Diplomarbeit

Symore - A System For Mobile Replication

Entwicklung und Implementierung eines optimistischen
Datenbankreplikationssystems für MANETs

Frank Bregulla
bregulla@inf.fu-berlin.de

Freie Universität Berlin
Institut für Informatik
Fachbereich Mathematik und Informatik
Betreuer: Prof. Dr.-Ing. Heinz Schweppe
Dipl. Inf Manuel Scholz

Abgabe: 25. Juli 2006

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Motivation . . . 1
- 1.2 Aufgaben und Ziele der Diplomarbeit . . . 2
- 1.3 Aufbau der Arbeit . . . 4

2 Grundlagen **5**
- 2.1 Mobile Ad-Hoc Netze . . . 5
- 2.2 Replikation . . . 6
  - 2.2.1 Aktualisierung . . . 7
  - 2.2.2 Synchronisation . . . 8
  - 2.2.3 Entwurfsmöglichkeiten optimistischer Replikationssysteme . . . 12

3 Verwandte Replikationssysteme **15**
- 3.1 Zustandstransfersysteme . . . 15
- 3.2 Semantische Konflikterkennung: Bayou und MobiSnap . . . 17
- 3.3 Epidemic Algorithms in Replicated Databases . . . 19
- 3.4 Multiversionsansatz . . . 21
- 3.5 IceCube . . . 22
- 3.6 Joyce . . . 23

4 Replikation in Symore **25**
- 4.1 Systemablauf . . . 25
- 4.2 Uhrensynchronisation . . . 30
  - 4.2.1 Rasterzeit . . . 32
  - 4.2.2 Totalordnung der Ereignisse . . . 34
  - 4.2.3 Synchronisationsalgorithmus . . . 35
- 4.3 Konflikterkennung . . . 37
  - 4.3.1 Konflikt . . . 37
  - 4.3.2 Phänomene . . . 41
  - 4.3.3 Datenstruktur: Vorgängergraph . . . 43
- 4.4 Konfliktlösung . . . 46
  - 4.4.1 Konfliktlösungsalgorithmen . . . 47
  - 4.4.2 First-Wins-Head-Algorithmus . . . 49

4.4.3 Konfliktlösung mittels Prioritätswerten . . . 52
4.4.4 Last-Wins-Tail-Algorithmus . . . 54
4.4.5 Berücksichtigung von Lesemengen . . . 56
4.5 Commit . . . 56
4.5.1 Korrektheit . . . 58
4.5.2 Implizites Commit . . . 59
4.5.3 Forced Commit . . . 60
4.5.4 Scheduled Commit . . . 61
4.5.5 Inaktivität . . . 61
4.5.6 Löschen aus dem Vorgängergraphen . . . 62
4.6 Datenverteilung . . . 63
4.7 Gruppenverwaltung . . . 64
4.7.1 Gruppenbeitritt . . . 64
4.7.2 Verlassen der Gruppe . . . 65
4.7.3 Ausschluss aus der Gruppe . . . 65
4.8 1-Kopien-Serialisierbarkeit . . . 66

5 Implementierung 71
5.1 Übersicht . . . 71
5.1.1 Architektur des Replikationsmanagers . . . 72
5.1.2 Interne vs. externe Implementierung des Replikationssystems . . . 73
5.2 Von einer SQL-Anweisung zur Datenverteilung . . . 74
5.2.1 Parser . . . 74
5.2.2 Ausführung einer lokalen Anweisung . . . 75
5.2.3 Lokales Commit . . . 80
5.3 Empfang und Ausführung von Ereignissen . . . 82
5.3.1 Aktualisierung der lokalen Datenbanksicht . . . 85
5.3.2 Benachrichtigung der Anwendung über Änderungen an der vorläufigen Sicht . . . 86
5.4 Korrektheit und Tests . . . 87

6 Beispielanwendung: MobileWiki 89
6.1 Anwendungsfälle . . . 89
6.2 Reihenfolge von Abschnitten . . . 92
6.3 Beobachter . . . 94
6.4 Fazit . . . 94

7 Skalierbarkeit 97
7.1 Systemparameter . . . 97
7.2 Analyse . . . 100
7.2.1 Bestimmung der Konfliktwahrscheinlichkeit . . . 101
7.3 Gray: Dangers of Replication . . . 107

8 Zusammenfassung und Ausblick **111**
8.1 Zusammenfassung 111
8.2 Ausblick 113

Literatur **114**

A Benutzung von Symore **119**
A.1 Konfigurationsparameter 119
A.2 Eigene Konfliktlösungsalgorithmen 121
A.3 Eigene Datenverteilung 121
A.4 Erweiterungen von SQL 122
A.4.1 Datendefinitionssprache 122
A.4.2 Automatische UUID-Generierung 122
A.4.3 Interne Prozeduren 122

B SQL-Grammatik **123**

Abbildungsverzeichnis

2.1 Systemarchitekturen replizierter Datenbanksysteme 8

4.1 Architektur von SYMORE 26
4.2 Basisworkflow des Replikationssystems SYMORE 29
4.3 Knoten und die Zeitdifferenzen zwischen ihren Uhren 33
4.4 Bestimmung der Zeitdifferenz von Knoten B zu Knoten A 35
4.5 Bestimmung der Zeitdifferenz über mehrere Hops 36
4.6 Konflikt durch kausal parallele Updates auf A und B 38
4.7 Konfliktelemente unterschiedlicher Granularität 39
4.8 Konflikt durch kausal parallele Updates auf A und B unter Berücksichtigung von Leseoperationen 40
4.9 Ausschnitt eines Vorgängergraphen 44
4.10 Last-Wins-Tail-Konfliktlösung 55
4.11 Historie eines Replikationsmanagers 58
4.12 Löschen von Transaktionen aus dem Vorgängergraphen 63

5.1 Paketdiagramm des Replikationsmanagers 72
5.2 Klassendiagramm des Paketes symore.sql.lang 76
5.3 Vereinfachter Objektbaum einer Select-Anweisung 77
5.4 Klassendiagramm des Pakets Event 83
5.5 Nebenläufigkeit in der lokalen Datenbank 85

6.1 Hauptfenster der Anwendung MOBILEWIKI 90
6.2 Konzeptuelles Datenbankschema des MOBILEWIKI 90

7.1 Beziehungen zwischen Parametern, die die Konflikthäufigkeit beeinflussen. 98
7.2 Auftreten von Konflikten im Modell (ü – Übertragung; t – Transaktion) . . 101
7.3 Konfliktwahrscheinlichkeit in Bezug auf das Verhältnis E(U) zu E(T') . . . 106
7.4 E(D) in Bezug auf Datenbankgröße und Transaktionslänge 106
7.5 P(K) mit E(U)/E(K)=3 in Bezug auf Knotenanzahl und Datenbankgröße . 107
7.6 P(K) mit E(U)/E(K)=1 in Bezug auf Knotenanzahl und Datenbankgröße . 107
7.7 Vergleich der Konfliktwahrscheinlichkeiten nach Gray und nach der Analyse in 7.2. 108

1 Einleitung

1.1 Motivation

Unsere heutige Zeit ist geprägt von Mobilität. Viele Menschen verbringen ihre Zeit an wechselnden Orten oder sind häufig unterwegs. Es entsteht der Wunsch, jederzeit und an jedem Ort auf Informationen zugreifen können. Sei es, dass man schon auf dem Weg zum Büro die aktuellsten Börsenkurse abfragt, einen Eintrag in dem zentralen Arbeitsgruppen-Terminkalender vornimmt, Außendienstmitarbeiter zusammen mit Kunden Vertragsdaten auch fernab vom Büro in einer zentralen Datenbank ändern oder Touristen Hintergrundinformationen über bestimmte Sehenswürdigkeiten vor Ort abrufen. Unterstützt wird dieser mobile Datenzugriff durch immer leistungsfähigere mobile Computer wie Notebooks, PDAs oder Handys, deren Akkulaufzeiten zudem immer länger werden.

Kabelgebundener Netzwerkzugriff ist nicht überall möglich, aber die weite Verfügbarkeit von Infrastruktur-Funknetzwerken wie z.B. GSM, GPRS oder UMTS, ermöglichen den fast flächendeckenden Onlinezugriff auf zentrale Daten. Der Vorteil des Einsatzes dieser Techniken ist es, immer die aktuellsten Daten abrufen und ändern zu können. Sie haben allerdings den Nachteil, (noch) recht teuer zu sein und nicht die Leistungsfähigkeit (Bandbreite, Verzögerung) von kabelgebundenen Netzen zu bieten. Vor allem aber sind solche Netze zwar weit verbreitet, aber eben nicht immer und überall verfügbar. Um zu verhindern, dass in solchen Fällen kein Arbeiten möglich ist, weil auf wichtige Daten nicht zugegriffen werden kann, werden diese Daten auf das mobile Gerät kopiert (repliziert). Dort kann offline, d.h. ohne aktive Netzwerkverbindung, mit ihnen gearbeitet werden. Erst wenn eine Verbindung mit der Datenbank auf einem zentralen Server wieder möglich ist, werden die replizierten Daten mit dieser synchronisiert. Wurden sie parallel auf dem Server, oder von anderen mobilen Geräten geändert, müssen die dadurch entstandenen Konflikte erkannt und gelöst werden.

In dem im Folgenden vorgestellten System SYMORE – **Sy**stem for **mo**bile **re**plication – wird ein weiterer Aspekt des mobilen Datenzugriffs betrachtet. In diesem System existiert kein zentraler Server, der letztendlich die Autorität über die Daten hat. Jedes mobile Gerät, das an einer Replikationsgruppe teilnimmt, speichert eine Kopie der Daten. Lokale Änderungen dieser werden sofort über ein Ad-hoc-Funknetz an alle Geräten der Replikationsgruppe, die sich in Reichweite des Funknetzes befinden, verteilt. Diese Änderungsnachrichten werden von den empfangenden Geräten zwischengespeichert, um sie später auch an diejenigen weiterverteilen zu können, die sich momentan nicht in

Funkreichweite befinden oder ausgeschaltet sind (epidemische Verteilung).

Jedes der hier vorgestellten Mobilitätsszenarien ist für bestimmte Anwendungen mehr oder weniger geeignet. Eine Versicherung wird die Notebooks ihrer Vertreter mit Software ausstatten, die entweder online direkt die Daten der zentralen Datenbank abruft und ändert, oder ein serverbasiertes Replikationsverfahren einsetzen. Eine Peer-to-Peer-Replikation über mobile Ad-Hoc-Netze (MANETs), wie sie SYMORE bietet, ist dagegen überall dort sinnvoll, wo eine Kommunikationsinfrastruktur fehlt, zu teuer ist oder aus anderen Gründen nicht verwendet werden kann, Änderungen an Daten aber möglichst schnell an andere Geräte verteilt werden sollen. Ein Einsatzgebiet für SYMORE sind beispielsweise Feldforschungsszenarien. Eine Gruppe von Wissenschaftlern (Archäologen, Biologen, Geologen, u.a.) macht Untersuchungen in einem abgelegenen Gebiet, wobei jeder Forscher seine Entdeckungen und Ergebnisse gleich in seinem mobilen Computer speichert. Hier besteht für die begrenzte Zeitspanne der Untersuchung kein Kontakt zu einem zentralen Server und einer zentralen Datenbank. Zur besseren Koordination sollen die Ergebnisse jedes einzelnen Forschers aber möglichst schnell an alle anderen verteilt werden. Hierbei könnte SYMORE zum Einsatz kommen. Weitere Einsatzfelder für solch eine Peer-to-Peer-Replikation und -synchronisation in spontanen, autonomen Funknetzen bestehen im Katastrophenmanagement, bei e-learning-Systemen oder mobilen Spielen. Auch Touristeninformationssysteme profitieren davon, wenn Nutzer beispielsweise Empfehlungen und Bewertungen zu einzelnen Sehenswürdigkeiten lokal und autonom auf ihren mobilen Geräten vornehmen können u. diese, direkt und auch ohne Kontakt zu einem zentralen Server mit benachbarten Nutzern austauschen können. In all diesen Bereichen kann SYMORE dafür sorgen, dass auf jedem mobilen Gerät immer auf einen gemeinsamen, möglichst aktuellen Datenbestand zugegriffen werden kann und lokal Änderungen an diesem vorgenommen werden können. Dadurch dass sich jeder mit jedem synchronisiert, können solche Änderungen schnell im Netz verteilt werden.

1.2 Aufgaben und Ziele der Diplomarbeit

Das Ziel der vorliegenden Diplomarbeit ist es, ein leichtgewichtiges, optimistisches Datenbankreplikationssystem für **m**obile **A**d-**H**oc-**Net**ze (MANETs) zu entwerfen und prototypisch zu implementieren. Wie im vorigen Abschnitt beschrieben, soll es vollkommen dezentral arbeiten, d.h. ohne einen zentralen Server auskommen. Ein optimistisches und asynchrones Replikationsverfahren soll eingesetzt werden.

Alle Daten sind auf jedes mobile Gerät repliziert. Es soll möglich sein lokal und autonom mit diesen zu arbeiten, auch wenn gerade keine Verbindung zu einem Netzwerk und damit zu den anderen Geräten besteht. Änderungen an diesen Daten sollen anschließend zu den anderen Geräten der Replikationsgruppe übertragen werden, sobald sich diese in Funkreichweite befinden. Dazu sollen verschiedene Verteilungsverfahren eingesetzt werden können.

Das zu entwickelnde System soll die transaktionale Ausführung von Operationen ermöglichen. Eine Transaktion besteht hier aus einer Menge von Operationen, die auf lokale Datenbankelemente zugreifen. Eine Transaktion soll atomar ausgeführt werden. Im Konfliktfall sollen alle ihre Operationen gemeinsam abgebrochen werden. Dieses ist für viele Anwendungen wichtig, um die Konsistenz der Daten in der Datenbank zu wahren.

Dadurch, dass Transaktionen autonom und lokal ausgeführt und erst nachträglich zu anderen Geräten übertragen werden, können Konflikte entstehen und Verletzungen der Isolationseigenschaft auftreten. Welche Konflikte und Nebenläufigkeitsanomalien eintreten können soll analysiert werden. Eine Lösung um diese zu erkennen und zu behandeln soll entworfen werden. Dabei soll das System in zweierlei Hinsicht Flexibilität bieten. Auf der einen Seite soll ein Nutzer entscheiden können, welche Arten von Konflikten erkannt werden sollen. Die für die jeweilige Anwendung nötigen Konsistenzgarantien können somit individuell ausgewählt werden. Die schwächste Bedingung, die garantiert werden soll, ist *Eventual Consistency*. Diese Garantie besagt, dass letztendlich jedes Gerät von allen auf anderen Geräten durchgeführten Änderungsoperationen erfährt und zu dem gleichen, global konsistenten Datenbankzustand gelangt, wenn nirgends mehr neue Transaktionen ausgeführt werden und das Netz nicht permanent partitioniert ist. Zwischenzeitlich können die Datenbankzustände der Geräte differieren. Zusätzlich soll gezeigt werden, dass unter Berücksichtigung sämtlicher durch kausal parallele Operationen auftretenden Phänomene auch 1-Kopien-Serialisierbarkeit erreicht wird.

Auf der anderen Seite soll Flexibilität dadurch erreicht werden, dass zur Lösung von Konflikten verschiedene, auch benutzerdefinierte Verfahren eingesetzt werden können.

Die für die Konflikterkennung und Konfliktlösung nötigen Datenstrukturen und Algorithmen sollen entwickelt werden. Ausgangspunkt dafür sind die in [Sch05a] vorgestellten Datenstrukturen des *Vorgängerbaumes (precedence tree)* zur Erkennung von Konflikten und der *SkewMatrix* zur Zeitsynchronisation der Uhren der mobilen Geräte. Diese Datenstrukturen sind in [Rö06] bei der Implementierung eines Objektreplikationssystems erprobt worden. Es soll in dieser Arbeit untersucht werden, wie sie den Bedürfnissen der Replikation bei relationalen Datenbanksystemen angepasst werden können. Der Vorgängerbaum speichert Verweise auf die Version eines Objektes, die von einer Änderungsoperation überschrieben wurde. Dadurch kann erkannt werden, wenn zwei Operationen kausal parallel die gleiche Version überschrieben haben. Die SkewMatrix speichert die Zeitdifferenzen der Uhren der verschiedenen an der Replikation beteiligten Geräte. Dieses ist nötig, um lokale Zeitpunkte eines Gerätes in lokale Zeitpunkte anderer Geräte umrechnen zu können. Diese Umrechnung kann fehlerhaft sein, da die Bestimmung der Zeitdifferenz nie exakt erfolgen kann. Eine Lösung zum Umgang mit diesem Problem wird in der Arbeit vorgestellt.

Um die Tauglichkeit des entworfene Replikationssystems zu demonstrieren, soll es schließlich in Java prototypisch für den Einsatz auf mobilen Geräten implementiert werden. Es soll als Bibliothek in Anwendungen eingebettet werden können. Zur Demonstration des Systems soll außerdem eine Beispielanwendung, die dieses Replikationssystem

nutzt, entwickelt werden.

Schließlich soll untersucht werden, wie das entwickelte System skaliert. Dazu wird ein Modell entwickelt, mit dem sich analysieren lässt, wie sich die Konfliktwahrscheinlichkeit bei unterschiedlichen Werten der Systemparameter verhält.

Der Schwerpunkt des im Folgenden zu entwickelnden Konzepts und dessen Implementierung liegt auf den Problemen, die sich durch die Replikation ergeben, wie Konflikterkennung, Konfliktlösung und dem Erreichen der verschiedenen Konsistenzbedingungen. Aspekte wie Persistenz und Recovery sind nicht Fokus der Arbeit.

1.3 Aufbau der Arbeit

Nachdem nun die Ziele dieser Diplomarbeit vorgestellt wurden, werden im folgenden Kapitel die Grundlagen für das hier entwickelte System, die beiden Bereiche *MANETs* und *Datenbankreplikation*, behandelt. In Kapitel 3 werden verwandte Arbeiten über optimistische und asynchrone Replikation vorgestellt. Neben replizierten Datenbanksystemen werden dabei auch verteilte Objektreplikationssysteme betrachtet. Diese Ansätze und Systeme werden mit dem in dieser Arbeit entwickelten System verglichen. In Kapitel 4 wird zunächst ein Überblick über den Aufbau und die Funktionsweise SYMORES gegeben und anschließend auf alle Teilaspekte im Detail eingegangen. Nach der Vorstellung der Funktionsweise werden in Kapitel 5 implementierungsspezifische Aspekte betrachtet. Eine Beispielanwendung – MOBILEWIKI – wird entwickelt, die SYMORE zur Replikation von Daten verwendet. Diese wird in Kapitel 6 vorgestellt. In Kapitel 7 wird die Skalierbarkeit des Systems theoretisch betrachtet und eine Formel zur Berechnung der Konfliktwahrscheinlichkeit hergeleitet. Zum Abschluss werden in Kapitel 8 die wichtigsten Aspekte des Systems noch einmal zusammengefasst und die erreichten Ziele aufgeführt. Des Weiteren wird aufgezeigt, wie der entwickelte Prototyp zu einem Produktivsystem ausgebaut werden kann und wie das hier vorgestellte Replikationskonzept weiter entwickelt werden könnte.

2 Grundlagen

MANETs und replizierte Datenbanken bilden die Grundlage für das vorgestellte verteilte Replikationssystem. In diesem Kapitel werden sowohl die Eigenschaften von MANETs behandelt als auch wichtige Begriffe im Bereich der replizierten Datenbanken eingeführt. Die Ansätze pessimistischer und optimistischer Synchronisation sowie synchroner und asynchroner Aktualisierung werden miteinander verglichen.

2.1 Mobile Ad-Hoc Netze

Nach [CM99] besteht ein MANET aus kooperierenden, autonomen und mobilen Geräten wie Laptops oder PDAs, die drahtlos miteinander kommunizieren. Sie nutzen keine vorhandene Kommunikationsinfrastruktur, sondern kommunizieren spontan („ad-hoc“) miteinander. Dabei fungiert jedes Gerät gleichzeitig als Router, der Daten für entfernte Geräte weiterleitet. Aus der Mobilität der Teilnehmer im MANET ergibt sich, dass sich die Netzwerktopologie dynamisch verändern kann. Teilnehmer können sich temporär aus der Kommunikationsreichweite anderer Teilnehmer entfernen und kurzzeitige Netzpartitionierungen entstehen lassen. Durch die relative Störanfälligkeit der drahtlosen Verbindung ergeben sich besondere Anforderungen an Kommunikationsprotokolle und Routingalgorithmen. Auch ist die Übertragungskapazität verglichen mit heutigen drahtgebundenen Netzwerken noch recht gering. Es handelt sich um ein geteiltes Medium, dessen Bandbreite sich alle in Funkreichweite befindlichen Nutzer teilen. Das vorgestellte Replikationsprotokoll soll also möglichst effizient mit der zur Verfügung stehenden Bandbreite umgehen und die Eigenschaft, dass jeder Medienzugriff tatsächlich ein (1-Hop)-Broadcast ist, für sich ausnutzen.

Weiterhin haben mobile Geräte nur begrenzte Energiereserven. Mit Geräten, die ausfallen, weil ihr Akku leer ist, ist also zu rechnen. Außerdem zeichnen sich Geräte, die in MANETs eingesetzt werden, typischerweise dadurch aus, dass sie nur über eine beschränkte Speicherkapazität und eine relativ geringe Rechenleistung verfügen.

Auch Sicherheitsaspekte spielen in MANETs eine wichtige Rolle. Drahtlose Verbindungen können leicht belauscht werden und sind anfällig für Dienstblockaden (denial of service). Auch sollten Daten, die auf mobilen Geräten gespeichert werden, vor unautorisiertem Zugriff geschützt werden, da diese kleinen Geräte besonders leicht gestohlen werden können. Diese Sicherheitsaspekte werden in der vorliegenden Arbeit nicht behandelt, da sie ein eigenes Forschungsfeld bilden.

2.2 Replikation

In diesem Abschnitt wird auf Grundlagen replizierter Datenbanken eingegangen. Zentrale Begriffe dieses Bereichs, die für die folgenden Kapitel von Bedeutung sind, werden eingeführt.

Ein *repliziertes Datenbanksystem* ist ein verteiltes Datenbanksystem, bei dem *logische Datenelemente (data items)* als *Kopien* oder *lokale Datenelemente (stored data items)* auf verschiedenen lokalen Datenbanken gespeichert sind. Diese Datenbanken befinden sich in der Regel auf unterschiedlichen Rechnern und sind über ein Netzwerk miteinander verbunden. Ein Datenelement ist ein atomarer Bestandteil einer Datenbank, der erzeugt oder gelöscht oder dessen Wert verändert werden kann. Aus den Datenelementen einer lokalen Datenbank und deren Werten ergibt sich ihr *lokaler Datenbankzustand*. Die Aufgabe eines replizierten Datenbanksystems ist es, dafür zu sorgen, dass die lokalen Datenbankzustände aller Datenbanken dieses Systems möglichst gleich sind.

Stationäre replizierte Datenbanksysteme bestehen im Normalfall aus einer Menge von *Datenmanagern (DM)* und einem zentralen *Transaktionsmanager (TM)*. Ein solches System ist in Abbildung 2.1 (a) dargestellt. Ein DM verwaltet eine lokale Datenbank und führt dort Lese- und Schreiboperationen auf Kopien logischer Datenelemente aus. Der TM dient als Bindeglied zwischen Nutzer und repliziertem Datenbanksystem. Er übersetzt Lese- und Schreiboperationen auf logischen Datenelementen einer Nutzertransaktion in Operationen auf Kopien dieser Elemente und sendet sie an die entsprechenden DM. Der TM sorgt dafür, dass Nutzer das replizierte Datenbanksystem wie ein einzelnes Datenbanksystem wahrnehmen. Dazu muss er u.a. sicherstellen, dass eine Transaktion immer die aktuelle Version eines Datenelementes liest, etwa indem er Schreiboperationen auf Datenelemente immer in Schreiboperationen auf alle Kopien dieser Elemente übersetzt.

Es gibt verschiedene Gründe für den Einsatz von Replikation. Ein Ziel ist es, die Leistungsfähigkeit stationärer Datenbanksysteme zu steigern. Indem Anfragen auf verschiedene Knoten verteilt werden, kann ein Lastausgleich geschaffen und der Transaktionsdurchsatz erhöht werden. Auch kann durch Replikation auf geographisch nahe Server eine Verbesserung des Durchsatzes und der Verzögerungszeiten bei der Datenabfrage erzielt werden. Weiterhin kann durch Replikation eine Backup-Kopie einer Datenbank möglichst aktuell gehalten oder Daten in ein Data Warehouse übertragen werden. Eine weitere wichtige Aufgabe von Replikation ist die Erhöhung der Verfügbarkeit eines Datenbanksystems. Fällt ein Knoten aus, können andere Knoten, auf die die Daten repliziert worden sind, weiterhin Transaktionen bearbeiten. Der angebotene Dienst kommt nicht vollständig zum Erliegen. Wird die Transaktionsverarbeitung auch bei Ausfall eines Knotens aufrecht erhalten, so müssen allerdings besondere Vorkehrungen zur Wahrung der Korrektheit beim Wiederanlauf dieses Knotens getroffen werden.

Wenngleich Replikation auch in stationären Systemen von Vorteil sein kann, so bietet sich ihre Verwendung besonders in mobilen Systemen an. Mobile Datenbanksysteme haben das Ziel, Datenzugriff auf einem mobilen Gerät auch dann zu ermöglichen, wenn

dieses gerade nicht mit dem Netzwerk und einem zentralen Datenbankserver verbunden ist. Zu diesem Zweck werden Daten, die auch lokal verfügbar sein sollen, auf das mobile Gerät repliziert.

Laut [HTKR05] sind die zwei wichtigsten Systemarchitekturen bei mobilen Datenbanksystemen die *erweiterte Client/Server-Architektur* und die *Middleware-Architektur mit einem Replikationsserver*. Bei ersterer gleicht jeder Client seine replizierten Daten sporadisch direkt mit einem zentralen Server ab, bei letzterer ist ein zentraler Replikationsserver zwischengeschaltet. Dieser steuert die Kommunikation zwischen Client und Server und kann so zwischen verschiedenen Datenbanksystemen auf beiden Seiten vermitteln. Weiterhin kann er die Serverdatenbank vor unberechtigtem Zugriff schützen und diese eventuell zu Stoßzeiten entlasten, indem er Synchronisationsdaten selbst zwischenspeichert. In diesen Systemen ist Replikation nicht transparent. Ein Nutzer ist sich bewusst, dass durch Transaktionen, die auf mobilen, unverbundenen Geräten stattfinden, Konflikte auftreten können, die bei der Synchronisation erkannt werden. Die erweiterte Client-Server-Architektur ist in Teil (b) der Abbildung 2.1 dargestellt.

Das in dieser Arbeit entwickelte System SYMORE ist ebenfalls ein mobiles Datenbanksystem. Replikation wird gleichfalls eingesetzt, um jederzeit Zugriff auf gemeinsame Daten auf nicht permanent verbundenen mobilen Geräten zu gewährleisten. Das Einsatzszenario von SYMORE ist ein MANET, in dem eine Gruppe mobiler Geräte direkt miteinander kommuniziert und kooperiert. Im Gegensatz zu den eben genannten mobilen Datenbanksystemen gibt es in diesem Szenario keinen zentralen Server, der die Daten verwaltet und auch keinen zentralen TM. Alle Daten sind auf jedes mobile Gerät repliziert. Nutzer greifen über lokale Replikationsmanager (RM) auf die lokalen Kopien der replizierten Daten zu. Alle RM kooperieren miteinander und sorgen durch direkte Synchronisation (peer-to-peer) dafür, dass die lokalen Datenbestände möglichst untereinander konsistent gehalten werden. Dargestellt ist dieses in Teil (c) der Abbildung 2.1.

2.2.1 Aktualisierung

Die erhöhte Verfügbarkeit und höhere Leistung – im Wesentlichen für Leseoperationen – replizierter Datenbanksysteme erhält man nicht umsonst. Es muss erheblicher Aufwand betrieben werden, um alle Kopien des replizierten Datenbestandes konsistent zu halten. Ändert eine Transaktion eine Kopie eines logischen Datenelementes, so müssen auch alle anderen Kopien aktualisiert werden. Die verschiedenen Datenbankzustände sollen schließlich nicht zu sehr auseinanderdriften. Diese Aktualisierung kann entweder *synchron (eager)* oder *asynchron (lazy)* geschehen.

Bei der synchronen Aktualisierung werden alle Knoten als Teil der Änderungstransaktion selbst aktualisiert. Änderungen an Datenelementen werden erst nach einem Commit sichtbar. Dieses wird erst abgeschlossen nachdem die Transaktion alle Kopien der manipulierten Datenelemente aktualisiert hat. Somit sind die lokalen Datenbankzustände immer in einem konsistenten Zustand. Nachteilig an diesem Verfahren ist, dass im Normalfall alle Knoten erreicht werden müssen, um eine Transaktion erfolgreich zu beenden.

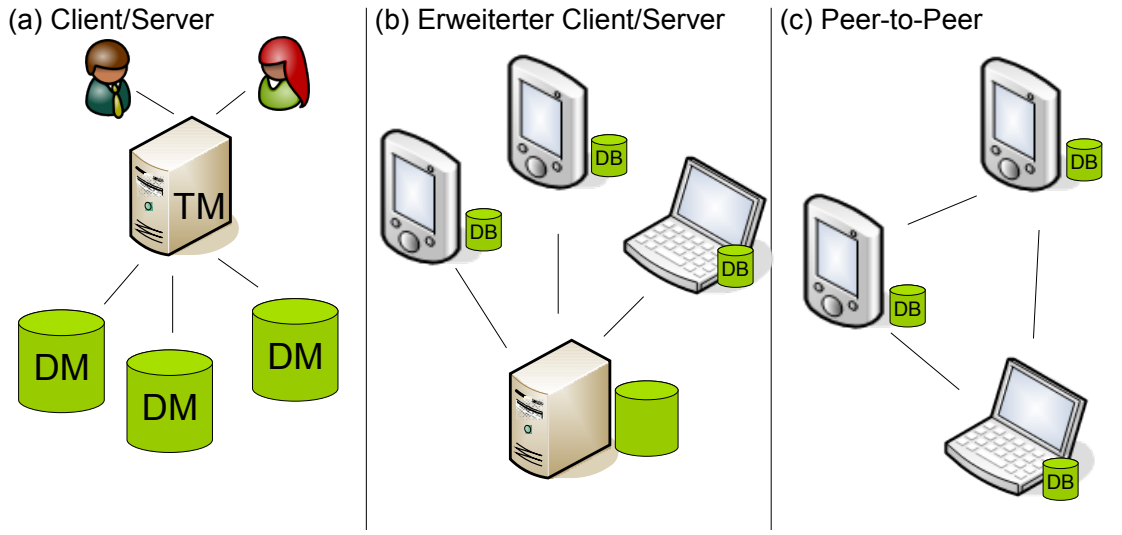

Abbildung 2.1: Systemarchitekturen replizierter Datenbanksysteme

Die Transaktionsausführung wird also verzögert, solange ein Knoten oder die Netzwerkverbindung ausgefallen ist.

Bei der asynchronen Aktualisierung werden andere Knoten erst verzögert aktualisiert. Die ursprüngliche Transaktion manipuliert Kopien logischer Datenelemente auf einem lokalen Datenbanksystem. Erst nach deren Beendigung werden deren Änderungen auch auf den anderen Knoten eingebracht. Dabei wird in Kauf genommen, dass Nutzer zeitweilig mit inkonsistenten Daten arbeiten.

Eine synchrone Aktualisierung kann in mobilen Systemen nicht sinnvoll eingesetzt werden, da häufig nicht alle Knoten gleichzeitig erreicht werden können und sich die Transaktionsausführung stark verzögern würde. Deshalb werden Kopien normalerweise asynchron aktualisiert.

2.2.2 Synchronisation

Außer für die Aktualisierung aller Kopien zu sorgen, muss ein repliziertes Datenbanksystem parallele Zugriffe von Transaktionen auf unterschiedliche Kopien der gleichen Datenelemente koordinieren. Ohne jegliche Kontrolle der Nebenläufigkeit könnten parallel ablaufende Transaktionen beispielsweise Kopien der gleichen logischen Datenelemente unterschiedlicher Knoten auf unterschiedliche Weise ändern. Dieses würde dazu führen, dass die verschiedenen lokalen Datenbankzustände immer weiter auseinanderdrifteten.

Es werden zwei Ansätze unterschieden, wie parallele Transaktionen auf unterschiedlichen Geräten synchronisiert werden können. *Pessimistische Synchronisationsverfahren* vermeiden Konflikte und Nebenläufigkeitsanomalien, die durch parallelen Zugriff auf Ko-

pien der gleichen Datenelemente entstehen, und sorgen dafür, dass eine Transaktion immer den aktuellen Wert eines Datenelementes liest. *Optimistische Synchronisationsverfahren* dagegen verhindern zunächst nicht, dass Konflikte auftreten können und dass veraltete Versionen von Datenelementen gelesen werden. Diese Verfahren gehen von der optimistischen Annahme aus, dass potentiell konfliktverursachende Schreiboperationen auf gleiche Datenelemente selten sind. Erst im Nachhinein, nach Abschluss der eigentlichen Transaktion, wird geprüft, ob Konflikte aufgetreten sind. Diese müssen dann entsprechend behandelt werden. Wo bei pessimistischer Synchronisation Transaktionen blockiert werden, um Konflikte zu vermeiden, müssen diese Konflikte bei optimistischer Synchronisation im Nachhinein gelöst werden.

Die Ansätze der pessimistischen und optimistischen Synchronisation werden im Folgenden genauer betrachtet.

Pessimistische Synchronisation

Pessimistische Synchronisation wird im Normalfall in stationären Systemen mit permanenter und zuverlässiger Netzwerkverbindung zwischen TM und allen beteiligten DM eingesetzt. Der Einsatz von Replikation soll hier vor dem Nutzer unter dem Gesichtspunkt der Serialisierbarkeit der Transaktionsausführungen verborgen bleiben. Diese Systeme garantieren *1-Kopien-Serialisierbarkeit (1-copy serializability)*, die folgendermaßen definiert ist: „Ein globaler Schedule S auf einer replizierten Datenbank ist 1-Kopieserialisierbar, wenn es einen seriellen Schedule auf einer nicht-replizierten Datenbank gibt, der den gleichen Effekt erzeugt wie S auf dem replizierten Datenbestand.“ [SH99, S. 590].

Um diese Konsistenzgarantie erfüllen zu können, arbeiten die meisten pessimistischen Synchronisationsverfahren mit globalen Sperren. Bevor eine Transaktion ein Datenelement ändern kann, muss sie für dieses eine Sperre anfordern. So wird verhindert, dass Transaktionen parallel auf unterschiedliche Kopien der gleichen logischen Datenelemente zugreifen. Werden im Rahmen der Transaktion alle Kopien eines logischen Datenelementes zusammen geändert, können in diesen Systemen keine Konflikte auftreten und die globalen Schedules sind serialisierbar. Damit eine Transaktion Operationen ausführen kann, ist also eine Kommunikation mit anderen Knoten nötig. Ist eine Netzwerkverbindung temporär nicht vorhanden, ist in dieser Zeit die Transaktionsausführung je nach System nicht möglich oder zumindest stark eingeschränkt.

Wie bei zentralisierten sperrbasierten Verfahren besteht zudem die Gefahr von Verklemmungen, deren Erkennung und Auflösung im verteilten Fall aufwändig ist.

Es gibt verschiedene sperrbasierte, pessimistische Synchronisationsverfahren. Diese unterscheiden sich hauptsächlich darin, wo und wieviele Sperren angefordert werden müssen, um ein bestimmtes Datenelement zu lesen oder zu schreiben. Einige der häufigsten Ansätze werden hier kurz vorgestellt.

Zentralisiertes Sperren: Hier verwaltet ein zentraler Server die Sperren für alle repli-

zierten Datenelemente. Eine Transaktion, die eine Kopie eines logischen Datenelementes ändern will, muss von diesem Server eine Sperre für das Datenelement anfordern.

Primärkopie-Verfahren: Bei diesem Verfahren werden nicht alle Sperren von einem einzigen zentralen Server verwaltet, sondern jedem logischen Datenelement ist ein bestimmter Knoten als Primärkopie zugeordnet. Um Änderungen an diesem logischen Datenelement vornehmen zu können, muss erst dessen Sperre von diesem ihm zugeordneten Knoten angefordert werden. Durch die Verteilung der Sperrverwaltung auf mehrere Knoten wird ein besserer Lastausgleich als beim zentralisierten Sperren erzielt.

