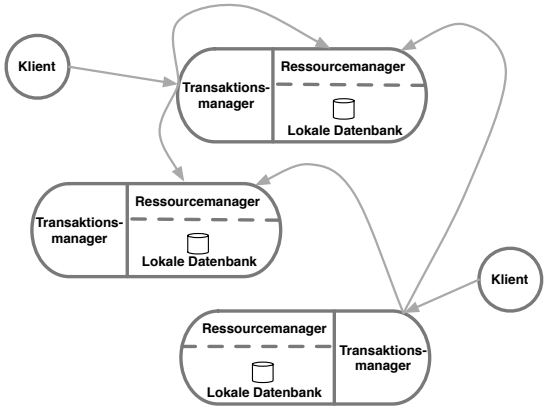

Abbildung 2.7: Systemmodell für verteilte Transaktionen

Datenbankserver kommunizieren miteinader, um die korrekte Ausführung von Transaktionen zu ermöglichen. Die kommunizierenden Prozesse werden in Ressource- und Transaktionsmanager eingeteilt (Abbildung 2.7). Der Ressourcenmanager ist diejenige Komponente eines Datenbankservers, der bei der Ausführung verteilter Transaktionen den Zugriff auf die lokale Datenbank kontrolliert. Transaktionsmanager sind für die Koordination einzelner Transaktionen unter Verwendung mehrerer Ressourcemanager zuständig. Klienten führen einzelne Transaktionen aus, in dem sie an ihren Transaktionsmanager die auszuführenden Basisoperation senden. Ressource- und Transaktionsmanager werden nach einem Ausfall wiederhergestellt, Klienten nicht. Klienten erfahren stets den Endzustand einer von ihnen ausgelösten Transaktion, wenn sie nicht abstürzen.

Jeder Server verfügt über eine lokale Uhr für die Erzeugung von Zeitstempeln. Die Uhren aller Server sind lose synchronisiert. Dies kann leicht als Bestandteil des regulären Nachrichtenaustausches realisiert werden.

Zur Vereinfachung wird im Folgenden angenommen, dass jeder Prozess auf einem eigenen Knoten ausgeführt wird und deshalb beide Begriffe in diesem Abschnitt synonym gebraucht.

2.5.1 Nebenläufigkeitskontrolle

Eine *globale Historie* ist eine Historie, deren Projektion bezüglich der Basisoperation einzelner Knoten äquivalent mit der lokalen Historie des jeweiligen Knotens ist. Die von den Schedulern der einzelnen Serverknoten erzeugten Ablaufpläne erfüllen jeweils ein (lokales) Korrektheitskriterium. Meist ist dies Konfliktserialisierbarkeit, worauf sich das Folgende auch beschränkt.

Für die korrekte Ausführung verteilter (globaler) Transaktionen ist es erforderlich, dass die globale Historie ebenfalls konfliktserialisierbar ist. Dies ist dann gegeben, wenn eine serielle, konfliktäquivalente globale Historie existiert. Diese Definition fordert somit globale (und damit auch lokale) Konfliktserialisierbarkeit und Konfliktäquivalenz zwischen globalen und lokalen Ablaufplänen [66].

Zeitstempel

Einige Verfahren erforden die Verwendung eindeutiger Zeitstempel. Eindeutige Zeitstempel können durch die Erweiterung lokaler Zeitstempel um eine eindeutige Kennung für den jeweiligen Knoten erzeugt werden.

Wenn die lokalen Uhren stark abweichen, werden einige Knoten stets zukünftige Zeitstempel generieren und sich dadurch gegenüber Knoten mit korrekt synchronisierter Uhr durchsetzen. Dieses Problem kann durch den Einsatz von Lamport-Uhren [38] behandelt werden.

Damit externen, miteinander kommunizierende Beobachter (Nutzer) des Systems keine Widersprüche in der Ausführungsreihenfolge von Operationen feststellen können, ist es notwendig, die Lamport-Zeit lose an die reale Uhrzeit zu koppeln. Dafür wird bei der Erzeugung eines neuen Zeitstempels gewartet, bis die lokale Uhrzeit größer als der größte, lokal bekannte, vorher im System benutzte Zeitstempel ist. Operationen mit Zeitstempeln stark abweichender Uhren werden zurückgewiesen, um längeres Blockieren einzelner Knoten zu verhindern [54].

Verteiltes Zwei-Phasen-Sperrprotokoll

Die Verteilung von Zwei-Phasen-Sperrprotokollen ist zunächst einfach. Jeder Knoten führt lokal Lese-, Schreib- und Sperroperationen aus. Die Verteilung erfordert jedoch eine gesonderte Behandlung der Freigabe von Sperren. Der Beginn der Entsperrphase muss global synchronisiert werden, um die Konfliktserialisierbarkeit der globalen Historie zu gewährleisten. Dies kann durch gegenseitige Benachrichtigung nach dem Ende der letzten Datenoperation realisiert werden. Die Entsperrphase beginnt, wenn die letzte Datenoperation eines beliebigen Knotens abgeschlossen wurde [66].

Strenges Zwei-Phasen-Sperren eignet sich besonders gut für die Verteilung, da alle Sperren bis zum Ende der Transaktion gehalten werden. Entsperren wird durch die Terminierungsoperation ausgelöst und erfordert somit keinen zusätzlichen Kommunikationsaufwand [64].

Alle nicht strengen Zwei-Phase-Sperrprotokolle erfordern zusätzlich einen Mechanismus für die verteilte Erkennung von Verklemmungen.

Verteilte Erkennung von Verklemmungen

Die Erkennung von Verklemmungen (deadlocks) erfordert bei verteilter Transaktionsverwaltung den Einsatz zusätzlicher Techniken. Einzelne Knoten besitzen nur begrenzte Kenntnis des Wartegraphen. Knoten müssen daher mit der nötigen Information für die Erkennung von Verklemmungen versorgt werden. Außerdem können empfangene Daten zum Zeitpunkt ihres Eintreffens bereits ungültig sein. Dann werden Verklemmungen ggf. fälschlich erkannt, obwohl der entsprechende Zyklus des Wartegraphen mittlerweile durch einen Transaktionsabbruch auf einem anderen Knoten aufgehoben wurde. Das kann jedoch nicht vermieden werden. Man unterteil die Verfahren in zwei Gruppen, Kantenverfolgung (edge chasing) und Pfadverteilung (path pushing) [66].

Verteilte, optimistische Nebenläufigkeitskontrolle

Zunächst können die Operationen ohne zusätzliche Koordination auf den jeweiligen lokalen Knoten ausgeführt werden. Kritisch ist jedoch die Validierungsphase. Um Serialisierbarkeit zu gewährleisten ist es erforderlich, dass die lokalen Validierungen globaler Transaktionen von allen Knoten in der selben Reihenfolge ausgeführt werden.

Mögliche Techniken hierfür sind der Einsatz eines verlässlichen Broadcastmediums, Validierug durch einen zentralen Knoten unter Verwendung eindeutiger Zeitstempel für das Transaktionsende und zirkulierende Tokens (nach [64]). Diese Ansätze sind jedoch in verteilten Systemen mit hoher Knotenanzahl in einem WAN ungeeignet.

AGRAWAL ET AL. [2] beschreiben eine andere Möglichkeit. Jeder Knoten n verwaltet einen streng monoton steigenden Zeitstempel $t_v(n)$ für die größte, bisher validierte Transaktion. Vor dem Validieren schlägt der Transaktionsmanager jedem betroffenen Knoten einen Validierungszeitpunkt t vor. Knoten stimmen dem Validierungszeitpunkt zu, falls $t > t_v(n)$ und aktualisieren $t_v(n)$. Falls alle Knoten zustimmen, wird validiert. Andernfalls wird das Verfahren solange wiederholt, bis ein passender Zeitstempel ermittelt werden kann oder bis die Transaktion abgebrochen wird.

GRUBER [28] beschreibt eine Optimierung für die Wahl des Zeitstempels. Die Bestätigung jeder Basisoperation enthält das aktuelle $t_v(n)$ des antwortenden Knotens. Der vorgeschlagene Validierungszeitpunkt wird so gewählt, dass $t > max_{n \in N}(t_v(n))$ ist. Ein so gewähltes t wird mit hoher Wahrscheinlichkeit in der ersten Runde von allen Knoten akzeptiert.

Verteilte Multiversionierung

Multiversion Concurrency Control kann direkt als verteiltes Verfahren für die Nebenläufigkeitskontrolle eingesetzt werden..

Lediglich das Löschen alter Versionen muß an die Verteilung angepasst werden, da einzelne Server den Zeitstempel der ältesten aktiven Transaktion nicht kennen und somit nicht wissen, ob eine Version gelöscht werden kann. Eine triviale Lösung ist, allen Servern den Startzeitpunkt neuer Transaktionen vor der Ausführung der ersten Datenoperation mitzuteilen. Dieser Ansatz skaliert jedoch nicht und verzögert unnötig die Transaktionsausführung. Alternativ nimmt man an, dass die Uhren sich maximal um Δ_t voneinander unterscheiden. Dadurch kann garantiert werden, dass der minimale Zeitstempel einer aktiven Transaktion zum Zeitpunkt t nicht kleiner als $t - \Delta_t$ ist und daher *nur* die diesem Zeitpunkt zugeordnete Version und alle auf sie folgenden Versionen eines Datenobjektes gespeichert werden müssen. Ältere Versionen können hingegen gelöscht werden, ohne die Korrektheit des Nebenläufigkeitskontrollverfahrens zu verletzen [54].

WEIHL [65] beschreibt verschiedene weitere Verfahren für das Löschen alter Versionen in verteilten Datenbanksystemen.

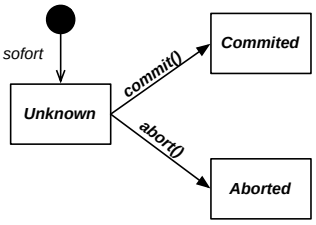

Abbildung 2.8: Zustandsdiagramm für atomares Festschreiben

2.5.2 Verteiltes, atomares Festschreiben

Atomares Festschreiben erfordert es, pro Transaktion den Zustandsautomaten aus Abbildung 2.8 zu realisieren. Dabei müssen folgende Bedingungen eingehalten werden: 1) Zustandsübergänge werden atomar und in endlicher Zeit ausgeführt. 2) Die Transaktion wird nur dann festgeschrieben, wenn dies nicht die Einhaltung der ACID-Garantien verletzt. 3) Der Endzustand des Automaten ist verfügbar, bis die Transaktionsausführung vollständig abgeschlossen wurde.

Verteiltes, atomares Festschreiben erfordert, dass letztendlich Einigkeit zwischen *allen* an einer Transaktion beteiligten Knoten besteht. Nach einer Zeit der Ungewissheit, in der einigen Prozessen der Endzustand der Transaktion nicht bekannt ist, wird die Transaktion entweder von allen Prozessen abgebrochen oder aber festgeschrieben. Festschreiben erfordert dabei die Zustimmung aller Prozesse, da nur sie über die nötige Information verfügen, um zu entscheiden, ob wegen lokaler Konflikte oder anderer Fehler abgebrochen werden muss. Das bedeutet auch, dass aufgrund der Verteilung die Prozesse einen Teil ihrer Autonomie aufgeben müssen und nicht mehr unabhängig das Festschreiben von Transaktionen entscheiden können [66]. Der Endzustand von Transaktionen wird deshalb in Protokollen für verteiltes, atomares Festschreiben meist durch einen Koordinator[12] zentral verwaltet. Dieser regelt Festschreiben und Abbrechen der jeweiligen Transaktion und informiert bei einer Änderung alle betroffenen Prozesse.

Verteiltes Zwei-Phasen-Festschreiben

Two-Phase-Commit (2PC) ist der am häufigsten eingesetzte Algorithmus für verteiltes, atomares Festschreiben. Zu Beginn des Festschreibens wird an alle beteiligten Knoten eine Prepare-Nachricht gesendet, die die Knoten dazu auffordert, dass Festschreiben vorzubereiten. Die Knoten antworten auf diese Nachricht mit „Ja“, falls aus ihrer lokalen Sicht Festschreiben möglich ist. Diese Antwort ist eine Garantie gegenüber dem Koordinator, dass er ab sofort die Transaktion festschreiben kann. Antwortet ein Knoten mit „Nein“, so wird die Transaktion sofort durch den Koordinator abgebrochen. Sobald der Koordinator „Ja“-Antworten von allen Knoten erhalten hat, werden diese über das endgültige Festschreiben der Transaktion informiert.

Der Hauptnachteil von 2PC besteht darin, dass bei einem Ausfall des Transaktionsmanagers kein Fortschritt mehr möglich ist und 2PC blockiert.

[12] Koordinator und Transaktionsmanager sind oft identisch.

Dieses Problem wird in der Praxis durch die Erhöhung der Ausfallsicherheit des Transaktionsmanagers oder durch den Einsatz nichtblockierender Protokolle, wie beispielsweise Three-Phase-Commit (3PC), behandelt. Eine andere Technik für den Umgang mit einem Ausfall des Koordinators besteht darin, andere Ressourcemanager nach dem Endzustand der Transaktion zu fragen. Diese Technik wird kooperatives Beenden (cooperative termination) genannt. Sie erfordert, dass die beteiligten Ressourcemanager sich gegenseitig kennen. Die entsprechende Information kann jedoch leicht als Bestandteil der Prepare-Nachricht verbreitet werden.

Als letzte Notfall-Massnahme wird gelegentlich auch das Abbrechen nach einer hinreichend großen Wartezeit eingesetzt. Dabei wird potentiell ein inkonsistenter Zustand des Systems erzeugt, um Blockierung durch verteiltes atomares Festschreiben aufzulösen und das Abarbeiten wartender Transaktionen zu ermöglichen.

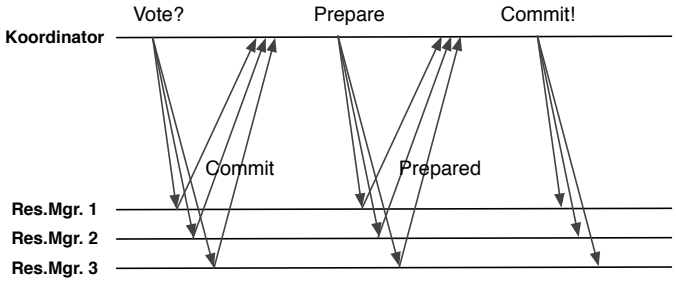

Abbildung 2.9: Three-Phase-Commit

Nichtblockierende, mehrphasige Verfahren

Three-Phase-Commit (3PC) verhindert das Blockieren von 2PC durch die Einführung einer weiteren Phase und die Nutzung von Timouts (Abbildung 2.9). Das Protokoll beginnt mit der Wahlphase. Der Koordinator fordert alle Ressourcemanager dazu auf, ihm mitzuteilen, ob sie die Transaktion festschreiben möchten. Wenn alle Ressourcemanager zustimmen, kann festgeschrieben werden. Andernfalls wird die Transaktion abgebrochen. Das Wahlergebnis wird in der folgenden Ausbreitungsphase (dissemination phase) den Ressourcemanagern mitgeteilt. Wenn ein Knoten während der Ausbreitungsphase nicht antwortet, wird die Transaktion abgebrochen. Erst wenn alle Ressourcemanager den Erhalt des Ergebnisses bestätigt haben, wird durch den Koordinator dass endgültige Festschreiben ausgelöst (Entscheidungsphase). Bei einem durch Timeout erkannten Ausfall des Transaktionsmanagers in der Entscheidungsphase wird die Transaktion von Ressourcemanagerknoten festgeschrieben.

Der Nachteil von 3PC besteht in der wesentlich größeren Anzahl von benötigten Nachrichten und der höheren Latenz. Deshalb hat es für Datenbankmanagementsysteme nur geringe praktische Bedeutung.

GUERRAOUI ET AL. [30] beschreiben ein anderes, nicht-blockierendes Zwei-Phasen-Protokoll. Während der Prepare-Phase werden Bestätigungsnachrichten nicht nur an den Koordinator, sondern an alle Knoten gesendet. Wenn ein Knoten Bestätigungsnachrichten von allen anderen Knoten empfangen hat, sendet er eine

Pre-Commit-Nachricht an alle Knoten. Sobald ein Knoten von jedem anderen Knoten eine Pre-Commit-Nachricht empfangen hat, wird die Transaktion auf diesem Knoten festgeschrieben. Wird von einem beliebigen Knoten der Ausfall eines anderen Knotens vermutet, wird für die Fehlerbehandlung ein uniformer Konsens initiiert. In der Bestätigungsphase wird dabei durch den initiierenden Knoten der Transaktionsabbruch, in der Pre-Commit-Phase das Festschreiben vorgeschlagen. Dieses Verfahren hat den Vorteil, dass es nur zwei Phasen benötigt und nicht blockiert, auch wenn der Koordinator ausfällt. Andererseits wird eine wesentliche größere Anzahl von Nachrichten versendet. Falls ein Broadcastmedium zwischen allen beteiligten Ressourcemanagern verfügbar ist, kann dies jedoch vernachlässigt werden.

Einphasige Protokolle: MVCC mit Commit-Records und NB-SPAC

REED [54] verwendet in MVCC (Unterabschnitt 2.4.3) für das verteilte, atomare Festschreiben von Transaktionen explizite Commit-Records. Commit-Records speichern für jede Transaktion ausfallsicher den Zustandsautomat aus Abbildung 2.8 und ermöglichen es, Zustandsübergänge atomar auszuführen.

MVCC ist ein pessimistisches Verfahren für die Nebenläufigkeitskontrolle, Ressourcemanager sind stets im Prepared-Zustand. Schreiboperationen erzeugen entweder potentielle neue Versionen, die mit einer Kennung für den Commit-Record versehen sind, oder lösen den Abbruch der Transaktion durch eine entsprechende Markierung des Commit-Records aus, falls die Version nicht erzeugt werden kann. Commit-Records werden erst nach dem Ende der letzten Basisoperationen ihrer Transaktion als festgeschrieben markiert.

Das Verfahren erlaubt gleichermaßen aktive Benachrichtigung und passives (on-demand) Auslesen des Commit-Records. Änderungen am Zustand des Commit-Records führen zu einer aktiven Benachrichtigung der ihm bekannten Ressourcenmanager. Sobald alle Ressourcemanager den Endzustand der Transaktion erfahren haben, kann der Commit-Record gelöscht werden. Festschreiben bewirkt eine Aktivierung potentieller Versionen, Abbrechen führt zu ihrer Löschung. Leseoperationen, deren Ergebnis eine noch nicht endgültige Version ist, überprüfen den Zustand des Commit-Records und warten ggf. auf eine Änderung. Sobald eine potentielle Version aktiviert wird, werden alle wartenden Leseoperationen abgearbeitet. Wird die potentielle Version gelöscht, wird die nächstpassende Version gelesen. Für das Nichtblockieren ist die Verfügbarkeit des Commit-Records von zentraler Bedeutung. Diese kann durch Replikation gewährleistet werden.

Das Protokoll ist einphasig, da das Senden von Prepare-Nachrichten entfällt. NB-SPAC [1] generalisiert diesen Ansatz für beliebige Nebenläufigkeitskontrollverfahren, deren Ablaufpläne rigoros sind.

2.5.3 Zusammenfassung

In diesem Kapitel wurden strukturierte Overlay-Netzwerke und Grundlagen der Transaktionsverarbeitung mit den Schwerpunkten Nebenläufigkeitskontrolle und verteilte Transaktionsverwaltung beschrieben. Für beide Themen wurden Systemmodelle und mögliche Fehlerklassen diskutiert.