ROWA(A) (Read One, Write All (Available)): Hier verwaltet jeder Knoten Sperren für seine lokalen Kopien der Datenelemente. Um ein Datenelement zu lesen, kann eine Sperre von einem beliebigen Knoten angefordert werden. Wird ein Datenelement geschrieben, sind hingegen Sperren von allen Knoten nötig. So können Kopien eines Datenelementes parallel gelesen werden, aber nur eine Transaktion kann ein Datenelement zur Zeit ändern. Es entfällt das Nadelöhr eines zentralen Servers. Leseoperationen erfordern nur einen sehr geringen Aufwand, Schreiboperationen hingegen einen sehr hohen. Bei der Variante ROWAA genügt es, zum Schreiben Sperren von allen gerade verfügbaren Knoten anzufordern. Problematisch ist hierbei allerdings Datenkonsistenz zu wahren, wenn zeitweilig nicht verfügbare Server wieder verfügbar werden.

Majority Locking: Hier müssen sowohl zum Lesen als auch zum Schreiben eines Datenelementes von einer Mehrheit der Knoten Sperren gehalten werden. Der Nachteil besteht darin, dass selbst zum Lesen eines Datenelementes recht viele Sperren angefordert werden müssen. Von Vorteil ist, dass auch in einem partitionierten Netz noch Updates möglich sind. In der Partition mit der Mehrheit der Knoten kann weiter mit den Daten gearbeitet werden und es besteht keine Gefahr, dass in anderen Partitionen gleichzeitig Kopien derselben logischen Datenelemente geändert und damit Konflikte erzeugt werden. Eine Verallgemeinerung dieses Verfahrens ist das *Quorum Consensus* Verfahren, bei dem jedem Knoten ein nichtnegatives Gewicht zugewiesen wird. Um ein Datenelement zu ändern, müssen nun nicht von der Mehrheit der Knoten Sperren gehalten werden. Es genügt, so viele Knoten zu sperren, dass deren Gewicht zusammen mehr als die Hälfte des Gesamtgewichts ergibt.

Optimistische Synchronisation

Bei den oben beschriebenen pessimistischen Synchronisationsverfahren wird davon ausgegangen, dass im Normalfall alle Knoten erreichbar sind. Die Nicht-Erreichbarkeit eines Knotens wird als seltenes Ereignis betrachtet. In mobilen Systemen ist die temporä-

re Nicht-Verfügbarkeit eines Knotens hingegen der Normalfall. Dieser Eigenschaft wird die optimistische Synchronisation am besten gerecht. Bei der lokalen Ausführung von Operationen, die auf replizierte Datenelemente zugreifen, kann in mobilen Umgebungen nicht gewartet werden, bis alle nötigen Sperren gewährt sind. Ähnlich wie bei optimistischen Nebenläufigkeitskontrollprotokollen in zentralisierten Datenbanken werden deshalb bei optimistischen Synchronisationsverfahren keine globalen Sperren angefordert, damit eine Transaktion ein Datenelement lesen oder schreiben kann. Eine Transaktion kann somit ohne Synchronisation auf einem Knoten eine Kopie eines logischen Datenelementes ändern. Bevor diese Änderung auf einem anderen Knoten eingebracht worden ist, kann dort ebenfalls eine Kopie dieses Datenelementes geändert worden sein. Dass durch solche kausal parallelen Operationen Konflikte entstehen, wird bewusst akzeptiert. Durch Wahl und Design von Anwendungen, die mit den replizierten Daten arbeiten, sollten potentiell konfliktverursachende Schreiboperationen auf den gleichen Datenelementen minimiert werden.

Der optimistische Ansatz hat den Vorteil, dass auch dann mit lokalen Daten auf einem Knoten gearbeitet werden kann, wenn dieser gerade nicht mit dem Netz verbunden ist oder sich in einer Partition befindet, der nicht die Mehrheit aller Knoten angehören. Jeder Knoten arbeitet autonom. Zunächst ist keinerlei Koordination und Kommunikation mit anderen Knoten nötig, um lokal Operationen auszuführen. Änderungsoperationen werden im Normalfall asynchron, erst nach Abschluss der eigentlichen Transaktion ausgetauscht. Erst im Nachhinein werden eventuelle Konflikte durch kausal parallele Operationen festgestellt und behandelt.

Wie für pessimistische gibt es auch für optimistische Replikationssysteme verschiedene Realisierungsmöglichkeiten. Diese unterscheiden sich unter anderem darin, wie und wo Konflikte festgestellt werden, ob Transaktionen unterstützt werden und wie Commitentscheidungen getroffen werden.

Mobile Datenbanken, die Replikation in einem erweiterten Client-Server-Szenario verwenden und auch einen optimistischen Ansatz zur Synchronisation einsetzen, sind in den letzten Jahren kommerziell verfügbar geworden und werden häufig eingesetzt (z.B. [IBM04], [iAn04]). Das in dieser Arbeit entwickelte Replikationssystem ist ein vollständig dezentrales repliziertes Datenbanksystem, das sich nicht mit einer zentralen Serverdatenbank, sondern peer-to-peer mit gleichberechtigten Datenbanken auf mobilen Geräten in einer Replikationsgruppe synchronisiert. Solche Systeme sind bisher nur in wenigen Forschungssystemen zu finden.

Die Aspekte, die in solchen Systemen mit Hinblick auf optimistische Synchronisation zu beachten sind, wie Konflikterkennung, Konfliktlösung und das Erreichen eines gemeinsamen konsistenten Datenbankzustandes, unterscheiden sich nicht grundsätzlich von denselben Aspekten in generellen optimistischen Peer-to-Peer-Objektreplikationssystemen. Bevor einige wichtige dieser optimistischen Replikationssysteme vorgestellt werden, werden im nächsten Abschnitt die generellen Entwurfsmöglichkeiten solcher Systeme aufgezeigt.

2.2.3 Entwurfsmöglichkeiten optimistischer Replikationssysteme

Saito und Shapiro [SS05] definieren sechs Kriterien, anhand derer sich die meisten optimistischen Replikationssysteme klassifizieren lassen. Dieses geschieht nicht primär mit dem Fokus auf Datenbanksystemen sondern in Bezug auf replizierte Systeme im Allgemeinen. Aus diesem Grund sprechen sie nicht von Datenelementen sondern von Objekten. Diese sechs Kriterien und die für SYMORE getroffene Wahl werden nun vorgestellt.

Anzahl der Schreiber: In optimistischen, replizierten Datenbanksystemen werden Operationen auf lokalen Kopien von Objekten durchgeführt. Verschiedene Systeme unterscheiden sich danach, auf welchen Kopien dieses möglich ist. Existierende Systeme reichen von *Single-Master*-Systemen, bei denen zunächst nur Kopien auf einem ausgezeichneten Knoten, dem „Master“, geändert werden dürfen und diese Änderungen anschließend an die anderen Knoten verteilt werden, bis hin zu *Multimaster*-Systemen, in denen mehrere oder alle Kopien unabhängig voneinander geändert werden dürfen.

SYMORE ist ein Multimaster-System. Jede Kopie eines Datenelementes kann unabhängig von den anderen Kopien geändert werden. Dieses ist wichtig in mobilen Umgebungen mit dynamischen Netzwerkpartitionierungen, bei denen ein zuverlässiger direkter Zugriff auf einen Master-Knoten nicht garantiert werden kann.

Datenübertragung: Zustand- vs. Operation: Nachdem lokale Kopien geändert wurden, müssen diese Änderungen auf die anderen Knoten übertragen und auf deren Kopien angewendet werden. Zum einen kann dazu der Zustand eines geänderten Objektes übertragen werden, zum anderen die Operation selbst, die dieses Objekt manipuliert hat. Eine Übertragung des Zustands ist im Normalfall einfacher, da die Kopie des geänderten Objektes auf dem empfangenden Knoten einfach überschrieben werden kann. Bei Operationsübertragungssystemen muss die Reihenfolge der Operationen beachtet werden. Knoten müssen sich auf eine Reihenfolge einigen, in der sie empfangene Operationen auf ihren Kopien ausführen, um einen konsistenten Zustand zu erreichen. Je nach Art der Operation kann die Übertragung dieser wesentlich effizienter sein als die Übertragung des kompletten Zustandes eines Objektes.

SYMORE überträgt die SQL-Anweisungen lokal initiierter Transaktionen, ist also ein Operationsübertragungssystem. Da sich eine einzelne SQL-Anweisung auf viele Datenelemente auswirken kann, werden somit häufig weniger Daten übertragen, als würde der neue Zustand aller geänderten Elemente übertragen.

Scheduling: syntaktisch vs. semantisch Eigene und empfangene fremde Operationen müssen von Knoten auf konsistente Weise geordnet werden, um äquivalente Zustände aller Kopien zu erreichen. Häufig ist diese Ordnung erst einmal vorläufig und kann sich ändern, wenn weitere Operationen und damit zusätzliche Informationen über parallel stattgefundene Aktivitäten empfangen werden. Syntaktisches

Scheduling verwendet nur die Informationen wann, wo und von wem Operationen ausgeführt worden sind, um sie zu ordnen und Konflikte zu erkennen. Semantisches Scheduling hingegen kann die Bedeutung von Updates und Beziehungen zwischen diesen mit einbeziehen. So kann ein semantischer Scheduler in einem Buchungssystem, in dem nur noch ein Platz verfügbar ist, Updates 1 und 2, die jeweils einen Platz buchen und Update 3, das eine Buchung storniert, in der Reihenfolge 1, 3, 2 ordnen und so alle Updates erfolgreich ausführen.

SYMORE verwendet das einfachere syntaktische Scheduling. Operationen, die nicht miteinander in Konflikt stehen, werden auf allen Knoten anhand ihrer ursprünglichen Ausführungsreihenfolge geordnet und ausgeführt, ohne auf die Semantik der Updates einzugehen. Dieses hat den Vorteil, dass die Art und Weise, wie auf zentrale Datenbanken zugegriffen wird, auch im verteilten Fall beibehalten werden kann. Gleiches gilt für die damit verbundene Serialisierbarkeitstheorie. Bei semantischem Scheduling müssten alle Datenzugriffe mit Informationen über deren Semantik angereichert werden.

Konfliktbehandlung: Konflikte entstehen, wenn Vorbedingungen von Operationen nicht erfüllt werden können. Pessimistische Systeme verhindern Konflikte von vornherein. Andere Systeme, wie z.B. verteilte Verzeichnissysteme, ignorieren Konflikte und überschreiben den Zustand von Objekten immer mit dem letzten Update. Systeme mit *syntaktischer Konflikterkennung* definieren Konflikte als parallele Updates auf ein Objekt und erkennen diese mittels syntaktischer Kriterien, wie z.B. Vektoruhren [Mat89] oder Vorgängergraphen. SYMORE gehört in diese Gruppe, wobei allerdings gezielt parallele Updates auf bestimmte Datenelemente als kommutativ zugelassen werden können. Auch ist eine gewisse Steuerung darüber möglich, welche Updates als miteinander in Konflikt stehend betrachtet werden. So können parallele Zugriffe auf verschiedene Datenelemente, die zu einem Konfliktelement zusammengefasst wurden, als Konflikt aufgefasst werden. In Systemen, die semantische Kriterien anwenden, entstehen unter Umständen weniger Konflikte, da ein Konflikt anhand der Verletzung bestimmter, updatespezifischer Vorbedingungen und nicht durch parallele Änderungen eines Objekts erkannt wird. Andererseits ist die semantische Konflikterkennung hochgradig anwendungsspezifisch und erfordert einen höheren Aufwand, sowohl bei dem Anwendungsentwickler als auch in Bezug auf nötige Rechenzeit.

Verteilungsstrategien: Updates, die auf Kopien von Objekten eines Knotens lokal ausgeführt worden sind, müssen auch auf allen anderen Kopien ausgeführt werden. Dazu müssen sie an alle Knoten, die weitere Kopien enthalten, verteilt werden. Verteilungsstrategien unterscheiden sich zum einen darin, ob sie für feste Netzwerktopologien entworfen wurden oder, wie die vielfach eingesetzten epidemischen Algorithmen, an eine sich dynamisch ändernde Netztopologie angepasst sind. Weiterhin kann Verteilung danach klassifiziert werden, welcher Grad an Synchronität erreicht

wird. Dieses ist abhängig von der Geschwindigkeit und Häufigkeit des Nachrichtenaustauschs. Der Nachrichtenaustausch kann periodisch stattfinden oder manuell initiiert werden und Push- oder Pull-Protokolle verwenden. In dem vorliegenden System wird ein epidemisches Verteilungsprotokoll für MANETs genutzt, wobei sowohl Push- als auch Pull-Verfahren für eine möglichst schnelle Verteilung mit möglichst geringem Aufwand sorgen.

Konsistenzgarantien: Optimistische Replikationssysteme nehmen eine gewisse temporäre Divergenz der Kopien auf den einzelnen Knoten in Kauf. Die meisten Systeme, so auch SYMORE, garantieren *Eventual Consistency*. Dies bedeutet, dass irgendwann, wenn keine neuen Updates mehr ausgeführt werden und alle Knoten miteinander verbunden sind, alle Kopien einen gleichen, konsistenten Zustand erreichen. Dazu müssen bestimmte Commit-Verfahren eingesetzt werden, die sicherstellen, dass alle Knoten alle Operationen bzw. Transaktionen in der gleichen Reihenfolge anwenden und Konflikte auf die gleiche Weise gelöst werden. Manche Systeme schränken die Divergenz ein, die sie zwischen Kopien zulassen, und stellen beispielsweise sicher, dass der Zustand einer bestimmten Kopie garantiert nicht älter als eine bestimmte Zeitspanne ist.

SYMORE garantiert zusätzlich zu Eventual Consistency 1-Kopien-Serialisierbarkeit. Das bedeutet, dass die letztendliche Ausführungsreihenfolge aller Transaktionen, die festgeschrieben werden, global serialisierbar ist und deren Effekte der Ausführung auf einer zentralisierten Datenbank entsprechen.

3 Verwandte Replikationssysteme

Im Bereich der optimistischen und asynchronen Replikation in mobilen Umgebungen sind bereits viele Forschungsarbeiten veröffentlicht worden. Auch weiterhin finden viele Forschungsaktivitäten zu diesem Thema statt. Systeme und Konzepte reichen von einfachen *Zustandstransfersystemen (state transfer systems)*, die nur einzelne Datenbankreihen betrachten, über komplexere Systeme, die Transaktionen unterstützen und Konflikte anhand von semantischen Kriterien erkennen und einen Primary-Copy-Ansatz verwenden, um Änderungen festzuschreiben, bis hin zu komplett dezentralisierten Systemen. Im Folgenden wird ein Überblick über einige der wichtigsten Arbeiten und Konzepte in diesem Bereich gegeben und ein Vergleich der Systeme mit SYMORE vorgenommen. Jedes System hat für bestimmte Einsatzzwecke seine Berechtigung. Daher ist es schwierig eine Bewertung vorzunehmen. Trotzdem soll versucht werden, herauszuarbeiten, welche Aspekte der vorgestellten Systeme aus welchen Gründen von SYMORE ähnlich oder anders realisiert werden.

3.1 Zustandstransfersysteme

Wingman [Ham] und Lotus Notes [LKBH+88] sind dezentrale, optimistische Zustandsübertragungssysteme, die es gestatten, auf jedem Knoten autonom Änderungen an Kopien von gemeinsamen Objekten vorzunehmen (update anywhere). Beide Systeme garantieren, dass der Zustand dieser Objekte zu einem konsistenten Zustand konvergiert (Eventual Consistency). Dazu übertragen sie im Hintergrund die lokal geänderten Daten zu anderen Knoten. Transaktionen in dem Sinn einer Gruppierung von Operationen, die atomar ausgeführt werden sollen, werden nicht unterstützt. Wingman erkennt Konflikte bei parallelen Änderungen einzelner Datensätze in einem relationalen Datenbanksystem und Notes bei parallelen Änderungen von Dokumenten in einem Dokumenten-Datenbanksystem. Dazu setzt Wingman Versionsvektoren ein (genannt „lineage“). Notes hingegen verwendet einfache Sequenznummern und Zeitstempel.

Wingman speichert zu jeder Datenbankzeile in einem zusätzlichen Feld `lineage` die Information, welcher Knoten welche Version dieser Zeile zuletzt geschrieben hat. Bei jeder lokalen Änderung einer Zeile wird der Versionswert, der der lokalen Knoten-Id zugeordnet ist, erhöht. Synchronisieren sich zwei Knoten A und B, wird immer die Version einer Zeile übernommen, die den höchsten Versionswert im Feld `lineage` enthält. Es wird zusätzlich erkannt, ob Konflikte durch kausal parallele Änderungen an dieser Zeile entstanden sind. Dieses wird signalisiert. Ein Konflikt liegt vor, wenn die Gewinner-

version eine niedrigere Versionsnummer für ein Update eines Knotens X enthält als die Verliererversion oder ein Knoten im Lineage der Verliererversion nicht im Lineage der Gewinnerversion vorkommt. Auf die Verliererversion sind Updates angewendet worden, die die Gewinnerversion nicht kennt und die nach der Synchronisation von A und B verloren gehen. Ein Nachteil dieses Verfahrens ist, dass der Versionswert einer Zeile auf einem Knoten durch viele lokale Updates auf diese künstlich so erhöht werden kann, dass diese Zeile im Konfliktfall garantiert gewinnt. Auch kann der Fall auftreten, dass eine Version in einem Konfliktfall zunächst verliert, aber ein weiteres Update, das auf dieser Verliererversion basiert, als Gewinner aus einem späteren Konflikt hervorgeht.

Notes wählt ein einfacheres Vorgehen, um Konflikte zu erkennen. Bei jeder Änderung an einem Datenelement wird deren Versionswert erhöht. Ähnlich wie Wingman wird im Konfliktfall die Version mit der höchsten Versionsnummer (die Version mit den meisten Updates) gewählt und die alte Version überschrieben. Kausal parallele Änderungen werden nicht erkannt.

Das „Bengal Database Replikation System“ [EMRP01] erweitert ähnlich wie SYMORE relationale Datenbanksysteme um Unterstützung für optimistische Peer-to-Peer-Replikation. Es erlaubt unverbundenen Nutzern Updates auf ihrer lokalen Kopie auszuführen und verteilt die geänderten Datenbankzeilen, sobald der lokale Knoten wieder mit anderen verbunden ist. Dadurch ist eine hohe Verfügbarkeit der Daten gewährleistet. Es besteht allerdings die Gefahr, dass durch parallele Updates auf die gleichen Datenbankzeilen Konflikte entstehen, die später gelöst werden müssen. Wie bei Wingman und Notes werden keine Transaktionen unterstützt. Updates werden mittels eines Gossiping-Verfahrens verteilt, bei dem immer Paare von Knoten Updateinformationen austauschen. Dieses hat den Vorteil, dass nicht jeder Knoten mit jedem anderen direkt kommunizieren muss. Konflikte werden durch Versionsvektoren erkannt. Im Konfliktfall werden die mit anderen in Konflikt stehenden Reihen separat gespeichert. Konflikte können entweder automatisch mittels *conflict resolvers* oder manuell gelöst werden. Bei der manuellen Konfliktlösung können entweder die in Konflikt stehenden Reihen zusammengeführt oder eine daraus ausgewählt werden. Aus der Arbeit zu Bengal geht leider nicht hervor, wie bei dieser Konfliktlösung vermieden wird, dass die lokalen Datenbankzustände der verschiedenen Knoten auseinanderdriften. Ein Konfliktlösungsalgorithmus müsste gewährleisten, dass Konflikte auf jedem Knoten gleich gelöst werden, egal in welcher Reihenfolge Updates empfangen werden.

Im Gegensatz zu dem hier vorgestellten Systemen unterstützt SYMORE Transaktionen, da viele Anwendungen verlangen, dass mehrere Operationen gemeinsam atomar ausgeführt werden können. Auch wenn nur eine dieser Operationen mit anderen in Konflikt steht und abgebrochen werden muss, sollen alle diese zusammengehörigen Operationen gemeinsam abgebrochen werden, damit die Datenbankkonsistenz gewahrt bleibt.

Auch werden die Operationen und nicht der Zustand der geänderten Datenelemente übertragen, weil dieses häufig einen geringeren Datenaufwand erfordert. Außerdem sind so auch kommutative Operationen möglich, die einen neuen Wert in Bezug auf einen

alten errechnen. Bei diesen ist die Reihenfolge ihrer Ausführung egal und es gibt Anwendungen, bei denen auch kausal parallele kommutative Operationen zugelassen werden können.

3.2 Semantische Konflikterkennung: Bayou und MobiSnap

Bayou ist ein optimistisches repliziertes Datenverwaltungssystem für mobile Umgebungen, das Eventual Consistency garantiert. Es ermöglicht eine anwendungsspezifische Konflikterkennung und Lösung.

Alle Datenelemente sind auf alle Knoten repliziert und Kopien dieser können lokal gelesen und vorläufig geschrieben werden. Jedem replizierten Datenelement ist ein sogenannter *Primary* zugeordnet, ein Knoten der dieses Element verwaltet. Änderungen werden durch Transaktionen vorgenommen, die eine Menge von Datenelementen atomar manipulieren können. Eine lokale Transaktion kann dabei aber nur Datenelemente manipulieren, die von demselben Primary verwaltet werden. Je zwei Knoten tauschen die ihnen bekannten Transaktionsdaten miteinander aus. Dieses Verfahren wird in Bayou paarweises Anti-Entropy-Verfahren genannt. Auf diese Weise werden beliebige, sich dynamisch ändernde Netzwerktopologien unterstützt. Wie in SYMORE werden Transaktionen lokal initiiert, vorläufig ausgeführt und verteilt. Jeder Knoten, der Transaktionen von anderen Knoten empfängt, führt diese ebenfalls lokal vorläufig aus. Jeder vorläufigen Transaktion ist ein global eindeutiger Zeitstempel zugeordnet, der die Reihenfolge bestimmt, in der vorläufige Transaktionen ausgeführt werden. Dieser Zeitstempel orientiert sich an der lokalen Uhr des jeweiligen Knotens, wobei eine Synchronisation der Uhren aller Knoten allerdings nicht stattfindet. Um Eindeutigkeit zu gewährleisten, enthält er zusätzlich die global eindeutige Geräte-Id.

Es kann nötig sein, dass vorläufig ausgeführte Transaktionen zunächst rückgängig gemacht werden, wenn andere Transaktionen mit früheren Zeitstempeln empfangen werden. Anschließend werden alle Transaktionen in der richtigen Reihenfolge erneut ausgeführt.

Jeder Transaktion in Bayou ist eine Vorbedingung in Form eines Prädikats zugeordnet. Dieses definiert, welche Werte die Kopien der Datenelemente eines Knotens haben dürfen, damit die Transaktion dort erfolgreich ausgeführt werden kann. Ist diese Vorbedingung nicht erfüllt, werden mittels einer zu der Transaktion gehörenden Merge-Prozedur alternative Änderungsoperationen versucht und geprüft, ob dadurch die Vorbedingung erfüllt werden kann. Nur im Erfolgsfall wird die Transaktion ausgeführt. Auf die hier beschriebene Weise erlangt jeder Knoten schließlich die gleiche vorläufige Sicht.

Um Transaktionen letztendlich festzuschreiben, ist es nötig die Reihenfolge, in der sie auf einem Knoten ausgeführt werden sollen, definitiv zu bestimmen. Dazu wird ein Primary-Commit-Verfahren eingesetzt. Wird eine Transaktion von deren Primary empfangen, wird auch dort versucht, die Transaktion auszuführen. Ist dieses erfolgreich, wird die Nachricht, dass diese Transaktion festgeschrieben wurde und in welcher Rei-

henfolge dieses im Hinblick auf andere Transaktionen geschehen ist, ebenfalls mittels Anti-Entropy verteilt. Muss die Transaktion abgebrochen werden, weil die Vorbedingung auch nach Anwendung der in der Merge-Prozedur definierten Alternativen nicht erfüllt werden kann, wird diese Abbruch-Entscheidung ebenfalls verteilt.

Anders als SYMORE, verwendet Bayou eine semantische Konfliktlösung. Die Ausführungsreihenfolge von Transaktionen wird bestimmt durch die Reihenfolge, in der sie ihren Primary erreichen. Während auch dieses System den paarweisen Austausch von Transaktionen ermöglicht und Transaktionsergebnisse sofort vorläufig verfügbar gemacht werden, ist zum letztendlichen Commit anders als in SYMORE, ein ausgezeichneter Primary nötig. Dessen Einsatz hat den Vorteil, dass Transaktionen ohne globale Abstimmung festgeschrieben werden können. Dieses kann geschehen, selbst wenn momentan nicht alle Knoten aktiv sind. Es besteht natürlich die Gefahr, dass der Primary selbst ausfällt. In SYMORE sind zu einem Commit Informationen aller Knoten der Replikationsgruppe nötig. Fällt ein Knoten aus, muss er aus der Replikationsgruppe ausgeschlossen werden, damit die verbleibenden Knoten weiter Commitentscheidungen für Transaktionen treffen können. In Bayou müsste für einen ausgefallenen Primary ein neuer bestimmt werden.

Auch müssen Transaktionsergebnisse und Ordnungsentscheidungen im Netz verteilt werden. Darauf kann in SYMORE verzichtet werden, weil jeder Knoten lokal selbstständig Commitentscheidungen trifft. Außerdem werden Transaktionen in Bayou in der Reihenfolge festgeschrieben, in der sie von dem zugehörigen Primary empfangen werden. Diese Reihenfolge entspricht, anders als in SYMORE, also nicht notwendigerweise der, in der diese Transaktionen lokal initiiert worden sind.

Auch unterscheiden sich die Bedeutungen von Konflikten. Bayou erkennt keine Konflikte auf Grund von kausal parallelen Änderungen an gleichen Datenelementen sondern ausschließlich anhand semantischer, anwendungsspezifischer Kriterien. Das hat den Vorteil, dass flexibler definiert werden kann, wann Transaktionen in Konflikt stehen. Nachteilig ist jedoch der höhere Aufwand. Ein Anwendungsentwickler muss sich für jede Transaktion sehr genau überlegen, welche Bedingungen erfüllt sein müssen, damit die Transaktion auch später auf dem Primary erfolgreich ausgeführt werden kann und Konsistenzkriterien erfüllt bleiben. Außerdem muss die Vorbedingung bei jeder Neuausführung der Transaktion abermals geprüft werden.

MobiSnap [PBM+00] ähnelt Bayou, da es ebenfalls Transaktionen verwendet, denen Konsistenzbedingungen in Form von Vor- und Nachbedingungen mittels Prädikaten über die aktuellen Datenwerte zugeordnet sind. Im Gegensatz zu Bayou wird hier das erweiterte Client-Server-Modell eingesetzt, in dem Transaktionen zunächst vorläufig auf Clients ausgeführt und erst später auf einen zentralen Server übertragen werden. Empfängt der Server eine Clienttransaktion, so prüft er, ob deren Konsistenzbedingungen weiterhin erfüllt werden können und schreibt die Transaktion im Erfolgsfall fest. Er sorgt außerdem dafür, dass diese Transaktion anschließend an andere Clients verteilt wird.

In MobiSnap wird zusätzlich ein Reservierungsmechanismus für mobile Transaktio-

nen bereitgestellt. Damit können stärkere Garantien gegeben werden, ob eine Transaktion letztendlich erfolgreich festgeschrieben werden kann. So kann z.B. für einen Client das Recht reserviert werden, Zeilen mit bestimmten vordefinierten Werten in die Datenbank einzufügen.

3.3 Epidemic Algorithms in Replicated Databases

In [AAS97] wird ein System zur Verwaltung replizierter Daten vorgestellt, das 1-Kopien-Serialisierbarkeit garantiert. Zusätzlich werden zwei Varianten dieses Systems präsentiert, die diese Konsistenzgarantie lockern und damit mobilen Umgebungen besser angepasst sind. Wie in SYMORE werden Operationen auf einzelnen Datenelementen nicht separat betrachtet, sondern werden zu Transaktionen, die atomar eine Gruppe von Operationen umfassen (und eine Menge von Datenelementen manipulieren), zusammengefasst. Die Grundannahme der hier vorgestellten Algorithmen ist es, dass kausal parallele Änderungen an gleichen Datenelementen und damit Konflikte zwischen Transaktionen unwahrscheinlich sind.

Allen in dieser Arbeit präsentierten Varianten ist gemeinsam, dass Transaktionen zunächst lokal auf einem Knoten ausgeführt werden und lokale Kopien der replizierten Datenelemente lesen und schreiben. Anschließend werden sie mittels eines epidemischen Algorithmus an die anderen Knoten der Replikationsgruppe verteilt. Um festzustellen, ob zwei Transaktionen nacheinander stattgefunden haben und damit in einer *Happened-before*-Beziehung stehen oder ob sie kausal parallel stattgefunden haben, werden Vektoruhren eingesetzt. Parallel ausgeführte Transaktionen können miteinander in Konflikt stehen. Genau wie in SYMORE ist dieses der Fall, wenn sie auf die gleichen logischen Datenelemente zugegriffen haben und mindestens eine von ihnen dieses Datenelement geändert hat. Um dieses zu erkennen, werden bei der initialen Ausführung einer Transaktion deren Lese- und Schreibmenge bestimmt und zusammen mit dem Vektoruhr-Zeitstempel und den Transaktionsdaten verteilt.

Zunächst stellen die Autoren einen pessimistischen Replikationsalgorithmus vor, der 1-Kopien-Serialisierbarkeit garantiert. Nach der lokalen Ausführung einer Transaktion werden die dabei gehaltenen lokalen Lesesperren freigegeben. Die lokalen Schreibsperren werden aber weiterhin gehalten und die Transaktion wird verteilt. Lokal ist diese Transaktion nun in einem *Pre-Commit-Zustand*. Empfängt ein Knoten eine Transaktion, wird dort anhand ihres Vektorzeit-Zeitstempels geprüft, ob sie kausal parallel zu anderen lokal vorliegenden, pre-committeten Transaktionen stattgefunden hat. Ist dieses der Fall, werden die Lese- und Schreibmengen der parallelen Transaktionen auf Konfliktoperationen hin untersucht. Wird ein Konflikt festgestellt, werden alle in Konflikt stehenden Transaktionen abgebrochen. Andernfalls werden für die neu empfangene Transaktion dort ebenfalls Schreibsperren angefordert und sie wird dort ebenfalls in den Pre-Commit-Zustand überführt.

Da Transaktionen an alle Knoten verteilt werden, werden lokal letztendlich überall die

gleichen Konflikte festgestellt und die gleichen Transaktionen abgebrochen. Eine Transaktion kann lokal definitiv festgeschrieben werden, wenn lokal bekannt ist, dass jeder Knoten aus der Replikationsgruppe alle Nachrichten zumindest bis zu dem Vektorzeit-Zeitstempel der festzuschreibenden Transaktion empfangen hat und lokal diese Transaktion in keinem Konflikt zu einer anderen steht. Dieses gilt, da das logbasierte epidemische Verteilungsprotokoll garantiert, dass Nachrichten eines Knotens in der Reihenfolge empfangen werden, in der sie auf ihrem Knoten erzeugt worden sind. Weiß also ein Knoten S_i, dass Knoten S_k alle Nachrichten bis zum Zeitstempel einer Transaktion t empfangen hat, so weiß S_i auch, dass er selbst alle Nachrichten von S_k bis zu diesem Zeitpunkt empfangen hat. S_i hat die Bestätigungsnachricht über den Empfang von t erhalten, und somit wegen des Verteilungsprotokolls auch alle Nachrichten, die S_k vor der Bestätigung erzeugt hat.

Das bisher präsentierte Verfahren kann leicht modifiziert werden, um ein optimistischeres Replikationsverfahren zu erhalten. So können die Schreibsperren nach einem Pre-Commit direkt aufgehoben werden, damit sofort mit den geänderten Daten weitergearbeitet werden kann und nicht auf die Commitentscheidung gewartet werden muss. Dieses führt allerdings genau wie in SYMORE dazu, dass Daten gelesen werden können, die noch nicht festgeschrieben wurden und es zu kaskadierenden Transaktionsabbrüchen kommen kann. Um Serialisierbarkeit der festgeschriebenen Transaktionen zu bewahren, darf eine Transaktion nur festgeschrieben werden, wenn alle Transaktionen, von denen sie abhängt, festgeschrieben wurden. Diese Bedingung gilt auch in SYMORE.

Eine weitere Modifikation des hier vorgestellten Replikationsverfahrens sieht vor, nicht mehr auf eine verteilte Commitentscheidung zu warten, sondern bei Beendigung der lokalen Transaktionsausführung deren Effekte direkt in der lokalen Datenbank festzuschreiben. Erst dann wird die Transaktion wie gehabt verteilt. Wird nun, genau wie vorher, ein Konflikt festgestellt, ist es nicht mehr möglich, die in Konflikt stehenden Transaktionen abzubrechen, da sie bereits lokal festgeschrieben worden sind. Stattdessen wird eine anwendungsspezifische Konfliktbehandlungsroutine aufgerufen, deren mögliche Funktionsweise in der Arbeit aber nicht erläutert wird. 1-Kopien-Serialisierbarkeit ist hier nicht mehr gewahrt.

Viele Aspekte des hier vorgestellten Systems, besonders der optimistischen Variante, in der Sperren nach dem Pre-Commit freigegeben werden, sind den Eigenschaften SYMOREs sehr ähnlich. Es werden weder explizite Commit- oder Abortnachrichten verschickt, noch Nachrichten, die eine Transaktionsordnung vorgeben.

Beide Systeme verwenden Lese- und Schreibmengen um Konflikte zu erkennen, die durch die parallele Ausführung von Transaktionen verursacht wurden. Kausale Abhängigkeiten werden in SYMORE dadurch erkannt, dass Verweise auf Vorgängerversionen der von einer Transaktion gelesenen oder geschriebenen Elementen zusammen mit der Transaktion selbst verteilt werden. Das hier vorgestellte System von Agrawal, Abbadi und Steinke verwendet Vektoruhren um kausal parallele Transaktionen und damit potentielle Konflikte zu erkennen. Vektoruhren enthalten pro Teilnehmer der Replikations-

gruppe einen logischen Zeitstempel. Dieser hat meistens die Form eines einfachen Zählers. Bei jeder lokalen Initiierung einer Transaktion wird der Eintrag des Knotens, der diese ausgelöst hat, um eins erhöht. Durch Vergleich zweier Vektoruhren wird erkannt, ob die zugehörigen Aktionen kausal nacheinander oder parallel ausgeführt worden sind. Ersteres ist der Fall, wenn jeder Eintrag einer Vektoruhr größer oder gleich dem entsprechenden Eintrag der anderen Vektoruhr ist (die Vektoruhr dominiert die andere), letzteres, wenn diese Bedingung nicht erfüllt ist. Vektoruhren haben den Nachteil, dass ihre Größe linear von der Größe der Replikationsgruppe abhängt. Außerdem ist es schwierig mittels Vektoruhren ein dynamisches Verlassen und Hinzukommen von Knoten zu realisieren.

Die Konflikterkennung mittels Vorgängerbeziehungen von Transaktionen benötigt weniger Daten als die Konflikterkennung mittels Vektoruhren, wenn die Replikationsgruppe größer ist als die Anzahl der Elemente, die eine Transaktion im Mittel schreibt und liest. Sei k die Größe der Replikationsgruppe und n die Anzahl der Schreib- und Leseoperationen einer Transaktion. Die Datenmenge, die zusammen mit einer Transaktion verteilt werden muss, ist hier proportional zu $k + n$. In SYMORE werden die Schreib- und Lesemengen jeweils mit der entsprechenden Vorgängertransaktion für jeden Eintrag verteilt. Hier ist die Datenmenge proportional zu $2 * n$. Dieses ist effizienter, wenn $nk < 2n$ gilt. Ist also die Replikationsgruppe größer als die Anzahl der Konfliktelemente, die eine Transaktion im Mittel schreibt und liest, muss SYMORE weniger Daten verteilen, als das System, das Vektoruhren einsetzt.

In dem System von Agrawal et al. wird eine Transaktion nur dann festgeschrieben, wenn kein Konflikt vorliegt. Ansonsten werden alle Transaktionen, die sich miteinander in Konflikt befinden, abgebrochen. In SYMORE bleibt dagegen eine der an einem Datenelement in Konflikt stehenden Transaktionen erhalten. Dieses führt insgesamt zu weniger Transaktionsabbrüchen. Weiterhin sind in SYMORE verschiedene Konfliktlösungsstrategien konfigurierbar, die bestimmen, welche Transaktionen abgebrochen werden und welche nicht. Im Gegensatz zu dem System von Agrawal et al. kann in SYMORE der Anwender einstellen, mit welcher Granularität Konflikte erkannt werden sollen. Hier wird zu der Granularität der Datenelemente keine Aussage gemacht. Insgesamt verfolgen beide Systeme einen ähnlichen Ansatz, wobei SYMORE die flexiblere Lösung bietet.

3.4 Multiversionsansatz

In [PB99] und [PB04] wird ein Algorithmus vorgestellt, um Transaktionen, die parallel, vorläufig auf lokalen Client-Kopien ausgeführt wurden, endgültig auf die zentrale Datenbank auf einem Server anzuwenden. Dabei wird Snapshot-Multiversions-Serialisierbarkeit garantiert. Dieses Serialisierbarkeitskriterium ist schwächer als 1-Kopien-Serialisierbarkeit, so dass auch Transaktionen festgeschrieben werden können, die ansonsten abgebrochen werden müssten.

In dem hier vorgestellten Modell liest eine lokale Transaktion immer den letzten lo-

kal vorliegenden, festgeschriebenen Wert eines Datenelementes. Diese Transaktion wird später auf den Server übertragen und dort erneut ausgeführt. Dabei muss sie in Bezug auf andere Transaktionen von anderen Knoten serialisiert werden. Bei Multiversions-Serialisierbarkeit ist dieses möglich, solange die Datenelemente in ihrer Schreibmenge nicht überschrieben worden sind, nachdem diese Transaktion diese Datenelemente gelesen hat. Ist eine Multiversions-Serialisierung nicht möglich, wird eine benutzerdefinierte Konfliktlösungsprozedur aufgerufen. Diese enthält als Eingabe die Lese- und Schreibmenge der zu serialisierenden Transaktion sowie die aktuellen Werte der Datenelemente der Serverdatenbank. Die Konfliktlösungsprozedur berechnet nun eine neue Schreibmenge (Datenelemente und Werte), die auf die Datenbank angewendet wird.

In [PB99] wird knapp auf eine Erweiterung des vorgestellten erweiterten Client-Server-Systems zu einem Peer-to-Peer-System eingegangen. Allerdings wird nicht ausreichend beschrieben, wie ein benutzerdefinierter Konfliktlösungsalgorithmus aussehen kann, so dass weiterhin Multiversions-Serialisierbarkeit gewahrt bleibt. Transaktionen können in beliebiger Reihenfolge von den verschiedenen Knoten empfangen werden. Somit muss auch der Konfliktlösungsalgorithmus für eine Transaktion auf unterschiedliche Eingabemengen angewendet werden, was im Normalfall unterschiedliche Ergebnisse produziert. SYMORE verwendet einen Konfliktlösungsalgorithmus, um Konflikte durch Abbruch bestimmter kausal paralleler Transaktionen zu bestimmen. Eventuell wäre es möglich, auch einen Konfliktlösungsalgorithmus zu entwickeln, der dieses schwächere Serialisierbarkeitskriterium einsetzt. Die dazu notwendigen Lese- und Schreibmengen der Transaktionen stehen dem Konfliktlösungsalgorithmus zur Verfügung.

3.5 IceCube

IceCube [KRSD01] ist ein generisches, log-basiertes Objektreplikationssystem, dessen Fokus auf der Zusammenführung (reconciliation) verteilt stattgefundener Transaktionen liegt. Seine Aufgabe ist es vorläufige Updates, die initial, autonom und eventuell parallel auf lokalen Kopien der Objekte ausgeführt wurden, auf einem Server zusammenzuführen. Dabei müssen bestimmte Bedingungen zwischen den Transaktionen und für die Zustände der Objekte gewahrt bleiben. Ähnlich wie in Bayou und MobiSnap sind Updates Transaktionen mit Vor- und Nachbedingungen. Diese Transaktionen werden nicht in einer festgelegten Ordnung (z.B. Zeitstempelreihenfolge) ausgeführt. Es wird stattdessen eine in Bezug auf Transaktionsabbrüche optimale Reihenfolge berechnet. Zusätzlich zu dynamischen Ordnungsbedingungen (den Vorbedingungen) werden dazu statische herangezogen. Statische Bedingungen hängen nicht von dem momentanen Zustand der replizierten Objekte ab und gelten immer zwischen bestimmten Transaktionen. Sie unterscheiden sich somit von den dynamischen Bedingungen, deren Erfüllung immer gegen einen bestimmten Zustand der replizierten Objekte getestet wird. Eine statische Bedingung ist beispielsweise, dass zwei Transaktionen kommutativ sind oder sich gegenseitig ausschließen, eine dynamische, dass der Wert eines Feldes des replizierten Objektes sich

in einem bestimmten Wertebereich befindet.

Der hier vorgestellte Algorithmus bestimmt die optimale Transaktionsordnung in drei Phasen. In der ersten Phase werden alle möglichen Reihenfolgen bestimmt, so dass die statischen Bedingungen zwischen Transaktionen erfüllt sind. Dadurch verringert sich die Anzahl möglicher Ausführungsreihenfolgen. Gegen die so bestimmten Reihenfolgen werden in einer zweiten Phase die dynamischen Bedingungen geprüft, indem die Vorbedingungen gegen Kopien der replizierten Objekte ausgeführt werden. Aus allen Reihenfolgen, die nun sowohl die statischen, als auch die dynamischen Bedingungen erfüllen, wird in der dritten Phase die, im Hinblick zu nötigen Transaktionsabbrüchen optimale, ausgewählt.

Durch die Verwendung von statischen und dynamischen Bedingungen für die Wiederausführung lokaler Transaktionen, werden hier Ansätze der syntaktischen und der semantischen Konfliktlösung miteinander verbunden. Der Ansatz diese beiden Konfliktlösungsverfahren zu kombinieren ist sehr interessant. Allerdings ist die Komplexität und benötigte Rechenzeit des IceCube-Algorithmus sehr hoch, so dass er in dieser Form für den Einsatz auf relativ leistungsschwachen mobilen Geräten nicht geeignet ist. Auch ist nicht klar, wie dieser Algorithmus in einem Peer-to-Peer-System wie SYMORE eingesetzt werden kann. Es müsste sichergestellt sein, dass IceCube auf jedem Knoten dieselbe Ordnung für die Ausführung der Transaktionen bestimmt.

3.6 Joyce

Joyce [OM05] ist ein Framework, das die Entwicklung von mobilen, sporadisch verbundenen, kollaborierenden Anwendungen vereinfacht. Es nutzt optimistische Replikation und basiert auf dem IceCube-Scheduler zur Konfliktlösung. Sein Fokus ist nicht Datenbankreplikation, sondern Objektreplikation. Die folgenden vier Aktivitäten werden mit Joyce von der Applikation in das Framework verlagert: Modellierung, Kommunikation, Konfliktlösung und Konsistenzwahrung. Mittels Joyce können Anwendungen auf mobilen Geräten auf Kopien der gleichen Objekte zugreifen und diese gleichzeitig und asynchron ändern. Modifikationen sind *Aktionen*, die auf den Objekten ausgeführt werden. Jede Aktion besteht nicht nur aus der Operation auf dem verteilten Objekt, sondern zusätzlich aus einer Menge von Randbedingungen.

Randbedingungen sind semantische Invarianten einer Anwendung. Diese müssen gewahrt werden, wenn parallele Änderungen zusammengeführt werden. Es gibt Log-Randbedingungen und Objekt-Randbedingungen.

Objekt-Randbedingungen beschreiben die Nebenläufigkeitssemantik der verteilten Objekte selbst, wie z.B. die Beziehungen zwischen Klassen von Aktionen (den statischen Randbedingungen in IceCube). Log-Randbedingungen dagegen drücken Nutzerintentionen und Anwendungssemantik aus und beschreiben die Beziehungen zwischen Exemplaren von Aktionen (den dynamischen Bedingungen). Mögliche Objekt-Randbedingungen sind „ist kommutativ“, „hilft“, „behindert“, „ermöglicht“ und „ver-

hindert“. Log-Randbedingungen werden in zwei Kategorien unterteilt: Gruppierungs-Randbedingungen (Paketgruppierung oder alternative Gruppierung) und Ordnungs-Randbedingungen (starke oder schwache Ordnungen zwischen Exemplaren von Aktionen). Zusätzlich zu Aktionen gibt es *Meta-Aktionen*, die transparent von Joyce generiert werden. Meta-Aktionen beschreiben Bedingungen zwischen Aktionen wie z.B. „starke Abhängigkeit“. Aktionen und Meta-Aktionen werden in einem Log gespeichert und epidemisch verteilt.

Eine Anwendung, die mit dem Joyce-Framework gebaut wurde, sieht immer eine lokale Sicht der verteilten Objekte. Diese Sicht wird mittels des IceCube-Schedulers aus den Aktionen unter Berücksichtigung aller Randbedingungen berechnet. Diese lokale Sicht ist wie in allen optimistischen Replikationssystemen nicht immer konsistent mit der globalen Sicht. Es wird allerdings Eventual Consistency garantiert. Dazu werden Aktionen schließlich wie in IceCube auf einer Primärkopie ausgeführt. Dort wird die endgültige Entscheidung getroffen, ob diese Aktionen erfolgreich festgeschrieben werden können und in welcher Reihenfolge dieses geschieht. Diese Entscheidung wird schließlich an die anderen Kopien verteilt.

Dieses System geht einen Schritt weiter als SYMORE, indem es ein Framework für die Konsistenzhaltung verteilter Objekte liefert und explizit auf die Anwendungsentwicklung ausgerichtet ist. Aufgabe von SYMORE ist es hauptsächlich, Daten konsistent zu halten. Es wird nicht primär darauf eingegangen, wie Anwendungen mit den Eigenschaften, die sich durch optimistische Replikation ergeben, umgehen, während dieses das Hauptziel von Joyce ist. Einige replikationsspezifische Erweiterungen in Hinblick auf zentralisierte Datenbanksysteme bietet auch SYMORE. Insbesondere bietet es der Anwendung die Möglichkeit, verschiedene Beobachter zu installieren. Beobachter können installiert werden, um benachrichtigt zu werden, wenn für eine lokal initiierte Transaktion endgültig entschieden wurde, ob sie festgeschrieben oder abgebrochen wird. Ein anderer Typ von Beobachter wird benachrichtigt, wenn sich der (vorläufige) Datenbankzustand auf Grund des Empfangs von Transaktionen anderer Replikationsmanager ändert.

Da Joyce auf Ice-Cube aufbaut, werden dort syntaktische und semantische Kriterien zur Konflikterkennung und Lösung herangezogen. Genau wie Ice-Cube und anders als SYMORE wird hier allerdings eine ausgezeichnete Kopie als Primärkopie benötigt, um endgültige Commitentscheidungen für Updates treffen zu können und so letztendlich Eventual Consistency zu wahren.

4 Replikation in Symore

In diesem Kapitel wird die Funktionsweise des replizierten Datenbanksystems SYMORE vorgestellt. Zunächst wird der grobe Systemablauf erläutert, bevor anschließend auf alle Teilbereiche des Systems detailliert eingegangen wird.

4.1 Systemablauf

Wie bei allen Datenbanksystemen interagieren Nutzer mit dem vorliegenden System, indem sie Transaktionen, die Datenelemente lesen und schreiben, darauf ausführen. Es gibt keinen zentralen Transaktionsmanager, der Transaktionen entgegennimmt und dafür sorgt, dass deren Operationen in Operationen auf die entsprechenden Kopien übersetzt werden wie bei manchen zentralisierten Replikationssystemen. Stattdessen initiiert ein Anwender Transaktionen direkt auf seinem lokalen mobilen Gerät. Diese Geräte kommunizieren miteinander, um die Änderungsoperationen, die an lokalen Kopien vorgenommen worden sind, asynchron miteinander auszutauschen und um letztendlich einen global konsistenten Datenbankzustand zu erlangen.

Wie in Abschnitt 2.2 erläutert wurde, befindet sich auf jedem mobilen Gerät ein lokales Replikationssystem, bestehend aus *Replikationsmanager (RM)*, *lokalem Datenbanksystem* und *Verteilungskomponente*. Eine Instanz eines solchen Systems ist in Abbildung 4.1 dargestellt. Die lokale Datenbank enthält Kopien aller logischen Datenelemente in diesem verteilten System. Der RM nimmt Nutzertransaktionen entgegen und führt diese auf der lokalen Datenbank aus. Anschließend sorgt er dafür, dass die durch diese Transaktionen verursachten Änderungen an den replizierten Daten auch den anderen RM der Replikationsgruppe bekannt gemacht werden. Dazu nutzt er eine Verteilungskomponente, die dafür sorgt, Nachrichten effizient an alle Geräte der Replikationsgruppe zu verteilen. Auf einem Gerät können prinzipiell mehrere RM ausgeführt werden. Im Folgenden wird aber davon ausgegangen, dass sich ein RM auf genau einem Gerät befindet.

Die generelle Funktionsweise des Replikationssystems ist in Abbildung 4.2 dargestellt und wird nun erläutert.

Von Nutzern initiierte Transaktionen werden als *lokale Transaktionen* von einem lokalen RM ausgeführt. Sie können nur auf lokale Datenelemente der lokalen Datenbank zugreifen. Eine Transaktion wird atomar und in Isolation zu anderen lokalen Transaktionen ausgeführt, allerdings ohne Synchronisation mit Transaktionen auf anderen RM. Es handelt sich hier also um ein optimistisches Synchronisationsverfahren.

Ein *lokales Commit* beendet schließlich eine lokale Transaktion. Für diese wurden Sper-

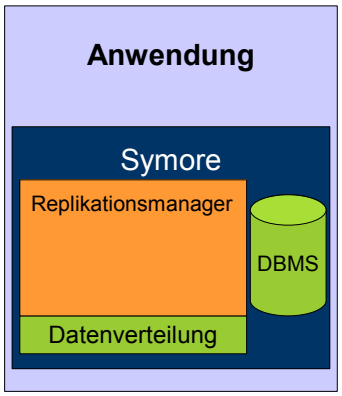

Abbildung 4.1: Architektur von SYMORE

ren von dem lokalen Datenbankmanager gehalten, die nun aufgehoben werden. Somit werden die Effekte der Transaktion lokal sichtbar. Dieses ist in b) in Abbildung 4.2 dargestellt. In a) haben zunächst alle lokalen Datenbanken A, B und C den gleichen Zustand, hier repräsentiert durch die zwei Smileys mit gleichem Gesichtsausdruck. In b) werden diese, durch je eine lokal initiierte Transaktion auf A und B, parallel und unabhängig voneinander geändert. Nachdem beide Transaktionen ein lokales Commit ausgeführt haben, sind diese Änderungen für alle lokalen Nutzer sichtbar.

Indem Transaktionsergebnisse möglichst schnell lokal sichtbar werden, wird die Verfügbarkeit der Daten erhöht. Es muss nicht erst auf die Aktualisierung aller anderen Kopien der geänderten Datenelemente und auf ein verteiltes Commit gewartet werden. In MANETs, wo Netzwerkverbindungen sehr unzuverlässig sind und sich einzelne Geräte häufig kurzzeitig nicht in Kommunikationsreichweite befinden, kann dieses ein langwieriger Vorgang sein. Während dieses Zeitraumes blieben die geänderten Datenelemente gesperrt und Transaktionen, die darauf zugreifen wollen, würden verzögert. Eine Transaktion, die ein lokales Commit ausgeführt hat, ist allerdings noch nicht endgültig festgeschrieben. Sie muss, wie im Folgenden erläutert wird, eventuell wieder zurückgenommen werden. Dieses kann kaskadierende Transaktionsabbrüche zur Folge haben, falls diese Transaktion Grundlage weiterer Transaktionen war.

In der Literatur wird der Zustand einer Transaktion, die ein solches lokales Commit ausgeführt hat, entweder als *Pre-Commit* (z.B. [AAS97]) oder als *tentative* (z.B. [SS05]) bezeichnet. In beiden Fällen bedeutet es, dass die lokale Transaktionsausführung zwar abgeschlossen ist, aber noch nicht endgültig über das Ergebnis der Transaktion entschieden wurde. Im Folgenden wird hier der Begriff „Pre-Commit“ für diesen Transaktionszustand verwendet, wohl wissend, dass hier nicht genau das Gleiche gemeint ist, wie bei der Verwendung dieses Begriffs im Rahmen von mehrphasigen Commitprotokollen (z.B. 2PC).

Nach dem lokalen Commit werden nun alle weiteren Kopien der Datenelemente asynchron aktualisiert. Die Informationen über die Schreiboperationen der lokalen Transak-

tionen werden dazu über die Verteilungskomponente an alle anderen RM der Replikationsgruppe verteilt. Dabei werden nicht die neuen Werte der geänderten Datenelemente übertragen, sondern die Schreiboperationen der lokalen Transaktion zusammen mit zusätzlichen Verwaltungsdaten. Diesen wird der global eindeutige Zeitstempel der initialen lokalen Transaktion hinzugefügt. Dieser enthält als einen Bestandteil den Zeitpunkt der Ausführung des lokalen Commits. Eine Transaktion ist somit eindeutig bezeichnet und kann global eindeutig in Bezug auf andere Transaktionen geordnet werden (genaueres siehe 4.2.2).

Die Änderungen durch die lokalen Transaktionen werden so nach und nach überall bekannt. In c) ist dargestellt, wie die Änderungen der auf *A* initiierten Transaktion an *B* und *C* verteilt werden.

Transaktionen ändern lokal und autonom lokale Datenelemente. Dadurch kann es vorkommen, dass verschiedene lokale Transaktionen auf unterschiedlichen RM unterschiedliche Kopien der gleichen logischen Datenelemente parallel ändern und so Konflikte verursachen. Diese Konflikte sind im Normalfall unerwünscht und müssen erkannt werden. Dieses geschieht lokal von jedem RM, sobald die Daten über die einen Konflikt verursachenden Transaktionen empfangen worden sind. In c) ist solch ein Konflikt auf *C* dargestellt. Eine Ursprungsversion der beiden Smileys ist parallel auf zwei RM geändert worden. Damit gibt es zunächst zwei neue Versionen.

Solche Konflikte müssen aufgelöst werden, um einen eindeutigen, konfliktfreien Datenbankzustand zu erhalten. Die Schreiboperationen von empfangenen Updatetransaktionen, die auf einem anderen RM ihr lokales Commit ausgeführt haben, werden beim Empfänger als eine neue Transaktion behandelt. Zur Unterscheidung zu den lokal von einem Nutzer initiierten *lokalen* Transaktionen werden sie als *Updatetransaktionen* bezeichnet. Um Konflikte aufzulösen entscheidet ein *Konfliktlösungsalgorithmus* (siehe Abschnitt 4.4), der lokal von jedem RM ausgeführt wird, welche Transaktionen diesen vorläufigen Datenbankzustand bilden. Dieses ist in d) dargestellt, wo auf RM *C* aus den zwei in Konflikt stehenden Transaktionen eine ausgewählt wird. Die ausgewählten Transaktionen werden hier als *aktiviert* bezeichnet, die Transaktionen, deren Effekte nicht an dem momentanen Datenbankzustand beteiligt sind, als *deaktiviert*. Diese Zustände sind nicht einem Commit oder Abort gleichzusetzen, da sie nur vorläufig sind und sich wieder ändern können. Eine weitere Updatetransaktion kann empfangen werden, die mit einer momentan aktivierten Transaktion in Konflikt steht. In diesem Fall kann der Konfliktlösungsalgorithmus entscheiden, die neu empfangene Updatetransaktion zu aktivieren und somit die andere zu deaktivieren. Verschiedene lokale Datenbanken können demnach zu einem gegebenen Zeitpunkt in unterschiedlichen Zuständen vorliegen, wenn ihre RM bisher unterschiedliche Transaktionen empfangen haben. RM, die Kenntnis von der gleichen Menge an Transaktionen haben, berechnen aber mittels des Konfliktlösungsalgorithmus die gleiche Menge an Transaktionen, die aktiviert sind und haben somit einen gleichen Datenbankzustand.

Wenn lokal eine endgültige Commitentscheidung für eine Transaktion getroffen wird,

muss sichergestellt sein, dass diese Transaktion von jedem RM festgeschrieben wird und ihre Reihenfolge überall gleich ist. Alle RM haben nun den gleichen Zustand. Dieses ist in e) dargestellt. Die endgültige Commitentscheidung kann mittels unterschiedlicher *Commitstrategien* herbeigeführt werden. Diese Strategien werden in Kapitel 4.5 vorgestellt.

Kommunikation zwischen verschiedenen Geräten ist in dem vorliegenden System nur nötig, um Ergebnisse lokaler Transaktionen zu verteilen und in geringem Maße für eine bestimmte Commitstrategie. Es findet keine Kommunikation statt um Sperren anzufordern oder um verteilte Commitentscheidungen für einzelne Transaktionen zu treffen.

Aus dem hier beschriebenen Ablauf der Replikation ergeben sich die folgenden Teilbereiche, die in den nächsten Kapiteln genauer beschrieben werden:

Uhrensynchronisation: Es wird Echtzeit verwendet, um verteilt auftretende Ereignisse wie z.B. Transaktions-Pre-Commits oder Commitzeitpunkte lokal in eine global eindeutige Ordnung bringen zu können. Dazu ist es nötig, die Zeitdifferenzen zwischen den Uhren der Geräte, auf denen RM ausgeführt werden, zu ermitteln. In der vorliegenden Arbeit wird ein Synchronisationsprotokoll vorgestellt, sowie ein Verfahren, um trotz eines Restsynchronisationsfehlers eine global eindeutige, an der Echtzeit orientierte Ordnung der in diesem verteilten System auftretenden Ereignisse zu ermöglichen.

Konflikterkennung: Auf allen RM werden lokale Transaktionen ausgeführt, die lokale Datenelemente manipulieren. Da diese Ausführung optimistisch, ohne globale Nebenläufigkeitskontrolle erfolgt, können verschiedene Transaktionen auf verschiedenen lokalen RM die gleichen logischen Datenelemente kausal parallel manipuliert haben. Es können also Konflikte entstanden sein. Dieses muss zuverlässig erkannt werden, um das Ziel eines letztendlich konsistenten replizierten Datenbanksystems zu erreichen und ein unkontrolliertes Auseinanderdriften der lokalen Datenbankzustände zu verhindern.

Konfliktlösung: Ist ein Konflikt zwischen Transaktionen erkannt worden, muss dieser in irgendeiner Form gelöst werden. Die Konfliktlösung besteht hier darin, dass eine Teilmenge aller sich im Pre-Commit-Zustand befindlichen Transaktionen bestimmt wird, deren Transaktionen untereinander konfliktfrei sind. Die Ausführung dieser Transaktionen bildet den neuen, vorläufigen Datenbankzustand. Es können unterschiedliche Algorithmen verwendet werden, die anhand bestimmter Kriterien bestimmte Transaktionen für diese Teilmenge auswählen. Mögliche Kriterien sind der Transaktionszeitstempel oder die Id des die Transaktion erzeugenden RM.

Commit: Jeder RM muss für jede Transaktion schließlich entscheiden, ob sie endgültig festgeschrieben oder abgebrochen wird. Dabei muss sichergestellt sein, dass jeder RM der Replikationsgruppe zu den gleichen Entscheidungen kommt, um eine konsistente Datenbanksicht und Eventual Consistency zu gewährleisten.

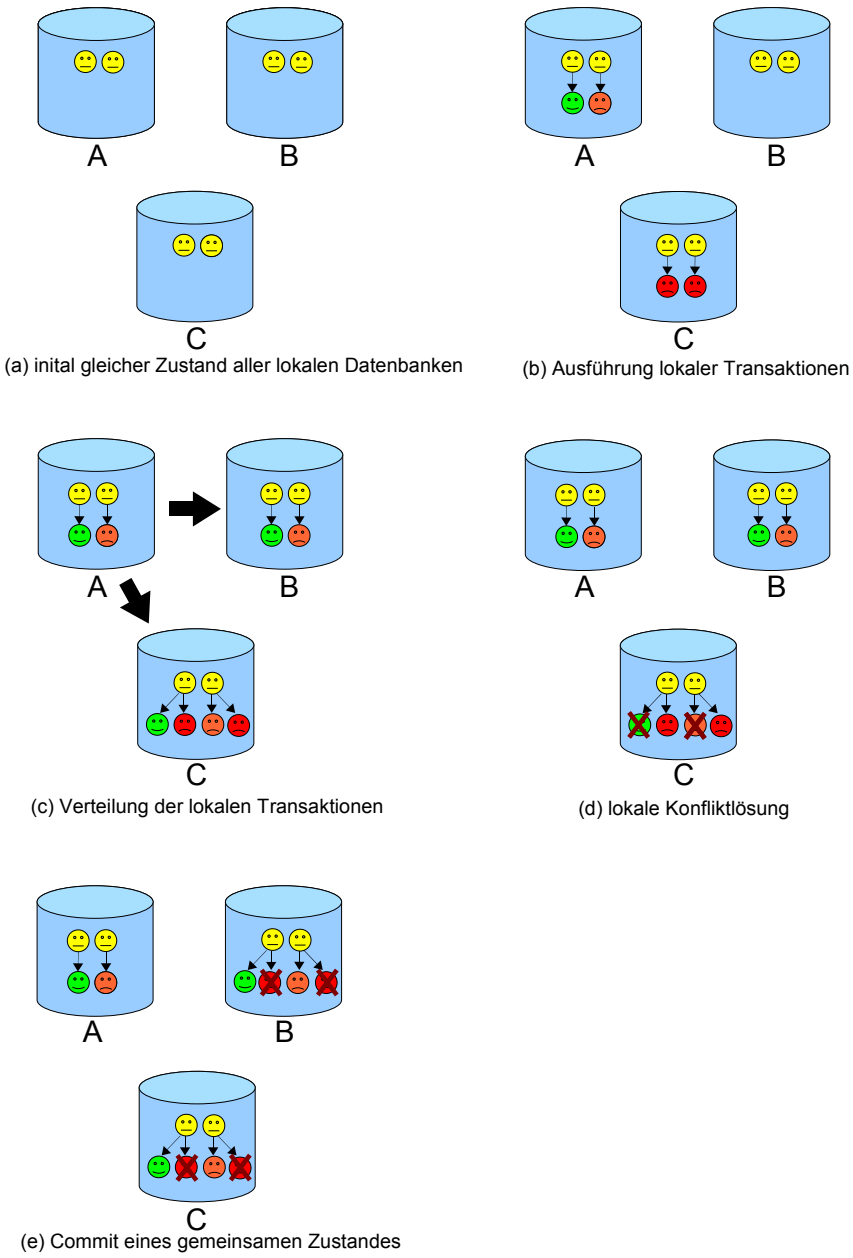

Abbildung 4.2: Basisworkflow des Replikationssystems SYMORE

Datenverteilung: Die lokal ausgeführten Transaktionen müssen zu den anderen RM gelangen. Dazu sollen Übertragungsverfahren eingesetzt werden, die speziell an die Eigenschaften von MANETs angepasst sind. Für den im Rahmen dieser Arbeit entwickelten Prototyp wird dazu eine Datenübertragungs- und Verteilungskomponente eingesetzt, die von Y. Schröder im Rahmen seiner Examensarbeit entwickelt wurde [Sch05b].

Gruppenverwaltung: Für die Commitstrategien ist es nötig zu wissen, welche RM Teil der Replikationsgruppe sind. Das Beitreten und Verlassen der Gruppe muss dabei mit der Ausführung der Commitprozesse abgestimmt sein. Außerdem ist es nötig, einen RM aus der Replikationsgruppe ausschließen zu können, wenn angenommen wird, dass dieser ausgefallen ist. Dabei ist wichtig, dass alle verbleibenden Gruppenmitglieder konsistent zu der gleichen Entscheidung gelangen.

4.2 Uhrensynchronisation

Für einige Teilbereiche des hier vorgestellten Systems ist es nötig, bestimmte lokal aufgetretene Ereignisse auch global konsistent in der Reihenfolge zu ordnen, in der sie ein externer Beobachter wahrgenommen hätte. Ein solches Ereignis ist beispielsweise das lokale Pre-Commit einer Transaktion. Konflikte zwischen Transaktionen können mittels einer totalen Ordnung entschieden werden. Wird als totale Ordnung die Echtzeit verwendet, so kann definiert werden, dass im Konfliktfall immer die ältere Transaktion gewinnt. Mit ihr in Konflikt stehende jüngere Transaktionen werden abgebrochen. Andere Ereignisse sind Commit-Ereignisse. Auch diese müssen global konsistent zu anderen Commit-Ereignissen oder Transaktionen geordnet werden.

Da die Ereignisse, die hier betrachtet werden, extern verursacht sind, soll deren lokale Ordnung auch mit der extern beobachteten Ordnung übereinstimmen. Nutzer haben die Erwartung, dass Ereignisse (z.B. Transaktionsausführungen), die (auch auf verschiedenen Geräten) hintereinander stattgefunden haben, auch in dieser Reihenfolge geordnet werden.

Logische Uhren, wie z.B. Vektoruhren können nicht eingesetzt werden. Vektoruhren definieren eine Kausalordnung auf Ereignissen, jedoch keine totale Ordnung. Sie betrachten Ereignisse, die nicht in demselben Prozess stattgefunden haben oder durch eine Sende-Empfangs-Beziehung miteinander in Verbindung stehen, als gleichzeitig. Es wird aber eine global konsistente Aussage darüber benötigt, welches der kausal gleichzeitigen Ereignisse real vor einem anderen stattgefunden hat.

In dem vorliegenden System wird also die Echtzeit als Ordnungskriterium für Ereignisse verwendet. Dabei besteht das Problem, dass in mobilen Netzen mit vielen verschiedenen autonomen Geräten nicht von synchronisierten Uhren ausgegangen werden kann.

Die lokalen Uhren der verschiedenen Knoten müssen demnach zunächst intern, also untereinander synchronisiert werden. Eine externe Synchronisation zu einer Referenz-

uhr, die beispielsweise koordinierte Weltzeit liefert, ist nicht nötig, da eine Replikationsgruppe ein geschlossenes System darstellt.

Beziehungen zwischen Uhren werden durch zwei Parameter gekennzeichnet: *Versatz* (engl. *skew*) und *Drift*. Der Versatz gibt an, wie groß der Betrag ist, um den sich die Werte zweier Uhren zu einem gegebenen Zeitpunkt unterscheiden. Die Drift gibt an, wie weit sich der Wert einer Uhr von einem Referenzwert über die Zeit wegbewegt, also wie schnell eine Uhr ihre Zeit weiterzählt im Vergleich zu einer Referenzuhr. Die Drift von Uhren hängt von verschiedenen physikalischen Faktoren ab, wie dem verwendeten Quarz oder der Umgebungstemperatur. Laut [CDK00] beträgt die Drift einer typischen PC-Uhr ungefähr 10^{-6} Sekunden/Sekunde. Das entspricht etwa einer Sekunde pro 11.6 Tagen. Trotz der besonderen Bedeutung der Drift bei lang laufenden Anwendungen wird sie für den hier entwickelten Prototypen vernachlässigt.

Im Folgenden wird darauf eingegangen, wie mit dem Versatz zwischen den einzelnen Uhren umgegangen werden kann. Die Uhren der jeweiligen Systeme können zwar einander angeglichen werden, eine exakte Übereinstimmung aller Uhren wird aber nie erreicht werden (siehe [LL84]). Selbst in zentralisierten Systemen mit einem zentralen Zeitgeber wird eine perfekte Synchronisation wegen der nichtdeterministischen Laufzeiten der Nachrichten, die zur Synchronisation ausgetauscht werden müssen, nicht erreicht.

Eine einfache Möglichkeit Uhren annähernd zu synchronisieren wäre die Ausstattung eines jeden Gerätes mit einem GPS-Empfänger oder einer Funkuhr. Mittels GPS kann die lokale Systemuhr mit einer Genauigkeit von etwa 340ns [Dan00] gegenüber einer Referenz-Weltzeituhr synchronisiert werden. Eine andere Möglichkeit stellt der Einsatz von Algorithmen bzw. Protokollen zur Uhrensynchronisation dar. Für das vorliegende System sollte solch ein Algorithmus leichtgewichtig und für den Einsatz in MANETs geeignet sein. Synchronisationsalgorithmen werden beispielsweise in [Cri89], [Rö01] oder [MBT04] beschrieben. Ein einfaches Zeitsynchronisationsprotokoll für MANETs, das hier eingesetzt wird, wird in 4.2.3 vorgestellt. Wie im Folgenden erläutert wird, muss der eingesetzte Synchronisationsalgorithmus eine konservative Schätzung der Ungenauigkeit des ermittelten Versatzes zwischen zwei Uhren liefern, um eine korrekte Totalordnung von Ereignissen sicherzustellen.

Wie in der Einleitung beschrieben, wird im vorliegenden System ein Synchronisationsverfahren eingesetzt, das die Datenstruktur der *SkewMatrix* verwendet. Anstatt die Werte der jeweiligen Systemuhren zu verändern, wird von jedem Knoten der Replikationsgruppe die Zeitdifferenz, die dessen lokale Uhr zu jeder lokalen Uhr eines anderen Knotens in der Replikationsgruppe hat, ermittelt und in dieser Matrix gespeichert. Dieses hat den Vorteil, dass die einzelnen RM sich nicht darauf einigen müssen, welche Zeit die „korrekte“ Referenzzeit ist. Würden zwei Gruppen, die sich intern auf eine gemeinsame Zeit geeinigt haben, zusammenkommen, müssten diese ansonsten entscheiden, welches nun die „korrekte“ Zeit ist.

Unter anderem aus diesem Grund ist auch die Verwendung von NTP [Mil92] oder SNTP [Mil06] zur Synchronisation der Uhren in SYMORE nicht sinnvoll. Durch NTP und

SNTP würde direkt die Systemzeit der Uhren angepasst. Auch synchronisieren NTP und SNTP mit einem zentralen Zeitgeber. Dieses ist auf Grund der dezentralen Struktur eines MANETs kein geeignetes Verfahren.

Wie beschrieben ist eine Uhrensynchronisation immer mit einem Restfehler behaftet. Dadurch kann nicht ohne weiteres sichergestellt werden, dass jeder RM alle Ereignisse in der gleichen Reihenfolge ordnet. Folgendes Beispiel soll dieses illustrieren:

Gegeben seien drei Replikationsmanager A, B, C. As Uhr habe einen realen Zeitunterschied von 60 Sekunden zu Bs Uhr und von 40 Sekunden zu Cs Uhr. Bs Uhr habe einen realen Zeitunterschied zu Cs Uhr von -20 Sekunden.

Könnten A, B und C ihre jeweiligen Zeitunterschiede paarweise exakt bestimmen, so gäbe es keine Probleme. Jeder RM könnte bei Erhalt einer Nachricht von einem anderen den Zeitstempel der Nachricht in seine lokale Zeit umrechnen indem er den jeweiligen Zeitunterschied addiert. Alle lokal erzeugten und empfangenen Ereignisse würden somit auf jedem RM gleich geordnet.

Die Zeitunterschiede zwischen den einzelnen RM können allerdings nicht ganz exakt bestimmt werden, sondern sind mit einem Fehler von maximal δ_{max} behaftet. Zur Illustration wird hier ein recht großer Fehler von jeweils +- 5 Sekunden verwendet. A habe einen gemessenen Zeitunterschied zu B von 65 Sekunden, zu C von 35 Sekunden. Der Zeitunterschied von B zu C sei mit 15 Sekunden bestimmt worden. Der Einfachheit halber seien die Zeitdifferenzen zwischen zwei RM symmetrisch angenommen. Jeder erzeuge nun ein Ereignis. A erzeuge Ereignis I zum lokalen Zeitpunkt 01:11:00 Uhr, B Ereignis III zum lokalen Zeitpunkt 01:11:55 und C Ereignis II zum lokalen Zeitpunkt 01:11:30, wie in Tabelle 4.2 fett dargestellt. Diese Tabelle zeigt außerdem die lokalen Zeiten, in die diese Ereignisse auf den jeweils anderen RM, mittels der vorher bestimmten Zeitdifferenzen, umgerechnet wurden. Man sieht hier, dass A die Ereignisse in der Folge III, II, I ordnen würde, B in der Folge II, III, I und C in der Folge II, I, III. Somit hätte jeder RM eine andere Ordnung bestimmt.

Dieses Problem, dass Ereignisse auf Grund von Synchronisationsungenauigkeiten auf verschiedenen Geräten unterschiedlich geordnet werden, ist bereits in einer früheren Arbeit, die das Synchronisationsverfahren mittels der SkewMatrix verwendet hat, beschrieben worden [Rö06]. Dort wurde die Wahrscheinlichkeit, dass solch ein Fehler auftritt, allerdings als vernachlässigbar eingestuft. In dem vorliegenden System ist die global eindeutige Ordnung von Ereignissen essentiell für die Korrektheit. Deshalb wurde ein einfaches Verfahren entwickelt, um trotz der oben beschriebenen Synchronisationsungenauigkeit eine global konsistente, an Echtzeit orientierte Ordnung von Ereignissen zu erreichen. Dieses Verfahren wird im folgenden Abschnitt vorgestellt.