In SONs sind der Umgang mit der Intransitivität des Kommunikationsmediums und inkonsistentem Routing ungelöste Probleme, die die Implementation eines Transaktionsverfahrens erschweren. Transaktionsverarbeitung erfordert die Einhaltung der ACID-Garantien durch geeignete Nebenläufigkeitskontrollverfahren und verteiltes, atomares Festschreiben. Konkrete Verfahren unterscheiden sich in ihren Eigenschaften. Für den Einsatz in einem bestimmten Szenario müssen sie gezielt ausgewählt und angepasst werden. Beim atomaren Festschreiben ist Fortschritt nur möglich, wenn kein beteiligter Ressourcemanager *endgültig* ausfällt, und daher eine genügend hohe Ausfallsicherheit kritischer Systemkomponenten notwendig.

Diese Punkte werden in Abschnitt 3.3 und Abschnitt 3.4 bei der Entwicklung eines Transaktionsverfahrens für strukturierte Overlay-Netzwerke aufgegriffen.

3 Transaktionen für strukturierte Overlay-Netzwerke

Dieses Kapitel beschreibt Techniken für die Realisierung von Transaktionen in einem strukturierten Overlay-Netzwerk.

Das relationale Modell ist die derzeit übliche Datenmodellierungstechnik für Informationssysteme. Relationale Datenmodelle können relativ einfach auf DHTs übertragen werden. Dies wird in Abschnitt 3.1 beschrieben. Anschließend werden Konsistenzanforderungen von gemeinschaftlichen Informationssystemen anhand von Beispielen untersucht. Zunächst wird dafür in Abschnitt 3.2 eine geeignete Notation für Anwendungsoperationen entwickelt. Für jedes Beispiel wird anschließend ein relationales Datenmodell aufgestellt, die wesentlichen Anwendungsoperationen algorithmisch beschrieben und daraus Konsistenzanforderungen abgeleitet.

Die darauf folgenden Abschnitte bilden den Kern dieser Arbeit. In Abschnitt 3.3 wird ein allgemeines Zellenmodell für strukturierte Overlay-Netzwerke entworfen, dass Replikation und Transaktionsunterstützung ermöglicht. Danach werden Abschnitt 3.4 die Ergebnise aus der Analyse der Beispielanwendungen aufgegriffen, um ein konkretes Transaktionsverfahren für strukturierte Overlay-Netzwerke nach dem Zellenmodell zu entwickeln.

3.1 Datenmodelle: Relationen und Distributed Hash Tables

Entity-Relationship-Modelle (kurz: ER-Modelle) [14] beschreiben inhaltliche Beziehungen zwischen den *Entitäten* mehrer *Entitätstypen*. So können etwa die Entitätstypen PERSON und LAND durch den *Beziehungstyp* HAT-BESUCHT verknüpft werden. Entitäten und Beziehungen werden näher durch atomare Attribute beschrieben (Name und Alter einer Person).

Beziehungen eines Beziehungstyps bestehen aus mehreren Entitäten. Beziehungstypen werden zusätzlich durch die Angabe von Kardinalitäten spezifiziert. Diese legen fest, wie vielen Entitäten eines Entitätstyps an einer Beziehung des Beziehungstyps teilnehmen können bzw. müssen.

Jeder Entitätstyp besitzt einen *Primärschlüssel*. Das ist eine minimale, ausgezeichnete Teilmenge aller Attribute, die die Entitäten des Entitätstyps eindeutig identifizieren.

3.1.1 Datenbank-Relationen

In relationalen Datenbanken werden Entitätstypen als Relationen dargestellt. Relationen entsprechen zweidimensionalen Tabellen mit einer festen Anzahl von Spalten. Jede Zeile einer Tabelle speichert eine einzelne Entität als Tupel aus ihren Attributen (Spalten). Das relationale Modell berücksichtigt keine Ordnung

zwischen Tabellenzeilen. Relationen enthalten aus Sicht des Modells lediglich die Menge aller Tupel (ihre *Extension*).

Beziehungen zwischen verschiedenen Relationen werden mittels Fremdschlüsseln modelliert. Darunter versteht man die Inklusion der Primärschlüsselattribute einer Relation A in einer Relation B. Diese Primärschlüsselattribute dienen in der Relation B als Verweis auf ein Tupel der Relation A und heißen dort Fremdschlüssel.[13] Beziehungstypen mit Kardinalitäten > 1 werden als eigene Relationen dargestellt, die die Attribute des Beziehungstyps und die Fremdschlüssel der bezogenen Entitäten (Relationen) speichert.

Relationale Datenbanken speichern neben den eigentlichen Datentabellen zusätzliche Indexstrukturen für den beschleunigten Zugriff. Es ist üblich, für jede Tabelle mindestens einen Index für den Primärschlüssel anzulegen, um effizient einzelne Tupel abfragen zu können. Zusätzliche Indexstrukturen dienen der Zugriffsoptimierung anwendungspezifischer Anfragen, wie beispielsweise Suchoperationen. Gerade Suchoperationen sind für gemeinschaftliche Informationssysteme besonders wichtig, da sie die Nutzung des von allen Anwendern gemeinsam aufgebauten Datenbestandes ermöglichen.

3.1.2 Relationen in Distributed Hash Tables

Das Datenmodell einer Distributed Hash Table ist wesentlich einfacher. Es gibt nur ein primäres Schlüsselattribut. Jedem Schlüssel wird ein Datum zugeordnet. Schlüssel werden dabei als binäre Brüche zwischen 0 und 1 interpretiert. Dies ermöglicht die Benutzung von Schlüsseln beliebiger Länge. Die Schlüssel einer Distributed Hash Table werden automatisch indiziert. Innerhalb der selben DHT existieren keine weiteren Indexstrukturen. Typische Basisoperationen sind put (key, $value$) und get (key).

Betrachtet man eine DHT als Datenbankrelation, so hat diese nur die beiden Attribute *Schlüssel* und *Wert*. *Schlüssel* ist der Primärschlüssel und wird durch das strukturierte Overlay-Netzwerk indiziert.

Möchte man umgekehrt eine Relation als DHT darstellen, kann man wie folgt vorgehen. Zunächst kodiert man alle Primärschlüsselattribute eines Tupels eindeutig als einen binären Primärschlüssel (Kompositschlüssel). Die Teilattribute müssen dabei extrahierbar bleiben. Eine solche Kodierung kann durch die Verwendung eines Trennsymbols oder von Feld- und Längenangaben erreicht werden. Analog verfährt man für Datenattribute. Das heißt alle Nichtprimärschlüsselattribute eines Tupels werden als ein einzelnes, binäres Datenattribut (Kompositwert) kodiert.

Durch diese Zweiteilung können die zusammengefassten Datenattribute eines Tupels einfach unter dem Kompositschlüssel in der Distributed Hash Table gespeichert werden. Eine schnelle Abfage einzelner Tupel ist möglich, da der Kompositschlüssel durch die DHT indiziert wird. Doch wie können weitere Relationen gespeichert werden?

Um eine weitere Relation zu speichern, benötigt man scheinbar eine weitere DHT. Das ist jedoch nicht nötig. Stattdessen wird jedem Primärschlüssel eine Id für

[13] Diese Darstellung ist vereinfacht. Statt dem Primärschlüssel kann auch ein beliebiger anderer Schlüsselkandidat in einer Fremdschlüsselbeziehung verwendet werden.

die zugeordnete Relation voranstellt und somit den Schlüsselraum der DHT für mehrere Relationen partitioniert. Die Aufteilung des Schlüsselraums bezüglich der Relationen ist nicht gleichmäßig, da die Größe einzelner Relationen sehr unterschiedlich ist. Diese Ungleichverteilung wird durch das Hashing der DHT behoben. Abbildung 3.1 veranschaulicht diese Vorgehensweise am Beispiel.

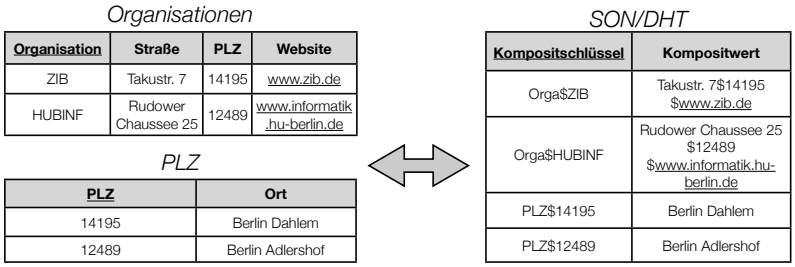

Abbildung 3.1: Speicherung mehrerer Relationen in einer DHT

Alternativ kann die Relations-Id auch an den Primärschlüssel angehängt oder anderweitig mit diesem gemischt werden. Dann gilt jedoch nicht mehr, dass die Tupel einer Relation im Primärschlüsselraum direkt aufeinander folgen. Für Bereichssuche in einem SON bedeutet das, dass Tupel anderer Relationen übersprungen werden müssen. Deshalb wird dieser Ansatz im Folgenden nicht weiter betrachtet.

Indexstrukturen und Ordnung

Eine zusätzliche Indexstruktur wird als Relation realisiert, die sowohl alle indizierten als auch alle Primärschlüsselattribute der indizierten Relationen speichert. Diese Attribute bilden gemeinsam den Primärschlüssel der Indexrelation und verweisen in der DHT symbolisch auf einen „leeren“ Wert, der die Existenz des beschriebenen Tupels anzeigt.[14]

Dies führt zu einer Dreiteilung der Attributmenge einer Relation in *Primärschlüsselattribute*, *Indexattribute* und *Datenattribute*. Die Konkatenation in festgelegter Ordnung von Relations-Id, Primärschlüsselattributen und ggf. Indexattributen ergibt für jedes Tupel den Schlüssel der DHT. Die Eindeutigkeit des so generierten Schlüssels muss bezüglich der Indexattribute durch die Anwendung und das eingesetzte Transaktionsverfahren gewährleistet werden. Es dürfen keine zwei Einträge einer Relation gespeichert werden, die sich in ihren Indexattributen unterscheiden jedoch in ihren Primärschlüsselattributen identisch sind.

Einige DHTs, wie zum Beispiel CAN [53], unterstützen mehrdimensionale Schlüssel aus mehreren Komponenten. Komponenten werden jeweils einzelnen Dimension des SON zugeordnet und diese getrennt voneinander indiziert.

Mehrdimensionale Schlüssel können direkt für die Realisierung zusätzlicher Indexstrukturen benutzt werden. Auf die ersten Dimensionen werden die Primärschlüsselattribute abgebildet. Weitere Dimensionen können für die zusätzliche Indizierung von Datenattributen verwendet werden. Nachteilig ist, dass mit steigender Anzahl der Dimensionen der Suchaufwand zunimmt. Die

[14] Später in dieser Arbeit wird dafür die Notation □ (Indikation) bzw. ⍁ (Löschung) verwendet

Skalierbarkeit des Verfahrens ist snicht gegeben, daher ist nur eine begrenzte Anzahl von indizierten Schlüsselkomponenten bzw. Dimensionen für jede Relation verfügbar.

Im Unterschied zum Relationenmodell existiert eine implizite Ordnung über den Schlüsseln jeder Dimension. Diese Ordnungen können für die Durchführung von Bereichssuche eingesetzt werden.[15]

Attributspezifische Basisoperationen

Bisher wurde davon ausgegangen, dass lediglich die Daten zu einem Primärschlüssel gelesen (get) bzw. geschrieben (put) werden sollen.[16] Für einige Datenstrukturen ist jedoch auch die Verwendung spezifischer Lese- und Schreiboperationen denkbar. So könnte z.B. test (*key*) die unnötige Übertragung der Daten verhindern, wenn nur ihr Vorhandensein getestet werden soll. Mit add (*key*, *item*) und remove (*key*, *item*) ließen sich Mengenoperationen auf dem Datum zu einem Primärschlüssel implementieren. Ein derartiges Schema wäre jedoch nicht mehr in der ersten Normalform (1NF) im Sinne des Relationenmodells.

Der Einsatz spezifischer Basisoperationen für bestimmte Attributtypen stellt eine Optimierung dar und ist orthogonal zu den grundsätzlichen Anforderungen an ein Transaktionsverfahren. Deshalb wird er in dieser Arbeit nicht weiter betrachtet.

3.2 Anwendungen: Datenmodelle und Datenkonsistenz

In diesem Abschnitt werden Beispielanwendungen untersucht. Für jedes Beispiel wird ein relationales Datenmodell und typische Anwendungsoperationen entwickelt und die notwenidgen Konsistenzbedingungen bestimmt. Es werden ein Wiki, ein Wiki mit Metadaten und Bookmark-Sharing betrachtet. Das Wiki stellt die geringsten und das Bookmark-Sharing die höchsten Anforderungen an ein Transaktionsverfahren.

Relationen werden analog zu dem im vorigen Abschnitt für DHTs beschriebenen Verfahren im SON gespeichert. Es wird davon ausgegangen, dass auf dem Schlüsselraum einfache Bereichsanfragen über einzelnen Relationen effizient ausführbar sind.[17] Außerdem wird angenommen, dass das SON Bulk-Operationen unterstüzt. Das sind Operationen, die auf den Daten einer endlichen Menge von Schlüsseln einer Relation ausgeführt werden (Multicast). Alle eingesetzten Basisoperationen können somit in drei Gruppen unterteilt werden: Singlecast, Bulk/Multicast oder aber Bereichsanfragen.

[15] Das resultiertende Datenmodell mit impliziter Ordnung auf allen Zeilen ähnelt dem Datenmodell von XML, jedoch ohne beliebig tiefe Hierarchisierung.
[16] Dies entspricht den Basisoperationen des Seitenmodells aus Unterabschnitt 2.3.1.
[17] Mögliche Techniken beschreiben [62, 4, 52].

3.2.1 Pseudocode-Notation der Algorithmen

Die Untersuchung von Anwendungsoperationen erfordert es diese aufschreiben zu können. Dafür wurde im Rahmen dieser Arbeit eine Pseudocode-Notation entwickelt, die die Verschriftung von Anwendungsoperationen auf strukturierten Overlay-Netzwerken mit Transaktionsunterstützung ermöglicht. Die Notation orientiert sich an der relationalen Algebra und versucht eine Balance zwischen deklarativen und imperativen Sprachelementen zu erreichen. Die einzelnen Operationen werden imperativ ausgeführt, die von ihnen bearbeiteten Daten aber deklarativ beschrieben. Dies ist analog zu *SQL*, welches jedoch wesentlich funktionsreicher ist.

Zusätzliches Ziel der Notation ist die Trennung und Sichtbarmachung unterschiedlicher relationaler Operationen (wie z.B. Projektion). SQL-Anweisungen sind deklarativ. In relationalen Datenbankmanagementsystemen entstehen dadurch Optimierungsmöglichkeiten für die konkrete Ausführung von Anfragen. In dezentralen Systemen sind derartige Optimierungen wesentlich schwieriger zu realisieren, da Anfrageoptimierung globales Wissen benötigt, z.B. über die Anzahl und Größe der Einträge einzelner Relationen. Derartiges Wissen ist oft nicht verfügbar, deshalb zielt die verwendete Notation darauf ab, die Anfrageoptimierung stärker dem Anwender zu überlassen.

Anweisungen, Funktionen und Transaktionen

Jede Zeile besteht aus einer Anweisung. Alle Anweisungen werden imperativ nacheinander ausgeführt. Prozedur- und Funktionsnamen werden in Sans-Serif geschrieben. Parameter und Variablennamen hingegen *kursiv* und Schlüsselworte **fett** gesetzt. Funktionen können mehrere Rückgabewerte besitzen. Diese müssen alle beim Funktionsaufruf konsumiert werden. Beispiel: $x, y \leftarrow$ Coordinates $(\text{„}ZIB\text{“})$, dabei ist „$\leftarrow$“ der Zuweisungsoperator.

Programmatisch werden die Anweisungen einer Transaktion durch die Befehlspaare **begin transaction** und **commit transaction** bzw. **abort transaction** umschlossen, um die zu einer Transaktion gehörenden Operationen zu markieren. Diese Markierung hat dynamische Ausdehnung (dynamic extent), das heißt sie ist mit dem jeweiligen Thread assoziert.[18]

Relationen und Tupel

Namen von Relationen werden in KAPITÄLCHEN geschrieben und durch ihre Primärschlüsselattribute indiziert. Die *Reihenfolge* wird dabei durch die Relation vorgegebenen. Leseoperationen auf Relationen ($\text{ADDRESS}_{\text{„}ZIB\text{“}}$) ergeben Tupel aus allen Attributen. Schreiboperationen verwenden nur Tupel aus Index- und Datenattributen als neue Werte ($\text{ADDRESS}_{\text{„}ZIB\text{“}} \leftarrow (\text{„}Takustrasse\ 7\text{“}, \text{„}Berlin\text{“})$). Indexrelationen zeigen lediglich das Vorhandensein von Tupeln an. Die Notation ist $\text{INDEXREL}_{key} \leftarrow \square$ bzw. $\leftarrow \not\square$. Letzteres entfernt ein Tupel aus einer Relation. Der logische Test auf Tupelexistenz ist daher $\text{RELATION}_{key} \neq \not\square$.

Einzelne Datenatome eines Relationstupels werden mittels global festgelegter Label ausgewählt. In dieser Arbeit sind das englischsprachige Bezeichner für

[18] Hingegen verwenden Kontrollstrukturen, wie z.B. **repeat** ... **until**, bei der Einfassung mehrerer Anweisung lexikalischen Gültigkeitsbereich (lexical scope).

die Attribute der jeweiligen Relationen. Beispielsweise selektiert t_{street} die Straße eines ADDRESS-Tupels. Labels dienen der Vermeidung von Zahlen bei der Adressierung von Tupelkomponenten. In einer konkreten Implementation der vorgestellten Algorithmen müssten Label entsprechend aufgelöst werden. Dies ist stets (statisch) möglich.

(a, b, c) ist ein künstliches Tupel aus drei Elementen. Dieses Tupel ist nicht direkt an eine Relation gebunden. Ist der Aufbau des Tupels im Kontext ersichtlich, werden auch die Komponenten künstlicher Tupel mittels Labels adressiert. „$\cdot$“ ist die Tupelkonkatenation. 1-Tupel und normale Werte sind nicht unterscheidbar. Es gilt $\{(a), (b)\} = \{a, b\}$. Dies vereinfacht gelegentlich die Algorithmen.

Umgebung, Bindung und Umgebungsanwendung

$\{label_1 \Rightarrow value_1,\ label_2 \Rightarrow value_2\}$ ist eine *Umgebung*. Eine Umgebung ist eine Menge von *Bindungen*. Bindungen b sind Tupel aus Labels und Attributwerten ($b = b_{label} \Rightarrow b_{value} = (b_{label}, b_{value})$). Umgebungen sind *wohlgeformt*, wenn sie keine zwei Bindungen mit gleichem Label enthalten. Wohlgeformte Umgebungen können mittels Labels indiziert werden und ergeben dann den Attributwert der entsprechenden Bindung. Falls für das gegebene Label keine Bindung in der Umgebung existiert, ist das Ergebnis $\perp$. Die später vorgestellten Algorithmen gehen davon aus, dass alle verwendeten Umgebungen (z.B. als Funktionsargumente) wohlgeformt sind.