4.2.1 Rasterzeit

Um sicherzustellen, dass jeder RM alle Ereignisse in der gleichen Reihenfolge ordnet, wird ein hier „*Rasterzeit*“ (*grid time*) genanntes Verfahren verwendet. Dabei wird die Auflösung der Zeit absichtlich auf die, durch δ_{grid} gegebene Auflösung herabgesetzt. δ_{grid}

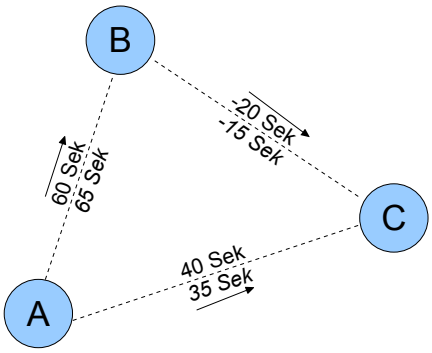

Abbildung 4.3: Knoten und die Zeitdifferenzen zwischen ihren Uhren (kursiv die gemessene Zeitdifferenz)

	I	II	III
A	**01:11:00**	01:10:55	01:10:50
B	01:12:05	01:11:45	**01:11:55**
C	01:11:35	**01:11:30**	01:11:40

Tabelle 4.2: lokale Zeiten der Ereignisse I, II und III auf den Knoten A, B, C

muss so gewählt werden, so es größer als δ_{max}, der maximal erlaubten Ungenauigkeit des bekannten Zeitunterschiedes zweier Uhren, ist.

Jeder Echtzeit-Zeitpunkt eines Ereignisses wird auf den nächsten, durch δ_{grid} teilbaren Wert abgerundet. Wird ein Ereignis empfangen, wird sein Zeitpunkt in die neue lokale Zeit umgerechnet, indem der vorher bestimmte Zeitunterschied zwischen der Sende- und der Empfangsuhr addiert wird. Dieser Zeitunterschied ist mit einem Fehler von maximal δ_{max} behaftet. Wird der nun erhaltene Zeitwert erneut in die Rasterzeit umgerechnet, also auf den nächsten, durch δ_{grid} teilbaren Wert abgerundet, so ist sichergestellt, dass der Fehler eliminiert ist. In dem seltenen Fall, dass zwei Ereignisse so kurz aufeinander folgen, dass ihre Zeitdifferenz kleiner als δ_{grid} ist, kann es passieren, dass beide dieselbe Rasterzeit zugewiesen bekommen. In diesem Fall wird anhand weiterer Kriterien, wie der RM-Id oder einer Sequenz-Id für eine totale Ordnung dieser Ereignisse gesorgt. Ereignisse, die innerhalb eines Zeitraumes von δ_{grid} auftreten, werden demnach eventuell nicht in der Reihenfolge geordnet, in der sie ein externer Beobachter wahrgenommen hätte. Für Ereignisse, die weiter als δ_{grid} auseinander liegen, stimmen diese beiden Ordnungen aber in jedem Fall überein.

Die Korrektheit dieses Verfahrens soll im Folgenden gezeigt werden.

Korrektheitsbeweis

Gegeben seien zwei Knoten A und B. t_A und t_B seien die ganzzahligen Werte der lokalen Uhren von A bzw. B zu einem gegebenen Zeitpunkt. $\text{offset}_{A,B}$ sei die tatsächliche Zeitdifferenz zwischen diesen beiden Uhren.

$$\text{Es gilt also: } t_A = t_B + \text{offset}_{A,B}\,. \tag{4.1}$$

Die Funktion $\text{grid}(t)$ berechnet für einen Zeitwert t die Rasterzeit, indem der Zeitwert t auf den nächst niedrigeren Rasterzeitwert gesetzt wird. Sie ist definiert als

Definition. $\text{grid}(t) = \lfloor \frac{t}{\delta_{\text{grid}}} \rfloor \cdot \delta_{\text{grid}}$

δ_{grid} ist dabei die Auflösung der Rasterzeit. Es wird vorausgesetzt, dass $\delta_{\text{grid}} > \delta_{max}$ ist, wobei δ_{max} der maximale Fehler bei der Bestimmung der Zeitdifferenz zweier Knoten ist.

Die Zeitdifferenz $\text{offset}_{A,B}$ selbst ist unbekannt. Bekannt ist nur

$$\text{offset}_{A,B_{measured}} = \text{offset}_{A,B} + \delta \tag{4.2}$$

also die Summe aus Zeitdifferenz und einem Fehler δ, mit $\delta \leq \delta_{max}$.

Wie oben definiert, beschreiben die Zeitpunkte t_A und t_B den gleichen Echtzeit-Zeitpunkt in dem System ihrer jeweiligen lokalen Uhr. Zu zeigen ist, dass diese Zeitpunkte nach Umrechnung in die lokale Rasterzeit einer Uhr, denselben Rasterzeit-Zeitpunkt in dem System dieser Uhr beschreiben. Jeder Knoten kann also alle Rasterzeit-Zeitpunkte von Ereignissen, die auf anderen Knoten aufgetreten sind, unter Wahrung der globalen Ordnung in die lokale Rasterzeit umrechnen. Dieses gelingt, solange die ermittelte Zeitdifferenz mit einem Fehler kleiner als δ_{max} behaftet ist.

Beweis.

$$\begin{aligned}
\text{grid}(t_A) &= \text{grid}(t_B + \text{offset}_{measured}) \\
\text{grid}(t_A) &= \text{grid}(t_B + \text{offset} + \delta) && \text{(Gleichung (4.2))} \\
\text{grid}(t_A) &= \text{grid}(t_A + \delta) && \text{(Gleichung (4.1))} \\
\lfloor \frac{t_A}{\delta_{\text{grid}}} \rfloor \delta_{\text{grid}} &= \lfloor \frac{t_A + \delta}{\delta_{\text{grid}}} \rfloor \delta_{\text{grid}} && \text{(Definition (4.8))} \\
0 &= \lfloor \frac{t_A + \delta}{\delta_{\text{grid}}} \rfloor - \lfloor \frac{t_A}{\delta_{\text{grid}}} \rfloor \\
0 &= \lfloor \frac{t_A}{\delta_{\text{grid}}} + \frac{\delta}{\delta_{\text{grid}}} \rfloor - \lfloor \frac{t_A}{\delta_{\text{grid}}} \rfloor \\
0 &< \lfloor \frac{t_A}{\delta_{\text{grid}}} + 1 \rfloor - \lfloor \frac{t_A}{\delta_{\text{grid}}} \rfloor && \text{(da } \frac{\delta}{\delta_{\text{grid}}} < 1\text{)} \\
0 &< \lfloor \frac{t_A}{\delta_{\text{grid}}} \rfloor + 1 - \lfloor \frac{t_A}{\delta_{\text{grid}}} \rfloor \\
0 &< 1
\end{aligned}$$

□

4.2.2 Totalordnung der Ereignisse

Jedem Ereignis im hier beschriebenen System wird der Zeitpunkt seines Auftretens in Rasterzeit zugeordnet. Je nach gewähltem δ_{grid} hat diese eine relativ grobe Auflösung. Einem Zeitstempel eines Ereignisses wird zusätzlich zu dessen Rasterzeit-Zeitpunkt die global eindeutige RM-Id des RM zugefügt, auf dem das Ereignis stattgefunden hat. So

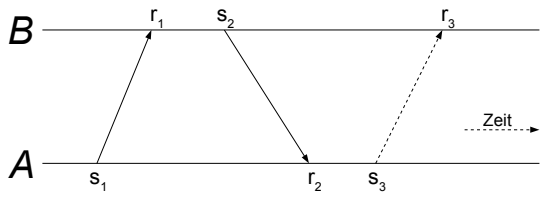

Abbildung 4.4: Bestimmung der Zeitdifferenz von Knoten B zu Knoten A

ist sichergestellt, dass Ereignisse, die auf verschiedenen RM zum gleichen Rasterzeit-Zeitpunkt aufgetreten sind, eindeutig geordnet werden können. Weiterhin verwaltet jeder RM einen lokalen Zähler, dessen Wert beim Auftreten eines Ereignisses ebenfalls dem Ereigniszeitstempel zugeordnet und anschließend um eins erhöht wird. Auf diese Weise sind auch Ereignisse, die lokal kurz hintereinander stattgefunden haben und somit ins gleiche Rasterzeitintervall fallen, eindeutig geordnet. Zusätzlich ermöglicht die lückenlose Nummerierung aller Ereignisse eines RM A, einem anderen RM B, festzustellen, ob er alle Ereignisse von A bis zu einem bestimmten Zeitpunkt empfangen hat, oder ob noch Ereignisse ausstehen.

Ein global eindeutiger Zeitstempel eines Ereignisses besteht also aus:

```
Rasterzeitwert | RM-Id | Zählerwert
```

4.2.3 Synchronisationsalgorithmus

Es wird nun ein Algorithmus vorgestellt, der für jeden RM seine Zeitdifferenz zu allen anderen RM in dessen Replikationsgruppe ermittelt. Der Algorithmus muss garantieren, dass der ermittelte Zeitunterschied mit einem Fehler kleiner als ein δ_{max} behaftet ist.

Damit ein Knoten die Zeitdifferenz zu einem anderen Knoten in seiner Kommunikationsreichweite ermitteln und den Fehler bestimmen kann, mit dem die ermittelte Zeitdifferenz maximal behaftet ist, sind zwei Nachrichten nötig. Wie in Abbildung 4.4 dargestellt, sendet ein Knoten A, der die Zeitdifferenz zu einem Knoten B ermitteln möchte, eine Nachricht an B. Diese Nachricht erhält bei A, kurz bevor sie A verlässt, einen Sendezeitstempel r_1. B empfängt diese Nachricht und ermittelt sofort nach deren Empfang den Empfangszeitstempel s_1. B schickt nun eine Antwortnachricht, die r_1 und den Sendezeitstempel dieser Nachricht s_2 enthält, an A. A empfängt diese Nachricht schließlich zum Zeitpunkt r_2. s_1 und r_2 sind dabei Zeitstempel in der lokalen Zeit von A, r_1 und s_2 in der lokalen Zeit von B.

A kann nun die Gesamtübertragungszeit (round trip time) rtt dieser beiden Nachrichten bestimmen mit $rtt = (r_2 - s_1) - (s_2 - r_1)$. Weiterhin kann A den Zeitunterschied zu B abschätzen mit $\delta_{A,B} = \frac{(r_1 - s_1) + (s_2 - r_2)}{2}$.

Dieser Wert ist mit einem Fehler von $\pm\frac{rtt}{2}$ behaftet. Da die Rasterzeitberechnung voraussetzt, dass der Fehler positiv ist, wird $\frac{rtt}{2}$ dazu addiert. Der ermittelte Wert der Zeitdifferenz von A und B ist also maximal rtt zu groß. Wenn rtt kleiner als δ_{max} ist, kann die so

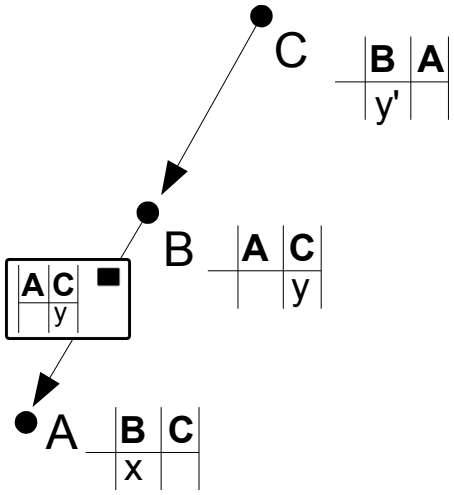

Abbildung 4.5: Bestimmung der Zeitdifferenz über mehrere Hops

erhaltene Zeitdifferenz genutzt werden, um die Zeitstempel von Ereignissen von B sicher in die lokale Rasterzeit von A umzurechnen. Dazu wird die ermittelte Zeitdifferenz auf den Zeitstempel der Nachricht addiert und anschließend die grid-Funktion angewendet. Ist $rtt > \delta_{max}$ muss das Synchronisationsprotokoll wiederholt werden.

Bisher kennt nur A den Zeitunterschied zu B, aber B nicht zu A. Sendet A eine weitere Nachricht an B, wie in Abbildung 4.4 gestrichelt dargestellt, in der die Zeitstempel s_2, r_2 sowie der Sendezeitpunkt dieser Nachricht s_3 enthalten sind, kann auch B anhand dieser drei Zeitstempel und dem Empfangszeitpunkt der Nachricht r_3 die Zeitdifferenz zu A bestimmen. Alternativ kann A direkt den Wert der von ihr berechneten Zeitdifferenz zwischen A und B an B schicken. Auf diese Weise sind die bestimmten Zeitunterschiede zwischen beiden Knoten symmetrisch.

Bisher wurde beschrieben, wie sich zwei Knoten direkt synchronisieren können. Es sollte aber auch möglich sein, Zeitstempel von Nachrichten in die lokale Zeit umzurechnen, die von Knoten initiiert wurden, die sich gerade nicht in unmittelbarer Kommunikationsreichweite befinden und zu denen noch keine Zeitdifferenz bekannt ist. Dieses wird in Abbildung 4.5 veranschaulicht. Es stellt kein großes Problem dar, wenn jeder Knoten die Zeitdifferenz zu seinen unmittelbaren Nachbarn kennt, oder leicht bestimmen kann. Eine Nachricht, die auf einem Knoten C initiiert wird, wird in einem 1-Hop-Radius verteilt und von Knoten B empfangen. B und C können direkt miteinander kommunizieren und so die Zeitdifferenz y und y' zwischen ihren Uhren bestimmen. Das Verteilungsprotokoll sorgt dafür, dass eine Nachricht letztendlich von jedem Knoten empfangen wird, indem Knoten, die Nachrichten erhalten haben, diese ggf. weitersenden. B sendet also irgendwann diese Nachricht weiter und hängt zusätzlich die Zeitdifferenz von seiner zu Cs Uhr und die konservative Abschätzung des bei der Ermittlung dieser Differenz erzielten Fehlers an. Damit kann nun ein Knoten A, der diese Nachricht empfängt, aus der Summe der Zeitdifferenzen x von A zu B und y von B zu C die Gesamtzeitdifferenz zu C bestim-

men. Auch der Gesamtfehler bildet sich aus der Summe der Fehler der Differenzen von A zu B und von B zu C. Ist dieser kleiner als ein voreingestellter Schwellenwert, kann A den Zeitunterschied zu C zusammen mit dem berechneten Fehler in seiner SkewMatrix speichern und die Nachricht in die lokale Zeit umrechnen. Schickt As Verteilungskomponente die Nachricht ebenfalls weiter, so wird die ermittelte Zeitdifferenz von A zu C an diese angehängt.

4.3 Konflikterkennung

In Abschnitt 4.1 wurde bereits erläutert, dass Konflikte durch kausal parallele Änderungen unterschiedlicher Kopien der gleichen logischen Datenelemente durch Transaktionen auf unterschiedlichen RM entstehen können. In diesem Abschnitt soll nun genauer auf diese Konflikte eingegangen werden. Es wird untersucht, welche Nebenläufigkeitsanomalien auftreten können und wie diese erkannt werden.

4.3.1 Konflikt

Abbildung 4.6 zeigt ein Beispiel eines Schreib-Schreib-Konfliktes. Hier wird durch Ausführung einer Schreiboperation einer Transaktion t_1 auf einer lokalen Datenbank A der Wert der lokalen Kopie x_A des logischen Datenelementes X geändert. Wenig später wird durch eine zweite Transaktion t_2 auf einer weiteren lokalen Datenbank B eine andere lokale Kopie x_B von X geändert, bevor die Änderung von t_1 dort empfangen worden ist. Durch die kausal parallelen Änderungen von X ist ein Konflikt entstanden, der erkannt werden muss. Würden die fremden Änderungen auf jedem lokalen RM nach deren Empfang einfach angewendet, würden die lokalen Datenbankzustände von A und B anschließend differieren. Auf A würden die Transaktionen t_1 und t_2 ausgeführt, auf B die Transaktionen t_2 und t_1. Hätte A die Änderung durch t_2 von B empfangen, bevor die lokale Transaktion t_1 ausgeführt wurde, wäre kein Konflikt entstanden. Gleiches würde gelten, wenn B die Änderung durch t_1 von A vor Ausführung von t_2 empfangen hätte.

Um einen global konsistenten Datenbankzustand zu erhalten, muss das verteilte Replikationssystem zumindest dafür sorgen, dass Schreiboperationen unterschiedlicher Transaktionen von jedem lokalen RM in der gleichen Reihenfolge ausgeführt werden. Damit wird aber nicht verhindert, dass Werte, die eine Schreiboperation erzeugt hat, von einer kausal parallelen unbemerkt überschrieben werden können (lost update). Um dieses zu verhindern erkennt SYMORE diese Schreib-Schreib-Konflikte. Ein Konfliktlösungsalgorithmus, wie er im Abschnitt 4.4 beschrieben wird, entscheidet daraufhin eine dieser Transaktionen abzubrechen. Wurde eine Transaktion schließlich definitiv abgebrochen, kann deren Urheber informiert werden.

Einen Sonderfall bilden Transaktionen, die kommutativ sind, d.h. unabhängig von ihrer Ausführungsreihenfolge dasselbe Ergebnis erzielen. Ein Beispiel wäre z.B. eine Transaktion, die den Wert eines Datenelementes um eins erhöht. Hierbei kann es für manche

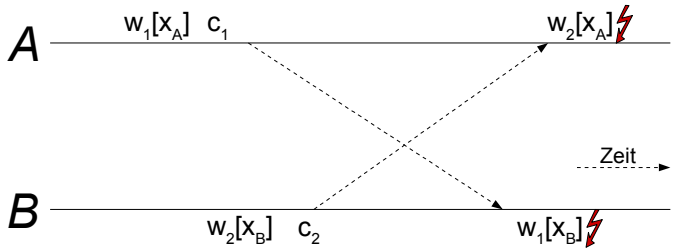

Abbildung 4.6: Konflikt durch kausal parallele Updates auf A und B

Anwendungen wünschenswert sein, kausal parallele Änderungen an diesem Datenelement nicht als Konflikt zu betrachten, sondern alle – auch kausal parallele – Operationen auszuführen. Da nicht der neue Zustand eines geänderten Datenelementes übertragen wird, sondern die Änderungsoperation selbst, können diese dazu einfach auf allen Knoten erneut ausgeführt werden.

Konfliktgranularität

SYMORE bietet Flexibilität bei der Erkennung von Konflikten. Wie eben anhand des Beispiels von kommutativen Transaktionen veranschaulicht, ist es nicht immer wünschenswert, alle parallelen Transaktionen mit Operationen auf gleichen Datenelementen als Konflikt zu behandeln. Deshalb werden solche Transaktionen nur dann als in Konflikt stehend aufgefasst, wenn dieses explizit gewünscht ist. Zunächst sorgt das Replikationssystem nur dafür, dass alle Transaktionsergebnisse überall in ihrer Transaktionszeitstempelreihenfolge ausgeführt werden und so überall ein gleicher Datenbankzustand erreicht wird.

Sollen parallele Änderungen an gleichen Datenelementen erkannt werden, muss dieses bei der Erzeugung der Datenbanktabelle angegeben werden. Es können dabei nicht nur parallele Änderungen gleicher logischer Datenelemente erkannt werden, sondern auch parallele Änderungen auf größeren Strukturen, wie kompletten Tabellen, Zeilen oder Spalten. So werden bei der Erzeugung einer Tabelle *Konfliktgranularitäten* definiert. Diese Konfliktgranularitäten bestimmen die Mengen von Datenelementen, bei denen Konflikte durch parallele Änderungen erkannt werden sollen. Ist als Konfliktgranularität etwa „Tabelle“ gewählt worden, so werden sämtliche parallelen Änderungen an dieser Tabelle als miteinander in Konflikt stehend aufgefasst. Viele Client-Server-Replikationssysteme erkennen Konflikte bei Manipulationen derselben Datenbankzeile. Um dieses Verhalten zu erreichen, kann „Reihe“ als Konfliktgranularität gewählt werden. Es werden dann parallele Änderungen an derselben Reihe als Konflikt aufgefasst, Änderungen an unterschiedlichen Reihen dagegen nicht. Analoges gilt für Spalten und Zellen als Konfliktgranularität.

Während mittels der Konfliktgranularität für eine Datenbank allgemein festgelegt wird, welche Datenelemente für die Konflikterkennung zusammengefasst werden, be-

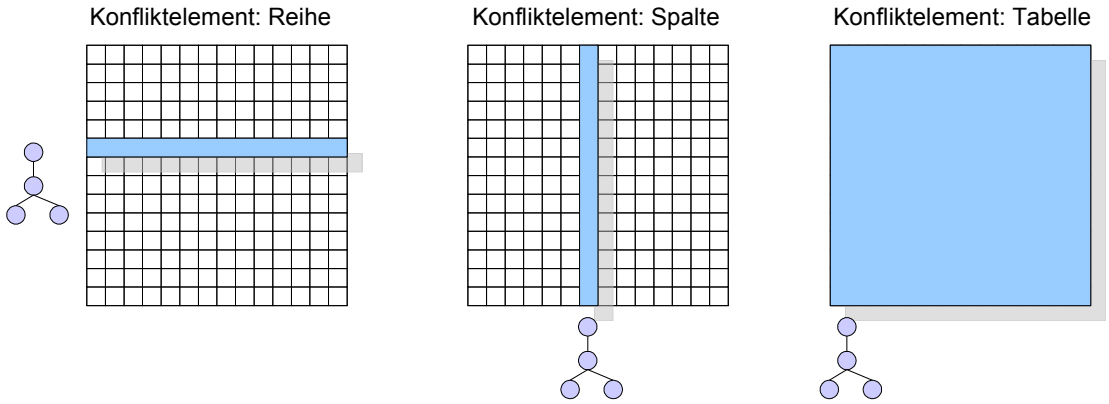

Abbildung 4.7: Konfliktelemente unterschiedlicher Granularität

zeichnet ein *Konfliktelement* eine bestimmte Menge konkreter Datenelemente. Ein solches Konfliktelement kann z.B. aus allen Datenelementen der gesamten Datenbank bestehen, aus einer bestimmten Datenzeile, aus einer bestimmten Datenspalte oder nur aus einem einzelnen Datenelement. Kausal parallele Änderungen von Datenelementen, die durch ein solches Konfliktelement zusammengefasst werden, werden von SYMORE als Konflikt aufgefasst. Der Begriff Konfliktelement wurde in Analogie zu dem Begriff Datenelement gewählt. Für die Konfliktbehandlung ist ein Konfliktelement und nicht ein Datenelement die atomare Einheit, von der parallele Änderungen erkannt werden sollen. Abbildung 4.7 veranschaulicht dieses.

Je nach Anwendung und Bedeutung einzelner Datenelemente kann ein Anwendungsentwickler die für die Anwendung günstigste Konflikterkennungsgranularität wählen. Je mehr Datenelemente ein Konfliktelement umfasst, desto weniger Daten müssen letztendlich zusammen mit den Transaktionen verteilt werden. Andererseits wird eine größere Zahl von Transaktionen als in Konflikt stehend betrachtet.

Eine Transaktion kann auf beliebige Datenelemente zugreifen und damit auch auf verschiedene Konfliktelemente. Eine Transaktion kann also mit unterschiedlichen Transaktionen an verschiedenen Konfliktelementen in Konflikt stehen.

Leseoperationen

Wie bereits beschrieben, werden in SYMORE nicht die neuen Werte (der Zustand) der von einer Transaktion geänderten Datenelemente übertragen, sondern die Operationen der ursprünglichen lokalen Transaktion. Diese werden als Updatetransaktion auf dem empfangenen RM später erneut ausgeführt. Dabei soll dasselbe Ergebnis wie bei der initialen Ausführung der Transaktion auf deren Ursprungsdatenbank erzielt werden. Die Werte, die von einer Transaktion geschrieben werden, hängen (zumindest im Modell) von den Werten ab, die in derselben Transaktion gelesen wurden. Es muss also sichergestellt

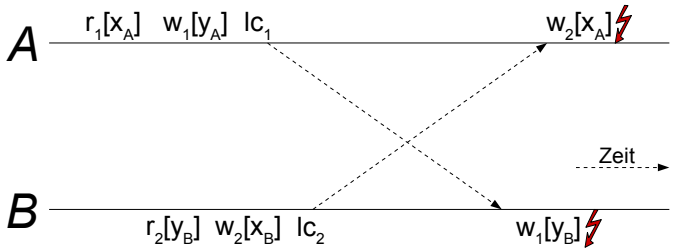

Abbildung 4.8: Konflikt durch kausal parallele Updates auf A und B unter Berücksichtigung von Leseoperationen

sein, dass die Werte aller dieser gelesenen Datenelemente auf der Datenbank des empfangenden RM den ursprünglichen Werten dieser Datenelemente auf der Ursprungsdatenbank entsprechen und nicht zwischenzeitlich durch andere Transaktionen geändert wurden. Dieses wird sichergestellt, indem zusätzlich Leseoperationen berücksichtigt werden. Ähnlich wie bei dem oben beschriebenen Schreib-Schreib-Konflikt besteht zwischen zwei Transaktionen ein Lese-Schreib-Konflikt, wenn eine Transaktion ein Datenelement liest, das eine andere parallel schreibt und die lesende Transaktion zusätzlich mindestens ein beliebiges Datenelement schreibt.

Abbildung 4.8 zeigt hierzu ein Beispiel. Ohne Berücksichtigung der Leseoperationen stünden die Transaktionen t_1 und t_2 nicht miteinander in Konflikt, da unterschiedliche logische Datenelemente geschrieben werden. Mit Berücksichtigung der Leseoperationen liegt aber ein Konflikt vor, da t_1 Datenelement y schreibt, welches parallel von t_2 gelesen wird und t_2 Datenelement x schreibt, welches parallel von t_1 gelesen wird.

Schreib- und Lesemenge

Um Konflikte zu erkennen, werden die *Schreibmengen (Writesets)* und ggf. *Lesemengen (Readsets)* von Transaktionen ermittelt. Wenn eine Transaktion ein Datenelement schreibt, wird das Konfliktelement, das dieses Datenelement enthält, zusammen mit dessen *Version* der Schreibmenge zugeordnet. Analoges gilt, wenn eine Transaktion ein Datenelement liest. Die Version eines Konfliktelementes entspricht genau der Transaktion, die zuletzt ein Datenelement dieses Konfliktelements geschrieben hat. Nur Transaktionen ändern Werte von Datenelementen und überführen sie so von einer Version in eine andere. Ein Element einer Schreib- oder Lesemenge ist also ein Zwei-Tupel, bestehend aus dem Konfliktelement und der letzten Version, bzw. der Transaktions-Id der Transaktion, die ein Datenelement dieses Konfliktelementes zuletzt geschrieben hat.

Die Ermittlung der Lesemenge und damit das Erkennen von Lese-Schreib-Konflikten ist in SYMORE optional. Je nach Anwendung kann entschieden werden, ob solch eine Erkennung nötig ist. Sie ist mit einem höheren Aufwand und einer größeren zu übertragenen Datenmenge verbunden.

Für die Konflikterkennung sind Transaktionen atomar und laufen in Isolation voneinander ab. Zwischenwerte, die sich im Rahmen der Transaktionsausführung ergeben

können, können außerhalb dieser Transaktion nicht beobachtet werden.

4.3.2 Phänomene

Da es sich bei SYMORE um ein Datenbanksystem handelt, werden auch die in klassischen Datenbanken bekannten *Phänomene* untersucht. Phänomene sind bestimmte Ausführungsfolgen von Operationen von Transaktionen (*Historien*), die zu *Mehrbenutzeranomalien* führen können. Mehrbenutzeranomalien sind Verletzungen der Isolationseigenschaft oder Inkonsistenzen der Datenbank. Die Diskussion in diesem Abschnitt orientiert sich an [HTKR05, S. 233ff.].

Die wichtigsten Phänomene sind *Dirty Write*, *Dirty Read*, *Non-repeatable Read* und *Phantome*. Im Folgenden werden diese beschrieben und es wird erläutert, wie sich SYMORE in Bezug auf diese verhält.

Transaktionen werden lokal serialisierbar ausgeführt. Phänomene können zwischen lokalen Transaktionen auf einem Knoten also nicht auftreten. Durch die nicht synchronisierte Ausführung von Transaktionen unterschiedlicher RM, die auf unterschiedlichen Kopien von Datenelementen arbeiten, sind solche Phänomene zwischen kausal parallelen Transaktionen aber möglich.

Dirty Write (P0): Dieses Phänomen liegt vor, wenn eine Transaktion t_j Datenelemente ändert, die zuvor von einer noch nicht abgeschlossenen Transaktion t_i geändert worden sind.

Dieses wird in SYMORE zunächst nicht verhindert. Transaktionsergebnisse werden sofort nach dem lokalen Pre-Commit für alle weiteren Transaktionen sichtbar, die somit den Wert eines Datenelementes, aufbauend auf einem noch nicht festgeschriebenen Wert dieses Elementes ändern können. Dieses Vorgehen ist eine Grundeigenschaft optimistischer Replikationssysteme. SYMORE stellt allerdings sicher, dass t_j nur dann endgültig festgeschrieben wird, wenn dieses auch für t_i der Fall ist.

Dirty Read (P1): Dieses Phänomen liegt vor, wenn eine Transaktion t_j ein Datenelement liest, das von einer Transaktion t_i, die noch nicht abgeschlossen ist, zuvor geschrieben wurde. Bricht t_i ab, hat t_j eine nicht dauerhafte Version gelesen.

Dieses Phänomen kann, genau wie P0, in SYMORE auftreten, da Änderungen von Transaktionen nach deren Pre-Commit sofort für weitere Transaktionen sichtbar werden. Um dieses zu erkennen, muss die Ermittlung der Lesemenge in SYMORE aktiviert sein. Mittels dieser wird festgestellt, von welcher Version eines Datenelementes und damit von welcher Schreiboperation eine Leseoperation abhängt. Die Transaktion, die diese Leseoperation enthält, wird erst festgeschrieben, wenn auch die Transaktion, die die Schreiboperation enthält, festgeschrieben wurde.

Non-repeatable Read (P2): Kann eine Transaktion während ihrer Ausführung auch ohne eigene Änderungen unterschiedliche Werte eines Datenelementes lesen, so liegt

dieses Phänomen vor.

In SYMORE verhindert der Synchronisationsmechanismus der lokalen Datenbank das Auftreten dieses Phänomens während der lokalen Ausführung einer Transaktion. Zusätzlich muss verhindert werden, dass eine Updatetransaktion andere Werte liest als die zugehörige initiale Transaktion. Dieses kann auftreten, wenn kausal parallel zu der lokalen initialen Transaktion auf einem anderen RM andere lokale Transaktionen oder Updatetransaktionen anderer RM ausgeführt worden sind. Erkannt wird dieses über die Lesemenge, die für die initiale Transaktion ermittelt wurde. Anhand dieser wird sichergestellt, dass eine zu dieser Transaktion gehörende Updatetransaktion auch auf dem fremden RM die gleiche Version der gelesenen Datenelemente vorfindet (siehe Abschnitt 4.4).

Ein Phänomen, das nicht auftreten kann, wenn P2 verhindert wird, aber zu dessen feineren Differenzierung dient, ist „Lost Update“. Dieses Phänomen tritt auf, wenn zwei noch nicht abgeschlossene Transaktionen dasselbe Datenelement ändern. Es entspricht den weiter oben beschriebenen Schreib-Schreib-Konflikten und wird über die Bestimmung der Schreibmenge jeder Transaktion von SYMORE erkannt.

Phantome (P3): Dieses Phänomen kann als eine besondere Form des Phänomens P2 angesehen werden. Im Gegensatz zu P2 geht es nicht um das Lesen eines veränderten Wertes eines Datenelementes, sondern um eine veränderte Ergebnismenge. Phantome treten auf, wenn eine Transaktion t_i Datenelemente ändert, in die Datenbasis einfügt oder aus dieser löscht, so dass sich die Ergebnismenge einer zweiten, kausal parallel ausgeführten Transaktion t_j ändert. t_j selektiert Datenelemente über ein Prädikat p. Fügt nun t_i kausal parallel weitere Elemente in die Datenbank ein, so kann es passieren, dass t_j mittels p bei der späteren erneuten Ausführung auf einem anderen Knoten eine veränderte Ergebnismenge erhält als bei ihrer initialen Ausführung.

Die Ergebnismenge eines Prädikats kann sich in SYMORE nicht durch die Änderung von Datenelementen ändern, ohne dass dieses – bei aktivierter Ermittlung der Lesemenge – erkannt wird. Es müssen aber zusätzliche (optionale) Maßnahmen getroffen werden, um auch Phantome durch kausal paralleles Einfügen in die Datenbank oder Löschen aus dieser zu erkennen. Zum einen wird das Prädikat der `Where`-Klausel einer SQL-Änderungsanweisung und in dieser enthaltener Unteranfragen nach der initialen Ausführung dieser Anweisung so umgeschrieben, dass sie auch bei einer erneuten Ausführung nur genau die initial bestimmten Zeilen selektiert. Dass diese Zeilen in der richtigen Version bei einer späteren Ausführung vorhanden sind, wird durch Verhinderung des Phänomens P2 sichergestellt. Zum anderen müssen Aggregatfunktionen und andere Funktionen – beispielsweise die Ermittlung der aktuellen Zeit –, die in einer Änderungsanweisung vorkommen können, in der Updatetransaktion durch ihren initial ermittelten Wert ersetzt werden, um

zu verhindern, dass bei erneuter Ausführung andere Werte ermittelt werden.

Ohne diese optionalen Maßnahmen können Phantome auch erkannt werden, indem die Konfliktgranularität so grob eingestellt wird, wie die von dem Prädikat umfasste Struktur. Handelt es sich bei dem Prädikat beispielsweise um eine Aggregatfunktion, die alle Zeilen einer Tabelle betrifft, wird mit der Konfliktgranularität „Tabelle“ sichergestellt, dass die Tabelle auch bei einer späteren Neuausführung des Prädikats auf einem anderen Knoten genau denselben Zustand wie bei der initialen Ausführung hat.

Das Problem der Vermeidung von Phantomen wird in Abschnitt 5.2.2 genauer diskutiert.

Abschließend lässt sich zusammenfassen, dass bei ausschließlicher Ermittlung von Schreibmengen die Phänomene „Dirty Write“ und „Lost Update“ erkannt werden. Werden zusätzlich die Lesemengen ermittelt, wird auch das Auftreten aller anderen Phänomene bis auf Phantome festgestellt. Um letzteres Phänomen zu vermeiden sind, wie beschrieben, zusätzliche Maßnahmen nötig. Die in Abschnitt 4.4 vorgestellten Konfliktlösungsalgorithmen sorgen schließlich dafür, Transaktionen auszuwählen, die abgebrochen werden müssen, um diese erkannten Konflikte aufzulösen.

4.3.3 Datenstruktur: Vorgängergraph

Die über die Schreib- und Lesemengen ermittelten Beziehungen zwischen Transaktionen müssen in einer Datenstruktur verwaltet werden. Dazu nutzt jeder RM einen gerichteten, azyklischen Graphen. Dieser Graph wird im Folgenden *Vorgängergraph* genannt. Er stellt explizit dar, welche Transaktion von welcher anderen früheren Transaktion abhängt, zeigt also die Vorgänger jeder Transaktion.

Der Vorgängergraph $G = (V, E)$ besteht aus einer Menge von Knoten $V = \{t_1, \cdots, t_n\}$ und einer Menge von gerichteten Kanten $(t_i, t_j) \in E$. Jeder Knoten repräsentiert eine Transaktion. Weiterhin gibt es die Menge KE, die alle bekannten Konfliktelemente enthält. Kanten $(t_i, t_j) \in E$ zwischen Knoten des Vorgängergraphen stellen Lese- oder Schreibbeziehungen zu anderen Transaktionen dar. Jeder Kante $e = (t_i, t_j) \in E$ ist durch die Funktion $on : E \rightarrow \mathcal{P}(KE \times \{r, w\})$ eine Menge von Konfliktelementen $k \in KE$ zugeordnet sowie die Information, ob Transaktion t_i k liest (r) oder schreibt (w). Wird in einer Transaktion ein Konfliktelement sowohl gelesen als auch geschrieben, wird nur die Information, dass das Element geschrieben wird, vermerkt.

Ein Ausschnitt aus einem solchen Vorgängergraphen ist in Abbildung 4.9 dargestellt. Die Kante $e = (TA_{14}, TA_{12})$ mit $on(e) = \{(KE_1, w)\}$ beispielsweise zeigt an, dass TA_{14} durch Konfliktelement KE_1 beschriebene Daten verändert, die vorher von TA_{12} geändert wurden.

Auch wenn ein Konfliktelement eine Menge von Datenelementen zusammenfasst, wird zur sprachlichen Vereinfachung im Folgenden der Begriff Konfliktelement wie der

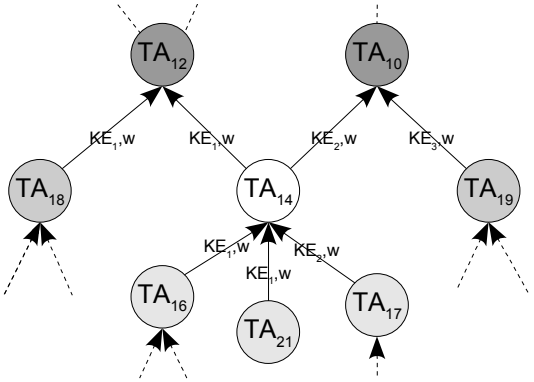

Abbildung 4.9: Beziehungen zu Knoten TA_{14}: $parents(TA_{14}) = \{TA_{12}, TA_{10}\}$; $children(TA_{14}) = \{TA_{16}, TA_{17}, TA_{21}\}$; $siblings(TA_{14} = \{TA_{18}\}$; TA_{19} ist kein Nachbarknoten von TA_{14}, da er ein anderes Konfliktelement manipuliert.

Begriff Datenelement verwendet. Ein Konfliktelement kann geändert werden, wenn gemeint ist, dass der Wert mindestens eines in diesem Konfliktelement enthaltenen Datenelementes geändert wird oder ein in diesem Konfliktelement enthaltenes Datenelement eingefügt oder gelöscht wird. Ebenfalls ist mit einem Knoten in dem Vorgängergraphen implizit auch die dem Knoten zugeordnete Transaktion gemeint. So kann von Konfliktelementen, die ein Knoten gelesen hat, gesprochen werden, obwohl tatsächlich eine dem Knoten zugeordnete Transaktion diese Konfliktelemente gelesen hat.

Jeder Knoten hat pro Konfliktelement maximal eine ausgehende Kante zu einem *Elternknoten*. Die Funktion $parent : V \times KE \rightarrow V$ liefert den Elternknoten, den ein Knoten an dem angegebenen Konfliktelement geändert hat, also $parent(t_i, k) = \{t_j \in V | \exists (t_i, t_j) \in E \wedge (k, w) \in on((t_i, t_j))\}$. Die Funktion $parents : V \rightarrow \mathcal{P}(V)$ liefert alle Elternknoten eines Knotens: $parents(t) = \{t_i \in V | \exists (t, t_i) \in E\}$

Weiterhin kann ein Knoten pro Konfliktelement beliebig viele eingehende Kanten von *Kindknoten* haben. Diese liefert die Funktion $allChildren : V \rightarrow \mathcal{P}(V)$ mit $allChildren(t) = \{t_i \in V | (t_i, t) \in E\}$. Nur die Kinder an einem Konfliktelement liefert $children : V \times KE \rightarrow \mathcal{P}(V)$, definiert als: $children(t, k) = \{t_i \in V | (t_i, t) \in E \wedge ((k, w) \in on(t_i, t) \vee (k, r) \in on(t, t_i))\}$.

Nachbarknoten eines Knoten t sind Knoten, die den gleichen Elternknoten an dem gleichen Konfliktelement wie t haben. Die Funktion $siblings : V \rightarrow \mathcal{P}(V)$ liefert alle Nachbarknoten eines Knotens. Sie ist definiert als: $siblings(t) = \{t_i \in V | \exists k \in KE, parent(t_i, k) \neq \varnothing \wedge parent(t, k) \neq \varnothing \wedge parent(t_i, k) = parent(t, k) \wedge t \neq t_i\}$.

Zusätzlich ist jedem Knoten der global eindeutige Zeitstempel seiner ihm zugeordneten Transaktion zugewiesen, wie in Abschnitt 4.2.2 beschrieben. In Abschnitt 4.1 wurde erwähnt, dass eine Transaktion in der Phase nach ihrem Pre-Commit und bis zum endgültigen Commit *aktiviert* oder *deaktiviert* oder zunächst *undefiniert* ist. Dieses wird in einem *Zustandsfeld* vermerkt.

Um definieren zu können, was ein Konflikt ist, wird zusätzlich die Menge $pred(t), t \in V$ aller Vorgängerknoten eines Knotens benötigt. $pred(t_i)$ ist definiert als $pred(t_i) = \{t_i, parents(t_i), parents(parents(t_i)), \cdots\}$.

Es folgt nun die Definition eines Konfliktes zwischen zwei Knoten in einem Vorgängergraphen. Dabei werden zunächst nur Schreib-Schreib-Konflikte betrachtet:

Ein *direkter Konflikt* $t_i \leftrightarrow t_j$ zwischen den Knoten $t_i, t_j \in V$ besteht gdw. $t_j \in siblings(t_i)$.

Ein *indirekter Konflikt* $t_1 \tilde{\leftrightarrow} t_2$ besteht gdw. $\exists t_{p1} \in pred(t_1), \exists t_{p2} \in pred(t_2)$ mit $t_{p1} \notin pred(t_2) \wedge t_{p2} \notin pred(t_1) \wedge t_{p1} \leftrightarrow t_{p2}$.

Werden zusätzlich Lesemengen betrachtet, liegen Konflikte auch vor, wenn ein Knoten ein Konfliktelement liest, das ein anderer schreibt. Analoges gilt für indirekte Konflikte.

Knoten werden nicht nur in dem Vorgängergraphen gespeichert. Zusätzlich werden Verweise auf diese in einer nach Zeitstempeln sortierten Liste `NodesByTimestampList` angelegt. Damit ist es effizient möglich, alle Knoten in Zeitstempel-Reihenfolge zu traversieren. Eine Map `activatedNodes` enthält zudem zu jedem Konfliktelement den in Zeitstempelreihenfolge letzten momentan aktivierten Knoten.

Mit den Informationen des Vorgängergraphen und den Zeitstempeln jedes Knotens können folgende Beziehungen zwischen den Knoten und damit zwischen den durch sie repräsentierten Transaktionen definiert werden:

1. Zeitliche Reihenfolge: $t_1 < t_2$, wenn t_1 älter ist als t_2 (hat kleineren Zeitstempel)
2. Hängt-ab-von(t_1,t_2): $t_1 \rightarrow t_2$, wenn t_1 auf Daten, geschrieben von t_2 aufbaut. Dieses ist gleichbedeutend mit $t_2 \in pred(t_1)$.
3. Konflikt-mit(t_1,t_2): $t_1 \tilde{\leftrightarrow} t_2$, wenn t_1 mit t_2 in (direktem oder indirektem) Konflikt steht (symmetrisch).

Einfügen in den Vorgängergraphen

Einfügen einer lokalen Transaktion: Eine neu erzeugte lokale Transaktion T_{new} muss in den Vorgängergraphen eingefügt werden. Dazu wird die Menge aller Konfliktelemente K bestimmt, deren Datenelemente im Zuge der Transaktionsausführung gelesen oder geschrieben wurden.

Für T_{new} wird ein zugeordneter Knoten V_{new} erzeugt und sein Zustand auf `activated` gesetzt. Es muss nun die Schreib- und eventuell die Lesemenge gebildet werden und der Knoten muss in den Vorgängergraphen eingefügt werden. Dazu wird für jedes Konfliktelement $k \in K$ der momentan aktivierte Knoten V_{ref} mittels der Map `activatedNodes` bestimmt. An diesen wird V_{new} mit der Operation $V_{ref}.appendChild(V_{new}, k)$ angehängt und schließlich an Stelle k in `activatedNodes` eingefügt.

Gibt es für ein $k \in K$ keinen Knoten in der Map `activatedNodes`, so wird V_{new} an den *Dummyknoten* gehängt. Der Dummyknoten ist immer aktiviert, hat den kleinsten

Zeitstempel aller Knoten und keine ihm zugeordnete Transaktion. Er dient als das Wurzelelement des Vorgängergraphen.

Der Zeitstempel von V_{ref} bildet zusammen mit dem Konfliktelement k ein Element der zu bildenden Schreib- oder Lesemenge. Die Transaktion T_{new} wird schließlich zusammen mit allen Elementen dieser Mengen verteilt.

Einfügen einer fremden Transaktion: Eine empfangene Updatetransaktion T_{recv} muss ebenfalls in den lokalen Vorgängergraphen eingefügt werden. Dazu wird zunächst geprüft, ob die in der Schreib- und Lesemenge von T_{recv} referenzierten Knoten im Graphen vorhanden sind. Es gibt zwei Gründe, warum dies nicht der Fall sein kann. Zum einen können Nachrichten in vertauschter Reihenfolge ankommen. Es kann vorkommen, dass eine Nachricht, die eine Transaktion enthält, deren Version von T_{recv} referenziert wird, noch nicht empfangen worden ist. Zum anderen können Knoten bereits aus dem Graphen gelöscht worden sein. Dieses geschieht im Zuge des Commitprozesses, in dem für eine Transaktion definitiv entschieden wird, ob sie festgeschrieben oder abgebrochen wird. Ist diese Entscheidung getroffen, wird der zugehörige Knoten unter bestimmten Bedingungen aus dem Vorgängergraphen gelöscht. Dieses wird in Abschnitt 4.5.6 genauer beschrieben.

Diese beiden Fälle müssen unterschieden werden. Im ersten Fall wird die Transaktion zurückgestellt und erst eingefügt, wenn alle referenzierten Transaktionen vorhanden sind. Im zweiten Fall wird die Transaktion an alle noch vorhandenen Referenzen angehängt. Fehlt an einem Konfliktelement eine Referenz, so wird die Transaktion an den Dummyknoten angefügt. Außerdem wird der Zustand der Transaktion sofort auf `deaktiviert` gesetzt und zusätzlich mittels des Feldes `definitiv` markiert, dass sich dieser Zustand nicht mehr ändern darf.

Sind alle referenzierten Knoten im Graphen vorhanden, wird der Knoten an diese angehängt und sein Zustand auf `undefiniert` gesetzt.

4.4 Konfliktlösung

Wie in Abschnitt 4.1 beschrieben, führt jeder lokale RM lokale Transaktionen aus und empfängt Daten über fremde Transaktionen anderer RM. Im vorigen Abschnitt wurde erläutert, wie Informationen über die kausalen Beziehungen aller lokal bekannten, auf irgendeinem RM initiierten Transaktionen in Vorgängergraphen dargestellt werden und so Konflikte erkannt werden.

Das Ziel des Replikationssystems ist es, eine hohe Verfügbarkeit der Daten zu gewährleisten. Deshalb sollen die Effekte von empfangenen Updatetransaktionen anderer RM möglichst sofort lokal sichtbar gemacht werden. Auch wenn Transaktionen miteinander in Konflikt stehen, soll nicht gewartet werden, bis global entschieden ist, welche dieser Transaktionen abgebrochen werden müssen und welche festgeschrieben werden können. Stattdessen wird vorläufig anhand der momentan lokal vorliegenden Informationen

bestimmt, welche Transaktion ihre Änderungen in den aktuellen lokalen Datenbankzustand einbringen kann und welche nicht. Ein Konfliktlösungsalgorithmus bestimmt dazu aus der Menge aller lokal vorliegenden Transaktionen, die sich im Pre-Commit-Zustand befinden, die Transaktionen, die aktiviert werden und somit die aktuelle vorläufige Datenbanksicht bilden.

4.4.1 Konfliktlösungsalgorithmen

Unterschiedliche Konfliktlösungsalgorithmen sind möglich. Sie unterscheiden sich durch die Kriterien, anhand derer sie die Menge der zu aktivierenden Transaktionen bestimmen. Damit jede lokale Datenbank letztendlich den gleichen Zustand erreicht, muss von jedem RM der gleiche Konfliktlösungsalgorithmus ausgeführt werden.

Als *Eingabe* enthält ein solcher Konfliktlösungsalgorithmus SOLVECONFLICT(`G, start, end`) den *Vorgängergraphen G*, einen *Startzeitpunkt (start)* (inklusive) und einen *Endzeitpunkt (end)* (exklusiv).

Der Algorithmus kann auf die *NodesByTimestamp-Liste* und die *ActivatedNodes-Map* zugreifen. Durch den Start- und Endzeitpunkt wird die Menge der Knoten im Vorgängergraphen eingeschränkt, deren Zustandsfeld der Konfliktlösungsalgorithmus ändern darf. Nur bei Knoten, deren Zeitstempel zwischen `start` und `end` liegt, ist dieses erlaubt. Der Algorithmus darf den Zustand eines Knotens nicht verändern, wenn dessen Zeitstempel außerhalb dieses Intervalls liegt.

Die *Ausgabe* des Konfliktlösungsalgorithmus ist der *Vorgängergraph*, bei dem der Zustand jedes Knotens zwischen *start* und *end* entweder aktiviert oder deaktiviert ist.

Die Teilmenge aller Transaktionen zwischen einem Start- und einem Endzeitpunkt sei im Folgenden mit $TA_{start,end}$ bezeichnet.

Durch die Ausführung der aktivierten Transaktionen soll sich ein konsistenter Datenbankzustand ergeben. Deshalb müssen nach der Ausführung des Konfliktlösungsalgorithmus folgende Kriterien gelten:

Korrektheitskriterien

1. Keine zwei Transaktionen $t_i, t_j \in TA_{0,end}$ stehen miteinander in Konflikt und sind aktiviert.

2. Für jede aktivierte Transaktion $t_i \in TA_{0,end}$ gilt, dass alle Transaktionen, von denen t_i abhängt, auch aktiviert sind.

3. Jede Transaktion $t_i \in TA_{0,end}$ ist entweder aktiviert oder deaktiviert.

4. Der Konfliktlösungsalgorithmus muss deterministisch sein. Für die gleiche Menge an Transaktionen in seiner Eingabe bestimmt er die gleiche Teilmenge der ak-

tivierten Transaktionen, unabhängig davon, in welcher Reihenfolge Transaktionen eingefügt worden sind.

Die obigen Kriterien beziehen sich auf Transaktionen im Intervall von 0 bis *end*, nicht nur von *start* bis *end*. Wie bei der Beschreibung der Commit-Algorithmen in Abschnitt 4.5 erläutert wird, ist für alle Transaktionen mit Zeitstempel vor dem *start*-Wert einer Konfliktlösungsausführung bereits eine definitive Commitentscheidung getroffen worden. Für diese Transaktionen in $TA_{0,start}$ gelten die oben genannten Kriterien bereits. Die Zustände dieser Transaktionen müssen bei der Berechnung der Zustände der Transaktionen zwischen *start* und *end* berücksichtigt werden, um insgesamt eine konsistente Datenbanksicht für alle Transaktionen in $TA_{0,end}$ zu erhalten.

Aus den oben genannten Kriterien folgt automatisch, dass für jede deaktivierte Transaktion $t_i \in TA_{0,end}$ gilt, dass jede Transaktion $t_j \in TA_{0,end}$, die von t_i abhängt, deaktiviert ist. Der Konfliktlösungsalgorithmus muss weiterhin sicherstellen, dass die Map `activatedNodes` nach Beendigung des Algorithmus zu jedem Konfliktelement einen Verweis auf die jüngste aktivierte Transaktion bzw. deren zugeordneten Knoten enthält, die dieses Konfliktelement schreibt. Ist solch ein Knoten nicht vorhanden, zeigt der Eintrag auf den Dummyknoten.

Das Kriterium 4 ist wichtig, um Eventual Consistency zu erreichen. Transaktionsdaten werden von unterschiedlichen RM zu unterschiedlichen Zeitpunkten empfangen und somit in unterschiedlicher Reihenfolge in den Vorgängergraphen eingeordnet. Ein Aufruf des Konfliktlösungsalgorithmus muss aber bei gleicher Eingabe immer das gleiche Ergebnis liefern. Zu beachten ist, dass für einen Konfliktlösungsalgorithmus im allgemeinen nicht gilt, dass solveConflict(G, t_1, t_2) gefolgt von solveConflict(G, t_2, t_3) das gleiche Ergebnis wie solveConflict$(G, t_1, t3)$ liefert. Diese Eigenschaft muss für den Commitprozess berücksichtigt werden.

Eine weitere Eingabe eines Konfliktlösungsalgorithmus können die Transaktionen sein, die seit der letzten Konfliktlösung neu in den Vorgängergraphen eingefügt worden sind. Dann kann ein Konfliktlösungsalgorithmus von diesen Transaktionen bzw. deren Repräsentation im Graphen ausgehend die neue Datenbanksicht berechnen. Er muss so nicht die neue Sicht für alle im Konfliktlösungsintervall befindlichen Transaktionen bestimmen. Bei der nachfolgenden Diskussion einiger Konfliktlösungsalgorithmen ist diese Möglichkeit der Einfachheit halber nicht berücksichtigt.

Es werden nun drei Konfliktlösungsalgorithmen vorgestellt und ihre Korrektheit gezeigt. Jeder Algorithmus verwendet dabei ein bestimmtes Kriterium, nach dem entschieden wird, wie ein Konflikt zwischen mehreren Transaktionen an einem Konfliktelement gelöst wird. Ein solches Kriterium kann z.B. der Transaktionszeitstempel, die Id des die Transaktion initiierenden RM, oder ein Prioritätsfeld sein. Da eine Transaktion beliebig viele Konfliktelemente manipulieren kann, kann sie auch mit beliebig vielen anderen eventuell pro Konfliktelement unterschiedlichen Transaktionen in Konflikt stehen. Ein Konfliktlösungsalgorithmus muss deshalb zusätzlich eine Reihenfolge festlegen, in der einzelne Konflikte gelöst werden.

4.4.2 First-Wins-Head-Algorithmus

Der First-Wins-Head-Algorithmus löst Konflikte zwischen Transaktionen an einem Konfliktelement mittels derer Transaktionszeitstempel. Die Transaktion mit dem kleinsten Zeitstempelwert wird aktiviert, alle anderen deaktiviert. Um Eindeutigkeit zu gewährleisten, werden alle Transaktionen in aufsteigender Zeitstempelreihenfolge betrachtet, beginnend mit der ältesten Transaktion. Bildlich gesprochen werden Konflikte in dem Konfliktgraphen von oben nach unten, von den Wurzeln zu den Blättern, gelöst.

Algorithmus FIRSTWINSHEAD (Alg. 1) zeigt die Funktionsweise in Pseudocode. Dabei verwendet er die Funktionen HASDEACTIVATEDPARENT und HASACTIVATEDSIBLING. Da diese Funktionen sehr einfach sind, sind sie hier nicht in Pseudocode angegeben. Die Funktion HASDEACTIVATEDPARENT prüft in einer Schleife, ob mindestens ein Elternknoten eines gegebenen Knotens schon deaktiviert ist. Die Funktion HASACTIVATEDSIBLING prüft, ob ein gegebener Knoten mindestens einen aktivierten Geschwisterknoten hat.

Der Einfachheit halber betrachtet der hier vorgestellte Algorithmus nur Schreib-Schreib-Konflikte. Eine Erweiterung, um auch Schreib-Lese-Konflikte zu bestimmen, ist einfach möglich und wird in Abschnitt 4.4.5 beschrieben. Es werden alle lokal bekannten Transaktionen mit Zeitstempeln zwischen `start-` und `end` betrachtet. Werden die neu in den Vorgängergraphen eingefügten Transaktionen zusätzlich angegeben, kann zunächst die transitive Hülle über alle mit ihnen in Verbindung stehenden Transaktionen gebildet werden. Der Konfliktlösungsalgorithmus wird dann auf dieser Menge ausgeführt.

In dem Algorithmus wird zunächst in der Schleife in Zeilen 4 bis 8 der Zustand jeder Transaktion, die nicht schon festgeschrieben ist, auf undefiniert gesetzt. In der anschließenden Hauptschleife, in der über alle Knoten in Zeitstempelreihenfolge iteriert wird, wird für jede undefinierte Transaktion entschieden, ob sie aktiviert oder deaktiviert werden kann. Aktiviert werden kann eine Transaktion t, wenn sowohl alle ihrer Elternknoten aktiviert sind als auch keiner ihrer Geschwisterknoten bereits aktiviert ist. Sind diese Bedingungen erfüllt, wird t aktiviert. Alle Geschwistertransaktionen von t werden sofort in der Schleife in Zeilen 14 bis 16 deaktiviert. Somit muss später, wenn diese Transaktionen nach Zeitstempelreihenfolge an der Reihe wären, für diese nicht überprüft werden, ob sie aktiviert werden können. Es wird somit vermieden für jede dieser Transaktionen nochmals über alle Eltern und Geschwisterknoten zu iterieren. Der letzte Schritt der Hauptschleife besteht darin, den Eintrag in der Map `activeNodes` für jedes Konfliktelement, das von dieser Transaktion geschrieben wurde, mit einem Verweis auf die aktuelle Transaktion zu aktualisieren.

Korrektheit

Das System sorgt dafür, dass die in Abschnitt 4.4.1 definierten Korrektheitskriterien für Transaktionen zwischen 0 und *start* zu Beginn des Konfliktlösungsalgorithmus SOLVECONFLICT(G, START, END) erfüllt sind. Ein korrekter Konfliktlösungsalgorithmus muss dafür sorgen, dass diese Kriterien nach dessen Ausführung für alle Transaktionen mit

Algorithmus 1 First-Wins-Head-Algorithmus

```
    procedure FIRSTWINSHEAD(start: timestamp, end: timestamp))
                                          ▷ Betrachte nur Knoten zwischen start u. end
        NodeSet ← NodesByTimestampList.subset(start, end)
        for all Node n_i ∈ NodeSet do        ▷ Setzte Zustand aller Knoten auf undefiniert
 5:         if !n_i.isDefinite() then
                n_i.setUndefined()
            end if
        end for
        for all Node n_i ∈ NodeSet do
10:         if n_i.isUndefined() then
                if (!hasDeactivatedParent(n_i)) and (!hasActivatedSibling(n_i, end)) then
                    n_i.setActivated()
                                    ▷ Deaktiviere alle Nachbarknoten (aus Performancegründen)
                    for all s_i ∈ n_i.getSiblings() do
15:                     s_i.setDeactivated()
                    end for
                                                        ▷ Aktualisiere activatedNodes-Map
                    for all ConflictItem c_i ∈ n_i.getConflictItems() do
                        setActiveNode(c_i, n_i)
20:                 end for
                else
                    n_i.setDeactivated()
                end if
            end if
25:     end for
    end procedure
```

Zeitstempeln zwischen 0 und *end* gelten.

Dieses soll hier mit Hilfe einer Schleifeninvariante über die Hauptschleife in Zeilen 9 bis 25 gezeigt werden.

T_i sei die Menge aller Transaktionen, die bis zu Beginn des i-ten Schleifendurchlaufs entschieden, also aktiviert oder deaktiviert worden sind. Es gilt die Invariante, dass vor dem i-ten Durchlauf für alle $t_i \in T_i$ gilt:

1. Wenn t_i aktiviert ist, dann gibt es kein $t_k \in T_i$, das aktiviert ist und mit t_i in Konflikt steht.

2. Für alle Vorgänger t_p von t_i gilt, dass wenn t_i aktiviert ist, dann auch t_p.

3. t_i ist entweder aktiviert oder deaktiviert.

Initialisierung: Vor dem ersten Schleifendurchlauf ist $T_i = TA_{0,start}$. Nach Voraussetzung sind die Korrektheitskriterien für $TA_{0,start}$ erfüllt.

Aufrechterhaltung: Bei jedem Schleifendurchlauf wird der in Zeitstempelreihenfolge nächste Knoten t_j ausgewählt. Da Knoten in Zeitstempelreihenfolge betrachtet werden, sind alle Knoten t_i mit Zeitstempel kleiner als $t_j.timestamp$ schon in T_i. Der Algorithmus prüft, ob t_j einen deaktivierten Elternknoten (*hasDeactivatedParent*) hat. Hat t_j keinen deaktivierten Elternknoten, müssen alle seine Elternknoten bereits aktiviert sein (wegen Bedingung 3 der Invariante). Wegen Bedingung 1 kann ein neuer Konflikt nur dann entstehen, wenn t_j aktiviert wird und einer seiner Geschwisterknoten schon aktiviert ist. Genau dieses wird in *hasActivatedSibling* geprüft. Ist kein aktivierter Geschwisterknoten vorhanden, kann t_j aktiviert werden. Wegen Bedingung 2 sind alle Vorgängerknoten der Elternknoten von t_j bereits aktiviert und mit der Aktivierung von t_j sind somit alle Vorgängerknoten von t_j aktiviert. Kann t_j nicht aktiviert werden, wird er deaktiviert. Nach Durchlauf der Schleife gelten alle Bedingungen der Invariante also für alle Knoten in $T_{i+1} = T_i \cup t_j$

Termination: Mit jedem Durchlauf wird ein weiterer Knoten der Menge $TA_{start,end}$ betrachtet und entschieden. Am Ende sind also alle Knoten dieser Menge betrachtet worden und damit gelten die drei Bedingungen der Invariante für die Menge $TA_{0,end}$. Damit sind gleichzeitig die Korrektheitskriterien 1, 2 und 3 erfüllt.

Alle Knoten werden in Zeitstempelreihenfolge abgearbeitet und ein Knoten wird höchstens einmal aktiviert. Wird er aktiviert, so wird für alle Konfliktelemente, die er schreibt, eine Referenz auf ihn in die Map `activatedNodes` eingefügt und ein eventuell vorhandener älterer Eintrag überschrieben. Es ist so sichergestellt, dass die `activatedNodes` immer einen Verweis zu dem jüngsten aktivierten Knoten an dem jeweiligen Konfliktelement enthält.

4.4.3 Konfliktlösung mittels Prioritätswerten

Bei dem First-Wins-Head-Algorithmus kann der Anwendungsentwickler keinen direkten Einfluss darauf nehmen, welche Transaktionen im Konfliktfall bevorzugt werden sollen. Bei diesem Algorithmus kann er einer Transaktion während der initialen Transaktionsausführung und vor der Ausführung des lokalen Commit eine bestimmte Priorität zuweisen. Der *Prioritätsalgorithmus* löst Konflikte primär anhand dieses *Prioritätswertes*. Die Transaktion mit dem kleinsten Prioritätswert gewinnt im Konfliktfall. Bei gleichen Werten wird anhand der Transaktionszeitstempel entschieden. Transaktionen werden bei diesem Algorithmus in Prioritätsreihenfolge abgearbeitet.

In Zeile 3 des Algorithmus 2 werden alle Transaktionen in dem betrachteten Intervall zwischen `start` und `end` nach ihren Prioritätswerten sortiert. Nachdem anschließend alle noch nicht definitiv entschiedenen Transaktionen auf `undefiniert` gesetzt worden sind, beginnt in Zeile 9 die Hauptschleife, in der über alle Knoten iteriert wird. Für jeden noch undefinierten Knoten wird mittels der Funktion CANACTIVATE rekursiv geprüft, ob kein Vorgängerknoten bereits deaktiviert ist und ob kein Geschwisterknoten eines Vorgängerknotens bereits aktiviert ist. Die Angabe des Endzeitpunktes `end` bei dieser Funktion ist nötig, da Geschwisterknoten, die jünger als `end` sind, nicht betrachtet werden sollen. Deren Zustand kann bei früheren Ausführungen dieses Algorithmus bereits geändert worden sein und ist in Zeile 4 nicht auf `deaktiviert` gesetzt worden.

Ergibt die Prüfung durch CANACTIVATE, dass die momentan betrachtete Transaktion aktiviert werden kann, so geschieht dieses in der Methode ACTIVATEPATH. Diese Methode aktiviert rekursiv alle noch undefinierten Vorgängerknoten der betrachteten Transaktion. Außerdem sorgt sie dafür, dass die entsprechenden Einträge in der aktivatedNodes-Map aktualisiert werden und für jedes Konfliktelement auf die jüngste aktivierte Transaktion zeigen.

Korrektheit

Die Korrektheit dieses Algorithmus wird mittels einer Schleifeninvariante gezeigt.

Sei die Menge T_i die Menge aller Transaktionen, die bis zu Beginn des i-ten Schleifendurchlaufs entschieden, also aktiviert oder deaktiviert worden sind.

Es gilt die Invariante, dass vor dem i-ten Durchlauf der Hauptschleife in Zeile 9 für alle $t_j \in T_i$ gilt:

1. t_j ist entweder aktiviert oder deaktiviert.
2. Für alle Vorgänger t_p von t_j gilt, wenn t_j aktiviert ist, dann auch t_p.
3. Wenn t_j aktiviert ist, dann gibt es kein $t_k \in T_i$, das aktiviert ist und mit t_i in Konflikt steht.

Initialisierung: $T_i \leftarrow TA_{0,start}$. Für diese Menge gelten die Korrektheitskriterien und damit die Schleifeninvariante nach Voraussetzung.

Algorithmus 2 Prioritäts-Konfliktlösungsalgorithmus

```
procedure PRIORITYCONFLICTSOLVING(start: timestamp, end: timestamp))   ▷
Betrachte nur Knoten zwischen start u. end
    NodeSet ← NodesByTimestampList.subset(start, end)
    sortByPriority(NodeSet)          ▷ Sortiere alle Knoten aufsteigend nach Priorität
    for all Node n_i ∈ NodeSet do        ▷ Setzte Zustand aller Knoten auf undefiniert
5:      if (!n_i.isDefinite()) then
            n_i.setUndefined()
        end if
    end for
    for all Node n_i ∈ NodeSet do
10:     if n_i.isUndefined() then
            if !canActivate(n_i, end) then
                n_i.setDeactivated()
            else
                activatePath(n_i)
15:         end if
        end if
    end for
end procedure
```

Aufrechterhaltung: Für T_i gelten alle Bedingungen der Invariante. Zu zeigen ist, dass diese Kriterien nach einem Schleifendurchlauf auch für $T_{i+1} = T_i \cup t_i \cup T_{dep}$ gelten, mit $t_i \in TA_{start,end}$ und $T_{dep} \subseteq TA_{start,end}$.

In jedem Schritt wird der Knoten t_i mit dem nächst höchsten Prioritätswert ausgewählt. Ist t_i noch undefiniert, wird mittels `canActivate` rekursiv geprüft, ob t_i oder einer seiner Vorgängerknoten $t_p \in T_{0,t_i.timestamp}$ mit einem schon aktivierten Knoten in Konflikt steht und ob ein Knoten, von dem t_i bzw. ein t_p abhängt, schon deaktiviert ist. Es kann nicht vorkommen, dass ein Knoten von zwei noch nicht entschiedenen Knoten abhängt, die selbst miteinander in Konflikt stehen, was von `hasActivatedSiblings` nicht erfasst würde. In diesem Fall hätten diese Knoten schon einmal gleichzeitig aktiviert sein müssen, damit ein weiterer Knoten von diesen beiden abhängen kann. Dieses ist nicht möglich.

Sind die beiden Bedingungen, die `canActivate` prüft, falsch, so können t_i und alle noch undefinierten Vorgängerknoten von t_i aktiviert werden. All diese undefinierten Vorgängerknoten von t_i werden in der Menge T_{dep} zusammengefasst. Die Aktivierung aller Knoten in $t_i \cup T_{dep}$ geschieht mittels der rekursiven Prozedur `activatePath`.

Die Bedingung 2 ist somit erfüllt, da entweder der Vorgänger eines Knotens $t \in t_i \cup T_{dep}$ schon vor dem aktuellen Schleifendurchlauf aktiviert war, oder jetzt aktiviert

worden ist. Da auch kein Knoten in dieser Menge einen aktivierten Nachbarknoten hatte und ein Vorgängerknoten nicht Nachbarknoten eines Nachfolgers sein kann, ist auch Bedingung 3 erfüllt. Bedingung 1 gilt ebenfalls, da alle Knoten in $t_i \cup T_{dep}$ aktiviert worden sind.

Liefert `canActivate false`, so wird t_i deaktiviert und die Menge T_{dep} bleibt leer. Auch so sind die drei Bedingungen der Invariante für $T_i \cup t_i$ erfüllt.

Termination: Jede Transaktion in $TA_{start,end}$ ist genau einmal entschieden worden und wurde zu T_i hinzugefügt. Dieses ist entweder geschehen, als sie bei der Abarbeitung der Transaktionsliste in Prioritätsreihenfolge an der Reihe war, oder vorher im Zuge der rekursiven Aktivierung der Vorgängerknoten eines Knotens t_i. Es gilt also $T_i = TA_{0,end}$ und die Invariante gilt für alle Transaktionen in T_i. Damit gelten auch die Korrektheitskriterien 1, 2 und 3.

Es muss nun gezeigt werden, dass für jedes Konfliktelement in der Map `activatedNodes` ein Verweis auf die jüngste aktivierte Transaktion gespeichert ist.

Dieses gilt, da, nachdem ein Verweis für ein bestimmtes Konfliktelement einmal eingefügt worden ist, dieses höchstens mit einem Verweis auf eine jüngere Transaktion, die ebenfalls dieses Konfliktelement schreibt, überschrieben wird. Transaktionen werden zwar nicht in ihrer Zeitstempelreihenfolge abgearbeitet, wird eine Transaktion t_p, die nach Prioritätsordnung an der Reihe ist, aber aktiviert, so werden alle ihre Vorgängertransaktionen ebenfalls aktiviert. Beginnend mit der ältesten Vorgängertransaktion werden dann in der Funktion ACTIVATEPATH für jede Transaktion die entsprechenden Einträge in `activatedNodes` vorgenommen. Ein solcher Eintrag wird also höchstens von einer jüngeren Transaktion überschrieben. Da für jede aktivierte Transaktion die Einträge in `activatedNodes` vorgenommen werden, enthält sie schließlich auch für jedes Konfliktelement den entsprechenden Verweis.

Optimierung

Eine Optimierung des hier vorgestellten Algorithmus ist möglich, indem in der Funktion ACTIVATEPATH auch alle Geschwister eines aktivierten Knotens und deren Kindknoten deaktiviert werden. Diese Knoten können nicht mehr aktiviert werden, da sie mit einem aktivierten Knoten in Konflikt stehen, bzw. von einem deaktivierten Knoten abhängen. Deaktiviert man sie gleich hier, anstatt erst dann, wenn sie nach Prioritätsreihenfolge an die Reihe kämen, sparte man das rekursive Überprüfen in CANACTIVATE. Dieses kann teuer sein, wenn viele Vorgängerknoten überprüft werden müssen. , ob sie aktiviert werden können.

4.4.4 Last-Wins-Tail-Algorithmus

Der Last-Wins-Tail-Algorithmus ähnelt sehr dem Prioritätskonfliktlösungsalgorithmus. Deshalb wird darauf verzichtet, diesen Algorithmus ebenfalls in Pseudocode anzuge-

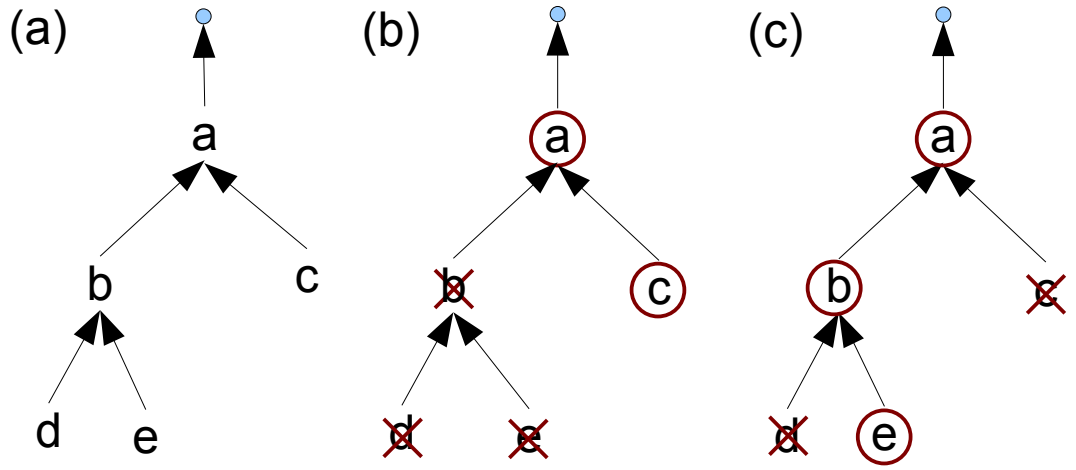

Abbildung 4.10: (a) Vorgängergraph ohne Konfliktlösung, (b) und (c) nach Konfliktlösung

ben. Anstatt einen Konflikt zwischen zwei Transaktionen anhand ihrer Prioritätsfelder zu entscheiden, wird der Konflikt anhand ihrer Transaktionszeitstempel entschieden. Hier gewinnt die Transaktion mit dem größeren Zeitstempel, also die, die zuletzt initiiert worden ist. Die Reihenfolge, in der alle Transaktionen betrachtet werden, ist hier die Zeitstempelreihenfolge, beginnend mit der jüngsten Transaktion (mit höchstem Zeitstempelwert). Bildlich gesprochen werden Konflikte von unten nach oben in dem Vorgängergraphen gelöst, also genau in umgekehrter Reihenfolge wie bei dem Head-First-Wins-Algorithmus. Dieses könnte aus Nutzersicht das natürlichere Verhalten sein, da eine später vorgenommene Änderung an einem Konfliktelement gegen eine kausal parallele, aber frühere Änderung im Konfliktfall gewinnt.