Umgebungen können direkt auf Relationstupel $\text{ADRESSE}_{\text{„ZIB“}} \overset{\text{UPDATE}}{\leftarrow} \{street \Rightarrow \text{„}Schlossalle\ 5\text{“}\}$ angewendet werden. Dabei wird für jede Bindung, die ihrem Label entsprechende Komponente auf den Wert der Bindung gesetzt. Es ist ein Fehler, wenn bei der Erzeugung eines Relationstupels durch Umgebungsanwendung ein benötigtes Attribut fehlt, da eine erforderliche Bindungen keinen Wert hat ($\perp$ ist).

Projektion

π ist der Projektionsoperator für einzelne Tupel und Umgebungen. Π ist der Projektionsoperator für Relationen und Tupelmengen.

$$\begin{aligned} \Pi_{attr_1, \ldots, attr_n}(M) &= \{\pi_{attr_1, \ldots, attr_n}(t) \mid t \in M\} \\ &= \{ (\, t_{attr_1}, \ldots, t_{attr_n}) \mid t \in M\} \end{aligned}$$

Wo dies eindeutig ist, werden die Projektionsoperatoren auch ohne explizite Klammern verwendet. Π^{prim}_{Rel}, Π^{data}_{Rel}, Π^{index}_{Rel}, $\Pi^{\neg prim}_{Rel}$ projizieren jeweils die Primärschlüssel-, Daten-, Index- oder aber die Nicht-Primärschlüsselattribute einer Relation REL.

Bereichsanfragen

Eine Bereichsanfrage (range query, auch: Bereichssuche) über einer Relation wird durch ein Fragezeichen „?“gekennzeichnet und mittels *Set-Comprehension* und Projektion ausgedrückt. Beispielsweise ergibt

$$\Pi_{orga,\, street,\, city}\{t \mid t \in \textsc{Adress}^{?}_{\text{„ZI“}<orga<\text{„ZZ“}}\} \text{ (oder kurz } \textsc{Adress}^{?}_{\text{„ZI“}<orga<\text{„ZZ“}})$$

alle Tupel von Organisationen aus ADRESSE deren Organisationsnamen lexikalisch zwischen „ZI“ und „ZZ“ liegen. Dabei stehen im Index jeweils Bedingungen für jedes Primärschlüsselattribut (repräsentiert durch das jeweilige Label) und optional für weitere Indexattribute in der durch die Relation vorgegebenen Reihenfolge.

Der *künstliche Attributwert* $*$ kann verwendet werden, um beliebige Attributwerte zu beschreiben. Dies ist praktisch, um Attributwerte über Funktionsargumente optional einzuschränken. $\text{DHT}^{?}_{key_1=\text{„}a\text{“},\, key_2}$ ist verkürzt für $\text{DHT}^{?}_{key_1=\text{„}a\text{“},\, key_2=*}$.

Gelegentlich sollen die Ergebnisse einer Anfrage sortiert werden. Beispiel:

$$\Pi_{orga,\, street,\, city}\{t \mid t \in \textsc{Address}^{?}_{\text{„ZI“}<orga<\text{„ZZ“}}\}^{\overleftarrow{orga},\, \overrightarrow{street}}_{\#<50}$$

sortiert zuerst absteigend nach dem Namen der Organisation und dann aufsteigend nach dem Straßennamen. Es werden die ersten 49 Ergebnisstupel ermittelt.

Die Elemente einer „sortierten Menge“ werden bei linearer Iteration (**for**) in der Sortierreihenfolge durchlaufen. Das Sortierkriterium der Ergebnisse einer Bereichsanfrage wird durch eine Liste von Attributen beschrieben. Ein Pfeil über dem Attribut gibt dabei die jeweilige Sortierrichtung an. Zusätzlich kann die maximale Anzahl der Ergebnisse beschränkt werden

Änderungs- und Löschoperationen über Schlüsselbereichen

Die Notation für Veränderungen von Tupeln, die durch eine Bereichsanfrage ausgewählt wurden, ist

$$t \overset{\text{UPDATE}}{\longleftarrow} \pi^{\neg prim}_{\text{REL}_t}(\text{Umgebung, ggf. konstruiert aus } t) \mid \forall t \in \text{Bereichsanfrage}$$

Veränderungen der Primärschlüsselattribute sind dabei untersagt! Dies kann, wie hier, durch die Verwendung geeigneter Projektionen sichergestellt werden.

Es können auch die Ergebnistupel einer Bereichsanfrage gelöscht werden. Die Notation verwendet die Löschoperation für einzelne Tupel: $t \leftarrow \not\emptyset \mid \forall t \in$ Bereichsanfrage.

Bulk-Operationen

Bulk-Schreiboperationen werden eingesetzt, um eine *vorberechnete* Menge von Tupeln zu verändern. Für Hinzufügen bzw. Überschreiben oder aber Löschen von Tupeln wird

$$\forall t \in \text{Tupelmenge} : \text{RELATION} \overset{+}{\leftarrow} t \text{ bzw. } \forall t \in \text{Tupelmenge} : \text{RELATION} \overset{-}{\leftarrow} t$$

geschrieben. Die Notation ist eine syntaktische Vereinfachung gegenüber der Verwendung von „←“, die im Beispiel eine Dekonstruktion des Tupels erfordern würde.[19]

Bulk-Updates sind ebenfalls möglich

$$\forall t \in \text{Tupelmenge} : \ \text{RELATION}_{t_{primAttr_1}, \ldots, t_{primAttr_n}} \leftarrow \pi^{\neg prim}_{\text{RELATION}}(t)$$

Rechtsseitig darf dabei ausschließlich lesend auf den lokalen Zustand des umschließenden Algorithmus zugegriffen werden!

Syntaxtabelle

Tabelle 3.1 fasst die Notation noch einmal zusammen.

Tabelle 3.1: Pseudocode-Notation im Überblick

Notation	Bedeutung
Procedure Proc $(arg_1, arg_2, \ldots, arg_n)$	Prozedurdeklaration
Function Fun $(arg_1, arg_2 \overset{def}{=} \text{„Wert“}, \ldots, arg_n)$	Funktionsdeklaration mit Defaultwert „Wert“ für arg_2
$x, y \leftarrow$ Coordinates $(\text{„}ZIB\text{“})$	Funktionsaufruf und Zuweisung mehrerer Rückgabewerte
begin transaction ... **commit (abort) transaction**	Einfassung von Operationen einer Transaktion
$\text{ADDRESS}_{\text{„}ZIB\text{“}}$	Relationstupel lesen
$\text{ADDRESS}_{\text{„}ZIB\text{“}} \leftarrow (\text{„}Takustrasse\ 7\text{“}, \text{„}Berlin\text{“})$	Relationstupel schreiben
$(a) \cdot (b, c) = (a, b, c)$	Tupel und Tupelkonkatenation
$\{label_1 \Rightarrow value_1,\ label_2 \Rightarrow value_2\}$	Umgebung aus zwei Bindungen
Umgebung U: **if** $U_{label} = \perp$ **then** ...	Wert einer Bindung extrahieren ($\perp$ bedeutet, dass U keine Bindung für *label* enthält)
$\text{ADRESSE}_{\text{„}ZIB\text{“}} \overset{\text{UPDATE}}{\leftarrow} \{street \Rightarrow \text{„}Schlossalle\ 5\text{“}\}$	Umgebungsanwendung
$\Pi_{attr_1, \ldots, attr_n}(M) = \{\pi_{attr_1, \ldots, attr_n}(t) \mid t \in M\} = \{(t_{attr_1}, \ldots, t_{attr_n}) \mid t \in M\}$	Projektion
$\Pi^{prim}_{Rel}, \Pi^{data}_{Rel}, \Pi^{index}_{Rel}, \Pi^{\neg prim}_{Rel}$	Projektion von Primärschlüssel-, Daten-, Index- und Nicht-Primärschlüsselattributen
$\text{DHT}^{?}_{key_1=\text{„}a\text{“}, key_2} = \text{DHT}^{?}_{\overrightarrow{orga}, \overrightarrow{street}}{}_{key_1=\text{„}a\text{“}, key_2=*}$	Einfache Bereichsanfrage ($*$ steht für einen beliebigen Wert)
$\text{ADDRESS}^{?}_{\text{„}ZI\text{“}<orga<\text{„}ZZ\text{“}\ \#<50}$	Bereichsanfrage mit Sortierung und Beschränkung der Anzahl der Ergebnisse
$t \overset{\text{UPDATE}}{\leftarrow} \{\text{Umgebung konstruiert aus } t\} \mid \forall t \in \text{Anfrage}$	Veränderung (Update) mittels Bereichsanfrage
$t \leftarrow \not\emptyset \mid \forall t \in \text{Anfrage}$	Löschen mittels Bereichsanfrage
$\forall t \in \text{Tupelmenge} : \text{RELATION} \overset{+}{\leftarrow} t \text{ bzw. } \overset{-}{\leftarrow} t$	Bulk-Hinzufügen bzw. Löschen
$\forall t \in \text{Tupelmenge} : \text{REL}_{t_{primAttr_1}, \ldots, t_{primAttr_n}} \leftarrow \pi^{\neg prim}_{\text{REL}}(t)$	Bulk-Updates

[19] $\forall t \in \text{Tupelmenge} : \ \text{RELATION}_{t_{primAttr_1}, \ldots, t_{primAttr_n}} \leftarrow \square \text{ bzw. } \leftarrow \not\emptyset$

3.2.2 Wiki

In der Einleitung wurde bereits das prinzipielle Datenmodell eines Wikis erklärt. Eindeutigen Seitennamen werden ihre jeweiligen (textuellen) Inhalte zugeordnet, die dann insbesondere auch Hyperlinks auf andere Seiten des Wikis beinhalten können. Moderne Wikis[20] beschränken sich jedoch nicht auf diese Basisfunktionalität. Häufig bieten sie zusätzliche Navigationshilfen für den Zugriff auf einzelne Seiten:

- Suche nach Seitennamen,
- Automatisch erzeugte Verzeichnisse (insbesondere *recent changes*: Zuletzt bearbeitete Seiten),
- *Backlinks* (Navigationshilfe, die für eine Seite anzeigt, welche anderen Seiten auf diese verweisen),
- Kategorisierung/*Tagging* (Zuordnung von Schlüsselwörtern (Tags) zu Seiten und Suche nach Seiten, denen bestimmte Tags zugeordnet wurden),
- Volltextsuche nach Seiteninhalten.

Solche Navigationshilfen sind eine wichtige Ergänzung, da sie das Finden relevanter Inhalte vereinfachen. Weitere, oft realisierte Funktionen sind:

- Anhänge (Bilder, Dokumente etc.),
- Mehrbenutzerbetrieb (Authentifzierung, Nutzerverwaltung, Änderungsmoderation, Diskussionsunterstützung),
- Anti-Spam-Mechanismen (erfordert meist Mehrnutzerbetrieb),
- Versionskontrolle (erfordert meist Mehrnutzerbetrieb),
- Metadaten für Seiten und Anhänge (z.B. Geoinformation),
- Seitenvorlagen (Unterstützung für die effiziente Erzeugung standardisierter Seiten, wie z.B. Sever-Dokumentationsseiten in einem Wiki für die Administration eines IntraNets).

Nicht jede Wiki-Software implementiert diese Anforderungen vollständig. Die überwiegende Mehrzahl der Funktionen wird aber von modernen Wiki-Systemen bereitgestellt. Im Folgenden werden Mehrbenutzerbetrieb und dementsprechend Anti-Spam-Mechanismen und Versionskontrolle von Wiki-Seiten nicht näher behandelt, da die hierfür erforderlichen Authentifizierungs- und Sicherheitsmechanismen in typischen SONs derzeit nicht verfügbar und, wie bereits in der Einleitung erwähnt, nicht Gegenstand dieser Arbeit sind.

Seitenvorlagen und Anhänge können als Wikiseiten anderen Typs modelliert werden. Die Speicherung und Indizierung einer begrenzten, vorab festgelegten Menge von Metadatenattributen (z.B. Geokoordinaten) für jede Wikiseite kann durch den Einsatz von einem SON mit mehrdimensionalen Schlüsseln realisiert werden. Die Modellierung beliebiger Metadaten beschreibt Unterabschnitt 3.2.3. Das letzte Änderungsdatum ist ein Metadatum und die Erzeugung des

[20] z.B. TikiWiki: http://tikiwiki.org, Mediawiki: http://www.mediawiki.org, MoinMoin: http://moinmoin.wikiwikiweb.de, TWiki: http://twiki.org, KWiki: http://www.kwiki.org, SnipSnap: http://snipsnap.org

Recent Changes-Verzeichnis entspricht somit einer Bereichsanfrage. Analog kann zumindest nach Seiten gesucht werden, deren Namen einen gemeinsamen Präfix besitzen. Die Realisierung von Backlinks und Volltextsuche erfordert jedoch den Einsatz zusätzlicher Indexstrukturen, die ggf. gleichzeitig mit dem Inhalt einer Seite geändert werden müssen. Tagging wird in Unterabschnitt 3.2.4 besprochen und deshalb hier ebenfalls vernachlässigt.

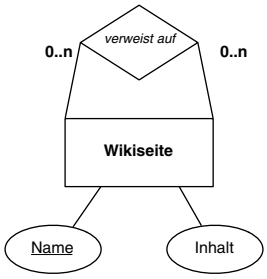

Abbildung 3.2: ER-Modell: Wiki

Abbildung 3.2 zeigt ein ER-Modell für ein einfaches Wiki, das bei entsprechender Indizierung Recent changes, Suche nach Seiten mit gleichem Namenspräfix und Backlinks unterstützt und damit exemplarisch alle Anforderungen außer Mehrbenutzerbetrieb, Spamschutz, Versionskontrolle und Tagging abdeckt. Dieses ER-Modell kann mit zwei Relationen gespeichert werden:

Tabelle 3.2: Relationenmodell: Wiki

Relation	Primärschlüsselattribute	Indexattribute	Daten
SEITENINHALT	Seitenname (*pageName*)	Änderungsdatum (*ctime*)	Seiteninhalt (*content*)
BACKLINKS	Referenzierende Seite (*referencing*), Referenzierte Seite (*referenced*)	-	-

Basisoperationen

Die entscheidende Operationen für das beschriebene Wiki sind WikiRead (*name*), WikiWrite (*name*, *content*) und RecentChanges. WikiRead (Algorithmus 3.1) liefert neben dem Inhalt einer Seite auch alle Backlinks auf diese zurück. RecentChanges (Algorithmus 3.2) kann direkt als Bereichsanfragen realisiert werden. WikiRead und WikiWrite (Algorithmus 3.3) entsprechen den DHT-Operationen put und get.

Das gleichzeitige Schreiben einer Seite durch mehrere Nutzer führt zu Schreibkonflikten. Wikis lösen dieses Problem durch zwei Techniken. Zunächst werden alle Schreibzugriffe auf eine Seite mittels Sperren oder Anfragequeues serialisiert. Vor der Durchführung des Schreibens wird geprüft, ob der dem Nutzer zuletzt bekannte Seiteninhalt zwischenzeitlich modifiziert wurde und somit ein Konflikt vorliegt. Ist dies der Fall, wird der Schreibzugriff abgebrochen und der Nutzer über die zusätzlichen Modifikationen informiert. Er hat dann Gelegenheit, seine Änderungen anzupassen (Merge) und anschließend den Prozess zu wiederholen, indem er die neue Fassung speichert.

Algorithmus 3.1 WikiRead: Inhalt und Backlinks einer Wiki-Seite lesen

```
1: function WikiRead (pageName)
2:     begin transaction
3:         content ← π_content(SEITENINHALT_pageName)
4:         backlinks ← Π_referenced(BACKLINKS?_referencing=pageName, referenced)
5:     commit transaction
6:     return content, backlinks
7: end function
```

Algorithmus 3.2 RecentChanges: Letzte Änderungen eines Wikis abfragen

```
1: procedure RecentChanges (beforeTime, limit)
2:     begin transaction
3:         result ← {SEITEINHALT?_pageName, ctime>beforeTime}^(←ctime)_#<limit
4:     commit transaction
5:     return result
6: end procedure
```

Da gleichzeitige Änderungen nur sehr selten auftreten, ist diese Lösung für die Konfliktbehandlung praktisch völlig ausreichend. Für die Umsetzung mit Transaktionen, ist es erforderlich, die Schreiboperation zu modifizieren: WikiWrite (*name*, $content_{old}$, $content_{new}$) überschreibt eine Seite nur dann, wenn ihr derzeitiger Inhalt mit $content_{old}$ identisch ist.

Diese Technik heißt Vergleichen und Vertauschen (compare and swap, kurz: CAS) [33] und kann auf modernen Mikroprozessoren als atomare Maschineninstruktion realisiert werden. Hier wird sie durch den Transaktionsmechanismus ermöglicht und eingesetzt, um die Dauer von Schreibtransaktionen zu minimieren. Ohne den Einsatz von CAS muss die Seite für die gesamte Bearbeitungsdauer gesperrt werden. Wird dafür eine Schreibsperre genutzt, ist kein gleichzeitiges Lesen möglich. Wird zunächst eines Lesesperre erworben und diese später in eine Schreibsperre umgewandelt, können Konflikte auftreten und zum Abbruch der Transaktion führen. Dies entspricht dem Einsatz von CAS bei WikiWrite (Algorithmus 3.3). Der Ansatz erfordert jedoch einen Scheduler, der die Umwandlung von Lese- in Schreibsperren (lock promotion) unterstützt. Alternativ kann auch Multiversionierung (MVCC) eingesetzt werden. Ohne CAS innerhalb der Transaktion führt das aber auch zur Blockierung neuerer Lesetransaktionen.

Die Konfliktbehandlung in Wikis ist ähnlich zur Nebenläufigkeitskontrolle von Versionskontrollsystemen. Das Entdecken eines Konflikts bei der Einspielung von Änderungen im zentralen Repository entspricht dem Abbruch des Schreibzugriffs. Versionskontrollsysteme bieten zusätzlich eine Heuristik, die das automatische Zusammenführen von Änderungen ermöglicht und damit die Anzahl von Abbrüchen/Konflikten minimiert. Die Ergänzung von Wikis um eine solche Heuristik ist möglich.

Konsistenz

Zusätzlich müssen bei jeder Änderung des Seiteninhaltes alle Indexstrukturen, z.B. für Backlinks, angepasst werden. Ist es notwendig, dies atomar auszuführen?

Algorithmus 3.3 WikiWrite: Neuen Inhalt einer Wiki-Seite schreiben und Backlinks aktualisieren

```
1:  procedure WikiWrite (pageName, content_old, content_new)
2:      refs_old ← Refs (content_old)
3:      refs_new ← Refs (content_new)
4:      refs_del ← refs_old \ refs_new          — Vorberechnung
5:      refs_add ← refs_new \ refs_old
6:      txStartTime ← CurrentTimeUTC()
7:      begin transaction
8:          if π_content(SEITENINHALT_pageName) = content_old then
9:              SEITENINHALT_pageName = (txStartTime, content_new)
10:             ∀t ∈ {(ref, pageName) | ref ∈ refs_add} : BACKLINKS ←+ t
11:             ∀t ∈ {(ref, pageName) | ref ∈ refs_del} : BACKLINKS ←− t
12:         else
13:             abort transaction
14:         end if
15:     commit transaction
16: end procedure
```

Das hängt davon ab, wie die Indexstrukturen benutzt werden. Da sowohl Volltextsuche als auch Backlinks *primäre Navigationsinstrumente* für den Zugriff auf gespeicherte Inhalte sind, ist Aktualität geboten und damit die Verwendung regelmässiger („lazy“) Indizierungsprozesse keine akzeptable Lösung.

In Bezug auf Volltextsuche kann man dagegen argumentieren, dass durch Recent changes (Algorithmus 3.2) ja bereits der Zugriff auf aktuelle Inhalte möglich ist. Dies greift jedoch zu kurz, da mancher Nutzer eines Wikis diese Funktion nicht kennt und das Fehlschlagen der Suche dann zusätzliche Navigationsschritte erfordert. Außerdem widerspricht es Erfahrungen des Nutzers mit typischen Suchfunktionen („Was da ist, kann auch gefunden werden“). Benutzer erwarten von Navigationsinstrumenten wie Suche und Backlinks vollständige Ergebnisse.

Mögliche Konsistenzbedingungen für ein Wiki sind daher:

W1 Jede Seite in SEITENINHALTE speichert stets den aktuellsten Seiteninhalt. Dieser wird durch WikiRead gelesen und durch WikiWrite modifizert.

W2 Es ist einem Nutzer nicht möglich, eine Seite zu überschreiben, deren aktuellen Inhalt er nicht gesehen hat.

W3 BACKLINKS enthält stets ausschließlich und vollständig alle Backlinks bezüglich der aktuellen Seiteninhalte aller in SEITENINHALTE gespeicherten Seiten.

Diese Bedingungen verlangen bereits relativ starke Eigenschaften von einem Transaktionsmechanismus! Die Algorithmen 3.1 und 3.3 implementieren WikiRead bzw. WikiWrite mittels eines (ACID-) Transakionsmechanismus. Lediglich einige Unteroperationen (z.B. bei WikiWrite: Schreiben des Inhalts, Aktualisierung des Index) können in ihrer Reihenfolge vertauscht werden.

3.2.3 Wiki mit Metadaten

In diesem Abschnitt wird das beschriebene Wiki um Metadaten erweitert. Dabei wird zwischen zwei Arten von Metadaten unterschieden: Systemmetadaten und Nutzermetadaten. Systemmetadaten werden auf eine Indexdimension des strukturierten Overlay-Netzwerks abgebildet und somit direkt in der Relation SEITENINHALT gespeichert. Systemmetadaten sollten deshalb Attribute beinhalten, die für die überwiegende Mehrzahl der Wikiseiten verfügbar sind und häufig abgefragt werden, wie beispielsweise Geokoordinaten in einem Wiki für touristische Sehenswürdigkeiten einer Stadt.

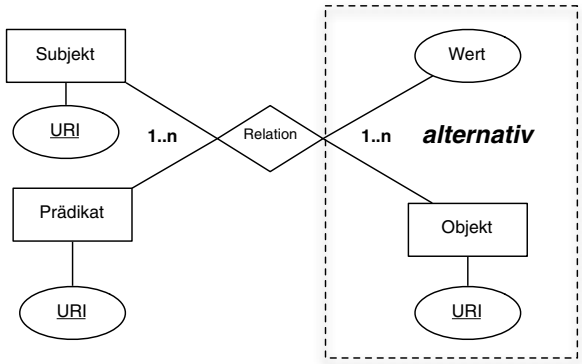

Abbildung 3.3: ER-Modell: Metadaten

Die Anzahl der verfügbaren Indexdimensionen ist begrenzt und daher können nicht alle Metadatenattribute auf Indexdimensionen abgebildet werden. Deshalb ist die Verwendung einer sekundären Relation METADATEN nötig, die für jede Seite zusätzliche Nutzermetadaten speichert. Mit Geokoordinaten als exemplarischen Systemmetadaten ergibt sich folgendes Relationenmodell (Tabelle 3.3), das sich für die Speicherung von Nutzermetadaten an das RDF-Modell [37] (Abbildung 3.3) anlehnt:

Tabelle 3.3: Relationenmodell: Wiki mit Metadaten

Relation	Primärschlüsselattribute	Indexattribute	Daten
SEITENINHALT	Seitenname (*pageName*)	Änderungsdatum (*ctime*), Längengrad, Breitengrad	Seiteninhalt (*content*)
BACKLINKS	Referenzierende Seite (*referencing*), Referenzierte Seite (*referenced*)	-	-
METADATA	Seitenname (*pageName*), Attributname (*attrName*)	Wert (*attrValue*)	-

Algorithmus 3.4 GetSysMetadata: Systemmetadaten einer Wiki-Seite lesen

```
1: function GetSysMetadata (pageName)
2:     begin transaction
3:         result ← SEITENINHALT_pageName
4:     commit transaction
5:     return {attr_1 ⇐ result_attr_1, ..., attr_n ⇐ result_attr_n}
6: end function
```

Algorithmus 3.5 SetSysMetadata: Systemmetadaten einer Wiki-Seite schreiben

Require: *changeEnv* beinhaltet nur Bindungen der Systemmetadatenattribute

```
1: procedure SetSysMetadata (pageName, content_old, changeEnv)
2:     begin transaction
3:         if π_content(SEITENINHALT_pageName) = content_old then
4:             SEITENINHALT_pageName ⟵(UPDATE) changeEnv
5:         else
6:             abort transaction
7:         end if
8:     commit transaction
9: end procedure
```

Basisoperationen und Konsistenz

Die Behandlung von Metadaten erfordert neue Basisoperationen: GetMetadata und SetMetadata, die aus Gründen der Vereinfachung folgend zusätzlich in GetSysMetadata (Algorithmus 3.4), SetSysMetadata (Algorithmus 3.5) und GetUserMetadata (Algorithmus 3.6), SetUserMetadata (Algorithmus 3.7) unterteilt werden.

Für die Suche in Sytemmetadaten genügt – wie bei der Suche nach aktuellsten Wiki-Seiten – eine Bereichsanfrage. Die Suche in Nutzermetadaten erfordert hingegen die *Verknüpfung von Daten aus zwei Relationen*. Dies wird im nächsten Unterabschnitt über Bookmark-Sharing-Systeme behandelt.

Hinsichtlich der Konsistenz sind neben den Bedingungen des vorigen Unterabschnittes zusätzliche Anforderungen notwendig:

W4 METADATA enthält stets ausschließlich und vollständig alle hinterlegten Metadaten.

W5 Es ist einem Nutzer nicht möglich, die Metadaten einer Seite zu überschreiben, deren aktuellen Inhalt er nicht gesehen hat.

Die erste Bedingung gewährleistet, dass der Nutzer stets auf die zu einem Seiteninhalt zugehörigen Metadaten zugreift. Die zweite Bedingung verhindert, dass unzulässige Metadatenänderungen in Unkenntnis des aktuellen Seiteninhaltes durchgeführt werden.

GetSysMetadata und SetSysMetadata sind analog zu WikiRead bzw. WikiWrite. Beide Operationen verwenden jedoch Umgebungen. Dies verhindert, dass bei SetSysMetadata alle Metadatenattribute neu geschrieben werden müssen und begünstigt so in Zusammenhang mit einem entsprechendenden Transaktionsverfahren die nebenläufige Ausführung mehrerer Schreiboperationen auf Metadaten.

Algorithmus 3.6 GetUserMetadata: Nutzermetadaten einer Wiki-Seite lesen

```
1: function GetUserMetadata (pageName)
2:     begin transaction
3:         result ← {t_attrName ⇐ t_attrValue | t ∈ METADATA?_name=pageName, attrName}
4:     commit transaction
5:     return result
6: end function
```

Algorithmus 3.7 SetUserMetadata: Nutzermetadaten einer Wiki-Seite schreiben

Require: *changeEnv* beinhaltet nur Bindungen der Nutzermetadatenattribute

```
1: procedure SetUserMetadata (pageName, content_old, changeEnv)
2:     begin transaction
3:         if π_content(SEITENINHALT_pageName) = content_old then
4:             ∀(anAttrName ⇐ anAttrValue) ∈ changeEnv :
5:                 METADATA_pageName, anAttrName ← anAttrValue
6:         else
7:             abort transaction
8:         end if
9:     commit transaction
10: end procedure
```

GetUserMetadata ist die erste Operation, die Bereichssuche benötigt, um ihr Ergebnis zu ermitteln. Dies folgt aus der Unterstützung beliebiger Nutzermetadatenattribute mittels einer zusätzlichen Relation. Effiziente Bereichssuche ist möglich und damit die Operation auch praktisch umsetzbar.

SetUserMetadata erfordert hingegen die Durchführung einer Bulk-Operation: das *parallele* Schreiben mehrerer vorberechneter Tupel. Dies kann bespielsweise durch eine Multicast-Operation des strukturierten Overlay-Netzwerks realisiert werden.

Abermals ist ersichtlich, dass viele Schritte der Anwendungsoperationen keine besonderen Anforderungen bzgl. der Reihenfolge ihrer Ausführung stellen. Die einzige Ausnahme bilden die Set-Operationen, die zuerst den bekannten und den derzeitigen Seiteninhalt vergleichen, bevor Änderungen durchgeführt werden. Sowohl die Bulk-Operationen als auch die Range Queries erfordern lediglich, dass sie atomar und vollständig auf alle adressierten Tupel angewendet werden.

3.2.4 Bookmark-Sharing

Typische Bookmark-Sharing-Systeme, wie etwa delicious,[21] erlauben die Verwaltung der eigenen Browser-Bookmarks eines Nutzers über eine Webseite. Das hat den Vorteil, dass sie dadurch zeit- und ortsunabhängig verfügbar sind. Zu jedem Bookmark werden Nutzername, Titel, Beschreibung und URL, sowie das Datum der letzten Änderung des Eintrags hinterlegt.

Die zentrale Funktion von Bookmark-Sharing-Systeme ist die Kategorisierung von Bookmarks mittels Tagging. Jedem Eintrag werden mehrere Tags (Schlüsselwörter) zugeordnet. Tags können einerseits benutzt werden, um die eigenen

[21] http://del.icio.us

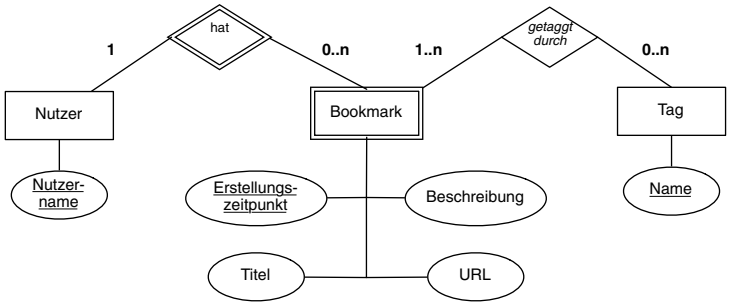

Abbildung 3.4: ER-Modell: Bookmark-Sharing

Bookmarks zu strukturieren, andererseits ermöglichen sie, die gezielte Suche nach Bookmarks anderer Nutzer. Die Zuordnung von Tags zu eigenen Bookmarks erzeugt somit Mehrwert für alle Nutzer des Systems. Typische Suchanfragen sind: 1) Die Suche nach Nutzern, die einen Bookmark auf die selbe URL angelegt haben, und die Möglichkeit, deren Bookmarks anzuzeigen und zu durchsuchen. 2) Die Suche nach Bookmarks anderer Nutzer, denen bestimmte Tags zugeordnet wurden. Für jede Suche besteht die Möglichkeit, die neuesten (bzgl. des Erstellungsdatum) n Ergebnisse sortiert abzufragen und dadurch neue Einträge zu relevanten Themen zu finden.[22]

Abbildung 3.4 zeigt ein mögliches ER-Modell für einen derartigen Dienst. Bookmarks werden als schwache Entitäten in Abhängigkeit ihres Nutzers modelliert. Das ER-Modell führt direkt zum Relationenmodell (Tabelle 3.4).

Tabelle 3.4: Relationenmodell: Bookmark-Sharing

Relation	Primärschlüsselattribute	Indexattribute	Daten
BOOKMARKS	Nutzername (*userName*), Erstellungszeitpunkt (*ctime*)	URL (*url*)	Titel (*titel*), Beschreibung (*descr*)
TAGS	Tag-Name (*tag*), Bookmark-Fremdschlüssel: Nutzername (*userName*), Erstellungszeitpunkt (*ctime*)	-	-

Basisoperationen und Konsistenz

Bookmarks werden wesentlich seltener geändert als Wiki-Seiten. Daher genügt es, das Bearbeiten eines Bookmarks als Löschen und Neuanlegen zu modellieren. Die entsprechenden Algorithmen werden hier nicht detailliert behandelt, da sie analog zu den Algorithmen der vorigen Abschnitte sind.

Wie die Wiki-Beispielanwendungen erfordert auch Bookmark-Sharing die Aktualisierung sekundärer Indexstrukturen (TAGS). Die Konsistenzbedingungen sind daher:

[22] Suchanfragen können oft in Form von RSS-Dateien abonniert werden. Mögliche Implementationstechniken dafür sind in einem SON der Einsatz eines Publish-Subscribe oder aber Client-Polling.

B1 BOOKMARKS enthält stets ausschließlich und vollständig alle dem System bekannten Bookmarks und die über sie gespeicherten Daten (Titel, URL, Beschreibung, Nutzer, Datum der Erzeugung).

B2 TAGS enthält stets ausschließlich und vollständig die zu einem Bookmark gehörenden Tags.

Algorithmus 3.8 QueryTags: Bookmarks nach Tags und Nutzern abfragen

```
1:  function QueryTags (anUserName =def= *, tags =def= ∅, limit)
2:      begin transaction
3:          if tags = ∅ then
4:              result ← {b | b ∈ BOOKMARKS?_{userName=anUserName, ctime}}^{←ctime}_{#<=limit}
5:          else
6:              ∀tag ∈ tags :
7:                  tagged_tag ← Π_{userName, ctime}{t | t ∈ TAGS?_{tagName=tag, userName=anUserName, ctime}}^{←ctime}_{#<=limit}
8:              tagged ← {t | t ∈ ⋂_{tag∈tags} tagged_tag}^{←ctime}_{#<=limit})
9:              ∀t ∈ tagged : result_t ← BOOKMARK_{t_userName, t_ctime}      – Bulk-Anfrage
10:             result ← ⋃_{t∈tagged} result_t
11:         end if
12:     commit transaction
13:     return result
14: end function
```

Suche

QueryTags (Algorithmus 3.8) ermittelt die *limit* aktuellsten Bookmarks, optional eingeschränkt in Bezug auf eine endliche Menge von Tags oder einen bestimmten Nutzer. Die Anfrage wirkt zunächst ähnlich zu RecentChanges bei Wikis. Die Eingrenzung des Ergebnisses mittels Tags führt aber zu neuen Schwierigkeiten.

Zuerst werden mittels einer Bereichsanfrage alle passenden Bookmarks für jeden einzelnen Tag bestimmt. Danach werden programmatisch Bookmarks ausgefiltert, denen nicht alle Tags zugeordnet wurden. Das kann als eine Form von Gruppierung aufgefass werden.[23] Abschließend wird das Endergebnis gebildet, indem mittels einer Bulk-Operation den *limit* jüngsten Bookmark-Einträgen noch fehlende Attribute aus BOOKMARKS zugeordnet werden. Diese Verknüpfung entspricht einem Natural Join zwischen BOOKMARKS und TAGS. Die Durchführung von Joins in SONs ist jedoch nicht Gegenstand dieser Arbeit. Deshalb wird die Verknüpfung in Algorithmus 3.8 programmatisch durchgeführt.

Sowohl die Bereichssuche über TAGS, als auch die Join-Operation über BOOKMARKS können in einzelne, unabhängige Teilschritte zerlegt werden. Diese können jeweils parallel ausgeführt werden. Die Einhaltung der Reihenfolge beider Hauptoperationen ist jedoch notwendig, um zwischenzeitlich Bookmarks aus der Ergebnismenge auszufiltern, die nicht allen geforderten Tags genügen.

[23] Eine effiziente Realisierung dieser Funktion auch mittels traditioneller, relationaler Datenbankmanagementsysteme eine nicht triviale Aufgabe.

3.2.5 Anforderungen und Abgrenzung

Die vorgestellten Algorithmen haben ähnliche Eigenschaften:

1. Die einzelnen Algorithmen müssen atomar, dauerhaft, isoliert und korrekt ausgeführt werden, wenn die Konsistenzbedingungen ihrer Anwendung nicht verletzt werden sollen.
2. Die Algorithmen bestehen aus relativ wenigen, teilweise komplexen Operationen. Operation sind Lese- oder Schreiboperationen über
   a) einzelnen Tupeln,
   b) einer festen, vorberechneten Menge von Tupeln (Bulk-Operation),
   c) oder aber einer mittels einer Bereichsanfrage beschriebenen Menge von Tupeln

   einer einzigen Relation.
3. Das Ergebnis von Lesoperationen wird oft zusätzlich durch eine anschließende Projektion eingeschränkt.
4. Operationen können in der Reihenfolge ihrer Ausführung oft vertauscht werden.
5. Die für die Durchführung von Operationen notwendigen Unterschritte können fast immer parallel ausgeführt werden.

Diese Eigenschaften geben den Rahmen für das zu entwickelnde Transaktionsverfahren vor. Es sollte neben der linearen Abfolge von Operationen gerade auch Parallelität innerhalb einer Transaktion gut unterstützen. Die harte Einhaltung der ACID-Garantien ist aufgrund der in den vorangegangenen Abschnitten aufgestellten Konsistenzbedingungen der Beispielanwendungen zwingend erforderlich.

QueryTags, der im letzen Unterabschnitt vorgestellte Algorithmus, stellt die Grenze für in dieser Arbeit betrachtete Transaktionsverfahren auf strukturierten Overlays dar. Die Behandlung von Joins, Aggregation und Gruppierung verbleiben als Aufgaben für die Zukunft.

3.3 Verteilung und Kommunikation im Zellenmodell

In Kapitel 2 wurden ausführlich sowohl bekannte Transaktionsverfahren als auch strukturierte Overlay-Netzwerke besprochen. Die Betrachtung typischer Anwendungen für SONs im vorigen Abschnitt 3.2 hat gezeigt, dass Anwendungskonsistenz durch den Einsatz von Transaktionen gewährleistet werden kann. In den folgenden Abschnitten wird beschrieben, was nötig ist, um Transaktion in einem strukturierten Overlay-Netzwerk zu ermöglichen.

3.3.1 Tupelverteilung

Das klassische Modell für verteilte Transaktionen (Abschnitt 2.5) ordnet Ressourcen (Daten) den für sie zuständigen Ressourcemanager und jeder Transaktionen

einen Transaktionsmanager zu. Anwendungen kommunizieren direkt mit ihrem Transaktionsmanager und dieser wiederum mit den Ressourcemanagern, um eine verteilte Transaktion auszuführen. Ressourcemanager koordinieren jeweils *exklusiv* den Zugriff auf ihre lokale Datenbank [66].

Die Topologie eines strukturierten Overlay-Netzwerks bestimmt bereits die Verteilung von Daten auf Knoten. Für die Entwicklung eines Transaktionsverfahren muß der Zugriff auf diese Daten durch einen Ressourcemanager geregelt werden. Der Einsatz eines zentralisierten Ressourcemanagers für das gesamte Overlay würde aber die Vorteile eines SONs, wie Verfügbarkeit, Autonomie und Lastverteilung, durch die Einführung eines Single-Point-Of-Failure aufheben. Deshalb ist es nötig, dass der Ressourcemanager ebenfalls verteilt realisiert wird. Mögliche vorstellbare Ansätze für diese Verteilung sind:

1. Replikation des Ressourcemanagers auf mehrere Knoten,
2. Partitionierung des zentralen Ressourcemanagers in Teilkomponenten.