Bei diesem Algorithmus erkennt man, dass die weiter oben angesprochene Eigenschaft, dass SOLVECONFLICT(G, t_1, t_2) gefolgt von SOLVECONFLICT(G, t_2, t_3) im Allgemeinen nicht SOLVECONFLICT(G, t_1, t_3) entspricht, gilt. In Abbildung 4.10 (a) ist ein einfacher Vorgängergraph mit Transaktionen a bis e in dieser Zeitstempelreihenfolge dargestellt. Würde SOLVECONFLICT(G, $a.timestamp$, $d.timestamp$) (exklusive d) ausgeführt, so würde Transaktion c aktiviert und b deaktiviert. Ein anschließendes Ausführen von SOLVECONFLICT(G, $d.timestamp$, $e.timestamp + 1$) müsste nun auch d und e deaktivieren, da sie sich auf die bereits deaktivierte Transaktion b beziehen. Dieses Ergebnis ist in (b) dargestellt. Würde direkt SOLVECONFLICT(G, `a.timestamp`, `e.timestamp+1`) ausgeführt, so wären Transaktionen a, b und e aktiviert und c und d deaktiviert, wie in (c) zu sehen ist.

4.4.5 Berücksichtigung von Lesemengen

Die hier vorgestellten Konfliktlösungsalgorithmen haben nur Schreib-Schreib-Konflikte berücksichtigt. Eine Erweiterung, um auch Lesemengen und sich daraus ergebende Schreib-Lese-Konflikte zu erkennen, wird nun vorgestellt. Es gibt Stellen im Algorithmus, an denen über alle Nachbarknoten eines Knotens n iteriert wird. Hier muss nun unterschieden werden, ob n diesen Knoten schreibt, oder liest. Schreibt n den Knoten, wird zusätzlich zu den schreibenden über alle lesenden Nachbarn iteriert. Liest n den Knoten, wird über alle schreibenden Nachbarn iteriert. Lesende Nachbarn spielen in diesem Fall keine Rolle. An Stellen, an denen über alle Elternknoten von n iteriert wird, müssen die Eltern einbezogen werden, von denen n liest. Bei Iterationen über Kindknoten werden auch die Kinder berücksichtigt, die das Datenelement lesen, das n ändert.

4.5 Commit

Nachdem in den vorangegangenen Abschnitten beschrieben wurde, was Konflikte sind und wie sie erkannt und aufgelöst werden, wird nun dargestellt, wie endgültige Commitentscheidungen für Transaktionen getroffen werden.

Bisher wurde die Menge der vorläufig aktivierten und deaktivierten Transaktionen betrachtet, die auf jedem RM lokal vorliegen und deren Beziehungen untereinander in dem Vorgängergraphen repräsentiert sind. Für diese Transaktionen sollte so schnell wie möglich definitiv entschieden werden, ob sie festgeschrieben werden können oder abgebrochen werden müssen, damit kaskadierende Transaktionsabbrüche möglichst vermieden werden. Diese Entscheidung wird lokal von jedem RM getroffen, anhand von Informationen, die er von anderen RM der Replikationsgruppe erhalten hat. Jeder RM muss dabei zu den gleichen Ergebnissen gelangen. Überall müssen die gleichen Transaktionen festgeschrieben und die gleichen Transaktionen abgebrochen werden, um einen global konsistenten Datenbankzustand zu erhalten.

Dieses grundlegende Konsistenzkriterium von SYMORE ist *Eventual Consistency*. Es wird hier in Anlehnung an Saito und Shapiro [SS05] wie folgt definiert:

1. Zu jedem Zeitpunkt und für jeden Replikationsmanager gibt es ein Präfix einer Historie, das äquivalent zu einem Präfix einer Historie jedes anderen Replikationsmanagers ist. Dieses ist das *festgeschriebene Präfix*.

2. Das festgeschriebene Präfix jedes Replikationsmanagers wächst monoton.

3. Alle nicht abgebrochenen Operationen in dem festgeschriebenen Präfix erfüllen ihre Vorbedingungen.

4. Für jede ausgeführte Operation befindet sich irgendwann entweder die Operation selbst oder die Information über den Abbruch dieser Operation in dem festgeschriebenen Präfix.

Diese Definition lässt absichtlich Raum für verschiedenste Implementierungen. Je nachdem wie der Äquivalenzbegriff und wie Vorbedingungen von Operationen im konkreten System definiert sind, erfüllen so unterschiedliche Systeme wie z.B. das Usenet oder Bayou [PSTT96] die Kriterien für Eventual Consistency.

In dem vorliegenden System entspricht eine Operation einer Transaktion. Die folgenden Vorbedingungen müssen gelten, damit eine Transaktion aktiviert werden kann: Alle Vorgänger dieser Transaktion sind aktiviert und die Transaktion steht mit keiner aktivierten Transaktion in Konflikt (siehe auch Abschnitt 4.4.1).

Eine lokale Historie eines RM besteht aus den festgeschriebenen sowie aus den definitiv abgebrochenen Transaktionen in ihrer Commitreihenfolge und den darauf folgenden aktivierten, sich im Pre-Commit-Zustand befindlichen Transaktionen in ihrer Zeitstempelreihenfolge. Während sich das Präfix der Historie mit den festgeschriebenen Transaktionen nicht mehr ändert, kann sich der noch nicht festgeschriebene Teil dynamisch verändern. Um Eventual Consistency zu erreichen, muss das Commit-Protokoll garantieren, dass ein Präfix der festgeschriebenen Transaktionen der Historie eines RM äquivalent zu einem Präfix der Historie jedes anderen RM ist. Für äquivalente Historien gelten hier die gleichen Kriterien wie für herkömmliche Datenbanksysteme. Zwei Historien sind äquivalent, wenn sie die gleiche Menge von Transaktionen beinhalten und Konfliktoperationen der nicht abgebrochenen Transaktionen in derselben Reihenfolge ausführen [KE04, S. 303].

Im Folgenden wird der generische Commit-Algorithmus, der in SYMORE eingesetzt wird, beschrieben und anschließend gezeigt, dass er die oben genannten Kriterien für Eventual Consistency erfüllt.

Jeder RM kann autonom, auf Grund der Ereignisse, die er von allen anderen RM empfängt, entscheiden, welche Transaktionen festgeschrieben oder abgebrochen werden sollen. Eine explizite Koordination mit einem Master oder anderen RM mit dem Ziel, bestimmte Transaktionen festzuschreiben, findet nicht statt. Der Commit-Algorithmus trifft nicht für einzelne Transaktionen Commitentscheidungen, sondern für alle Transaktionen, deren Pre-Commit-Zeitstempel in einem *Commitintervall* liegen. Dieses Commitintervall ist begrenzt durch den *letzten Commitzeitpunkt* und einen neu gewählten *aktuellen Commitzeitpunkt*, wie in Abbildung 4.11 dargestellt. Für alle Transaktionen mit einem Zeitstempel vor dem letzten Commitzeitpunkt ist eine Commitentscheidung bereits getroffen. Sie sind entweder festgeschrieben oder abgebrochen worden. Alle Transaktionen nach diesem Zeitpunkt befinden sich im Pre-Commit-Zustand und sind entweder gerade aktiviert oder deaktiviert. Sobald alle Transaktionen, die von irgendwelchen RM der Replikationsgruppe initiiert worden sind und deren Zeitstempel kleiner als dieser aktuelle Commitzeitpunkt ist, lokal empfangen worden sind, wird der Konfliktlösungsalgorithmus bis zu diesem Zeitpunkt ausgeführt. Alle Transaktionen, die jetzt durch diesen aktiviert worden sind, werden festgeschrieben, alle, die in diesem Intervall deaktiviert worden sind, werden definitiv abgebrochen. Diese Transaktionen und die ihnen zugeordneten Daten können nun aus der `NodesByTimestamp`-Liste und unter bestimmten

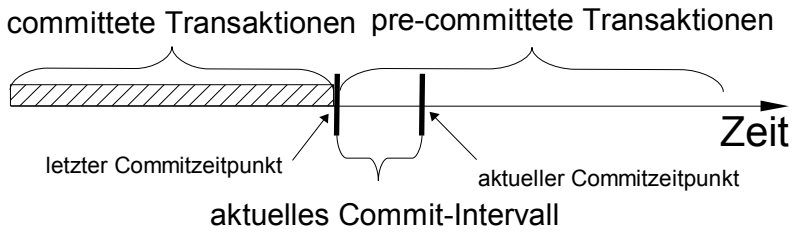

Abbildung 4.11: Historie eines Replikationsmanagers

Voraussetzungen (siehe Abschnitt 4.5.6) aus dem Vorgängergraphen entfernt werden. Nachdem dieser Commitdurchlauf abgeschlossen ist, wird ein neuer Commitzeitpunkt bestimmt und das Prozedere beginnt erneut.

Anhand der Daten des eindeutigen Zeitstempels, der jeder Transaktion zugeordnet ist (Abschnitt 4.2.2), kann bestimmt werden, ob alle Transaktionen mit Zeitstempel bis zu dem aktuellen Commitzeitpunkt, die auf einem bestimmten RM initiiert worden sind, bereits empfangen wurden. Ist bereits eine Transaktion dieses RM mit höherem Zeitwert als dem Commitzeitpunkt empfangen worden und auch alle Transaktionen vor dieser, erkennbar an der lückenlosen Sequenznummernfolge der Transaktionen, kann es keine weiteren, noch nicht empfangenen Transaktionen von diesem RM vor diesem Commitzeitpunkt geben.

Der Schlüssel zu Eventual Consistency liegt bei diesem Verfahren in der Wahl der Commitzeitpunkte. Wie in Abschnitt 4.4 bereits erläutert und an dem Beispiel in 4.4.4 verdeutlicht wurde, ist das Ergebnis einer Folge von Ausführungen des Konfliktlösungsalgorithmus im Normalfall davon abhängig, welche Zwischenschritte gewählt werden. Würden verschiedene RM eine Folge unterschiedlicher Commitzeitpunkte wählen, so würden die Konfliktlösungsalgorithmen eventuell unterschiedliche Transaktionen aktivieren oder deaktivieren. Die nun aktivierten Transaktionen würden festgeschrieben und die deaktivierten abgebrochen. Somit würde sich der Teil des Schedules mit den Transaktionen, für die Commitentscheidungen getroffen sind, auf verschiedenen RM unterscheiden. Da diese Entscheidungen definitiv sind, würde Eventual Consistency nicht erreicht. Der Commitprozess muss also dafür sorgen, dass jeder RM die Ausführung des Konfliktalgorithmus, die schließlich zu Commitentscheidungen von Transaktionen führt, mit den gleichen Start- und Endzeitpunkten vornimmt. Eine Ausnahme hiervon bildet der First-Wins-Head-Algorithmus, wie im nächsten Abschnitt gezeigt wird.

4.5.1 Korrektheit

Unter der Bedingung, dass die Wahl der Commitzeitpunkte konsistent erfolgt, kann nun gezeigt werden, dass das hier verwendete Commitverfahren im Sinn der oben gegebenen Definition von Eventual Consistency korrekt ist.

Das festgeschriebene Präfix einer Historie eines RM ist der Teil der Historie mit Opera-

tionen bis zu dem Zeitpunkt, bis zu dem auf allen RM eine Commitentscheidung bereits getroffen wurde. Der minimale letzte Commitzeitpunkt eines RM bestimmt diesen also für alle RM. Diese festgeschriebenen Präfixe aller Historien aller RM sind äquivalent, da sie genau die gleichen festgeschriebenen und abgebrochenen Transaktionen in gleicher Reihenfolge enthalten. Dieses gilt, da, wie oben beschrieben, der Konfliktlösungsalgorithmus von allen RM mit den gleichen Eingaben ausgeführt wird und somit nach Korrektheitskriterium 4 (Abschnitt 4.4) gleiche Ergebnisse erzielt.

Die Bedingung 2, dass das festgeschriebene Präfix jedes RM monoton wächst, gilt, da die Verteilungskomponente garantiert, dass irgendwann die Daten über alle irgendwo lokal initiierten Transaktionen von jedem RM empfangen werden (Abschnitt 4.6). Initiiert ein RM über eine einstellbare Zeitspanne keine neue Transaktion, so wird eine Dummy-Nachricht versandt. Somit wissen die anderen RM, dass sie nicht mit dem Commit auf noch unbekannte Transaktionen dieses RM warten müssen. Werden nun regelmäßig neue Commitzeitpunkte bestimmt, schreitet die Ausführung des Commitprozesses fort. Damit wächst auch das festgeschriebene Präfix jedes RM monoton. Wie diese Commitzeitpunkte bestimmt werden, wird in den nächsten Abschnitten erläutert.

Auch erfüllen die nicht abgebrochenen Operationen in dem festgeschriebenen Präfix jedes RM ihre Vorbedingungen (3). Dieses folgt daraus, dass alle Transaktionen festgeschrieben werden, die bei der vorangehenden Ausführung des Konfliktlösungsalgorithmus bis zu dem Commitzeitpunkt aktiviert worden sind. Dieser Algorithmus garantiert, dass nach dessen Ausführung für alle Transaktionen im Intervall von 0 bis zum Commitzeitpunkt die Vorbedingungen erfüllt sind.

Die Erfüllung der letzten Bedingung, dass jede Operation in dem festgeschriebenen Präfix berücksichtigt werden muss (4), folgt aus Bedingung 2 und daraus, dass Transaktionen nach ihrer Zeitstempelreihenfolge geordnet sind. Commitentscheidungen werden bis zu einem Commitzeitpunkt getroffen und dieser schreitet immer weiter fort. Auch das festgeschriebene Präfix wächst monoton, so dass irgendwann jede Transaktion Teil des festgeschriebenen Präfixes ist.

Unter der Bedingung, dass Commitzeitpunkte korrekt gewählt werden, erfüllt das hier beschriebene Commitverfahren also die Anforderungen an Eventual Consistency. Um diese Commitzeitpunkte zu bestimmen, bietet SYMORE drei unterschiedliche Commitprozesse. Diese werden im Folgenden näher erläutert.

4.5.2 Implizites Commit

Dieses Commit-Protokoll funktioniert nur zusammen mit Konfliktlösungsalgorithmen wie „First-Wins-Head“, bei denen das globale Commitergebnis einer Transaktion bestimmt werden kann, sobald alle existierenden älteren Transaktionen empfangen worden sind. Im Gegensatz zu den meisten anderen Konfliktlösungsalgorithmen spielt es bei diesem Konfliktlösungsalgorithmus keine Rolle, wie die konkreten Commitzeitpunkte gewählt werden, solange für alle Transaktionen auf allen RM letztendlich Commitentscheidungen getroffen werden.

Bei dem First-Wins-Head-Algorithmus werden die Transaktionen in ihrer Zeitstempelreihenfolge von der ältesten zur jüngsten Transaktion abgearbeitet. An Konfliktstellen in Vorgängergraphen gewinnt immer die älteste Transaktion. Ein RM R kann für eine Transaktion t also spätestens dann lokal eine Commitentscheidung treffen, wenn er von jedem anderen RM in seiner Replikationsgruppe alle Transaktionen bis zu dem Zeitstempel von t erhalten hat. Dann weiß R sicher, dass es keine weitere ältere, noch unbekannte Transaktion geben kann, die die Commitentscheidung von t noch verändern könnte.

Beweis

Angenommen eine Transaktion t_2 mit $t_2.timestamp > t.timestamp$ verändere das Commit-Ergebnis von t. Dazu müsste diese Transaktion an mindestens einem Konfliktelement mit t in Konflikt stehen. t_2 stehe mit t entweder in direktem Konflikt oder in indirekten Konflikt.

1. Indirekter Konflikt: Es gibt eine Transaktion t_3, die Vorfahre von t und t_2 ist. Da t_3 Vorfahre von t ist, ist t_3 älter als t und ist somit schon bekannt. Da Konflikte beginnend mit der ältesten Transaktion entschieden werden, wird vor t_2 und t entschieden, ob t_3 aktiviert oder deaktiviert wird. t_2 hat auf diese Entscheidung keinen Einfluss.

2. Direkter Konflikt: t und t_2 haben in einem Vorgängergraphen den gleichen Elternknoten t_3. Es muss gelten, dass $t_2.timestamp < t.timestamp$, damit der Konfliktlösungsalgorithmus t_2 aktiviert und nicht t. Dieses steht in Widerspruch zur Annahme.

In SYMORE speichert jeder RM in einem Vektor `SummaryVector` zu jedem anderen RM den Zeitstempel des Ereignisses, bis zu dem auch alle früheren Ereignisse dieses RM lückenlos empfangen worden sind. Ein Commit wird ausgelöst, wenn durch Empfang einer Transaktion oder eines anderen Ereignisses der älteste Zeitpunkt, der für irgendeinen RM in diesem Vektor gespeichert ist, erhöht wird. Eine Commitentscheidung kann dann für alle Transaktionen, die sich im Pre-Commit-Zustand befinden und deren Zeitstempel vor diesem Zeitpunkt liegen, getroffen werden.

4.5.3 Forced Commit

Wie oben beschrieben ist es bei Einsatz der meisten Konfliktlösungsalgorithmen wichtig, dass Commitzeitpunkte auf allen RM gleich gewählt werden. Eine Möglichkeit dieses zu erreichen ist es, Anwender selbst bestimmen zu lassen, zu welchem Zeitpunkt eine Commitentscheidung herbeigeführt werden soll. Ein Anwender führt dazu auf seinem lokalen RM einen *Forced-Commit*-Befehl aus. Daraufhin wird ein Forced-Commit-Ereignis generiert, dem der Zeitstempel der Ausführung des Forced-Commit-Befehls zugewiesen wird und das, genau wie normale Transaktionen auch, an die anderen RM verteilt wird.

Jeder RM wählt nun übereinstimmend den diesem Ereignis zugeordneten Zeitstempel als neuen Commitzeitpunkt, wodurch Eventual Consistency gewahrt bleibt.

Es stellt auch kein Problem dar, wenn von verschiedenen RM unabhängig, kurz hintereinander Forced-Commit Ereignisse generiert werden, ohne dass sie die jeweils anderen schon erhalten haben. Ein Forced-Commit ist ein Ereignis wie eine Transaktion auch. Das bedeutet, ein RM führt das eigentliche Commit erst aus, wenn er alle Ereignisse von allen RM bis zu dem Forced-Commit-Zeitpunkt empfangen hat. Ist unter diesen Ereignissen ein weiteres, früheres Forced-Commit-Ereignis, wird die Commitentscheidung zuerst bis zu dessen Zeitstempel getroffen. Auf diese Weise werden Commitentscheidungen auf allen RM zwar zu unterschiedlichen Echtzeit-Zeitpunkten aber überall mit der gleichen Eingabe getroffen. So ist sichergestellt, dass überall die gleiche Ausgabe produziert wird und die verschiedenen Datenbankzustände konvergieren.

4.5.4 Scheduled Commit

Eine weitere Möglichkeit Commitzeitpunkte zu bestimmen besteht darin, automatisch, regelmäßig, zu festgesetzten Zeitpunkten auf allen RM den Commitprozess auszuführen. Das Commitintervall $commit_{int}$ legt die Zeitspanne zwischen zwei Commits fest und muss auf jedem RM gleich festgelegt werden, um Eventual Consistency zu gewährleisten. Mit $CommitTS = (\lfloor \frac{time}{commit_{int}} \rfloor + 1) * commit_{int}$ lässt sich lokal der initiale Commitzeitpunkt bestimmen. Ausgehend von diesem Zeitpunkt werden in $commit_{int}$-Abständen weitere Commits ausgeführt.

Auch hier wird ein Commit mit $commitTS$ in die lokale Ereigniswarteschlange gestellt. Das Commit wird erst ausgeführt, wenn von allen RM alle Nachrichten bis zu diesem Zeitpunkt empfangen worden sind. Damit ist sichergestellt, dass der Commitalgorithmus überall auf der gleichen Eingabe ausgeführt wird.

Der Vorteil dieses Verfahrens gegenüber dem Forced Commit ist es, dass Commitentscheidungen automatisch getroffen werden, ohne Eingriff des Anwenders. Außerdem werden keine expliziten Commitnachrichten verschickt.

4.5.5 Inaktivität

Allen Commitalgorithmen ist gemein, dass sie nur Commitentscheidungen für Transaktionen bis zu Zeitpunkten treffen können, bis zu denen sie Informationen über die Ereignisse aller anderen RM der Replikationsgruppe empfangen haben. Ein RM, der lange inaktiv ist und keine lokalen Transaktionen erzeugt, verzögert also das Commit aller.

Aus diesem Grund verteilt ein RM nach einer einstellbaren Untätigkeitszeitspanne eine *Dummynachricht*. Diese Nachricht besagt, dass in dem Zeitraum von der letzten regulären Nachricht dieses RM bis zum Zeitpunkt der Dummynachricht keine Ereignisse auf diesem RM stattgefunden haben. Dummynachrichten müssen nur verteilt werden, wenn durch andere RM Aktivität in der Replikationsgruppe stattfindet. Erzeugt kein RM neue Ereignisse, so müssen auch keine weiteren Dummynachrichten verteilt werden.

Jeder RM merkt sich den Zeitstempel des letzten lokalen Ereignisses t_e, das er erzeugt hat. Wird eine fremde Nachricht (außer einer Dummynachricht) empfangen, wird geprüft, ob deren Zeitstempel hinter t_e liegt. In diesem Fall wird ein Timer gestartet, der nach $t_e + dummyMessageInterval$ ausgelöst wird und dabei eine Dummynachricht verschickt. Der Zeitstempel t_e wird anschließend auf den Zeitpunkt des Abschickens der Dummynachricht gesetzt. Wird ein eigenes reguläres Ereignis verteilt, während ein Timer gestellt ist, wird der Timer gelöscht.

4.5.6 Löschen aus dem Vorgängergraphen

Zusätzlich zur Konfliktlösung müssen irgendwann Knoten von festgeschriebenen und abgebrochenen Transaktionen aus dem Vorgängergraphen gelöscht werden. Nach jeder Ausführung eines Commit-Protokolls wird dazu der Algorithmus TRIM-GRAPH ausgeführt.

Es kann Transaktionen geben, deren Zeitstempel hinter dem letzen Commitzeitpunkt liegt und die sich auf Transaktionen vor diesem Zeitpunkt beziehen. Würden nach einem Commit nun alle Knoten vor diesem Zeitpunkt aus dem Vorgängergraphen gelöscht werden, fehlten Bezugspunkte. Auch eine neu empfangene Transaktion könnte nicht ohne weiteres in den Vorgängergraphen eingefügt werden. Somit kann auch nicht entschieden werden, ob sie aktiviert oder deaktiviert werden sollte, da diese Entscheidung von dem Zustand ihrer Eltern und Geschwisterknoten beeinflusst wird.

Der Löschalgorithmus entfernt deshalb nicht jeden Knoten, dessen Zeitstempel vor dem letzten Commitzeitpunkt liegt, aus dem Graphen. Jeder Knoten, der festgeschrieben und damit auch aktiviert ist und den höchsten Zeitstempel aller Knoten mit Zeitpunkt vor dem Commitzeitpunkt an einem Konfliktelement hat, bleibt erhalten. Er wird an den Wurzelknoten dieses Konfliktelementes gehängt. Es ist nur noch der Zustand der zugeordneten Transaktion von Bedeutung. Alle anderen Daten dieser Transaktion werden gelöscht. Verweise von Knoten mit einem Zeitstempel nach dem Commitzeitpunkt, die sich auf Knoten beziehen, die gelöscht worden sind, werden auf die Elternknoten der gelöschten Knoten umgesetzt.

Grafik 4.12 veranschaulicht dieses Vorgehen. In Teil (b) wurde der Löschalgorithmus bis zum Zeitpunkt von Transaktion f einschließlich ausgeführt. Transaktionen a und e werden gelöscht, weil es Transaktion f als die letzte aktivierte Transaktion an dem Konfliktelement III gibt. Transaktion c wird gelöscht, weil sie deaktiviert ist. Transaktion d zeigt nun auf die Vorgänger von c und, da Transaktion a aus dem Graphen entfernt wurde, damit auf die Wurzelelemente der Konfliktelemente II und III. Transaktion b kann noch nicht gelöscht werden, weil sie die letzte aktivierte Transaktion an Konfliktelement II ist.

Eine neu empfangene Transaktion, die sich auf mindestens einen gelöschten Knoten bezieht, muss in jedem Fall deaktiviert werden. Entweder ist ihr Elternknoten im Rahmen des Commitprozesses abgebrochen worden und ist deswegen aus dem Graphen gelöscht worden, oder ihr Elternknoten war zwar aktiviert, hatte aber einen Kindknoten,

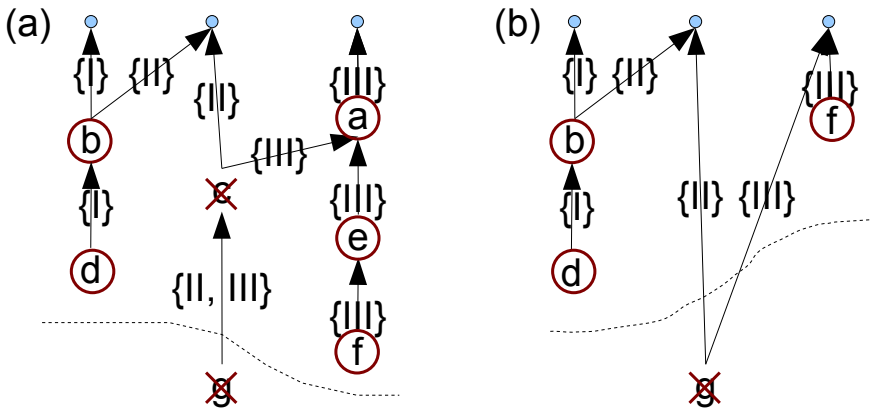

Abbildung 4.12: Vorgängergraph mit Transaktionen $a, \ldots, g$ (in dieser Zeitreihenfolge) an den Konfliktelementen I, II, III. (a) Vorgängergraph vor dem Trimming, (b) nach Trimming bis Zeitpunkt f

der ebenfalls aktiviert war und festgeschrieben wurde. Im ersten Fall muss der neue Knoten nach Korrektheitskriterium 2 deaktiviert werden, im zweiten Fall nach Kriterium 1. Nur wenn sich die neu ankommende Transaktion auf den in Zeitstempelreihenfolge letzten festgeschriebenen Knoten an einem Datenelement bezieht, kann ihr Zustand nicht sofort entschieden werden und wird auf „undefiniert“ gesetzt. In beiden Fällen wird die Transaktion an die vorhandenen referenzierten Knoten angehängt und in die sortierte Knotenliste eingefügt.

Um zu entscheiden, ob ein Referenzknoten im Graphen fehlt, weil er schon gelöscht worden ist oder weil er noch nicht empfangen wurde, ist es nötig, zu jeder Referenz den Zeitstempel des referenzierten Knotens zu übertragen. Damit ist klar, dass eine Referenz, die im Graphen fehlt und deren Zeitstempel kleiner als der Commitzeitpunkt ist, schon gelöscht worden ist. Eine Referenz mit einem größerem Zeitstempel als dem Commitzeitpunkt ist noch nicht empfangen worden. Da Zeitstempel global eindeutig sind, genügt es, referenzierte Transaktionen mit diesem Zeitstempel eindeutig zu bezeichnen. Eine zusätzliche Transaktions-Id ist nicht nötig.

4.6 Datenverteilung

Lokale Transaktionen und Commit-Ereignisse müssen möglichst effizient an alle Mitglieder der Replikationsgruppe verteilt werden. Ein Verteilungsverfahren muss damit umgehen können, dass einzelne Knoten vorübergehend nicht verbunden sind und dass sich die Netztopologie dynamisch verändert. Weiterhin sollten die charakteristischen Eigenschaften von Funknetzen ausgenutzt werden (geteiltes Medium, jede Nachricht ist implizit Broadcast) und insgesamt sparsam mit Nachrichten umgegangen werden.

In mobilen Umgebungen werden häufig *epidemische Algorithmen* zur Datenverteilung angewendet. Epidemische Algorithmen sind dadurch gekennzeichnet, dass Daten nach-

richtenweise verteilt werden und einzelne Knoten Nachrichten auch für andere Knoten vorhalten. So kann ein Knoten gespeicherte Nachrichten auf Anfrage an andere Konten weiterleiten (pull) oder er kann seine gesammelten Nachrichten entweder in periodischen Abständen oder bei Kontakt mit anderen Knoten verteilen (push). Auf diese Weise empfangen auch Knoten, die nur sporadisch mit dem Netzwerk verbunden sind, alle Nachrichten. Nachrichten verbreiten sich wie Viren bei einer Epidemie von einem Knoten zum nächsten. Epidemische Algorithmen garantieren im Normalfall nicht, wie schnell eine bestimmte Nachricht verteilt wird, sondern nur, dass jede Nachricht bei nicht permanent partitioniertem Netz irgendwann von jedem Knoten empfangen wird.

In SYMORE wird ein solches epidemisches Datenverteilungsverfahren verwendet. Es nutzt eine Kombination von adaptiven Push- und Pull-Protokollen. Nachrichten werden garantiert irgendwann von jedem Knoten empfangen, es sei denn, sie werden mit einer so hohen Frequenz erzeugt, dass interne Puffer überlaufen und Nachrichten verworfen werden müssen. Genauere Informationen zur Funktionsweise des Datenverteilungsverfahrens finden sich in [Sch05b].

4.7 Gruppenverwaltung

In diesem Kapitel wird beschrieben, wie Verfahren aussehen können, mittels derer ein RM einer Replikationsgruppe beitreten, diese regulär verlassen oder aus dieser von anderen ausgeschlossen werden kann, falls er als ausgefallen betrachtet wird. Bei diesen Verfahren muss darauf geachtet werden, dass weiterhin Eventual Consistency gewahrt bleibt. Es dürfen insbesondere durch den Beitritt eines RM keine Inkonsistenzen im Rahmen des Commitprozesses auftreten. Auch beim Ausscheiden eines RM aus der Gruppe darf der Commitprozess nicht zum Erliegen kommen, weil auf Nachrichten des nicht mehr vorhandenen RM gewartet wird.

4.7.1 Gruppenbeitritt

Das System sollte eine Möglichkeit bieten, neue Gruppenmitglieder aufzunehmen. Ein neues Gruppenmitglied muss dabei den aktuellen globalen Datenbankzustand erhalten und seine Präsenz in der Gruppe allen Gruppenmitgliedern bekannt machen. Ein einfaches Protokoll wird im Folgenden beschrieben.

Jede Replikationsgruppe ist durch eine eindeutige Kennung bezeichnet. Es wird davon ausgegangen, dass der RM A, der der Replikationsgruppe beitreten möchte, diese Kennung kennt. Andernfalls könnten durch ein einfaches Protokoll die Gruppen-Ids der unmittelbaren Nachbarn ermittelt werden und daraus die gewünschte ausgewählt werden.

Zunächst sendet A eine Broadcastnachricht JOIN-REQUEST mit dem Beitrittswunsch an seine unmittelbaren Nachbarn. Diese antworten mit einer JOIN-REPLY-Nachricht, die ihren `SummaryVector` und ihre IP-Adresse enthält. Nach Ablauf einer kurzen

Wartezeit entscheidet sich der beitretende RM für den Knoten mit dem aktuellsten `SummaryVector` und baut eine TCP-Verbindung zu diesem, im Folgenden B genannt, auf. Über diese Verbindung fordert A den aktuellen Datenbankzustand von B an. Dieser sendet daraufhin seinen stabilen Zustand, sowie alle vorläufigen Transaktionen und die `SkewMatrix`. Während des Übertragungsvorgangs darf auf B für keine neuen Transaktionen ein lokales Commit ausgeführt werden, da diese Transaktionen nicht übertragen würden. Wurde die Übertragung erfolgreich abgeschlossen, verteilt B eine Nachricht ANNOUNCE-JOIN, in der der Gruppenbeitritt des neuen RM angekündigt wird. Ab diesem Zeitpunkt darf der neue RM aktiv an der Replikationsgruppe teilnehmen.

Dadurch, dass B den Beitritt des neuen RM ankündigt, ist gewährleistet, dass weiterhin jeder RM Commits mit derselben Gruppe an Teilnehmern und damit mit den gleichen Transaktionen, also auf der gleichen Eingabe, ausführt. Ein Commit nach ANNOUNCE-JOIN findet erst dann statt, wenn lokal alle Nachrichten aller Gruppenmitglieder bis zum Commitzeitpunkt vorliegen. D.h., auch die ANNOUNCE-JOIN-Nachricht muss vorliegen und somit wird der neue RM für das Commit einbezogen.

Das Protokoll muss natürlich mit möglichen Fehlerfällen umgehen können. Ein möglicher Fehlerfall liegt vor. wenn der RM B, mit dem sich A synchronisiert, während dieses Vorgangs ausfällt. In diesem Fall muss A den Beitrittsvorgang mit einem anderen RM wiederholen.

4.7.2 Verlassen der Gruppe

Um die Replikationsgruppe ordnungsgemäß zu verlassen, muss ein RM A eine LEAVE-Nachricht verteilen. Damit wissen alle anderen RM dieser Gruppe, dass sie für Commits, die Commitzeitpunkte nach dem Zeitpunkt der LEAVE-Nachricht haben, A nicht mehr einbeziehen müssen.

Da die verwendete Verteilungskomponente Nachrichten nur „so gut sie kann“ (best effort) verteilt und es keine Möglichkeit gibt, zu erfahren, ob eine an sie zur Verteilung übergebene Nachricht wirklich schon verteilt worden ist, sollte die LEAVE-Nachricht zusätzlich auf Anwendungsebene an As Nachbarn gesendet werden. Diese Nachbarn quittieren den Empfang und sorgen ebenfalls für die Weiterverteilung der Nachricht. Erst wenn A mindestens eine Bestätigung über den Empfang seiner LEAVE-Nachricht erhalten hat, darf er die Gruppe wirklich verlassen.

4.7.3 Ausschluss aus der Gruppe

Ist ein Knoten ausgefallen, wäre es wünschenswert, dessen RM aus der Replikationsgruppe ausschließen zu können. Trotz des Ausfalls eines RM können andere RM weiterhin Transaktionen ausführen und untereinander austauschen. Es können allerdings keine Transaktionen mehr durch den Commitprozess definitiv festgeschrieben werden. Wie in Abschnitt 4.5 beschrieben, ist jedem Commit ein fester Commitzeitpunkt zugeordnet und

das Commit wird erst ausgeführt, wenn von jedem RM alle Nachrichten bis zu diesem Zeitpunkt empfangen worden sind.

Der Ausschluss eines RM erfordert die Abstimmung aller verbleibenden Gruppenmitglieder. Es ist nicht möglich, dass ein RM R autonom entscheidet, dass ein anderer RM R_{fail} ausgefallen ist. Eventual Consistency wäre nicht länger gewahrt. Um dieses zu zeigen wird angenommen, dass R die Entscheidung, dass R_{fail} ausgefallen ist, zu einem bestimmten Zeitpunkt t_{fail} trifft. Es gibt dann einen Zeitpunkt t_{last}, bis zu dem R die letzte Nachricht von R_{fail} empfangen hat. Da R_{fail} als ausgefallen betrachtet wird, wartet R nicht mehr auf Nachrichten von R_{fail} für Commitzeitpunkte nach t_{last} um Commitentscheidungen zu treffen. R kann also eine Commitentscheidung treffen, sobald alle Nachrichten von allen RM außer R_{fail} bis zu dem Commitzeitpunkt erhalten wurden. Problematisch ist, wenn R_{fail} nicht wirklich ausgefallen ist, R dieses aber fälschlicherweise annimmt. Dann könnte ein anderer RM Nachrichten von R_{fail} mit Zeitstempeln nach t_{last} empfangen, die für Commitentscheidungen berücksichtigt werden. Das festgeschriebene Präfix dieses RM sieht also eventuell anders aus, als das von R. Damit ist Eventual Consistency nicht mehr gewährleistet.

Auch die Verteilung der Entscheidung durch R, dass R_{fail} ab Zeitpunkt t_{last} als ausgefallen angesehen wird, führt zu keiner Lösung. Ein anderer RM könnte eine Transaktion von R_{fail} mit Zeitstempel zwischen t_{last} und dem Zeitstempel der Nachricht über den Ausfall von R_{fail} schon empfangen und festgeschrieben haben, bevor die Meldung über den angenommenen Ausfall von R_{fail} empfangen wird.