Der erste Ansatz erscheint nicht praktikabel. Jede transaktionale Operation würde die Interaktion zwischen dem Transaktionsmanager, dem speichernden Knoten und dem replizierten Ressourcemanager erfordern. Dies ist eine offensichtlich sehr kommunikationsintensive Vorgehensweise! Die Lastanforderungen an die Knoten eines replizierten, zentralen Ressourcenmanagers wären sehr hoch. Ferner muss bei einem Ausfall eines Ressourcemanagerknotens ein Ersatzknoten mit dem vollständigen Zustand des Ressourcemanagers versorgt werden. Der Zustand eines Ressourcemanagers ist umfangreich und die Übertragung damit von unakzeptabel langer Dauer.

Wie sollte der Ressourcemanager aufgeteilt werden? Idealerweise so, dass die Kommunikation zwischen speicherndem Knoten und Ressourcemanager vollständig entfällt bzw. nur noch lokal erfolgt. Deshalb sollte jeder Ressourcemanagerknoten nur Zustand verwalten, der eindeutig Tupeln zugeordnet werden kann, die von ihm lokal gespeichert werden. Dadurch kann das Ressourcemanagent nach dem selben Prinzip wie die Daten im Overlay verteilt werden und das Transaktionsverfahren ist hinsichtlich der Speicherung lastverteilt. Davon ausgenommen ist das atomare Festschreiben, dass die Koordination aller Ressourcemanager durch einen Transaktionsmanager erfordert. Für alle Operationen, die lediglich auf einzelnen Tupeln arbeiten, ist jedoch keine zusätzliche Kommunikation zwischen speicherndem Knoten und verantwortlichem Ressourcemanagerknoten erforderlich.

Dieser Ansatz wird im Folgenden für die Verteilung von Tupeln und Ressourcemanagement verwendet. Transaktionen bestehen aus elementaren Lese- und Schreiboperationen auf einzelnen Tupeln. Tupel und die für das Transaktionsverfahren nötigen Datenstrukturen werden gemäß des Tupelschlüssels eindeutig auf Knoten des strukturierten Overlay-Netzwerks verteilt. Transaktionsabbruch und Festschreiben sind die einzigen tupelübergreifenden Operationen, die in diesem Modell betrachtet werden. Es entspricht somit dem Seitenmodell aus Unterabschnitt 2.3.1. Phantomprobleme während der Bereichssuche bzw. anderen Bulkoperationen werden nicht behandelt.

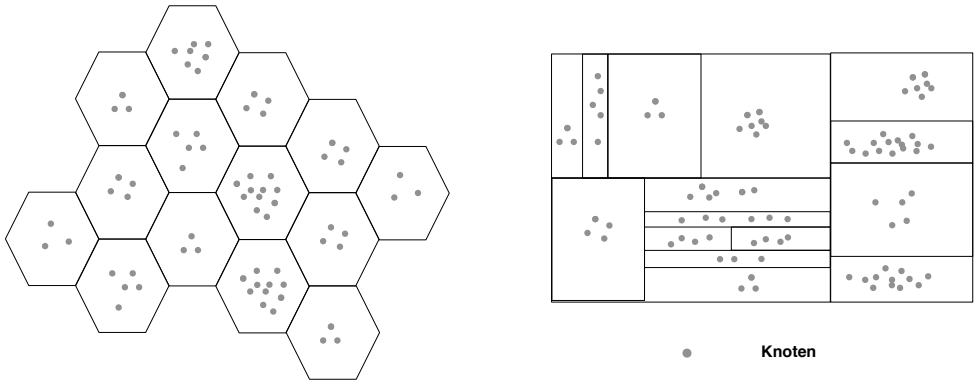

Abbildung 3.5: Schematische Darstellung des Zellenmodells: Sechseck- und Rechtecktopologien aus Zellen mit unterschiedlicher Knotenzahl

3.3.2 Zellenmodell: Routingkonsistenz durch Replikation

Vor der Auswahl eines geeigneten Transaktionsverfahrens, ist die Beschäftigung mit einem grundsätzlichen Problem strukturierter Overlay-Netzwerke nötig: Routingkonsistenz. Wie bereits in Unterabschnitt 2.2.4 beschrieben, garantiert konsistentes Routing, dass Nachrichten nur durch die jeweils zuständigen Knoten ausgeliefert werden. Für Transaktionen heißt das, dass eine Lese- und Schreiboperation auf einem einzelnen Tupel ausschließlich von dem Knoten ausgeführt wird, der für die Speicherung dieses Tupels zuständig ist. Ohne Routingkonsistenz brechen alle transaktionalen Garantien zusammen. Alte Werte werden verwendet, da neue Werte fälschlich auf einem nicht zuständigen Knoten gespeichert wurden und damit selbst durch korrekt zugestellte Lese-Nachrichten nicht gefunden werden können.

Derzeit ist kein strukturiertes Overlay-Netzwerk bekannt, dass Routingkonsistenz unter realistischen Bedingungen, wie insbesondere dem Auftreten von Knotenfluktuation (churn), gewährleistet. Bekannte Verfahren [25, 35] garantieren Routingkonsistenz nur, wenn Knoten sich stets ordnungsgemäß an- bzw. abmelden. Knotenfluktuation kann jedoch selbst in kontrollierten Umgebungen nicht verhindert werden. Im Umkehrschluss gilt, dass Knoten nicht unkontrolliert ausfallen dürfen, wenn Routingkonsistenz erforderlich ist. Doch wie kann das erreicht werden?

Replizierte Zustandsautomaten

Replizierte Zustandsautomaten (Replicated State Machines, kurz: RSM) sind eine Standardtechnik für die Replikation eines Dienstes auf mehrere Knoten mit dem primären Ziel der Ausfallsicherheit [38, 60]. Alle Knoten führen atomar identische Operationen in gleicher Reihenfolge aus. Dadurch sind stets alle Knoten im selben, replizierten Zustand. Um die als nächstes gemeinsam durchzuführende Operation zu bestimmen, wird meist eine Form von atomarem, geordnetem Broadcast (atomic bzw. total-order broadcast) eingesetzt. Diese Broadcast-Algorithmen basieren wiederum auf einem zugrunde liegenden Konsens-Algorithmus, wie zum Beispiel Paxos [39, 40, 41, 13].

Der Einsatz von replizierten Zustandsautomaten erhöht die Ausfallsicherheit. Solange eine Mehrheit der Knoten den Algorithmus korrekt ausführen und miteinander kommunizieren können, kann die RSM neue Operationen abarbeiten und wird nicht unkontrolliert ausfallen.[24] Das kann zumindest in einer kontrollierten Umgebung mit sporadischen Knotenausfällen garantiert werden.

Zellenmodell

Die Konstruktion eines strukturierten Overlay-Netzwerks aus „virtuellen“ Knoten (folgend: Replikazelle, oder kurz: Zelle), die jeweils einen replizierten Zustandsautomaten aus „echten“, physikalischen Knoten bilden, wird folgend als *Zellenmodell* bezeichnet. Das Zellenmodell ermöglicht die Maskierung von Knotenfluktuation durch die Wahl eines geeigneten Replikationsfaktors. Replikation verhindert, dass Zellen unkontrolliert ausfallen. Durch die Verwendung des Zellenmodells ist es somit prinzipiell möglich, Routingkonsistenz in einem strukturierten Overlay-Netzwerk zu garantieren.

Neben Routingkonsistenz ist Replikation auch für die Einhaltung der ACID-Garantien von Transaktionen notwendig (Unterabschnitt 3.4.3). Das Zellenmodell eliminiert die Notwendigkeit eines zusätzlichen Replikationsschemas für das Overlay-Netzwerk, wie beispielsweise symmetrische Replikation [26]. Es kann vermutet werden, dass der Einsatz von RSMs gegenüber bisher eingesetzten Techniken sogar effizienter ist, da er kein Routing innerhalb des Overlays erfordert. Stattdessen wird für die Replikation direkt das zugrunde liegende Kommunikationsnetzwerk verwendet. Die Verwendung ausfallsicherer Zellen erweist sich damit zweifach als sinnvolle Strategie.

Neben den genannten Vorteilen wirft das Zellenmodell auch neue Fragen auf. Es ist zu klären, wie physikalische Knoten auf Zellen verteilt werden, und wie die Kommunikation zwischen einzelnen Zellen abläuft.

3.3.3 Zellen, Knoten und Overlay-Topologie

Jede Zelle ist für einen bestimmtes, zusammenhängendes Teilgebiet des Schlüsselraums zuständig. Die Nachbarn einer Zelle u sind diejenigen Zellen, deren Teilgebiete, das Gebiet von u berühren. Die Gesamtheit aller Zellen deckt den Schlüsselraum vollständig ab. Darüber hinaus stellt das Zellenmodell keine Anforderungen an die eingesetzte Topologie. Beliebige Strukturen sind möglich (Abbildung 3.5).

Für die Verteilung von Knoten auf Zellen ist die Behandlung von zwei Ereignissen notwendig: Hinzufügen eines neuen Knotens und Ausfall eines vorhandenen Knotens. Im Extremfall beeinflussen diese Ereignisse die Topologie des Overlays. Wenn eine Zelle aus zu vielen Knoten besteht, ist es sinnvoll, sie aufzuteilen. Besteht sie aus zu wenigen Knoten, muss sie aufgelöst werden, um einen unkontrollierten Ausfall rechtzeitig zu verhindern. Diese dynamische Zuweisung von Knoten ermöglicht es, Knotenfluktuation im Zellenmodell zu maskieren.

[24] Dies folgt aus den eingesetzen Konsens-Algorithmen (siehe z.B. [40].)

Durch das Löschen von Zellen entstehen Lücken in der Abdeckung, die behoben werden müssen. Es ist wünschenswert, dass dieser Reparaturprozess ausschließlich lokal abläuft und maximal die Nachbarzellen der gelöschten Zelle betrifft. Andernfalls sind iterative Änderungen der Topologie notwendig. Eine lokale Änderung würde sich ausbreiten, bis sich das System wieder stabilisiert hat und vielfältige Grenzverschiebungen zwischen Zellen erfordern. Für die Transaktionsverarbeitung ist es aber wünschenswert, dass die Zuordnung von Tupeln zu Zellen während der Ausführung möglichst konstant bleibt und für die gewählte Overlay-Topologie das Entfernen und Verschmelzen von Zellen ausschließlich die Gebietsgrenzen zu ihren direkten Nachbarn beeinflusst.[25] In dieser Arbeit wird das als *Strukturstabilität* bezeichnet. Beispielsweise sind Overlay-Netzwerke mit Ringstruktur strukturstabil.

In den folgenden Abschnitten wird davon ausgegangen, dass das eingesetzte SON nach dem Zellenmodell arbeitet und eine beliebige, aber strukturstabile Topologie einsetzt. Als Nächstes wird ein mögliches Verfahren für die Verteilung von Knoten auf Zellen beschrieben. Das Ziel ist es dabei, stets die Ausfallsicherheit von Zellen zu gewährleisten.

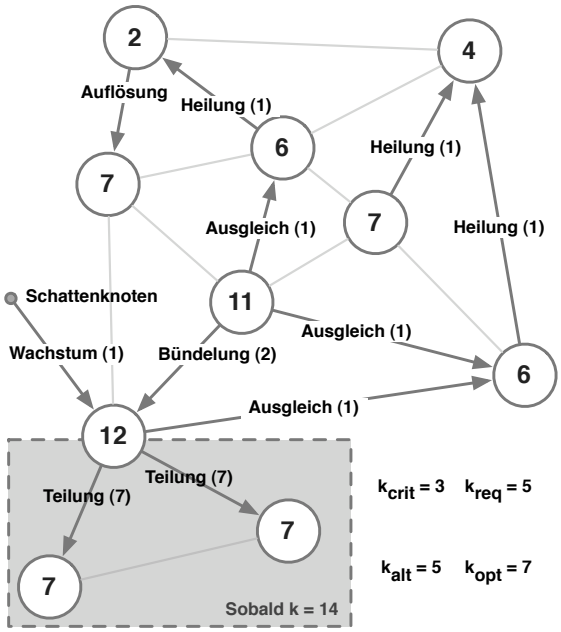

Abbildung 3.6: Beispiel für Knotenumverteilung im Zellenmodell

Knotenumverteilung im Zellenmodell

Soll ein neuer Knoten zu einer Zelle hinzugefügt werden, wählt dieser einen beliebigen Knoten als Paten aus und wird zu diesem zunächst als Schattenknoten hinzugefügt. Das bedeutet, dass ihm zwar bereits alle Nachrichten durch den Paten zugestellt werden, er jedoch noch nicht über eine vollständige Kopie des Systemzustands verfügt und noch keine Datenspeicherung und kein Routing im

[25] Genauer: Diejenigen Grenzlinien der Nachbarn, die diese mit den modifizierten Zellen geteilt haben

Overlay-Netzwerk ausführt. Der Schattenknoten wird über seinen Paten mit dem Zustand der Zelle synchronisiert. Der Pate signalisiert im Auftrag gegenüber der Zelle das Vorhandensein des Schattenknotens, das Ende des Abgleichs, als auch das Hinzufügen des Schattenknotens zur Zelle nach dem Ende des Abgleichs. Bei Ausfall des Patenknotens obliegt es dem Schattenknoten einen neuen Paten als Stellvertreter innerhalb der Zelle zu wählen und über diesen mit der Synchronisation fortzufahren.

Für jede Replikazelle bezeichne k_{crit}, die kritische, k_{req}, die erforderliche, k_{opt}, die optimale Anzahl von Knoten. Es gilt $k_{crit} \leq k_{req} < k_{opt}$. k_{alt} mit $k_{req} \leq k_{alt} < k_{opt}$ ist die Altruismusgrenze. Im Folgenden seien diese Parameter für das System konstant, eine Erweiterung um unterschiedliche Parameter für einzelne Zellen ist jedoch denkbar. Ferner kennt jede Zelle ihre direkten Nachbarzellen in der Topologie des Overlay-Netzwerks und die Anzahl der sie konstituierenden Knoten (Knotenzahl $k(u)$ der Zelle u). Die Überschusszahl $k^+(u)$ wird als $k(u) - k_{opt}$ definiert. Schattenknoten werden bei der Berechnung der Knotenzahl nicht berücksichtigt.

In regelmäßigen Abständen wird durch jede Zelle u folgende Menge von Umverteilungsregeln *nacheinander* abgearbeitet (als eine atomare Operation der RSM):

Auflösung Falls die Knotenzahl dieser Zelle unter k_{crit} sinkt, wählt sie eine Nachbarzelle mit maximaler Knotenzahl aus, und beginnt damit, ihren Zustand an diese zu übertragen. Sobald die Übertragung abgeschlossen ist, löst sich u selbst auf, in dem sie mit dem Nachbarknoten verschmilzt. Falls die eigene Knotenzahl zuvor jedoch wieder k_{crit} erreicht, wird dieser Vorgang abgebrochen.

Heilung Falls einzelne Nachbarzellen weniger als k_{req} Knoten besitzen und diese Zelle mehr als k_{alt} Knoten besitzt, werden Schattenknoten und Knoten an Nachbarknoten abgegeben, um dieses Manko zu beheben. Zellen mit kleinerer Knotenanzahl werden dabei bevorzugt. Die eigene Knotenzahl darf dabei jedoch nicht unter k_{alt} sinken. Knoten, die das Ziel einer Zellauflösung sind, wenden diese Regel nicht an.

Ausgleich Falls einzelne Nachbarzellen weniger als k_{opt} Unterknoten besitzen und diese Zelle mehr als k_{opt} Unterknoten besitzt, werden Schattenknoten und Unterknoten gleichmässig an Nachbarknoten abgegeben, um dieses Manko zu beheben. Die eigene Knotenzahl darf dabei jedoch nicht unter k_{opt} sinken.

Bündelung Falls die eigene Überschusszahl positiv ist und es Nachbarzellen gibt, deren Überschusszahl die eigene übersteigt, werden eigene Überschussknoten an diejenige Nachbarzelle abgegeben, deren Überschusszahl maximal ist.

Aufteilung Falls die eigene Knotenzahl $\geq 2k_{opt}$ ist, wird die Zelle in $\lfloor \frac{k(u)}{k_{opt}} \rfloor$ unabhängige Replikazellen aufgeteilt.

Wachstum Vorhandene Schattenknoten, die über eine vollständige Kopie des Zustands verfügen, werden durch eine Folgeoperation ihres Paten aktiviert.

Es ist notwendig, dass die Auswahl abgegebener Knoten und ihrer jeweiligen neuen Zellen deterministisch ist. Dies folgt bereits aus den Anforderungen

des zugrunde liegenden RSM-Ansatzes, da andernfalls der Zustand einzelner Knoten divergieren könnte. Außerdem darf ein Knoten nicht mehrfach der selben Zellen hinzugefügt werden, da sonst die Ausfallsicherheit dieser Zelle negativ beeinflusst wird. Eine solche Situtation kann eintreten, falls scheinbar verschiedene Knoten auf dem selben Computer ausgeführt werden oder falls das Overlay vorsieht, dass ein Knoten Bestandteil von mehr als einer Zelle sein kann.

Das beschriebene Verfahren für Knotenumverteilung im Zellenmodell zielt darauf ab, die verfügbaren Knoten gleichmäßig so auf Zellen zu verteilen, dass jede Zelle irgendwann aus genau k_{opt} Knoten besteht. Das Auslösen der Umverteilung könnte randomisiert alle t Sekunden erfolgen. Zusätzlich könnte ein Knoten, dessen Knotenzahl k unter k_{req} sinkt, Notfallumverteilungen bei seinen Nachbarn auslösen. Abbildung 3.6 enthält ein Beispiel. Zellen gelten als benachbart, wenn sie durch eine Kante verbunden sind. Für jede Zelle wird die Anzahl ihrer Knoten und die bei der nächsten Umverteilung angewendeten Regeln dargestellt (ausgehende Kanten). Dabei wird in Klammern angegeben, wie viele Knoten jeweils umverteilt werden.

Das Verfahren berücksichtigt nicht die Anzahl der Nachbarn. Es sind jedoch Overlay-Topologien und Parameter k_{req}, k_{alt}, k_{opt} vorstellbar, bei denen das gleichzeitige Auslösen der Heilungsregel durch alle Nachbarn einer Zelle zu viele neue Schattenknoten hinzufügen würde. Dies wäre möglich, falls für alle Nachbarn $n \in N(u)$ einer Zelle u mit $k(u) < k_{req}$ gelten würde, dass $\Sigma_{n \in N(u)} \min(\max(k(n) - k_{alt}, 0), k_{req} - k(u)) \geq 2k_{opt} - k(u)$ wäre.[26]

Eine solche Situation hätte nicht wünschenswerte, abrupte Veränderungen der Zellenstruktur zur Folge. Die Knotenanzahl einer unterversorgten Zelle würde plötzlich anschwellen. Im nächsten Schritt würde diese in mehrere Zellen aufgeteilt („Bombardement“). Eine mögliche Lösung besteht darin, dass entweder die geheilte Zelle in einem Schritt nur eine begrenzte Anzahl von Knoten annimmt oder aber, dass die heilenden Zellen nur eine begrenze Anzahl von Knoten abgeben.