Um einen RM auszuschließen, ist also eine explizite Abstimmung zwischen allen RM der Replikationsgruppe nötig. Ein RM, der annimmt, dass ein anderer ab Zeitpunkt t_{last} ausgefallen ist, kann diese Meldung verteilen. Erst wenn von allen anderen RM eine Bestätigung empfangen worden ist, dass auch dort keine Nachrichten von R_{fail} nach t_{last} vorliegen, darf wirklich, ohne Berücksichtigung von fehlenden Nachrichten dieses RM, das Commit stattfinden. Hat ein anderer RM Nachrichten mit einem späteren Zeitstempel von R_{fail} erhalten und bereits festgeschrieben, bestätigt er den Ausschlussversuch von R_{fail} nicht und R_{fail} wird auf keinem RM ausgeschlossen.

4.8 1-Kopien-Serialisierbarkeit

Neben Eventual Consistency garantiert SYMORE *1-Kopien-Serialisierbarkeit (1-copy-serializability, 1SR)*. Während Eventual Consistency zusichert, dass der Datenbankzustand jedes Knotens letztendlich gleich ist, trifft 1SR zusätzlich eine Aussage über die Historien, die in einem replizierten Datenbanksystem möglich sind. 1-Kopien-Serialisierbarkeit garantiert, dass die Effekte aller in einem replizierten Datenbanksystem ausgeführten Transaktionen die gleichen sind, als wenn sie sequentiell auf einer einzigen zentralisierten Datenbank ausgeführt worden wären.

Im Folgenden wird gezeigt, dass SYMORE 1SR erreicht, unter der Bedingung, dass Konflikte durch parallelen Zugriff auf Datenelemente erkannt werden. Es muss also Konflikt-

granularität „cell“ für alle Zellen aller Tabellen eingestellt sein und sowohl die Ermittlung der Schreib- als auch der Lesemengen aktiviert sein.

RD- und 1C Historien

Für diesen Beweis wird auf die Definition von 1SR von Bernstein [BHG87] zurückgegriffen. Dieser verwendet *Replicated-Data-Historien (RD-Historien)* und *1-Kopien-Historien (one-copy (1C) histories)* zur formalen Definition von 1SR.

Eine RD-Historie ist, vereinfacht dargestellt, eine Historie, bei der alle Lese- und Schreiboperationen von Transaktionen auf konkreten Kopien x_A von Datenelementen x arbeiten (z.B. $r_2(y_B), w_1(x_A)$). Wie bei Historien zentralisierter Datenbanksysteme bestimmt eine RD-Historie eine partielle Ordnung über den Operationen einer Menge von Transaktionen $T = \{T_0, \ldots, T_n\}$. Die Ordnungsrelation $<$ ergibt sich aus den Konfliktoperationen der in der Historie enthaltenen Transaktionen. Transaktionen in RD-Historien bestehen aus den gleichen Operationen wie Transaktionen in zentralisierten Datenbanken: Lesen (r), Schreiben (w), Commit (c) und Abort (a). Zwei Operationen stehen miteinadner in Konflikt, wenn beide auf die gleiche Kopie des gleichen Datenelementes zugreifen und mindestens eine Operation eine Schreiboperation ist. Im Folgenden werden ausschließlich Historien betrachtet, die nur Transaktionen enthalten, die ihr Commit ausgeführt haben.

Eine 1C-Historie dagegen ist eine Historie, deren Operationen nicht zwischen verschiedenen Kopien eines Datenelementes unterscheiden.

Definition von 1SR

Bernstein definiert nun:

Definition. *Eine RD-Historie ist 1-kopien-serialisierbar, wenn sie äquivalent zu einer seriellen 1-Kopien-Historie ist.*

Bernstein zeigt weiter, dass diese Äquivalenz dann gegeben ist, wenn eine RD-Historie H dieselben *Liest-von-Beziehungen (reads-from relationships)* wie eine serielle 1C-Historie H_{1C} hat. Außerdem muss die Ordnung der Transaktionen in H_{1C} mit der durch den Konfliktgraphen $SG(H)$ gegebenen übereinstimmen.

Die Liest-von-Beziehung zwischen zwei Transaktionen t_i, t_j einer Historie H ist definiert als: t_i liest x von t_j, wenn $w_j[x] < r_i[x]$ und $a_j \not< r_i[x]$ und, wenn es ein $w_k[x]$ mit $w_j[x] < w_k[x] < r_i[x]$ gibt, dann $a_k < r_i[x]$.

Um zu beweisen, dass ein System 1SR garantiert, muss gezeigt werden, dass jede RD-Historie, die dieses System produzieren kann, äquivalent zu einer 1-Kopien-Historie ist.

Transaktionsmodell

Das von Bernstein verwendete Systemmodell unterscheidet sich von dem von SYMORE. Um für SYMORE zu zeigen, dass 1SR gewährleistet ist, muss ein äquivalentes Transak-

tionsmodell definiert werden, das kompatibel mit den in der Definition verwendeten RD-Historien ist. Bernstein geht von einem zentralen Transaktionsmanager (TM) aus, auf dem Nutzer Transaktionen initiieren, die auf logische Datenelemente zugreifen. Der TM übersetzt die Operationen dieser Transaktionen in Operationen auf konkrete Kopien dieser Datenelemente, die auf verschiedenen Datenmanagern (DM) gespeichert sind. In SYMORE werden Transaktionen auf einem lokalen Replikationsmanager (RM) initiiert und greifen direkt auf bestimmte Kopien von Datenelementen dieses DM zu. Anschließend werden diese Transaktionen an andere RM verteilt. Dort werden dieselben Operationen nun auf die dortigen Kopien angewendet. Beziehungen zwischen Transaktionen, die bei der initialen Ausführung ermittelt wurden, werden ebenfalls verteilt.

In SYMORE gibt es demnach initiale lokale Transaktionen $\tilde{t}_{i/X}, X \in R = \{RM_1, \ldots, RM_n\}$ und anschließend je initialer Transaktion für jeden weiteren RM $Y, Y \in R \backslash X$ eine Updatetransaktion $t_{i/Y}$, die die Effekte der initialen Transaktion auch dort geltend macht. Die initiale und alle zugehörigen Updatetransaktionen können als Teiltransaktionen einer globalen Transaktion t_i betrachtet werden. Statt zwischen der lokalen Transaktionsausführung und der anschließenden Aktualisierung aller Kopien zu trennen, werden diese Aktionen als eine gemeinsame Transaktion betrachtet. Es wird davon abstrahiert, wo eine Transaktion ausgeführt wird und wie Transaktionsdaten verteilt werden.

Serialisierbare RD-Historie

Jeder RM in SYMORE enthält einen Vorgängergraphen (VG). Dieser bestimmt eine partielle Ordnung der lokal bekannten Teiltransaktionen über deren Vorgängerbeziehungen. 1SR bezieht sich nur auf das festgeschriebene Präfix jedes VG. Dieses wird, wie in Abschnitt 4.5 beschrieben, bis zu einem aktuellen Commitzeitpunkt durch Ausführung des Konfliktlösungsalgorithmus gebildet. Deshalb sind in ihm keine Konflikte zwischen aktivierten bzw. festgeschriebenen Transaktionen enthalten. Der Fall, dass $t_{i/X} = pred(t_{j/X})$ und $t_{i/X} = pred(t_{k/X})$ an einem gleichen Konfliktelement gilt, wird verhindert, indem $t_{j/X}$ oder $t_{k/X}$ abgebrochen wird.

Aufgrund der Anwendung des gleichen Konfliktlösungsalgorithmus bis zu den gleichen Zeitpunkten sind somit alle festgeschriebenen Präfixe aller VG aller RM gleich. Sie enthalten die jeweils lokalen Teiltransaktionen derselben globalen Transaktionen in derselben partiellen Ordnung. Durch topologisches Sortieren kann aus jedem Graphen eine lokale Historie gebildet werden, die diese partielle Ordnung beibehält.

Diese lokalen Historien können zu einer globalen RD-Historie H zusammengeführt werden. Dazu werden die einzelnen Teiltransaktionen unter Beibehaltung der durch die lokalen Historien definierten Ordnung so angeordnet, dass alle zusammengehörigen Teiltransaktionen hintereinander angeordnet sind. Zusätzlich muss gelten, dass jede initiale Teiltransaktion vor ihren Updatetransaktionen in H angeordnet ist.

Über einen Widerspruchsbeweis wird nun gezeigt, dass die so gebildete RD-Historie H serialisierbar ist. Ihr Konfliktgraph (serializability graph) $SG(H)$ enthält also keine

Zyklen:

Beweis. Es wird angenommen, dass H einen Zyklus der Länge 2 enthält. Dann gibt es Operationen $p, r \in t_i$ und $q, s \in t_j$ mit $t_i, t_j \in H$. Es gelte $p < q$ und $s < r$. p und q müssen auf einem gemeinsamen RM arbeiten und ein gemeinsames Datenelement manipulieren, damit sie ein Konfliktpaar bilden können. Dieser RM sei X mit $X \in R$. Demnach ist p in Teiltransaktionen $t_{i/X}$ und q in Teiltransaktion $t_{j/X}$ enthalten. Es gilt $t_{i/X} < t_{j/X}$ in H. Wären r und s ebenfalls in $t_{i/X}$ bzw. $t_{j/X}$ enthalten, so müsste $r < s$ gelten, da Teiltransaktionen sich nicht überlappen. Dieses steht im Widerspruch zur Annahme. Es gibt also eine Teiltransaktion $t_{i/Y}$, in der r enthalten ist und eine Teiltransaktion $t_{j/Y}$ mit s mit $Y \in R \backslash X$. Somit gilt nach der Annahme $t_{j/Y} < t_{i/Y}$.

Es folgt also, dass H sowohl $t_{i/X} < t_{j/X}$, als auch $t_{j/Y} < t_{i/Y}$ enthält. Dieses steht im Widerspruch dazu, dass, wie oben erläutert, die Ordnung der lokalen Teiltransaktionen aller Transaktionen in jedem VG gleich ist. Per Induktion kann dieser Widersruch ebenfalls für Zyklen mit einer Länge > 2 gezeigt werden. □

Äquivalenz zu 1C

In einem letzten Schritt wird nun gezeigt, dass die serialisierbare RD-Historie H äquivalent zu einer seriellen 1C-Historie ist.

Eine 1C-Historie H_{1C}, deren Transaktionsordnung die durch $SG(H)$ gegebene Ordnung berücksichtigt, ist seriell. Zusätzlich müssen zwischen H und H_{1C} die gleichen Liest-von-Beziehungen bestehen. Dabei muss gelten, dass wenn t_i ein Datenelement x von t_j in H liest, dann liest t_i dieses Datenelement von t_j auch in H_{1C}.

Alle Transaktionen können in H seriell geordnet werden. Ändert eine Transaktion eine Kopie eines Datenelementes in H, dann ändert die gleiche Transaktion auch alle anderen Kopien auf die gleiche Weise. Greift in H eine Teiltransaktion $t_{i/X}$ auf ein Datenelement x zu, das von einer Teiltransaktion $t_{j/X}$ geschrieben wurde, so greift auch jede andere Teiltransaktion $t_{i/Y}$ jedes anderen RM Y auf x, geschrieben von $t_{j/Y}$, zu. Sie erhält in jedem Fall die gleiche Version t_j dieses Datenelementes. In der äquivalenten 1C-Historie wird nicht zwischen verschiedenen Kopien eines Datenelementes unterschieden. Da Transaktionen in H serialisierbar ausgeführt werden, liest auch die Transaktion t_j in H_{1C} die gleichen Versionen wie ein beliebiges t_{j/X_m}.

Es ist somit gezeigt, dass auf Grund der Art, wie in Symore Konflikte erkannt und gelöst werden und eine global konsistente Ordnung aller Teiltransaktionen erreicht wird, alle Historien, die SYMORE produzieren kann, 1SR sind.

5 Implementierung

Nachdem im vorangegangenen Kapitel das Replikationskonzept vorgestellt wurde, wird nun dessen Umsetzung in einen Prototypen beschrieben.

Nutzer interagieren mit dem Replikationssystem über SQL-Anweisungen. Die Diskussion im vorangegangenen Kapitel hat aber ausschließlich elementare Lese- und Schreiboperationen betrachtet. Ein wichtiger Teil der Implementierung ist es deshalb, zu bestimmen, welche Daten- und Konfliktelemente von einer SQL-Anweisung gelesen oder geschrieben werden. Ein weiterer Aspekt betrifft den Umgang mit Nebenläufigkeit, die sich durch mehrere parallele lokale Nutzertransaktionen und der parallel dazu stattfindenden Verarbeitung empfangener Transaktionsdaten ergibt. Auch wurde noch nicht erläutert, wie Änderungen der vorläufigen Datenbanksicht tatsächlich ablaufen.

Zunächst wird in diesem Kapitel die statische Struktur der Implementierung auf Komponenten- und Paketebene vorgestellt, bevor anschließend die Verarbeitungsschritte in SYMORE von der Erzeugung bis zur Verteilung und vom Empfang bis zum Commit einer Transaktion erläutert werden. Dabei wird auf die eben genannten Aspekte an den entsprechenden Stellen im Arbeitsablauf eingegangen.

5.1 Übersicht

SYMORE implementiert die JDBC-Schnittstelle, wie sie im Java Community Process JSR 169 für das *Connected Device Configuration (CDC) Foundation Profile* von *J2ME* für kleine und mobile Geräte definiert ist. Diese ist eine strikte Teilmenge von *JDBC 3.0*. Vor allem veraltete Schnittstellen und Methoden wurden entfernt und einige Bereiche der API wurden an die besonderen Eigenschaften von Geräten mit begrenzten Ressourcen angepasst. Eine Anwendung kommuniziert über diese Schnittstelle mit SYMORE. Sie sieht das replizierte Datenbanksystem also wie einen herkömmlichen Java-Datenbanktreiber. Replikationsspezifische Erweiterungen, die gegenüber zentralisierten Datenbanken nötig sind, sind größtenteils als Erweiterung von SQL implementiert.

Der vorliegende Prototyp implementiert die elementar notwendigen Methoden der JDBC-API. Eine genaue Übersicht dieser befindet sich in der Javadoc-Dokumentation.

Wie bereits beschrieben, besteht die Systemarchitektur von SYMORE aus drei Hauptkomponenten (siehe Abschnitt 4.1). Die zentrale Komponente ist der Replikationsmanager (RM), der alle Aspekte der Replikation behandelt, wie Konflikterkennung, Konfliktlösung und das Erreichen von Eventual Consistency. Zum persistenten Speichern der Daten verwendet er intern ein leichtgewichtiges eingebettetes Java-Datenbanksystem.

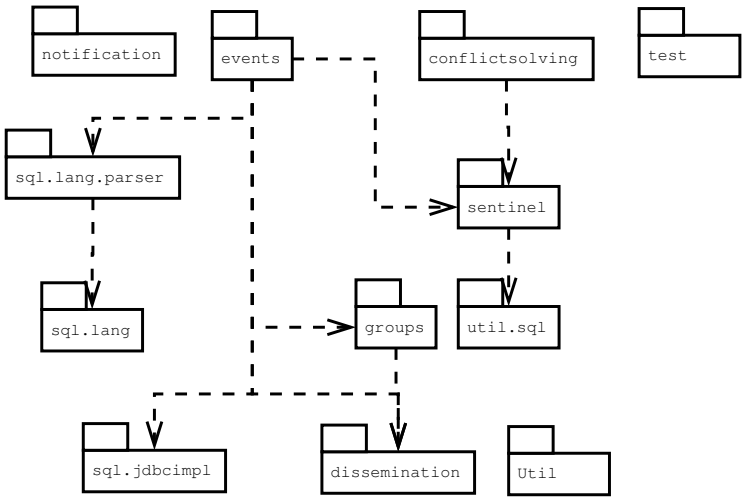

Abbildung 5.1: Paketdiagramm des Replikationsmanagers

In dem vorliegenden Prototyp wurde dazu Apache Derby verwendet, prinzipiell kann aber jede Datenbank eingesetzt werden, für die eine JDBC-Schnittstelle nach JSR-169 existiert und die das Isolationslevel „serializable“ unterstützt. Zur Kommunikation und effizienten Verteilung von Nachrichten in MANETs und zur Synchronisation mit anderen RM wird eine Komponente zur Datenverteilung eingesetzt. Verteilungskomponente und Konfliktlösungsalgorithmus sind konfigurierbar und können ausgetauscht werden, ohne das System neu übersetzen zu müssen. Das Replikationssystem selbst ist als Teil einer Anwendung in diese eingebettet.

5.1.1 Architektur des Replikationsmanagers

In Grafik 5.1 sind die wichtigsten Komponenten des RM und ihre Abhängigkeiten in einem UML-Paketdiagramm dargestellt. Die Pakete sind hauptsächlich nach fachlichen Kriterien gegliedert. Ihre Aufgaben werden im Folgenden erläutert.

sql.jdbcimpl: Dieses Paket enthält die Implementierungen der JDBC-Interfaces `DataSource`, `Connection` und `Statement`. Dieses sind die Klassen, mit denen ein Nutzer des Systems direkt interagiert.

symore.dissemination.*: Dieses Paket besteht aus generischen Klassen und Interfaces für die Datenverteilung und Synchronisation. Außerdem befinden sich hier konkrete Implementierungen von Datenverteilungskomponenten und Fabriken und Adapter, um auf die eingesetzte externe Datenverteilungskomponente aus [Sch05b] zugreifen zu können.

symore.sql.lang.*: Hier befindet sich die SQL-Grammatik sowie der daraus generierte Parser. Ebenfalls sind in diesem Paket alle Klassen der Syntaxbäume, die der Parser

aus SQL-Eingaben generiert, enthalten. Insbesondere befinden sich hier die Klassen, die die verschiedenen SQL-Anweisungen repräsentieren und damit Bestandteil einer Transaktion sind.

symore.groups: Klassen dieses Paketes verwalten die lokale Sicht auf die Replikationsgruppe. Informationen über die Mitglieder der Replikationsgruppe werden hier gespeichert, wie deren Id oder für jeden RM der Zeitstempel der letzten lückenlos empfangenen Nachricht. Außerdem befindet sich hier die `SkewMatrix`, die den Zeitunterschied zwischen der lokalen Uhr und der jedes RM speichert.

symore.sentinel: Hier befindet sich die Datenstruktur des Vorgängergraphen und alles, was zu deren Verwaltung, zur Definition von Konfliktgranularitäten und zur Bestimmung von Konfliktelementen nötig ist.

symore.conflictsolving: Dieses Paket enthält die Schnittstelle, die Konfliktlösungsalgorithmen implementieren müssen, sowie einige vordefinierte Konfliktlösungsalgorithmen.

symore.notification: Hier befinden sich die Interfaces, die eine Anwendung implementieren muss, um über Änderungen an der Datenbank mittels des Beobachter-Entwurfsmusters informiert zu werden. Außerdem enthält dieses Paket die Infrastruktur, die notwendig ist, um diese Änderungen zu ermitteln.

symore.events: Dieses Paket enthält Klassen, die die verschiedenen Ereignistypen (`Transaction`, `Commit`, `DummyMessage`) repräsentieren sowie den `EventManager`, der empfangene Ereignisse entsprechend behandelt.

5.1.2 Interne vs. externe Implementierung des Replikationssystems

Ein erster Entwurf des Replikationssystems sah vor, die Replikationslogik direkt in Derby oder eine andere unter einer Open Source Lizenz stehende Java-Datenbank zu integrieren. Dieses hätte den Vorteil, effizienter als eine externe Implementierung zu sein. Für die Replikationslogik ist es nötig, die SQL-Anweisungen des Anwenders zu parsen, da auf bestimmte Bestandteile dieser Anweisungen zugegriffen werden muss (siehe Abschnitt 5.2). Würde die Replikationslogik in das Datenbanksystem integriert, könnte auf dieses zusätzliche Parsen verzichtet werden. Außerdem könnte effizienter auf interne Datenbankinformationen wie das Data Dictionary zugegriffen werden. Auch könnte Code, der Funktionalität wie z.B. Cachingmechanismen oder die Generierung von eindeutigen Ids bereitstellt, wiederverwendet werden. Nachteilig an einem solchen Ansatz wäre, dass ein Wechsel auf eine neue Version der verwendeten Datenbank erschwert würde. Die eigenen Erweiterungen müssten mühsam mit dem neuen Datenbankcode zusammengeführt werden. Ein weiteres wesentliches Kriterium, das gegen diesen Ansatz spricht, ist die hohe Komplexität des Datenbanksystems und die Einarbeitungszeit, die nötig gewesen wäre, um die replikationsspezifischen Erweiterungen dort integrieren zu können.

Das vorliegende System verwendet deshalb eine separate Replikationskomponente, die intern eine relationale Datenbank zur persistenten Speicherung der Daten verwendet und mit dieser über deren JDBC-Schnittstelle kommuniziert. Neben geringerer Komplexität bietet dieses den Vorteil, dass prinzipiell jedes Datenbanksystem eingesetzt werden könnte. Nachteilig ist, dass einige Verarbeitungsschritte nun doppelt vorgenommen werden müssen, wie etwa das doppelte Parsen von SQL-Anweisungen.

5.2 Von einer SQL-Anweisung zur Datenverteilung

In den folgenden Unterabschnitten wird schrittweise der Verarbeitungsfluss erläutert, der durchlaufen wird, wenn eine SQL-Anweisung lokal auf dem Replikationssystem ausgeführt wird.

Wie bereits erwähnt, erscheint das Replikationssystem für die Anwendung wie ein normaler JDBC-Datenbanktreiber. Um Daten manipulieren oder abfragen zu können, muss eine Anwendung also zunächst über das `DataSource`-Objekt der Datenbank ein `Connection`-Objekt erhalten. Damit können `Statement`-Objekte erzeugt werden, auf denen die eigentlichen Anfragen ausgeführt werden. Wird die erste SQL-Anweisung einer neuen Transaktion mittels einer der `execute`-Methoden des `Statement`-Objektes ausgeführt, wird eine neue Transaktion gestartet. Intern wird ein `Transaction`-Objekt erzeugt, das alle zu der Transaktion gehörigen Daten zusammenfasst.

5.2.1 Parser

Als ersten Verarbeitungsschritt muss die als String vorliegende SQL-Anweisung geparst werden. Dieses geschieht, um die von der Anweisung betroffenen Konfliktelemente und Datenbankzeilen ermitteln zu können.

Der eingesetzte Parser wird von dem *Java Compiler Compiler (JavaCC)* [Jav06] generiert. Mit dessen Hilfe wird aus einer Eingabedatei, in der die Grammatik spezifiziert ist, ein Parser für diese erzeugt. Dieser erzeugt aus einer SQL-Anweisung in Textform einen Syntaxbaum in Form einer Objektstruktur. Diese repräsentiert die geparste Anweisung und ermöglicht so deren Weiterverarbeitung. In der Eingabedatei ist die Grammatik in einer Notation ähnlich der erweiterten Backus-Naur-Form (EBNF) spezifiziert, die zusätzlich mit Angaben zur Generierung eines die Eingabe repräsentierenden Objektbaumes versehen ist.

In dem vorliegenden Prototypen wird kein vollständiger SQL-Standard unterstützt. Die verwendete SQL-Grammatik basiert auf der Grammatik der Open-Source Java Datenbank „mckoi“ [Die04]. Sie wurde vereinfacht und angepasst, um einen für SYMORE günstigen Objektbaum zu erzeugen. Alle elementaren SQL-Anweisungen wie `Create Table`, `Drop Table`, `Select`, `Values`, `Insert`, `Update` und `Delete` werden unterstützt, wenn auch nicht in jedem Fall in dem vollen Umfang wie in SQL vorgesehen. Zusätzlich zu diesen werden Anweisungen unterstützt, die dazu dienen, die Replikation zu

steuern. Diese sind `Forced Commit`, um ein Forced Commit zu initiieren, `Synchronize`, um einen Synchronisationsvorgang der Datenverteilungskomponente zu initiieren, oder `Set Priority <int>`, um die Priorität einer Transaktion festzulegen. Die Anweisung `Create Table` wurde erweitert, um Konfliktgranularitäten definieren zu können. Für eine genaue Spezifikation der eingesetzten Grammatik sei auf den Anhang verwiesen.

Syntaxbaum

Der mittels JavaCC generierte Parser erzeugt aus einer SQL-Anweisung eine diese repräsentierende Objektstruktur. Der Parser ist möglichst einfach gehalten, so dass einige mögliche Syntax- und Semantikfehler in einer geparsten SQL-Anweisung deshalb von der vorliegenden Implementierung nicht entdeckt werden. Beispielsweise wird nicht überprüft, ob angegebene Datentypen gültig sind. Dieses verursacht keine Probleme, da Fehler in einer SQL-Anweisung gefunden werden, wenn sie schließlich auf der Hintergrunddatenbank ausgeführt wird. Der dort eingesetzte Parser prüft die Korrektheit der Anweisung und signalisiert Fehler, die dann entsprechend behandelt werden.

In Abbildung 5.2 ist ein UML-Klassendiagramm dargestellt. Dieses zeigt die Klassen in dem Paket symore.sql.lang, die für die Ausführung einer SQL-Anweisung verwendet werden. Der Übersicht halber sind nur die wichtigsten Methoden dargestellt. Die Klassen dieses Paketes gliedern sich in zwei Vererbungshierarchien. Die erste Hierarchie umfasst alle Klassen, die von `Exp` erben. Diese sind möglicher Bestandteil eines Objektbaumes, der Teile einer geschachtelten SQL-Anweisung repräsentiert. Eine Anweisung besteht aus `Operanden`, die durch `Operatoren` miteinander verbunden sind. Für die Repräsentation dieser Hierarchie wird das Collection-Entwurfsmuster angewendet. Operanden und Operatoren werden in der Klasse `Expression` zusammengefasst, die selbst auch ein `Operand` ist. So ist es möglich, beliebig geschachtelte Anweisungen darzustellen.

Die zweite Hierarchie umfasst alle Klassen, die von der Klasse `Statement` abgeleitet sind. Jedes Exemplar einer bestimmten Unterklasse von `Statement` ist die Wurzel eines erzeugten Objektbaumes und repräsentiert eine SQL-Anweisung. Es enthält die einzelnen Bestandteile dieser als Listen bzw. Operanden-Hierarchien.

5.2.2 Ausführung einer lokalen Anweisung

Jede dieser von `Statement` abgeleiteten Klassen überschreibt die Methode `execute` mit den für die entsprechende Anweisung spezifischen Aktionen. Nach dem Parsen einer Anweisung liefert der Parser ein Exemplar einer solchen Klasse zurück, auf dem anschließend diese Methode aufgerufen wird.

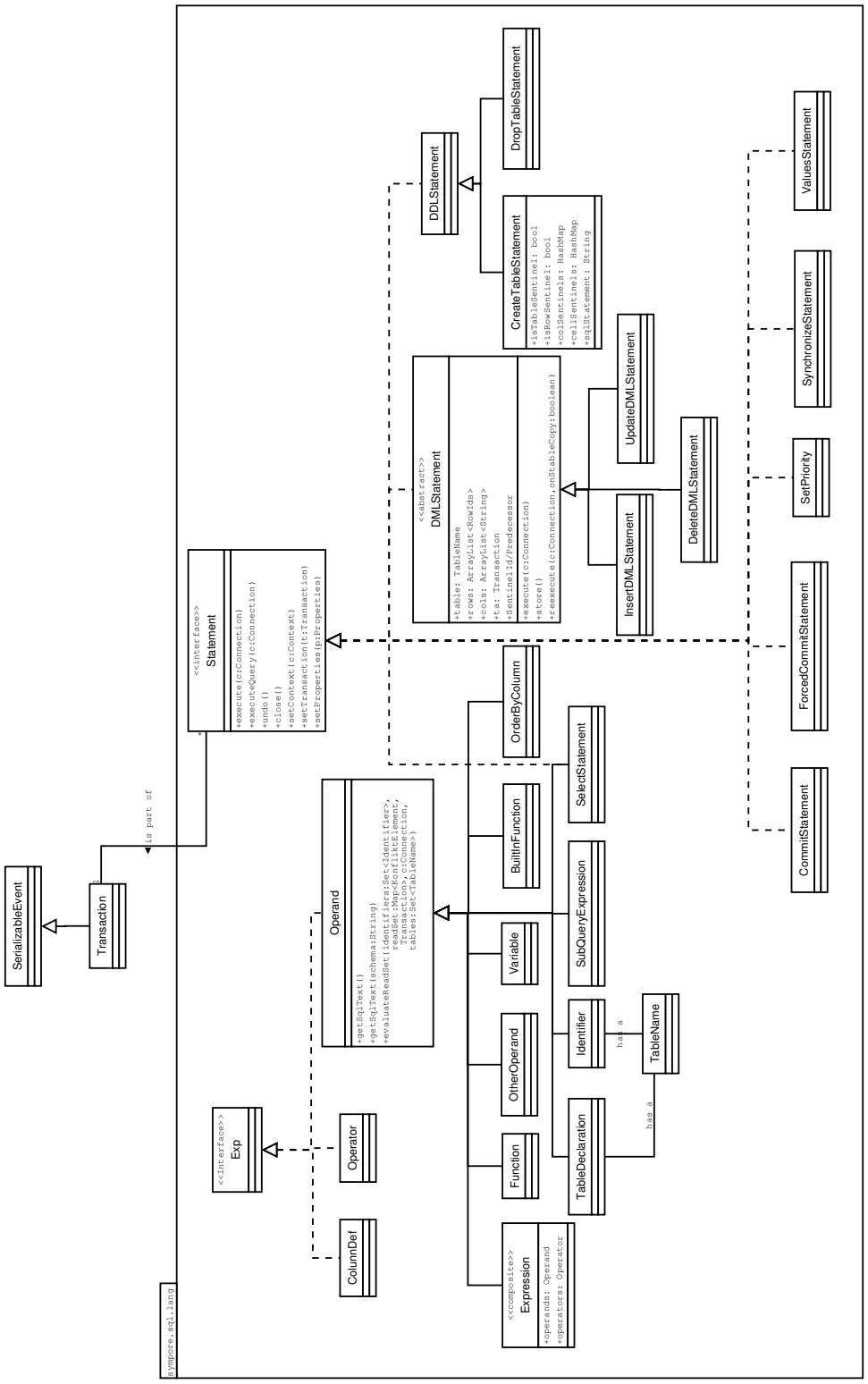

Abbildung 5.2: Klassendiagramm des Paketes symore.sql.lang

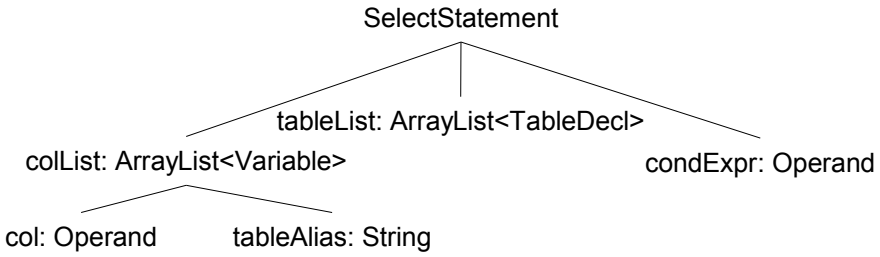

Abbildung 5.3: Vereinfachter Objektbaum einer Select-Anweisung

Lese- und Schreibmengen

Die Methode `execute` einer Anweisung der Datenmanipulationssprache (DML) von SQL sorgt dafür, die entsprechende Anweisung auf der vorläufigen Datenbanksicht auszuführen. Außerdem werden durch diese Methode alle Konfliktelemente bestimmt, die von dieser Anweisung gelesen oder geschrieben werden. Dieses ist notwendig, um die zugehörige Transaktion in die Vorgängergraphen einfügen und Konflikte feststellen zu können. Eine `Delete`-Anweisung schreibt Konfliktelemente, die Anweisungen `Update` und `Insert` können diese zusätzlich lesen. `Select`- und `Values`-Anweisungen lesen Konfliktelemente.

Im Folgenden wird zunächst die Ausführung einer `Select`-Anweisung beschrieben, um anschließend die Ausführung der `Update`-Anweisung exemplarisch für Anweisungen, die Daten manipulieren, zu erläutern. Die weiteren Datenmanipulationsanweisungen sind ähnlich realisiert. Weitere Anweisungen, wie z.B. `Create Table` oder `Drop Table`, werden hier nicht behandelt, da ihre Implementierung keine Besonderheiten aufweist.

Select: Ist die optionale Erkennung von Schreib-Lese-Konflikten über die Konfigurationsdatei oder einen Konfigurationsparameter für das Replikationssystem aktiviert, müssen die durch `Select`-Anweisungen gelesenen Konfliktelemente ermittelt werden.

Eine `Select`-Anweisung in SYMORE hat folgende Form:

```
SELECT [DISTINCT | ALL] <colExpr> (, <colExpr>)*
  FROM <table> (, <table>)*
  [WHERE <condExpr>] [ORDER BY <orderByList>]
```

Nach dem Parsen liegt diese in Form eines Objektbaumes vor, wie vereinfacht in Abbildung 5.3 dargestellt ist.

In Abschnitt 4.3 wurde erläutert, dass ein Konfliktelement eine Menge von Datenelementen beschreibt. Um zu bestimmen, von welchen Konfliktelementen eine `Select`-Anweisung liest, müssen zunächst die Datenelemente ermittelt werden, von denen sie liest. Alle Spalten, Reihen und Tabellen dieser Elemente müssen dazu bestimmt werden.

Der Parser erzeugt aus einer `Select`-Anweisung ein Objekt des Typs `SelectStatement`. Dieses enthält in der Liste `tableList` je einen einfachen mit dem zugehörigen Schema qualifizierten Tabellennamen. Unteranfragen an dieser Stelle werden nicht unterstützt und außer dem Kreuzprodukt mehrerer Tabellen keine weiteren Join-Formen wie „inner join“, „outer join“ usw. Die Bestimmung der Tabellen, von denen gelesen wird, ist somit recht einfach möglich.

Die in der `Select`-Anweisung angegebenen Spalten sind in der Liste `colList` gespeichert. Diese besteht aus Objekten des Typs `Variable`, die je ein Objekt des Typs `Operand` enthalten. Anders als in der Tabellenliste unterstützt die vorliegende Implementierung an dieser Stelle Unteranfragen, die einen skalaren Wert zurückliefern. Das konkrete `Operand`-Objekt ist deshalb nicht notwendigerweise ein einfacher Spaltenbezeichner. Es kann vom Typ `Expression` sein, und somit Unteranfragen oder Funktionen wie `AVG`, `SUM` usw. enthalten. Weiterhin kann es die Abkürzung „*“ enthalten, die alle Spalten bezeichnet. Um die Lesemenge zu bestimmen, müssen alle konkreten Spaltenbezeichner ermittelt werden. Falls ein Spaltenbezeichner nicht explizit mit der zugehörigen Tabelle qualifiziert wurde, muss auch deren Name ermittelt werden, da mehrere Tabellen Spalten gleichen Namens enthalten können.

Da jedes Element dieses möglicherweise geschachtelten Teilbaumes an einem Spaltenelement ein Untertyp von `Operand` ist, erbt es von diesem die Methode `evaluateReadSet(Set identifiers, Set readSet, Set tables, ..)`. Diese wurde für jeden Untertyp passend überschrieben und wird auf dem Wurzelelement mit einer leeren Menge `readSet` und leeren Menge `identifiers` aufgerufen. Die Menge `tables` enthält die Tabellennamen, die im vorherigen Schritt ermittelt wurden. Für jeden konkreten `Operand` bestimmt diese Methode die Spaltenbezeichner, die er repräsentiert, ermittelt zu welchen Tabellen diese gehören und fügt sie der Menge `identifiers` hinzu. Enthält ein `Operand` selbst `Operand`-Objekte, wird die Methode `evaluateReadSet` auf jedem dieser Objekte rekursiv aufgerufen. Im Falle von Unteranfragen werden so rekursiv deren Lesemengen bestimmt und der Menge `readSet` hinzugefügt.

Im nächsten Schritt werden alle Tabellen, aus denen keine Spalte in die Ergebnismenge projiziert wurde, aus der oben bestimmten Tabellenliste herausgefiltert. Anschließend werden für alle verbliebenen Tabellen die Zeilen bestimmt, von denen gelesen wird. In relationalen Datenbanken ist eine Datenbankreihe durch ihren Primärschlüssel eindeutig gekennzeichnet. Der Einfachheit halber und um das Datenaufkommen möglichst gering zu halten, werden in SYMORE keine beliebigen, benutzerdefinierten Primärschlüssel unterstützt. Ein Primärschlüssel ist Teil der Beschreibung eines Konfliktelementes und muss mit der Lese- und Schreibmenge übertragen werden. Stattdessen wird vorausgesetzt, dass jede Tabelle den Primärschlüssel „rowId“ enthält. Bei Erzeugung einer Datenbankzeile kann für diesen automatisch ein eindeutiger Wert bestimmt werden, er kann aber auch vom Nutzer explizit gesetzt werden.