Gebietsaufteilung

Das Verfahren für Knotenumverteilung im Zellenmodel behandelt die Zuordnung von Zellen zu Teilgebieten des Schlüsselraums nur insofern, als dies durch Verschmelzung und Aufteilung erforderlich ist.

Für ein strukturiertes Overlay ist es zusätzlich wichtig, verschiedenen Zellen unterschiedlich große Bereiche des Schlüsselraumes zuzuordnen, um Lastverteilung bezüglich der gespeicherten Datenmenge und der Anzahl der eingehenden Anfragen durchzuführen. Beim Aufteilen von Zellen kann dies berücksichtigt werden, in dem die Gebiete der neu entstehenden Zellen so gewählt werden, dass sie ungefähr gleich viele Tupel speichern.

Nimmt man an, dass eine Zelle Informationen über die Anzahl der von ihren Nachbarn gespeicherten Tupel verfügt, beispielsweise in Form eines Gradienten der Tupelverteilung in der Nachbarzelle orthogonal zur Richtung der gemeinsamen Gebietsgrenze, dann könnte sie Grenzgebiet der Nachbarzelle

[26] Ohne Berücksichtigung eventuell vorhandener, zusätzlich umverteilter Schattenknoten

übernehmen, falls von ihr zu wenige, bzw. eigenes Grenzgebiet abgeben, falls von ihr zu viele Tupel gespeichert werden.

3.3.4 Kommunikation im Zellenmodell

Folgend werden die Kommunikationserfordernisse unterschiedlicher Operationen in strukturierten Overlay-Netzwerken nach dem Zellenmodell analysiert, bevor dann integrativ im nächsten Abschnitt ein geeignetes Transaktionsverfahren für die Konsistenzerhaltung sowohl auf der Ebene der Routingstruktur als auch für die in Abschnitt 3.2 beschriebenen Anwendungen gewählt und angepasst wird.

Routing und Transaktionen

Der Einsatz von Zellen aus replizierten Zustandsautomaten hat seinen Preis. Selbst moderne Konsensalgorithmen wie Fast-Paxos erfordern im nicht-byzantinischen, fehlerfreien Fall $N(\lfloor 2N/3 \rfloor + 1)$ Unicast-Nachrichten und mindestens 3 Kommunikationsrunden [27] [41]. Dafür kann Routingkonsistenz und Replikation durch einen Mechanismus ermöglicht werden. Der Aufwand ist somit vertretbar.

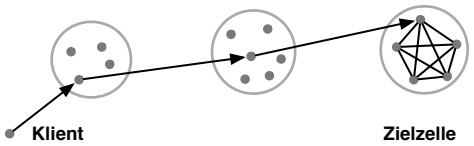

Abbildung 3.7: Routing im Zellenmodell

Die Durchführung einer RSM-Operation für jeden Routingschritt ist aus Performancegründen nicht akzeptabel. Da jedoch alle Unterknoten einer Zelle über identische Routingtabellen verfügen, kann die Weiterleitung als einfache Operation, das heißt mit der selben Geschwindigkeit des Routings unreplizierter, strukturierter Overlays erfolgen. Das entspricht aus transaktionaler Sicht der Verwendung schmutziger Leseoperationen (Unterabschnitt 2.3.1) für das Routing. Lediglich das Zustellen von Nachrichten muss als replizierte Operation der Zelle erfolgen (Abbildung 3.7). Falls dabei atomar festgestellt wird, dass die Zelle für die Nachricht nicht zuständig ist, wird die Weiterleitung fortgesetzt. Eine derartige Situation tritt ein, falls der empfangende Zellknoten bei der versuchten Auslieferung durch Netzwerkfehler noch nicht über der aktuellen Zustand der Zelle verfügt. Der Ansatz garantiert, dass eine Nachricht stets an die zuständige Zelle ausgeliefert wird. Es ist jedoch möglich, dass Nachrichten nicht zugestellt werden, falls ein Knoten während der Weiterleitung ausfällt.

Änderungen an der Routingtabelle, die mehrere Knoten involvieren, erfordern die Verwendung von Transaktionen. Das Transaktionsverfahren wird zweifach, das heißt sowohl für die Konsistenz des Routings als auch für die Ausführung von Anwendungstransaktionen, eingesetzt. Die Anforderungen sind jedoch jeweils verschieden. Transaktionen für Routingänderungen bestehen in der Regel ausschließlich aus wenigen, parallel ausführbaren Schreiboperationen. Die

[27] Halbe Roundtrips

Schreibmenge ist zu Beginn der Transaktion vollständig bekannt. Das wird bei der Wahl des Transaktionsverfahrens im nächsten Abschnitt berücksichtigt.

Zuverlässiges Senden

Neben regulärer Nachrichtenweiterleitung (Routing) und Transaktionen ist die Existenz eines zusätzlichen Kommunikationsmechanismus zwischen zwei Zellen wünschenswert, der erfolgreiche Kommunikation garantiert, jedoch nicht den vollen Aufwand einer Transaktion erfordert. Ein derartiger Mechanismus wird folgend als zuverlässiges Senden bezeichnet. Zuverlässiges Senden könnte beispielsweise für den Austausch von Metadaten zwischen Nachbarzellen, wie etwa die Anzahl der gespeicherten Datenpunkte, verwendet werden.

Zuverlässiges Senden beginnt mit dem (impliziten) Entstehen eines Kommunikationsbedürfnisses als Bestandteil der Ausführung einer replizierten Operation. Durch ein vorvereinbartes, deterministisches Schema ist festgelegt, welcher Unterknoten die Nachricht sendet (Bote).

Der Bote versendet die Nachricht unter Verwendung eines Kontaktknotens der Zielzelle, analog zum regulären Routing. Die erfolgreiche Zustellung der Nachricht bewirkt die Ausführung einer replizierten Operation der empfangenden Zelle. Der Bote wird darüber durch seinen Kontaktknoten informiert und löst seinerseits das Ausführen einer replizierten Bestätigungsoperation der sendenden Zelle aus. Kommt es zu einem Ausfall des Kontaktknotens, wird durch den Boten ein neuer Kontaktknoten gewählt. Fällt der Bote aus, wird durch das vorab festgelegte Auswahlschema automatisch ein neuer Bote bestimmt. Zusätzlich kann ein Bote auch signalisieren, dass er aufgrund von Netzwerkproblemen nicht in der Lage ist, die Nachricht zuzustellen. Dann wird ebenfalls ein neuer Bote bestimmt.

Die Verwendung des Pull-Prinzips für den Erhalt der Bestätigung begünstigt den Umgang mit Zuständigkeitsänderungen durch Zellverschmelzung und -teilung. Letztlich ist ausschließlich der Bote für die Ermittlung der Bestätigung zuständig. Dies kann mehrfaches Routen zum Ziel erfordern.

Zusammenfassung: Zustand und Kommunikation

Zusammenfassend kann der Zustand eines Unterknotens zweigeteilt werden: Replizierter Zustand der Zelle (Daten, Sperren, Routingtabellen) und flüchtiger Knotenzustand (Nachrichtenweiterleitung, lokale Zeit, Informationen über das lokale Netzwerk). Für die Kommunikation stehen verschiedene Primitiven zur Verfügung, aufsteigend geordnet nach Kommunikationsaufwand und Zuverlässigkeit:

1. Direkte Kommunikation zwischen einzelnen Knoten:
   Senden, ggf. Antwort empfangen,
2. Unzuverlässiges Senden von einem Knoten an eine Zelle mit atomarer Zustellung:
   Routing, Repliziertes Zustellen, opt. Antwortnachricht (Ausfall der Antwort führt zu erneutem Senden und erfordert ggf. Duplikatfilterung)

3. Zuverlässiges Senden zwischen Zellen mit atomarer Zustellung: Replizierte Sendeoperation, Routing, Replizierte Zustelloperation, Bestätigungsnachricht, Replizierte Bestätigungsoperation
4. Senden im Rahmen einer Transaktionen (Nächster Abschnitt)

3.4 Transaktionen im Zellenmodell

Das primäre Kriterium für den Entwurf eines Transaktionsverfahrens für strukturierte Overlay-Netzwerke nach dem im vorigen Abschnitt entwickelten Zellenmodell, ist die Minimierung des erforderlichen Kommunikationsaufwands im Sinne von Latenz und damit der Anzahl der erforderlichen Kommunikationsschritte. Latenzminimierung ist notwendig, da das Zellenmodell durch den Einsatz von Zustandsreplikation bereits gegenüber nicht-replizierten Overlays einen wesentlich höheren Kommunikationsaufwand erfordert und Latenz letztlich die Gesamtdauer der Transaktion maßgeblich bestimmt. Gleichzeitig sollte geringere Latenz möglichst nicht durch den Einsatz vieler zusätzlicher Nachrichten (N * N) erkauft werden, da sonst eine Überlastung der verfügbaren Netzwerkkapazität letztlich zu einer längeren Kommunikationsdauer führen kann.

3.4.1 Wahl eines Nebenläufigkeitskontrollverfahrens

Der Einsatz traditioneller, pessimistischer Verfahren (Unterabschnitt 2.4.1) ist weitgehend ungeeignet, um die Latenz zu minimieren, da das Ausführen jeder Basisoperation als replizierte Operation der betroffenen Zelle erfolgen muss. Ausgenommen hiervon sind Transaktionen, die nur aus einer Basisoperation bestehen. Derartige Transaktionen können direkt atomar durch den zuständigen Ressourcemanager bzw. die zuständige Zelle ausgeführt werden. Dieser Sonderfall wird folgend ausgeklammert, es könnte aber sinnvoll sein, die atomare Ausführung einzelner Basisoperationen auf einem Datum zusätzlich anzubieten.

Die Notwendigkeit des Einsatzes replizierter Operationen für jede Basisoperation gilt auch für Lese-Operationen in Multiversionierungsverfahren. Das Lesen einer Objektversion durch eine Transaktion mit dem Zeitstempel t erfordert das Speichern des maximalen Lesezeitpunktes dieser Objektversion, falls dieser vor t liegt (das ist wahrscheinlich). Lesen ist somit eine potentiell zustandsverändernde Operation und muss deshalb repliziert ausgeführt werden. Von Vorteil ist hingegen, dass Nur-Lese-Transaktionen in pessimistischen Multiversionierungsverfahren kein Festschreiben erfordern.

In allen nicht-starken Zwei-Phasen-Sperrprotokollen können Verklemmungen auftreten. Die Implementation eines Algorithmus für die verteilte Erkennung von Verklemmungen im Zellenmodell würde eine hohe Anzahl replizierter Operationen und zusätzliche Interzellenkommunikation erfordern. Starken Zwei-Phasen-Sperrprotokolle (SS2PL) benötigen ein vollständiges Vorabwissen über die Lese- und Schreibmenge von Transaktionen. Das ist eine große Einschränkung für Anwendungen, die erst aufgrund von innerhalb der Transaktion ausgeführten Leseoperationen Folgezugriffe bestimmen können. Derartige Transaktionen sind jedoch recht häufig. Ein typisches Beispiel hierfür sind einfache Lesezugriffe unter Verwendung von Indexrelationen.

Die Verwendung eines optimistischen Nebenläufigkeitskontrollverfahrens (Unterabschnitt 2.4.4) ermöglicht hingegen die Minimierung der Anzahl auszuführender, replizierter Operationen. Für jede beteiligte Zelle wird zufällig ein Stellvertreterknoten für die Transaktionsausführung gewählt. Über diese Stellvertreter wird dann unter Verwendung einfacher, direkter Nachrichten des zugrunde liegenden Kommunikationsmediums die Transaktion optimistisch ausgeführt. Erst während der Validierungsphase werden die Daten von den Stellvertreterknoten auf die relevanten Zelle repliziert. Dadurch ist der Ansatz unempfindlich gegenüber topologischen Veränderungen in der Zellenstruktur.

Einfache optimistische Nebenläufigkeitskontrolle hat auch Nachteile. Die Validierung einer Transaktion kann fehlschlagen und zum Abbruch der Transaktion führen. Dadurch werden insbesondere Transaktionen mit langer Ausführungszeit bestraft.

Hybride, optimistische Nebenläufigkeitskontrolle [64] behandelt das Problem unter der Annahme von Zugriffsinvarianz. Wie bereits in Unterabschnitt 2.4.5 beschrieben, werden während der ersten Validierungsphase alle benötigten Sperren erworben. Falls die Validierung fehl schlägt, wird die Transaktion erneut gestartet ohne die Sperren aufzugeben. Bei der erneuten Ausführung arbeitet die Transaktion somit exklusiv auf den Daten und wird erst im letzten Schritt erfolgreich validiert. Lediglich die Validierung erfordert den Einsatz einer replizierten Operation pro beteiligtem Ressourcemanager bzw. pro beteiligter Zelle. Damit erweist sich der Einsatz eines hybriden, optimistischen Verfahrens als ideal für die Nebenläufigkeitskontrolle eines verteilten Datenspeichers, der auf einem strukturierten Overlay-Netzwerk nach dem Zellenmodell basiert.

3.4.2 Absturzbehandlung

Ein interessanter Aspekt des Zellenmodells ist, dass Zellen zwar aufgelöst werden können, jedoch zuvor eine andere Zelle die Verwaltung der von ihr gespeicherten Daten übernimmt. Da vom Fail-Noisy-Systemmodell (Unterabschnitt 2.2.2) ausgegangen wird, werden einmal ausgefallene Knoten auch nicht wiederhergestellt. Die Implementation von Absturzbehandlung ist daher nicht erforderlich. Jedes gespeicherte Datum wird zu jedem Zeitpunkt durch eine Zelle aus korrekten Knoten gespeichert. Die dafür notwendige Ausfallsicherheit ist Aufgabe der verwendeten Algorithmen für Knotenverteilung und Zustandsreplikation von Zellen.

Aufgrund der dynamischen Änderung der Gebietsaufteilung zwischen Zellen ist es wünschenswert, für die Reintegration von Knoten ehemaliger, nicht länger existenter Zellen eine Nachfolgezelle so zu wählen, dass die Menge des neu zu übertragenden Zustands minimiert wird. Eine zukünftige Erweiterung um Unterstützung für Crash-Recovery könnte die Zeit bis zur vollständigen Reaktivierung eines kurzzeitig ausgefallenen Knoten verkürzen.

Darüber hinaus entsteht die Frage, wie mit dem ausgefallenen Datenbereich vollständig ausgefallener Zellen bis zur Wiederherstellung umgegangen werden sollte. Die einzige korrekte Antwort verlangt, dass der Datenbereich für die Verwendung bis zur Wiederherstellung vollständig gesperrt wird. Dies erscheint kaum praktikabel, weshalb es sinnvoll scheint, Ausfallsicherheit und

Absturzbehandlung ausschließlich durch den Einsatz von Zellen aus replizierten Zustandsautomaten zu garantieren.

3.4.3 Verteiltes, atomares Festschreiben

Verteiltes, atomares Festschreiben erfordert das zentrale Halten des Terminierungszustands der Transaktion (festgeschrieben oder abgebrochen). Dieser darf nur atomar geändert werden. In jedem Fall muss garantiert werden, dass unterschiedliche Ressourcemanager (oder gar unterschiedliche Replikate) sich einheitlich in Bezug auf das Festschreiben der Transaktion verhalten. Die Nichterfüllung dieser Forderung bewirkt die Vernichtung *sämtlicher* ACID-Garantien: Atomizität (Nichtgleichzeitiges Festschreiben), Isolation (spätere Transaktionen lesen ggf. alte Daten), Konsistenz (folgt aus fehlender Atomizität) und Dauerhaftigkeit (Teile der Schreibmenge werden nicht festgeschrieben).

Es gibt verschiedene Möglichkeiten, um einheitliches, verteiltes, atomares Festschreiben zu realisieren: Einsatz eines nicht-blockierenden Verfahrens, garantierte Ausfallsicherheit des Transaktionsmanagers und die Verwendung eines ausfallsicher gespeicherten Commit-Records. Die Benutzung des Zellenmodells ermöglicht bereits die ausfallsichere Speicherung des Transaktionszustands. Deshalb scheint auch hier die Verwendung eines hybriden Ansatzes sinnvoll.

Zwei-Phasen-Festschreiben mit ausfallsicherem Transaktionsmanager

Der Transaktionsklient generiert zu Beginn der Transaktion eine eindeutige Kennung, die Transaktions-Id und wählt einen beliebigen Unterknoten derjenigen Zelle aus, deren Teilgebiet für die Speicherung dieser Transaktions-Id zuständig ist.[28] Dieser Knoten wird der Transaktionsmanager. Alle durchzuführenden Operationen werden vom Klienten an den Transaktionsmanager gesendet und durch diesen ausgeführt. Beim Start der Transaktion wird durch den Transaktionsmanager ein Commit-Record (Abbildung 2.8) für die Transaktionskennung in der Zelle des Transaktionsmanagers erzeugt. Dieser kann benutzt werden, um bei einem Ausfall des Transaktionsmanagers den Terminierungszustand der Transaktion zu ermitteln oder diese abzubrechen. Der Transaktionsmanager führt nun die Transaktion unter Verwendung von 2PC aus. Nach der Bestätigung der Prepare-Nachricht durch alle beteiligten Ressourcemanager wird der Commit-Record geschrieben und erst danach die Festschreib-Nachricht gesendet. Ressourcemanager starten nach dem Senden der Prepare-Bestätigung einen Timer. Nachdem dieser abgelaufen ist, löst der Ressourcemanager einen Transaktionsabbruch durch Schreiben des Commit-Records aus. Falls dieser bereits festgeschrieben ist, wird dies allen Ressourcemanagern durch zustellenden Knoten mitgeteilt (d.h. derjenige Knoten, der die Anfrage des Ressourcemanagers als replizierte Operation zugestellt hat). Der Commit-Record ermöglicht somit die Wiederherstellung des Transaktionsmanagers nach einem Ausfall.

Der Vorteil des obigen Verfahrens besteht darin, dass es weitestgehend auf Zustellgarantien für Nachrichten verzichtet. Lediglich die verlässliche Modifikation des Commit-Records ist erforderlich. Dies wird bereits durch

[28] Alternativ könnte auch die Transaktions-Id so gewählt werden, dass sie in der ersten benutzten Zelle gespeichert wird

die Verwendung von replizierten Zellen garantiert. Der Einsatz eines nicht-blockierenden Verfahrens würde hingegen die verlässliche Zustellung der gesendeten Nachrichten mittels replizierter Operationen erfordern und damit einen nicht unerheblichen Mehraufwand verursachen.

Validierung

Da ein hybrides optimistisches Nebenläufigkeitskontrollverfahren vorgeschlagen wird, ist zusätzlich die globale Serialisierung von Validierungsphasen mit überlappenden Lese- und Schreibmengen notwendig. Ein entsprechendes Verfahren wurde bereits in Unterabschnitt 2.5.1 beschrieben.

Ressourcemanager und Zellen

Bisher wurde stillschweigenden davon ausgegangen, dass während der Abarbeitung einer Transaktion die Gebietsverteilung von Zellen und damit der Zuständigkeitsbereich von Ressourcemanagern konstant ist. Dies ist jedoch nicht der Fall.

Bei normalen Anfragen werden Zuständigkeitsänderungen durch die reguläre Nachrichtenweiterleitung (routing) behandelt. Für atomares Festschreiben muss gewährleistet sein, dass das Festschreibprotokoll alle von der Transaktion benutzten Daten umfasst. Dafür ist die ausschließliche Verwendung von Datenschlüsseln bei der Adressierung von Nachrichten des Festschreibprotokolls erforderlich. Aus Gründen der Effizienz ist es sinnvoll, dass Schlüsselbereiche in einer Bulknachricht zusammengefasst und durch den Einsatz eines entsprechenden Broadcastprotokolls im Overlay-Netwzerk verteilt werden. Betroffene Zellen generieren dann jeweils eine Antwort, die dass Teilgebiet (und damit den Gültigkeitsbereich) der antwortenden Zelle beinhaltet. Erst wenn der Transaktionsmanager Antworten für alle benutzten Daten bekommen hat, kann die nächste Phase des Festschreibprotokolls gestartet werden.

Zusätzlich ist es erforderlich, dass Veränderungen in der Gebietsstruktur von Zellen als atomare Zuständigkeitsänderungen bezüglich Daten- und Ressourcemanagement realisiert werden. Um dies effizient zu gestalten, könnte zunächst, ausserhalb einer Transaktion, der relevante Zustand zwischen den Zellen ausgetauscht werden. Zellen würden temporär replizierten Zustand speichern, für den sie nicht zuständig sind. In einem zweiten Schritt könnte dann innerhalb einer Transaktion atomar die Zuständigkeit gewechselt werden.

3.4.4 Optimierte Transaktionstypen

Nur-Lese-Transaktionen

Viele Anwendungsoperationen - und damit Transaktionen - bestehen lediglich aus Lese-Operationen. Derartige Nur-Lese-Transaktionen können gegenüber schreibenden Transaktionen optimiert ausgeführt werden, da sie kein aufwändiges atomares Festschreiben erfordern. Es wäre daher wünschenswert, das gewählte hybride optimistische Verfahren um einen Optimierung für Nur-Lese-

Transaktionen zu erweitern. Dazu wird folgend, analog zu ROMV (Unterabschnitt 2.4.5), eine Ergänzung von HOCC um Multiversionierung vorgeschlagen.

Für jedes Datenobjekt existieren mehrere Versionen, die jeweils mit dem Zeitstempel ihrer Erstellung versehen sind. Zusätzlich existiert stets eine Kopie der jüngsten, gültigen Version, die aktuelle Version. Lese- und Schreibsperre beziehen sich ausschließlich auf die aktuelle Version. Die aktuelle Version dient der Entkopplung von Multiversionierung und hybrid-optimistischer Nebenläufigkeitskontrolle. Die Idee des Verfahrens ist es, neu erzeugte Versionen stets so in die Zukunft zu verlegen, dass sie die Korrektheit vergangener Nur-Lese-Transaktionen nicht berühren.

Jeder Nur-Lese-Transaktionen wird ihr Startzeitpunkt zugeordnet. Das Lesen durch eine Nur-Lese-Transaktion wird *pessimistisch*, als replizierte Operation ausgeführt. Der Leseoperation wird die älteste, vor dem Start ihrer Transaktion erstellte, Version zugeordnet. Falls diese Version die aktuelle Version ist, wird der Zeitstempel der aktuellen Version auf den Startzeitpunkt der lesenden Transaktion gesetzt. Lesende Transaktionen erfordern kein atomares Festschreiben, da die von ihnen gelesenen Versionen stets unveränderlich sind. Leseoperationen, denen eine validierte, noch nicht festgeschriebene Version einer Transaktion zugeordnet wird, werden jedoch bis zum Ende dieser Transaktion durch Festschreiben oder Abbrechen verzögert. Dies ist nötig, da nach dem Beginn der Validierung der Zeitstempel aktueller Versionen nicht mehr geändert werden kann.

Schreibende Transaktionen arbeiten nach dem Verfahren der hybrid-optimistischen Nebenläufigkeitskontrolle unter ausschließlicher Verwendung der aktuellen Versionen. Jeder schreibenden Transaktion wird zu Beginn des atomaren Festschreibens ein Zeitstempel zugeordnet. Dieser muss größer sein, als der größte Zeitstempel einer aktuellen Version, die von der Transaktion benutzt (gelesen oder geschrieben) wurde. Beim Festschreiben wird allen benutzten, aktuellen Versionen dieser Zeitstempel zugeordnet. Danach wird eine Kopie aller modifizierten, aktuellen Versionen angelegt. Diese Kopie ist die neue, gültige Version, die zukünftig von Nur-Lese-Transaktionen gelesen wird.

Parallelität und Schachtelung

In Unterabschnitt 3.2.5 wurde für die Umsetzung von Anwendunsoperationen die parallele Ausführung von Untertransaktionen gefordert. Dafür muss zunächst unterschieden werden, auf welcher Ebene diese Parallelität benötigt wird. Parallelität auf der Ebene unabhängiger Teilschritte komplexer Operationen (Schreiben einzelner Tupel als Bestandteil einer Bulkoperation) stellt keine zusätzlichen Bedingungen an das Transaktionsverfahren. Es muss lediglich gewartet werden, bis alle parallelen Teilschritte nach einem Multicast im SON ausgeführt wurden, damit die Serialisierbarkeit nicht verletzt wird. Das genügt für die Mehrzahl der in Abschnitt 3.2 besprochenen Anwendungen.

Die parallele Ausführung mehrerer Operationen erfordert hingegen die Unterstützung von *Transaktionsschachtelung* (nested transactions). Für die Isolation der Untertransaktionen ist es erforderlich, dass sie jeweils optimistisch auf einer unabhängigen Kopie der Daten arbeiten und sich dadurch nicht gegenseitig

beeinflussen. Das kann lokal mittels Multiversionierung (Unterabschnitt 2.4.3) implementiert werden [54]. Die Validierungsphase beginnt erst, wenn alle parallelen Untertransaktionen abgeschlossen wurden. Dann werden die benötigten Sperren für alle Untertransaktionen erworben und zusätzlich überprüft, ob Konflikte zwischen parallelen Untertransaktionen aufgetreten[29] sind. Ist dies der Fall, wird die Transaktion abgebrochen. Andernfalls wird das Protokoll regulär fortgesetzt.

Pessimistische Schreibtransaktionen

Schreibtransaktionen, deren Lese- und Schreibmenge zu Beginn der Transaktion bereits bekannt sind, können statt der optimistischen Nebenläufigkeitskontrolle auch direkt das strenge Zwei-Phasen-Sperrprotokoll einsetzen. Dies verhindert das erneute Ausführen nach fehlschlagender Validierung, erfordert aber dafür die Verwendung replizierter Operationen für jeden Transaktionsschritt. Gerade Transaktionen, die aus einfachen, ausschließlich parallel ablaufenden Basisoperationen bestehen, könnten jedoch davon profitieren. Das gleichzeitige Ändern von Routingeinträgen in mehreren Zellen ist ein typisches Beispiel für eine derartige Transaktion.

3.4.5 Zusammenfassung

Das vorgestellte Transaktionsverfahren bietet somit nicht nur *eine* Möglichkeit des atomaren Zugriffs, sondern vier:

Atomare Einzeloperation werden als replizierte Operation auf einem Datum einer Zelle realisiert, die kein zusätzliches, verteiltes atomares Festschreiben erfordern.

Pessimistische, verklemmungsfreie Schreibtransaktionen basierend auf der Verwendung eines strengen Zwei-Phasen-Sperrprotokolls. Sie sind beispielsweise geeignet für parallele Änderungen an den Routingtabellen mehrerer Zellen, da in diesem Fall Sperrphase, Operationen, und Entsperrphase in die Prepare-Nachricht des verteilten, atomaren Festschreibens gebündelt werden können.

Optimistische, verklemmungsfreie Schreibtransaktionen unterstützen schreibende, ggf. lange dauernde Anwendungstransaktionen. Erfolgreiche Transaktionsausführung ist ab der ersten Wiederholung garantiert, falls Zugriffsinvarianz gegeben ist.

Pessimistische, Nur-Lese-Transaktionen erfordern kein verteiltes, atomares Festschreiben. Sie sind daher besonders geeignet, wenn Datentupel von vielen unterschiedlichen Zellen gelesen werden sollen.

Die Ausfallsicherheit der von Zellen gespeicherten Informationen wird benutzt, um die Wiederherstellbarkeit des Transaktionsmanagers zu gewährleisten. Für die Durchführung von verteilten, atomaren Festschreiben genügt deshalb der Einsatz von Zwei-Phasen-Festschreiben, erweitert um einen Timeout-basierten Mechanismus für die Reaktivierung des Transaktionsmanagers.

[29] Ganz analog zu State Transactional Memory [32]

4 Diskussion

Als Ergebnis dieser Arbeit wurde im vorigen Kapitel schrittweise ein Transaktionsverfahren für strukturierte Overlay-Netzwerke nach dem Zellenmodell entwickelt, dass Ausfallsicherheit durch den Einsatz replizierter Zustandsmaschinen gewährleistet.

Etna [47] ist ein Verfahren für die Implementation atomarer Objekte in einem Overlay mit Ringstruktur durch den Einsatz quorumbasierter Replikation. Dabei wird Paxos [39] als zugrundeliegender Konsensalgorithmus verwendet. Konfigurationen (beteiligte Replikaknoten) werden direkt im Overlay gespeichert. Alle Operationen werden durch eine Primärkopie ausgeführt und damit Möglichkeiten der Lastverteilung und Latenzminimierung verschenkt. Es wird nicht beschrieben, wie der atomare Speicher um Transaktionen erweitert werden kann. MESAROS ET. AL [44] beschreiben ein Transaktionsverfahren für strukturierte Overlays. Das Verfahren basiert auf einem einfachen Zweiphasensperrprotokoll und benutzt Transaktionsprioritäten für den Umgang mit Verklemmungen. Die Arbeit beschränkt sich auf die unrealistischen Annahmen, dass Kommunikation zwischen Knoten zuverlässig und geordnet ist und dass kein Replikationsmechanismus benutzt wird.

Das vorgestellte Zellenmodell behandelt beides, in dem es die Gewährleistung von Replikation und Ausfallsicherheit an die das Overlay bildenden Zellen delegiert: Jede Zelle ist eine replizierte Zustandsmaschine aus physikalischen Unterknoten. Dies ermöglicht das atomare, ausfallsichere Lesen- und Schreiben des Zustands der Zelle. Für Transaktionsverfahren ist der Einsatz von atomarer Replikation aus drei Gründen notwendig:

Routingkonsistenz des Overlays Replizierte, ausfallsichere Zellen maskieren Knotenfluktuation (churn) und ermöglichen somit Routingkonsistenz.

Einhaltung der ACID-Garantien Die ACID-Bedingungen erfordern das atomare Schreiben einer Mehrheit von Replikas. Zellen bilden dieses Erfordernis konzeptuell im Rahmen strukturierter Overlay-Netzwerke ab.

Verteiltes, atomares Festschreiben erfordert das atomare, ausfallsichere Speichern des Terminierungszustands der Transaktion. Der Einsatz von Replikazellen ermöglicht dies.

Statt der Verwendung von Zellen ist auch Replikation auf der Ebene von Knoten eines klassischen Overlay-Netzwerks möglich [26]. Verteiltes, atomares Festschreiben könnte dann durch den Einsatz nicht-blockierender Protokolle, wie Paxos- oder Dreiphasenfestschreiben, umgesetzt werden. Es bleibt jedoch offen, wie in einem solchen Modell Routingkonsistenz erreicht werden kann. Selbst wenn man dies vernachlässigt, bleibt als Nachteil, dass alle Nachrichten über das Overlay-Netzwerk zu den einzelnen Replikaten geroutet werden müssen.

Tabelle 4.1: Vergleich des Transaktionsverfahren

Transaktionstyp	**Einmalig für N beteiligte Zellen**	**Parallele Operation auf N Zellen**	**Gesamt für k serielle Operationen**
Atomares Schreiben	NL	NR	$1L + 1R$, da stets $k = 1, N = 1$
Nur-Lese-Transaktion	NL	NR	$NL + kNR$
Pess. + 2PC	$NL + 2NR$	NR	$NL + (k+1)NR$
Hyb.Opt. + 2PC	$NL + 2NR$	NU	$NL + (k-1)NU + 2NR$
Hyb.Opt. + 2PC + Valid.Fehler	$NL + 3NR$	$2NU$	$NL + (2k-2)NU + 3NR$
Vgl. ohne Zellen	**Einmalig für rN beteiligte Knoten**	**Parallele Operation auf rN Knoten**	**Gesamt für k serielle Operationen**
Atomares Schreiben	rNL	NR	$rL + 1R$, da stets $k = 1, N = 1$
Nur-Lese-Transaktion	rNL	NR	$rNL + kNR$
Pess. + 2PC	$rNL + 2NR$	NR	$rNL + (k+1)NR$
Hyb.Opt. + 2PC	$rNL + 2NR$	NU	$rNL + (k-1)NU + 2NR$
Hyb.Opt. + 2PC + Valid.Fehler	$rNL + 3NR$	$2NU$	$rNL + (2k-2)NU + 3NR$

Tabelle 4.1 vergleicht die Verfahren quantitativ hinsichtlich des Kommunikationsaufwands. Es werden Transaktionen aus k seriellen Operationen auf N Zellen betrachtet. Jede Operation arbeitet parallell auf den selben Zellen. U ist eine einfache, R eine replizierte Operation. L ist eine Routing-Operation im Overlay. Es wird davon ausgegangen, dass die Zellenstruktur während der Ausführung der Transaktion konstant ist. Für jedes Verfahren wird angegeben, welche und wieviele Operationen einmalig pro beteiligter Zelle, pro Operationsschritt der Transaktion und insgesamt benötigt werden. Die Gesamtzahl von Operationen berücksichtigt das Zusammenfassen von Nachrichten, wie etwa der letzten Datenoperation mit der Validierungs- und Prepare-Nachricht.

Falls $k > 1$ ist, wird die Anzahl replizierter Operationen durch den Einsatz des beschriebenen, hybriden Verfahrens reduziert. Dafür können aber bei einem Fehlschlagen der Validierung zustätzliche Routingschritte notwendig sein. Zum Vergleich wird in der zweiten Hälfte der Tabelle angegeben, wie hoch der Kommunikationsaufwand ist, wenn *oberhalb* des Overlays repliziert wird (Replikationsfaktor r). Es zeigt sich, dass der Einsatz von Zellen unnötige Routingschritte verhindert. Ohne Zellenstruktur sind stets rNL Routing-Nachrichten notwendig. Das Zellenmodell ermöglicht außerdem die Verwendung mehrerer Routingalternativen pro Routingschritt (Hop) und kann in Zusammenhang mit Techniken zur Latenzschätzung (z.B. [18]) benutzt werden, um die Gesamtroutingzeit zu minimieren.

Es gibt verschiedene Ansätze für die Implementation von replizierten Zustandsmaschinen: Quorumsysteme und atomarer Broadcast. EKWALL ET AL. [19] vergleichen beide, und kommen zu dem Ergebnis, dass atomarer Broadcast vorteilhafter ist, da er einerseits einen schwächeren Fehlerdetektor erfordert und andererseits den Kommunikationsaufwand reduziert, falls die Updatefunktion in einem Schritt übertragen wird. Im Vergleich dazu erfordern Quorumsysteme stets zwei Schritte: 1. Lesen, 2. Schreiben der *beim Klienten* berechneten Veränderungen. Eine Implementation von atomarem Multicast für Gruppen dynamischer Größe, wie sie für das Zellenmodell benötigt werden, beschreibt [59].

Der Einsatz replizierter Zustandsmaschinen erfordert im günstigen Fall $N(\lfloor 2N/3 \rfloor + 1)$ Unicast-Nachrichten und mindestens 3 Kommunikationsrunden [41]. Die Ur-

sache dafür ist, dass der RSM-Ansatz die Einhaltung einer totalen Ordnung zwischen allen ausgeführten, replizierten Operationen erfordert. Dies ist jedoch nicht immer notwendig. Für viele Paare von Nachrichten gilt, dass der von ihnen ausgelöste Effekt unabhängig von der Ausführungsreihenfolge beider Nachrichten ist. Die Nachrichten können kommutativ vertauscht werden. Generischer Broadcast [49] berücksichtigt dies durch Einführung einer Kommutationsrelation zwischen Nachrichtenpaaren. Die Zustellung kommutierender Nachrichten erfordert im günstigen Fall nur 2 Kommunikationsrunden. Generischer Brodcast erfordert jedoch, dass nicht mehr als ein $1/3$ der beteiligten Knoten ausfallen [50].

Die Kombination von generischem Broadcast mit Standardtechniken für Gruppenkommunikation [6] könnte die Nachteile des RSM-Ansatzes reduzieren. Für strukturierte Overlays kann Kommutierbarkeit zwischen Nachrichten leicht global festgelegt werden: 1) Nachrichten, die an disjunkte Tupelmengen adressiert sind, kommutieren 2) Nur-Lese-Operationen kommutieren 3) Sonst nichts. Eine andere Optimierungsmöglichkeit für replizierte Zustandsmaschinen ist das Ignorieren semantisch irrelevanter Nachrichten. PEREIRA ET. AL [51] beschreiben einen entsprechenden Ansatz im Kontext von massivparallelen Online-Spielen. Für die Zukunft bleibt somit eine Vielzahl von Möglichkeiten für die weitere Aufwandsminimierung des Einsatzes von Replikazellen.

Ein andere interessante Optimierungsmöglichkeit für die Verbesserung der Routinglatenz im Zellenmodell könnte darin bestehen, jeden Knoten gleichzeitig in mehreren nicht benachbarten Zellen zu verwenden. Bei der Nachrichtenweiterleitung würde ein derartiger Knoten dann eine „Abkürzung“ in eine weit entfernte Region des Schlüsselraums bieten.

Das vorgestellte Zellenmodell geht davon aus, dass Änderungen der Gebietsstruktur lokal begrenzt sind: Änderungen der Grenzlinie einer Zelle betreffen maximal ihre direkten Nachbarn. Diese Grundannahme über die Topologie ist durchaus nicht selbstverständlich. Beispielsweise können in CAN Situationen auftreten, in denen durch den Ausfall von Knoten Lücken in der Abdeckung entstehen können, die nicht durch ein Verschmelzen von ehemaligen Nachbarknoten behoben werden können: Das entstehende Gebiet wäre nicht mehr in der von CAN verwendeten Rechteckform.

Es ist jedoch die Meinung des Autors, dass es grundsätzlich notwendig ist, den Reparaturaufwand bei Änderungen in der Gebietsstruktur des Overlay-Netzwerks lokal zu begrenzen, um den damit verbundenen Kommunikationsaufwand zu minimieren. Je mehr Zellen von einer Strukturänderung betroffen sind, um so mehr Knoten müssen an der diese Änderung umsetzenden Transaktion teilnehmen. Außerdem ist es wünschenswert, dass zu Beginn einer Strukturänderung die Menge der betroffenen Zellen feststeht, da dies die Verwendung pessimistischer Schreibtransaktionen ohne Verklemmung (SS2PL) ermöglicht. Die Suche nach geeigneten Tesselationen für die Gebietsaufteilung mehrdimensionaler Schlüsselräume ist eine interessante Aufgabe für zukünftige Arbeiten.

Das entwickelte Transaktionsverfahren berücksichtigt direkt die Gegebenheiten des Zellenmodells, in dem es die Verwendung replizierte Operationen minimiert. Dies erforderte den Einsatz optimistischer Nebenläufigkeitskontrolle, da für pessimistische Verfahren die replizierte Ausführung jeder Operation nötig ist.

Der gewählte hybride Ansatz erlaubt einerseits die garantierte Ausführung langlaufender Transaktionen mit Zugriffsinvarianz, andererseits können reine Anfragetransaktionen auf schnell ausgeführte Nur-Lese-Operationen zurückgreifen, die kein atomares Festschreiben erfordern. Falls Lese- und Schreibmenge zu Beginn der Transaktion vollständig bekannt sind, kann alternativ auch direkt das starke Zweiphasensperrprotokoll verwendet werden, das den optimistischen Schreibtransaktionen zugrunde liegt. Dies ist beispielsweise bei Änderungen an den Routingtabellen oder aber einfachen Schreiboperationen mit paralleler Aktualisierung von Indices sinnvoll. Es ist somit möglich, für jede Anwendungsoperation ein passendes Transaktionsverfahren zu wählen. In Anlehnung an die Anwendungen aus Abschnitt 3.2, sind pessimistische Schreibtransaktionen für WikiWrite und SetSecMetadata, Nur-Lese-Transaktionen für WikiRead und GetSecMetadata und optimistische Schreibtransaktionen beispielsweise für DeleteBookmark geeignet.

Verschiedene Aspekte der Transaktionsverarbeitung, wie die Behandlung von Phantomproblemen, die Durchführung von Joins und die Integration der Transaktionsverarbeitung mit Mechanismen für Broadcast- und Bulkoperationen des Overlays, wurden in dieser Arbeit nicht berücksichtigt. Die Betrachtung wurde insgesamt auf das Seitenmodell beschränkt, dass lediglich elementare Lese- und Schreiboperationen auf einzelnen Datenobjekten untersucht. Für die Zukunft wäre die Erweiterung des entwickelten Verfahrens für das Objektmodell,[30] dass komplexe, hierarchisch gegliederte Operationsbäume mit Kommutationsinformation auf jeder Ebene betrachtet, ein sinnvoller, nächster Schritt. Dies würde einem Wechsel vom Datenanfrageparadigma (data-request) zum Funktionsanfrageparadigma (function-request) entsprechen, bei dem Ressourcemanager komplexe, typspezifische Operationen auf Datenobjekten ausführen. Die Integration dieses Ansatzes mit Replikationstechniken, die Kommutationsinformation berücksichtigen, wie etwa generischem Broadcast [49], könnte dazu beitragen, den Aufwand für die Transaktionsausführung signifikant zu senken.

Eine andere Optimierungsmöglichkeit entsteht durch die Verteilung der Transaktionslogik. Im vorgestellten Verfahren wird davon ausgegangen, dass die Transaktionslogik durch den Transaktionsmanager ausgeführt wird, der dafür Lese- und Schreibanfragen an die einzelnen Ressourcemanager sendet. Eine entsprechende Verlagerung einzelner Bestandteile des Kontrollflusses zu den Ressourcemanagern könnte benutzt werden, um die Kommunikationslatenz langlaufender Transaktionen zu reduzieren. Dies erfordert jedoch eine komplexe Verteilung der jeweils für die Fortsetzung des Kontrollflusses benötigten Daten zwischen allen beteiligten Ressourcemanagern. Die Anwendung bekannter analytischer Techniken aus dem Compilerbau ist ein vielversprechender Weg für die Entwicklung eines Algorithmus für die verteilte Ausführung des Kontrollflusses von Transaktionen.

In dieser Arbeit wurde ein Transaktionsverfahren für Overlay-Netzwerke entworfen, das einerseits die Behandlung von Routingkonsistenz, Ausfallsicherheit und Replikation ermöglicht, andererseits den dafür erforderlichen Kommunikationsaufwand minimiert. Dabei wurden die Erfordernisse typischer Anwendungen berücksichtigt. Neben den in diesem Kapitel angesprochenen Optimierungen verbleiben eine strengere Formalisierung des Transaktionsverfahrens und des Zellenmodells vor dem Hintergrund konkreter Topologien, sowie die Erstellung einer Implementation als interessante Aufgaben für die Zukunft.

[30] ausführlich beschrieben in [66]

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

Literaturverzeichnis

[1] ABDALLAH, Maha ; PUCHERAL, Phillippe: A Single-Phase Non-Blocking Atomic Commitment Protocol. In: *Lecture Notes in Computer Science* Bd. 1460. Springer, 1998

[2] AGRAWAL, Divyakant ; BERNSTEIN, Arthur J. ; GUPTA, Pankaj ; SENGUPTA, Soumitra: Distributed optimistic concurrency control with reduced rollback. In: *Distributed Computing* (1987), Nr. 2, S. 45–59

[3] BERNSTEIN, Philip A. ; GOODMAN, Nathan: Multiversion Concurrency Control – Theory and Algorithms. In: *ACM Transactions on Database Systems* 8 (1983), Dezember, Nr. 4, S. 465–483

[4] BHARAMBE, Ashwin R. ; AGRAWAL, Mukesh ; SESHAN, Srinivasan: Mercury: Supporting Scalable Multi-Attribute Range Queries. In: *SIGCOMM Comput. Commun. Rev.* 34 (2004), Nr. 4

[5] BINDEL, David ; CHEN, Yan ; EATON, Patrick ; GEELS, Dennis ; GUMMADI, Ramakrishna ; RHEA, Sean ; WETHERSPOON, Hakim ; WEIMER, Westley ; WELLS, Christopher ; ZHAO, Ben ; KUBIATOWICZ, John: OceanStore: An Extremely Wide-Area Storage System. 2000 (UCB/CSD-00-1102). – Forschungsbericht

[6] BIRMAN, Kenneth: Lightweight Causal and Atomic Group Multicast. In: *ACM Transactions on Computer Systemes* 9 (1991), Nr. 3, S. 272–314

[7] CAI, Wenyuan ; ZHOU, Shuigen ; QIAN, Weining ; XU, Linhao: C^2: a new overlay network based on CAN and Chord. In: *International Journal of High-Performance Computing* (2006). – to appear

[8] CASTRO, M. ; COSTA, M. ; ROWSTRON, A.: Performance and dependability of structured peer-to-peer overlays / Microsoft Research. 2003 (MSR-TR2003-94). – Forschungsbericht

[9] CASTRO, Miguel ; DRUSCHEL, Peter ; GANESH, Ayalvadi ; ROWSTRON, Antony ; WALLACH, Dan S.: Secure routing for structured peer-to-peer overlay networks. In: *Proc. of the 5th Usenix Symposium on Operating Systems Design and Implementation (OSDI '06)*, 2006

[10] CASTRO, Miguel ; DRUSCHEL, Peter ; KERMARREC, Anne-Marie ; NANDI, Animesh ; ROWSTRON, Antony ; SINGH, Atul: SplitStream: High-Bandwidth Multicast in Cooperative Environments. In: *SOSP'03*, 2003

[11] CHAN, Arvola ; FOX, Stephen ; LIN, Wen-Te K. ; NORI, Anil ; RIES, Daniel R.: The Implementation of An Integrated Concurrency Control and Recovery Scheme. In: *Proceedings of the 1982 ACM SIGMOD international conference on Management of data*, 1982, S. 184–191

[12] CHANDRA, Tushar D. ; HADZILACOS, Vassos ; TOUEG, Sam: The weekest failure detector for soliving consensus. In: *Journal of the Association for Computing Machinery* 43 (1996), Juli, Nr. 4, S. 685–722

[13] CHARRON-BOST, Bernadette ; SCHIPER, André: Improving Fast Paxos: being optimistic with no overhead. In: *IEEE Proc. 12th Pacific Rim Int. Symp. on Dependable Computing (PRDC)*, 2006, S. 287–295

[14] CHEN, Peter Phin-Shan: The Entity-Relationship Model – Toward a Unified View of Data. In: *ACM Transactions on Database Systems* 1 (1976), Januar, Nr. 1, S. 9–36

[15] CHEN, Wei ; LIU, Xuezheng: Enforcing Routing Consistency in Structured Peer-to-Peer Overlays: Should We and Could We? In: *5th International Workshop on Peer-to-Peer Systems (IPTPS'06)*, 2006

[16] COX, Russ ; DABEK, Frank ; KAASHOEK, Frans ; LI, Jinyang ; MORRIS, Robert: Practical, Distributed Network Coordinates. In: *Proceedings of the Second Workshop on Hot Topics in Networks (HotNets-II)* ACM SIGCOMM, 2003

[17] COX, Russ ; MUTHITACHAROEN, Athicha ; MORRIS, Robert: Serving DNS using Chord. In: *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002

[18] DABEK, Frank ; COX, Russ ; KAASHOEK, Frans ; MORRIS, Robert: Vivaldi: A Decentralized Network Coordinate System. In: *Proceedings of the ACM SIGCOMM '04 Conference*, 2004

[19] EKWALL, Richard ; SCHIPER, André: Replication: Understanding the Advantage of Atomic Broadcast over Quorum Systems. In: *Journal of Universal Computer Science* 11 (2005), Nr. 5, S. 703–711

[20] EL-ANSARY, Sameh ; ALIMA, Luc O. ; BRAND, Per ; HARIDI, Seif: Efficient Broadcast in Structured P2P Networks. In: *IPTPS'03*, 2003

[21] EL-ANSARY, Sameh ; HARIDI, Seif: An Overview of Structured P2P Overlay Networks. In: *Theoretical and Algorithmic Aspects of Sensor, Ad Hoc Wireless and Peer-to-Peer Networks*. CRC Press, 2005

[22] FISCHER, Michael J. ; LYNCH, Nancy A. ; PATERSON, Michael S.: Impossibility of Distributed Consensus with One Faulty Process. In: *Journal of the Association for Computing Machinery* 32 (1985), April, Nr. 2, S. 374–382

[23] FORD, Bryan ; SRISURESH, Pyda: Peer-to-Peer Communication Across Network Address Translator. In: *USENIX'05*, 2005

[24] FREEDMAN, Michael J. ; LAKSHMINARAYANAN, Karthik ; RHEA, Sean ; STOICA, Ion: Non-Transitive Connectivity and DHTs. In: *Proceedings of the Second Workshop on Real, Large Distributed Systems (WORLDS '05)*, 2005, S. 55–60

[25] GHODSI, Ali: *Distributed k-Ary System: Algorithms for Distributed Hash Tables*, KTH Stockholm, Diss., 2006

[26] GHODSI, Ali ; ALIMA, Luc O. ; HARIDI, Seif: Symmetric Replication for Structured Peer-to-Peer Systems. In: *The 3rd International Workshop on Databases, Information Systems and peer-to-Peer Computing*, 2005

[27] GRAY, Jim: The Transaction Concept: Virtues and Limitations. In: *Proceedings of the 7th International Conference on Very Large Databases*, 1981, S. 144–154

[28] GRUBER, Robert E.: *Optimistic Concurrency Control for Nested Distributed Transactions*, Massachusetts Institute of Technology, Diss., Juni 1989

[29] GUERRAOUI, Rachid ; RODRIGUEZ, Luís: *Introduction to Reliable Distributed Programming*. Berlin : Springer, 2006

[30] GUERRAOUI, Rachid ; SHIPER, André: The Decentralized Non-Blocking Atomic Commitment Protocol. In: *Proceedungs of the 7th IEEE Symposium on Parallel and Distributed Processing (SPDP '95)*, 1995

[31] HARRELL, Fox ; HU, Yuanfang ; WANG, Guilian ; XIA, Huaxia: *Survey of Locating and Routing in Peer-to-Peer Systems*. Online. http://www.cs.ucsd.edu/classes/fa01/cse221/projects/group15.pdf. Version: Dezember 2001

[32] HARRIS, Tim ; MARLOW, Simon ; JONES, Simon P. ; HERLIHY, Maurice: Composable memory transactions. In: *ACM Conference on Principles and Practice of Parallel Programming (PPoPP'05)*, 2005

[33] HERLIHY, Maurice: Wait-Free Synchronization. In: *ACM Transactions on Programming Languages and Systems* 13 (1991), Januar, Nr. 1, S. 124–149

[34] HERLIHY, Maurice ; MOSS, J. Eliot B.: Transactional Memory: Architectural Support for Lock-Free Data Structures. In: *Proceedings of the 20th Annual International Symposium on Computer Architecture*. 1993, S. 289–300

[35] JOHNSON, Stein E.: *Consistent lookup during Churn in Distributed Hash Tables*, Norwegian University of Science and Technology, Diss., 2005

[36] KARGER, D. ; LEHMAN, E. ; LEIGHTON, F. ; LEWIN, M. ; PANIGRAPHY, R.: Consistent hashing and randome trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In: *Proceedings of the 29th Annual ACM Symposium on Theory of Computing* ACM, 1997

[37] KLYNE, G. ; CARROLL, J. J.: Resource Description Framework (RDF): Concepts and Abstract Syntax / W3C. Version: Februar 2004. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/. 2004 (rdf-concepts). – W3C Recommendation

[38] LAMPORT, Leslie: Time, Clocks, and the Ordering of Events in a Distributed Systms. In: *Communications of the ACM* 21 (1978), Nr. 7

[39] LAMPORT, Leslie: The Part-Time Parliament. In: *ACM Transactions on Computer Systemes* 2 (1998), Mai, Nr. 16, S. 133–169

[40] LAMPORT, Leslie: Paxos made Simple. In: *ACM SIGACT News (Distributed Computing Column)* Bd. 32, 2001, S. 18–25

[41] LAMPORT, Leslie: Fast Paxos / Microsoft Research. 2nd revised edition. 2006 (MSR-TR-2005-112). – Forschungsbericht

[42] LYNCH, Nancy ; MERRITT, Michael ; WEIHL, William ; FEKETE, Alan: *Atomic Transactions*. Morgan Kaufmann Publishers, 1994

[43] MARKATOS, E. P.: Tracing a Large-Scale Peer to Peer System: An Hour in the Life of Gnutella. In: *The Second International Symposium on Cluster Computing and the Grid*, 2002

[44] MESAROS, Valentin ; COLLET, Raphael ; GLYNN, Kevin ; ROY, Peter V.: *A Transactional System for Structured Overlay Networks*. März 2005

[45] MILOJICIC, Dejan S. ; KALOGERAKI, Vana ; LUKOSE, Rajan ; NAGARAJA, Kiran ; PRUYNE, Jim ; RICHARD, Bruno ; ROLLINS, Sami ; XU, Zhichen:

Peer-to-Peer Computing / Hewlett-Packard. 2002 (HPL-2002-27). – Forschungsbericht

[46] MOHAN, C. ; PIRAHESH, Hamid ; LORIE, Raymond: Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. In: *SIGMOD '92: Proceedings of the 1992 ACM SIGMOD international conference on Management of data*. New York, NY, USA : ACM Press, 1992, S. 124–133

[47] MUTHITACHAROEN, Athicha ; GILBERT, Seth ; MORRIS, Robert: Etna: A Fault-tolerant Algorithm for Atomic Mutable DHT Data / Computer Science and Artificial Intelligence Laboratory. 2005 (MIT-CSAIL-TR-2005-044 and MIT-LCS-TR-993). – Forschungsbericht

[48] NAOR, Moni ; WIEDER, Udi: A Simple Fault Tolerant Distributed Hash Table. In: *Proceedings of the IPTPS 2003 / Lecture Notes in Computer Science*, 2003, S. 88–97

[49] PEDONE, Fernando ; SCHIPER, André: Generic Broadcast / Département d'Informatique Ecole Polytechnique Fédérale de Lausanne. 1999 (IC 1999-12). – Forschungsbericht

[50] PEDONE, Fernando ; SCHIPER, Andre: Lower Bounds on Generic Broadcast Algorithms / Laboratoire de Systèmes Répartis, Ecole Polytechnique Fédérale de Lausanne (EPFL). 2004 (LSR-REPORT-2004-013). – Forschungsbericht

[51] PEREIRA, José ; RODRIGUES, Luís ; OLIVEIRA, Rui: Reducing the Cost of Group Communication with Semantic View Synchrony. In: *Proceedings of Dependable Systems and Networks*, 2002, S. 293–302

[52] RAMABHADRAN, Sriram ; RATNASAMY, Sylvia ; HELLERSTEIN, Joseph M. ; SHENKER, Scott: Prefix Hash Tree - An Indexing Data Structure over Distributed Hash Tables / Intel Research Berkeley. 2004 (IRB-TR-03-011). – Forschungsbericht

[53] RATNASAMY, Sylvia ; FRANCIS, Paul ; HANDLEY, Mark ; KARP, Richard ; SHENKER, Scott: A Scalable Content-Addressable Network. In: *Proceedings of the ACM SIGCOMM 2001*, 2001

[54] REED, David P.: Naming and Synchronization in a Decentralized Computer System / MIT. 1978 (MIT-LCS-TR-205). – Forschungsbericht

[55] REED, David P.: Implementing Atomic Actions on Decentralized Data. In: *ACM Transactions on Computer Systems* 1 (1983), Februar, Nr. 1, S. 3–23

[56] RIPEANU, Matei: Peer-to-Peer Architecture Case Study: Gnutella Network. In: *1st IEEE International Conference on Peer-to-Peer Computing (P2P2001)*, 2001

[57] RITTER, Jordan: *Why Gnutella Can't Scale. No, Really.* Online. http://www.darkridge.com/~jpr5/doc/gnutella.html. Version: Februar 2001

[58] ROWSTRON, Antony ; KERMARREC, Anne-Marie ; CASTRO, Miguel ; DRUSCHEL, Peter: SCRIBE: The design of large-scale event notifications infrastructure. In: *Poceedings of 3rd International Workshop on Networked Group Communication (NGC2001)*, 2001

[59] SCHIPER, André: Dynamic group communication. In: *Distributed Computing* 18 (2006), Nr. 5, S. 359–374

[60] SCHNEIDER, Fred B.: The State Machine Approach: A Tutorial. / Department of Computer Science, Cornell University. 1986. (TR 86-800). – Forschungsbericht

[61] SCHÜTT, Thorsten ; SCHINTKE, Florian ; REINEFELD, Alexander: Structured Overlay without Consistent Hashing: Empirical Results / Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB). 2005 (ZR-05-40). – ZIB Technical Report

[62] SCHÜTT, Thorsten ; SCHINTKE, Florian ; REINEFELD, Alexander: Structured Overlay without Consistent Hashing: Empirical Results. In: *Proceedings of the Sixth Workshop on Global and Peer-to-Peer Computing (GP2PC'06)*, 2006

[63] STOICA, Ion ; MORRIS, Robert ; LIBEN-NOWELL, David ; KARGER, David ; KAASHOEK, M. F. ; DABEK, Frank ; BALAKRISHNAN, Hari: Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. (January 10, 2002)

[64] THOMASIAN, Alexander: Distributed Optimistic Concurrency Control Methods for High-Performance Transaction Processing. In: *IEEE Transactions on Knowledge and Data Engineering* 10 (1998), Januar/Februar, Nr. 1, S. 173–189

[65] WEIHL, William E.: Distributed version management for read-only actions (extended abstract). In: *PODC '85: Proceedings of the fourth annual ACM symposium on Principles of distributed computing*. New York, NY, USA : ACM Press, 1985. – ISBN 0–89791–168–7, S. 122–135

[66] WEIKUM, Gerhard ; VOSSEN, Gottfried: *Transactional Information Systems – Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. London : Academic Press, 2002

[67] XU, Zhichen ; ZHANG, Zheng: Building Low-maintenance Expressways for P2P Systems / Internet Systems and Storage Laboratory / Hewlett-Packard Laboratories Palo Alto. 2002 (HPL-2002-41). – Forschungsbericht