Um die Zeilen zu bestimmen, von denen gelesen wird, müssen also deren rowIds

ermittelt werden. Dazu wird eine SQL-Anweisung erzeugt, die mittels des Prädikats `condExpr` des Originalausdrucks die Datenbankzeilen selektiert, von denen gelesen wird und auf die rowIds aller gelesenen Tabellen projiziert:

`SELECT` t_1`.rowId,` t_2`.rowId, ...,` t_n`.rowId FROM` $t_1, t_2, \ldots, t_n$
`WHERE <condExpr>`

Mit diesen Informationen, den qualifizierten Spaltennamen, den Reihen und den Tabellen, können nun die Konfliktelemente bestimmt werden, die von dieser Anweisung gelesen werden. Dazu werden für jede Konfliktgranularität, die für eine durch diese Anweisung gelesene Tabelle definiert ist, die entsprechenden Konfliktelemente erzeugt. Diese sind durch Objekte des Typs `SentinelId` dargestellt.

Schließlich wird die eigentliche `Select`-Anweisung auf der Datenbank ausgeführt. War deren Ausführung erfolgreich, werden die Konfliktelemente in dem aktuellen Transaktionsobjekt gespeichert und das durch die Ausführung der eigentlichen `Select`-Anweisung erhaltene `SQL-ResultSet` an den Aufrufer zurückgegeben. Ist bei der Ausführung ein Fehler aufgetreten, wird dieser signalisiert und die Anweisung abgebrochen. Da noch keine Konfliktelemente an das Transaktionsobjekt weitergereicht wurden, befinden sich die Datenstrukturen weiterhin in einem konsistenten Zustand. Die Ausführung der `Select`-Anweisung ist hiermit beendet.

Update: Für SQL-Anweisungen, die Daten manipulieren, müssen die Konfliktelemente bestimmt werden, die geschrieben werden. Die Ermittlung dieser erfolgt ähnlich wie die Bestimmung der Konfliktelemente, die gelesen werden. Sie ist sogar einfacher, da durch solch eine Anweisung immer nur Felder einer einzigen Tabelle manipuliert werden können. Außerdem sind die Spalten, die geschrieben werden, explizit in der Anweisung angegeben. Es wird hier exemplarisch für die `Update`-Anweisung gezeigt, wie bei der Bestimmung der Schreibmenge vorgegangen wird.

Eine `Update`-Anweisung hat in SYMORE folgende Form:

```
UPDATE <table> SET <col>=<valExpr> (, <col>=<valExpr> )*
  [WHERE <condExpr>]
```

Zunächst werden alle Datenbankzeilen bestimmt, in denen diese Anweisung schreibt. Dazu wird die SQL-Anfrage `SELECT rowId FROM <table> WHERE <condExpr>` auf der Datenbank ausgeführt. Die Information, welche Tabelle und welche Spalten betroffen sind, befindet sich explizit in der `Update`-Anweisung und wurde durch den Parser bereits ermittelt. Mit diesen Informationen können nun die Konfliktelemente, die geschrieben werden, bestimmt und die entsprechenden `SentinelId`-Objekte erzeugt werden.

Die `Update`-Anweisung wird nun auf der vorläufigen Datenbanksicht ausgeführt. Sind hierbei keine Fehler aufgetreten, wird anschließend das Prädikat `<condExpr>` so umgeschrieben, dass es explizit die eben ermittelten Zeilen bezeichnet. Dieses ist notwendig, da diese Anweisung wahrscheinlich später erneut lokal oder entfernt ausgeführt

wird. Dabei sollen genau dieselben Zeilen selektiert werden wie bei ihrer initialen Ausführung. Durch paralleles Einfügen weiterer Zeilen in die Tabelle könnte eine erneute Ausführung ansonsten auch andere als die ursprünglichen Zeilen betreffen. Das neue Prädikat hat nun die Form `rowId IN` $\texttt{rowId}_1, \texttt{rowId}_2, \ldots, \texttt{rowId}_n$. Dieses Umschreiben ist für `Select`-Anweisungen, die nur lesen, nicht nötig, da sie nicht übertragen und nicht später erneut ausgeführt werden. Bei `Delete`-Anweisungen hingegen muss es erfolgen.

Während durch Umschreiben des Prädikates `<condExpr>` verhindert wird, dass die Anweisung bei erneuter Ausführung andere Datenbankzeilen schreibt, wird standardmäßig nicht verhindert, dass sie später andere Datenbankzeilen liest. Die `<valExpr>` sind beliebige Ausdrücke, deren Operanden sich auf die gerade zu ändernde Zeile oder, mittels Unteranfragen, auf beliebige Datenelemente der Datenbank beziehen können. Es ist zunächst Aufgabe des Anwendungsentwicklers dafür zu sorgen, dass keine unerwünschten Effekte entstehen, wenn eine Unteranfrage an diesen Stellen bei späteren Ausführungen andere Daten liest als bei der initialen Ausführung. Insbesondere muss verhindert werden, dass sie bei einer späteren Ausführung mehr als ein Element zurückliefert. Dieses ist an dieser Stelle nicht gültig und führt zum Abbruch der SQL-Anweisung.

Die Vermeidung des hier beschriebenen Phantomproblems (siehe auch Abschnitt 4.3.2) kann optional in SYMORE aktiviert werden. Dann werden auch alle `where`-Prädikate aller Unteranfragen von `Update`- und `Insert`-Anweisungen durch die initial durch diese Anfragen ermittelten Zeilen ersetzt. Zusammen mit der aktivierten Ermittlung der Lesemenge ist dann sichergestellt, dass diese Anweisungen nur dann erneut ausgeführt werden, wenn sie dieselben Daten lesen können. Zusätzlich werden auch Funktionen durch ihren initial ermittelten Wert ersetzt.

Sollen für die Konflikterkennung Leseoperationen mit herangezogen werden, müssen auch für die hier behandelten Anweisungen, die Daten schreiben, gelesene Konfliktelemente bestimmt werden. Dieses geschieht, wie oben exemplarisch für die `Select`-Anweisung erläutert, durch Ausführung der Methode `evaluateReadSet` auf den einzelnen `<valExpr>` repräsentierenden Objekten.

Für jede Anweisung, deren Lese- oder Schreibmenge ermittelt wird, darf sich der Datenbankzustand während der Ausführung seiner `execute`-Methode nicht ändern. Die `execute`-Methode ermittelt zunächst die gelesenen oder geschriebenen Konfliktelemente, bevor die eigentliche Anweisung auf der Hintergrunddatenbank ausgeführt wird. Ändert sich der Datenbankzustand zwischen der Ausführung dieser beiden Schritte, besteht die Gefahr, dass im ersten Schritt andere Konfliktelemente bestimmt wurden als im zweiten tatsächlich gelesen oder geschrieben wurden. Indem die Datenbank im Isolationslevel „serializable“ betrieben wird, wird dieses verhindert.

5.2.3 Lokales Commit

Eine Transaktion wird abgeschlossen, wenn lokal ein `Commit` oder `Rollback` ausgeführt wird. Im ersten Fall entspricht dieses einem Pre-Commit aus Sicht des

Replikationssystems. Mittels der vorher ermittelten Konfliktelemente und der Map `activatedNodes` wird bestimmt, welche Transaktionen die Vorgänger der aktuellen Transaktion sind. Anschließend wird, wie in Abschnitt 4.3.3 beschrieben, die aktuelle Transaktion in den Vorgängergraphen eingefügt und die Verweise in der Map `activatedNodes` für alle betroffenen Konfliktelemente auf diese gesetzt. Im zweiten Fall wird die Transaktion abgebrochen. Auf der Hintergrunddatenbank wird ebenfalls ein Rollback ausgeführt, um alle schon ausgeführten SQL-Anweisungen dieser Transaktion rückgängig zu machen.

Eine Transaktion wird durch ein Exemplar der Klasse `Transaction` repräsentiert. Diese erbt, wie alle anderen Ereignisse auch, von der Klasse `SerializableEvent`. Diese enthält einen eindeutigen Zeitstempel, wie in Abschnitt 4.2.2 beschrieben. Dieser wird während der Ausführung des Pre-Commits aus dem aktuellen Zeitpunkt, der lokalen RM-Id und einer Sequenznummer bestimmt und im Transaktionsobjekt gespeichert. Die Klasse `SerializableEvent` implementiert das Interface `Comparable` so, dass alle diese Objekte einfach anhand ihrer Zeitstempel verglichen und eindeutig geordnet werden können.

Zum Schluss der Behandlung des Pre-Commits wird das `Transaction`-Objekt an die Synchronisationskomponente zur Verteilung übergeben. Es werden die Schreib- und Lesemenge dieser Transaktion übertragen sowie alle SQL-Anweisungen, die im Rahmen dieser Transaktionsausführung Datenelemente manipuliert haben. Um die zu übertragenen Daten zu minimieren, wird darauf verzichtet, die erzeugten Objektbäume für jede Anweisung zu senden. Stattdessen wird jede Anweisung als String übertragen, aus dem der empfangende RM mittels seines Parsers die Objektstrukturen erneut erzeugt.

Nebenläufigkeit

Bei dem lokalen Commit müssen Effekte beachtet werden, die sich durch nebenläufige Aktionen anderer lokaler Transaktionen, aber auch durch Empfang und Behandlung empfangener Updatetransaktionen ergeben. So dürfen die Operationen, die während des lokalen Commits einer Transaktion ausgeführt werden, nicht überlappend mit dem lokalen Commit einer anderen Transaktion stattfinden. Die Bestimmung der Lese- und Schreibmenge findet in zwei getrennten Operationen statt, die zusammen mit der Bestimmung des Transaktionszeitstempels ununterbrechbar ausgeführt werden müssen. Könnte sich die Ausführung des lokalen Commits zweier Transaktionen überlappen, kann es passieren, dass eine Transaktion hinter einer anderen in dem Vorgängergraphen angeordnet wird, aber einen kleineren Zeitstempel als diese erhält.

Parallele lokale Änderungen an den gleichen Datenelementen sind durch den Synchronisationsmechanismus der verwendeten lokalen Datenbank ausgeschlossen. Dieser sorgt im Normalfall durch Sperren auf Datenzeilen dafür, dass lokale Transaktionen aus Datenbanksicht serialisierbar ausgeführt werden. Es kann darauf verzichtet werden, zwischen nichtsequentiell ausgeführten lokalen Transaktionen eine weitere Konflikterkennung, wie sie zwischen Transaktionen unterschiedlicher RM stattfindet, durchzuführen.

Da die Hintergrunddatenbank im Isolationslevel „serializable“ betrieben wird, ist gewährleistet, dass lokale Transaktionen serialisierbar ausgeführt werden. Im Konfliktgraphen werden diese hintereinander angeordnet. Zwei lokale Transaktionen können durchaus überlappend zwei unterschiedliche Zeilen einer Tabelle manipulieren, obwohl Konfliktgranularität „Tabelle“ für diese definiert ist. Bei kausal parallelen Transaktionen auf unterschiedlichen RM wird dieses als Konflikt erkannt, bei parallelen lokalen Transaktionen auf einem RM nicht. Da das lokale Pre-Commit ununterbrechbar ausgeführt wird, findet dieses bei einer dieser parallelen lokalen Transaktionen vor den anderen statt. Diese ändert die Felder der Map `activatedNodes` für Konfliktelemente, die sie geschrieben hat, mit einem Verweis auf ihren Transaktionszeitstempel, also in dem Beispiel auch das Konfliktelement für diese Tabelle. Die zweite Transaktion, deren Ausführung sich mit der ersten teilweise überlappt hat, führt ihr Pre-Commit nach dieser aus. Sie erhält bei der Ermittlung der Vorgängertransaktion für das Konfliktelement „Tabelle“ anhand der Map `activatedNodes` den Verweis auf die erste Transaktion, wird also hinter diese in den Vorgängergraphen eingefügt. Zwei Transaktionen können so nicht an einem Konfliktelement in der einen Reihenfolge, an einem anderen aber in umgekehrter Reihenfolge angeordnet werden.

5.3 Empfang und Ausführung von Ereignissen

Zusätzlich zu der Ausführung von lokalen Transaktionen werden Updatetransaktionen von anderen RM empfangen. Außer diesen Transaktionen verteilen RM Informationen über weitere lokale Ereignisse wie Forced Commits oder Dummy-Nachrichten. Diese Ereignisse werden in Form eines entsprechenden `Transaction`, `ForcedCommit` oder `DummyMessage`-Objekts, das von `SerializableEvent` erbt, versendet. Diese Hierarchie ist in dem Klassendiagramm in Abbildung 5.4 dargestellt. Alle diese Klassen erben den eindeutigen Zeitstempel von der Klasse `Event` und überschreiben die abstrakten Methoden `store` und `handle`. Hier wurde das Strategie-Entwurfsmuster angewendet, indem jedes Ereignis selbst definiert, wie es zwischengespeichert (store) und ausgeführt (handle) werden soll.

Ein empfangenes Ereignis wird von der Verteilungskomponente an den `EventManager` des Event-Paketes übergeben, der es zunächst zwischenspeichert. Dazu ruft er die Methode `store` auf diesem SerializableEvent-Objekt auf, so dass es sich selbst in die richtige Warteschlange einreihen kann (`inTransactions` bei Transaktionen und Dummy-Nachrichten, `inCommits` bei ForcedCommits). Außerdem wird der `SummaryVector`, der den Zeitstempel des letzten lückenlos empfangenen Ereignisses eines RM speichert, aktualisiert.

Die Bearbeitung der empfangenen Ereignisse übernimmt ein eigener Thread. Dieser läuft parallel zu den Nutzerthreads, die lokale Transaktionen ausführen. Dieser Ereignisbehandlungsthread blockiert, solange sowohl die `inTransactions`- als auch die `inCommits`-Warteschlangen leer sind. Wird in eine der beiden Warteschlangen ein Er-

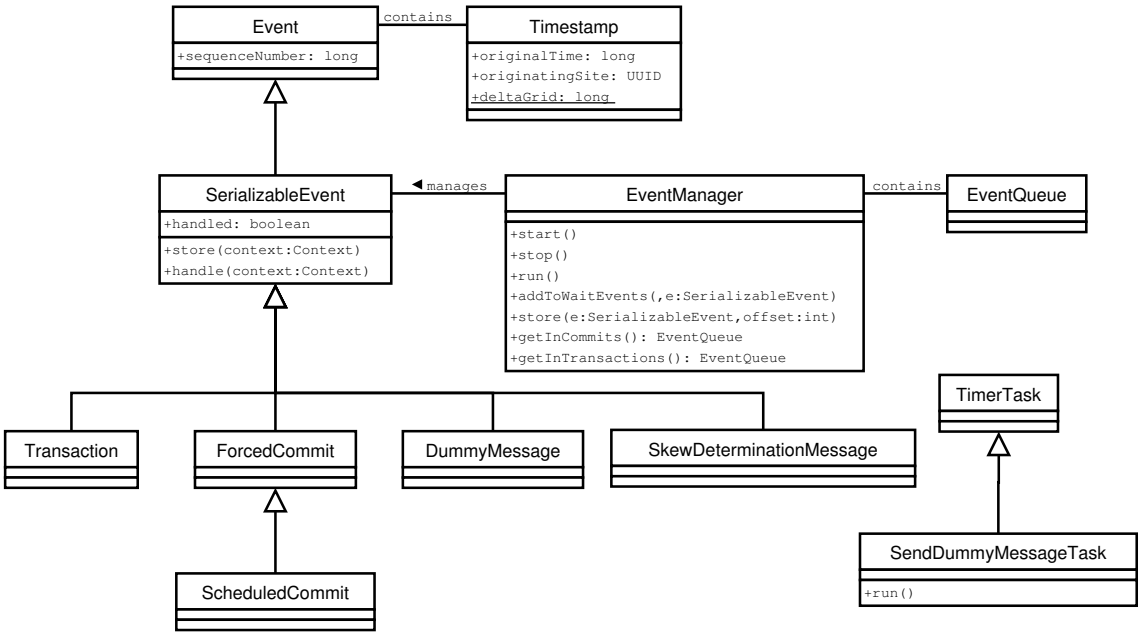

Abbildung 5.4: Klassendiagramm des Pakets Event

eignis eingefügt, wird der Ereignisbehandlungsthread aufgeweckt. Zunächst kopiert er alle neuen Transaktionen und Dummy-Nachrichten aus der `inTransactions`-Warteschlange in die lokale Transaktionswarteschlange und fügt sie dort sortiert ein. Anschließend werden die Ereignisse in dieser Warteschlange der Reihe nach durch Aufruf ihrer `handle`-Methode ausgeführt und im Erfolgsfall aus ihr gelöscht. Die Ausführung einer fremden Transaktion umfasst das Prüfen, ob alle Vorgängerknoten ihrer Lese- und Schreibmengen im Vorgängergraphen vorhanden sind und das eigentliche Einfügen in diesen im Erfolgsfall.

Im nächsten Schritt werden die Commitereignisse bearbeitet und die neue Datenbanksicht berechnet. Zunächst werden alle Commitereignisse aus der `inCommits`-Warteschlange in die nach Zeitstempeln sortierte Commitwarteschlange eingefügt. Commitereignisse darin können empfangene `ForcedCommit`-Ereignisse oder lokal erzeugte und in dieser Warteschlange gespeicherte `ScheduledCommit`-Ereignisse sein. Alle Commitereignisse in dieser Warteschlange werden nun der Reihe nach, ebenfalls durch Aufruf ihrer `handle`-Methode, ausgeführt. Wie in Abschnitt 4.5 erläutert wurde, wird dabei zunächst geprüft, ob die Daten aller Transaktionen aller RM bis zu dem aktuellen Commitzeitpunkt lokal vorliegen. Ist diese Bedingung erfüllt, wird der Konfliktlösungsalgorithmus über die Transaktionen mit Zeitstempeln in dem Intervall vom *letzten* Commitzeitpunkt bis zum *aktuellen* Commitzeitpunkt ausgeführt und das Commitereignis aus der Warteschlange entfernt. Alle nun aktivierten Transaktionen bleiben definitiv aktiviert und der *aktuelle* Commitzeitpunkt wird zum *letzten* Commitzeitpunkt.

Benutzer können über das `DataSource`-Objekt Beobachter für die Benachrichtigung

über Commitereignisse registrieren (Beobachter-Entwurfsmuster). Alle registrierten Beobachter werden nun über lokal initiierte Transaktionen, die jetzt permanent abgebrochen oder festgeschrieben worden sind, informiert.

Nachdem alle Commitereignisse auf diese Weise behandelt worden sind, wird die neue vorläufige Datenbanksicht bestimmt. Dazu wird der Konfliktlösungsalgorithmus für alle übrigen noch nicht festgeschriebenen oder abgebrochenen Transaktionen, die sich in der `NodesByTimestamp`-Liste mit Zeitstempeln größer als dem letzten Commitzeitpunkt befinden, ausgeführt. Der neue Datenbankzustand wird nun für alle Transaktionen sichtbar gemacht. Wie dieses geschieht, wird in Abschnitt 5.3.1 beschrieben. Alle Beobachter, die sich für die Benachrichtigung über geänderte Datenbankzeilen registriert haben, werden jetzt informiert. Anschließend werden nicht mehr benötigte Transaktionen durch Ausführung des Trimming-Algorithmus (siehe Abschnitt 4.5.6) aus dem Vorgängergraphen entfernt.

Nebenläufigkeit

Die Ausführung des Konfliktlösungsalgorithmus im Zuge der Commitbehandlung und der Neuberechnung der Datenbanksicht darf nicht parallel zu der Ausführung lokaler Transaktionen erfolgen. Der Konfliktlösungsalgorithmus ändert die Menge der aktivierten Transaktionen und die Map `activatedNodes`. Geschieht dieses parallel zu der Ausführung von Nutzertransaktionen, besteht die Möglichkeit, dass diese bei ihrem lokalen Commit völlig andere Vorgängertransaktionen bestimmen als sie tatsächlich hatten. Deshalb erfolgen die Aktionen des Ereignisbehandlungsthreads unter vollständigem Ausschluss von Nutzertransaktionen, wie in Abbildung 5.5 dargestellt ist. Der Ereignisbehandlungsthread fordert vor Ausführung dieser Aktionen eine exklusive Sperre an. Diese wird gewährt, sobald keine lokale Nutzertransaktion, die für ihre Ausführung eine geteilte Sperre halten muss, mehr aktiv ist.

Die Sperrverwaltung ist so implementiert, dass nicht eine Art von Sperre systematisch der anderen vorgezogen wird und somit z.B. lokale Nutzertransaktionen den Ereignisbehandlungsthread aushungern könnten. Jeder neuen Sperranfrage wird ein Prioritätswert in aufsteigender Reihenfolge zugeordnet. Anschließend wird sie in eine Prioritätswarteschlange eingereiht. Sperranfragen werden in der Reihenfolge ihrer Prioritätswerte abgearbeitet. Einer neuen Anfrage nach Gewährung einer geteilten Sperre wird nur stattgegeben, wenn sich keine exklusive Sperre mit niedrigerem Prioritätswert in der Warteschlange befindet. Indem auf den Prioritätswert eines Sperrtyps zusätzlich ein konstanter Wert addiert wird, kann der andere Typ gegenüber diesem bevorzugt behandelt werden. Eine Nutzertransaktion behindert in dem hier vorgestellten System natürlich das Ausführen von Commits und empfangenen Transaktionen solange, bis sie ihr lokales Pre-Commit ausführt.

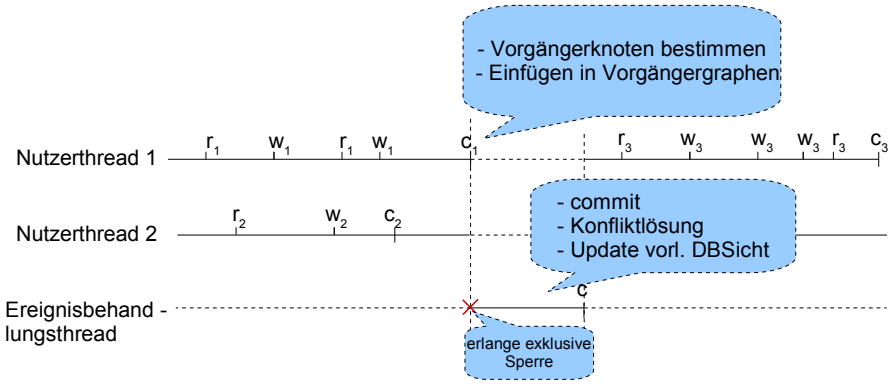

Abbildung 5.5: Nebenläufigkeit in der lokalen Datenbank

5.3.1 Aktualisierung der lokalen Datenbanksicht

Die Effekte der durch die Ausführung des Konfliktlösungsalgorithmus ermittelten aktivierten Transaktionen müssen für weitere Transaktionen sichtbar gemacht werden. Lokale SQL-Anweisungen sollen immer auf die aktuellen Werte der lokalen Datenelemente zugreifen.

Das in SYMORE implementierte Verfahren legt bei Erzeugung einer Nutzer-Datenbanktabelle eine Kopie dieser in einem separaten Schema an. Diese Tabelle enthält die Werte, die sich aus der Ausführung der definitiv festgeschriebenen Transaktionen ergeben. Dieses ist der stabile Zustand. Wird eine Transaktion definitiv festgeschrieben, so wird sie auf diesen Tabellen des stabilen Zustandes ausgeführt. Soll die vorläufige Datenbanksicht geändert werden, so wird der Inhalt dieser Tabellen komplett auf die Originaltabellen, die den vorläufigen Zustand enthalten, kopiert. Anschließend werden alle vorläufigen, aktivierten Transaktionen auf diesen Tabellen in Zeitstempelreihenfolge erneut ausgeführt. Dazu wird die Methode `reexecute` auf allen `Statements` dieser Transaktion aufgerufen.

Das hier beschriebene Verfahren ist recht einfach, verbraucht aber durch die Kopien der Tabellen für den stabilen Zustand etwa den doppelten Speicherplatz als wenn die Datenbank nicht repliziert wäre. Dadurch, dass die vorläufige Datenbanksicht immer materialisiert ist, können `Select`-Anfragen schnell beantwortet werden, das Erzeugen des neuen Datenbankzustandes nach Empfang fremder Transaktionen ist aber mit einigem Berechnungsaufwand verbunden. Dieser hängt davon ab, wieviele Transaktionen sich im Pre-Commit-Zustand befinden und erneut ausgeführt werden müssen, wenn sich der Datenbankzustand ändert.

Eine effizientere Implementierung könnte im Replikationssystem vermerken, welche Zeilen (Reihe/Tabelle) bei der (Neu-)Ausführung einer vorläufigen Transaktion geändert, eingefügt oder gelöscht wurden. Beim Zurücksetzen des vorläufigen Datenbankzustandes auf den stabilen Zustand müssten dann nur die geänderten und gelöschten

Zeilen kopiert und die eingefügten gelöscht werden.

Eine Implementierung könnte auch auf einer tieferen Schicht des Datenbanksystems ansetzen. Statt auf der Ebene von SQL-Anweisungen könnte sie auf der Ebene der physischen Datenverwaltung arbeiten. So könnte vermerkt werden, welche physischen Datenbankseiten durch vorläufig ausgeführte Transaktionen verändert wurden. Nur diese müssen kopiert werden, wenn die vorläufige Sicht auf die stabile zurückgesetzt wird. Ebenso könnte ein Copy-on-write-Verfahren angewendet werden, das nur Kopien der Datenbankseiten anlegt, auf denen vorläufige Transaktionen Datenelemente ändern wollen. Diese Verfahren verlangen aber einen Eingriff in das verwendete Datenbanksystem.

Würde in SYMORE nur eine Nutzertransaktion zur Zeit zugelassen, so könnte man die Nutzeroperationen und die Datenbankoperationen des Ereignisbehandlungsthreads über eine einzige, gemeinsame Transaktion der eingebetteten Datenbank ausführen. Bei der Neuberechnung der vorläufigen Sicht werden zunächst alle vorläufigen Transaktionen innerhalb dieser Transaktion erneut ausgeführt. Anschließend können alle neuen lokalen Nutzertransaktionen in der gleichen Hintergrundtransaktion ausgeführt werden, ohne dass auf dieser ein Commit ausgeführt wird. Muss eine veränderte vorläufige Sicht berechnet werden, wird die Hintergrundtransaktion zurückgesetzt, wodurch ihre Effekte auf der Datenbank rückgängig gemacht werden. Die Datenbank enthält jetzt wieder den gleichen Zustand wie zu Beginn der letzten Berechnung der vorläufigen Sicht. Eine zusätzliche Kopie jeder Tabelle ist für die stabile Sicht hier nicht nötig. Soll eine Transaktion definitiv festgeschrieben werden, wird sie einfach nach dem `Rollback` der aktuellen Hintergrundtransaktion und vor dem erneuten Ausführen der vorläufigen Transaktionen auf der Hintergrunddatenbank ausgeführt. Diese Ausführung wird mit einem Commit festgeschrieben.

5.3.2 Benachrichtigung der Anwendung über Änderungen an der vorläufigen Sicht

Die Anwendung, in die SYMORE eingebettet ist, kann über das `DataSource`-Objekt des Replikationssystems Beobachter vom Typ `ChangeListener` in dem Replikationssystem registrieren. Diese werden benachrichtigt, wenn sich eine spezifizierte Tabelle der vorläufigen Datenbanksicht auf Grund neu empfangener Transaktionen oder ausgeführter Commits nach Ausführung des Konfliktlösungsalgorithmus geändert hat. Je nach Anwendung kann solch ein Push-Mechanismus wesentlich effizienter sein als ein ansonsten nötiges periodisches Abfragen (pull) des Datenbankzustandes.

Um feststellen zu können, ob sich der Datenbankzustand geändert hat, wird vor Ausführung der Commits und Neuberechnung der vorläufigen Datenbanksicht eine Kopie des Zustands aller vorläufigen Transaktionen angelegt. Nach der Neuberechnung der vorläufigen Sicht wird diese mit dieser Kopie verglichen. Werden Änderungen festgestellt, so werden die registrierten Beobachter benachrichtigt. Die Benachrichtigungsmethode enthält eine Referenz auf ein Objekt, über das der Beobachter abfragen kann, welche Zeilen geändert, eingefügt oder gelöscht worden sind oder für welche Zeilen das

Einfügen oder Löschen rückgängig gemacht wurde. Da das Ermitteln dieser Informationen einigen Berechnungsaufwand erfordert, geschieht es verzögert (lazy) erst dann, wenn ein Beobachter diese Methoden aufruft. Genügt einem Beobachter die Information, dass eine bestimmte Tabelle geändert worden ist, wird somit Berechnungsaufwand gespart.

Da die Rückrufmethoden aus dem Hintergrundthread des Replikationssystems heraus erfolgen, darf eine Behandlung in der Anwendung nur sehr kurz sein. Auch dürfen in dieser Methode keine Datenbankoperationen ausgeführt werden, da es auf Grund der eingesetzten Synchronisation sonst zu Blockierungen kommt.

5.4 Korrektheit und Tests

Die Implementierung von SYMORE orientiert sich stark an der in Kapitel 4 beschriebenen Konzeption. Dort wurde für zentrale Aspekte bereits theoretisch deren Korrektheit gezeigt.

Um eine möglichst hohe Fehlerfreiheit der Implementierung zu gewährleisten wurden für viele Klassen Unit-Tests entwickelt, mit denen automatisiert die Korrektheit vieler Methoden geprüft werden kann.

SYMORE wurde so entwickelt, dass mehrere Exemplare des Systems parallel und unabhängig voneinander in der gleichen Java Virtual Machine (VM) laufen können. Dazu wurde bewusst auf die Verwendung von Singletons oder statischen Methoden und Variablen verzichtet. Somit ist es einfach auch funktionale Aspekte, die die Kommunikation zwischen zwei Knoten beinhalten, zu testen. Dazu werden in einem Programm zwei Exemplare von SYMORE gestartet und in beiden Transaktionen erzeugt. Da beide Exemplare in der gleichen VM laufen, kann anschließend leicht geprüft werden, ob die richtigen Transaktionen abgebrochen oder festgeschrieben worden sind und ob beide Datenbankzustände letztendlich gleich sind. Es muss dabei ausgeschlossen werden können, dass Fehler bei diesen Tests durch die externe Datenverteilungskomponente verursacht wurden. Deshalb wurde eine einfache Verteilungskomponente geschrieben, die Daten zuverlässig mittels TCP zwischen zwei RM überträgt. Auch die richtige Behandlung von Ereignissen, die in vertauschter Reihenfolge empfangen werden, kann mittels einer speziellen Verteilungskomponente die für diese Vertauschung sorgt, zuverlässig geprüft werden.

6 Beispielanwendung: MobileWiki

Um die Funktionsweise des Replikationssystems zu demonstrieren und dessen Tauglichkeit für den Einsatz in Anwendungen unter Beweis zu stellen, wurde eine einfache Beispielanwendung entwickelt. Die Wahl fiel dabei auf eine Art verteiltes, dezentrales Wikisystem mit Namen MobileWiki. Die Anwendung ist inspiriert von Tomboy [Gra06], einem Desktop Notizbuch für Linux und Unix Systeme. Ein Nutzer dort kann Notizen anlegen und diese ähnlich wie in Wiki-Systemen über Hyperlinks miteinander verbinden.

In dem vorliegenden System wird auf einer Menge mobiler Geräte je eine MobileWiki Anwendung ausgeführt. Diese verwendet das in dieser Arbeit vorgestellte verteilte Replikationssystem Symore, um ihre Daten zu verwalten. Der komplette Datenbestand des verteilten Systems liegt auf jedem Gerät als Kopie vor. Jeder Nutzer kann lokal und autonom neue Artikel anlegen oder löschen und bestehende verändern. Diese Aktionen werden anschließend über das Replikationssystem an die anderen Replikationsmanager übertragen und somit den anderen Teilnehmern bekannt gemacht.

Die Anwendung und das Datenmodell sind einfach, jedoch gut geeignet, um zu demonstrieren, wie Symore die Entwicklung verteilter Anwendungen ermöglicht und erleichtert. Dabei wird auch deutlich, wo die Grenzen eines replizierten Datenbanksystems liegen. Das MobileWiki besteht aus einer Menge von Artikeln, die jeweils in Abschnitte untergliedert sind. Es enthält bei seinem ersten Aufruf einen leeren Artikel (Start-Artikel). Dieser kann von beliebigen Teilnehmern mit Inhalten gefüllt werden. Weitere Artikel können wie unten beschrieben angelegt und aufgerufen werden.

In einem Fenster wird zu einem Zeitpunkt immer nur ein Artikel mit seinen Abschnitten angezeigt, wie in Abbildung 6.1 dargestellt ist. Zwischen Artikeln kann über Hyperlinks und den Zurück-Knopf navigiert werden. Der Text der Abschnitte kann direkt geändert werden. Anders als in vielen HTML-Wikis muss dazu keine spezielle Editieransicht aufgerufen werden.

Artikel und Abschnitte sind durch die Bezeichner `ArticleId` bzw. `SectionId` global eindeutig bestimmt. Das konzeptuelle Datenbankschema ist in Abbildung 6.2 dargestellt.

6.1 Anwendungsfälle

Im Folgenden wird eine Auflistung der für dieses System relevanten Anwendungsfälle (Use Cases) gegeben. Probleme, die bei der Implementierung einzelner Anwendungsfälle beachtet werden müssen, werden erläutert.

Abbildung 6.1: Hauptfenster der Anwendung MOBILEWIKI

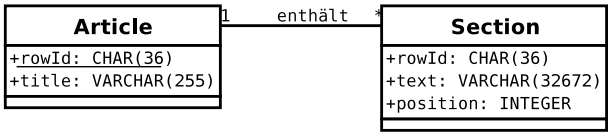

Abbildung 6.2: Konzeptuelles Datenbankschema des MOBILEWIKI

Artikel anlegen: Ein neuer Artikel wird erstellt, indem aus einem bestehenden Artikel ein Wort markiert und als *Verweis* definiert wird. Der Text des Verweises entspricht dem Titel des neu erstellten Artikels. Ein neuer Artikel enthält einen initial leeren Abschnitt.

Artikel löschen: Ein aktuell angezeigter Artikel kann gelöscht werden. Dabei werden auch Verweise auf diesen gelöscht. Nach dem Löschen wird der zuletzt vor dem gelöschten Artikel dargestellte Artikel angezeigt oder, falls dieser nicht (mehr) vorhanden ist, der Start-Artikel. Der Start-Artikel kann nicht gelöscht werden. So wird ausgeschlossen, dass kein neuer Artikel mehr angelegt werden kann.

Artikel über Verweis aufrufen: Wird auf einen vorhandenen Verweis geklickt, wird der Artikel, dessen Titel dem Text des Verweises entspricht, samt all seinen Abschnitten geladen und angezeigt. Der aktuelle Artikel wird in der Historie zu dem Vorgänger des nun angezeigten Artikels. Dieses ermöglicht über den Zurück-Knopf zu dem vorherigen Artikel zurückzukehren. Alle Wörter in den Abschnitten dieses Artikels, die einer Überschrift eines anderen Artikels in der lokalen Datenbank entsprechen, werden als Verweis zu diesen kenntlich gemacht.

Titel ändern: Der Titel eines Artikels kann geändert werden. Diese Änderung wird erst wirksam, wenn der Artikel gespeichert wird.

Abschnitt erstellen: In einen angezeigten Artikel kann ein neuer Abschnitt eingefügt werden, indem er an einen bereits vorhandenen angehängt wird. Ein neuer Abschnitt wird als leerer Abschnitt sofort in der Datenbank gespeichert, falls der zugehörige Artikel noch in der lokalen Datenbank vorhanden ist. Auf Grund kausal paralleler Aktivitäten anderer Teilnehmer kann der momentan angezeigte Artikel bereits aus der lokalen Datenbank gelöscht worden sein. Ist der zugehörige Artikel nicht mehr in der Datenbank vorhanden, wird der Nutzer darüber informiert. Über ein Dialogfenster werden ihm die Wahlmöglichkeiten angeboten, den aktuellen Artikel erneut abzuspeichern oder zu verwerfen. Der neue Abschnitt enthält eine von SYMORE generierte, global eindeutige Identifikationsnummer (`sectionId`) und einen Verweis auf den zugehörigen Artikel. Auf die Problematik der Reihenfolge von Abschnitten wird in Abschnitt 6.2 eingegangen.

Abschnitt löschen: Wird ein Abschnitt aus einem angezeigten Artikel gelöscht, so geschieht dieses ebenfalls direkt in der Datenbank. Auch hier ist es möglich, dass der zu löschende Abschnitt in der aktuellen lokalen Datenbank nicht mehr vorhanden ist.

Abschnitt speichern: Textuelle Änderungen an einem Abschnitt werden erst auf Aufforderung des Nutzers gespeichert und daraufhin an andere Geräte der Replikationsgruppe verteilt. Da derselbe Abschnitt von einem anderen Nutzer auf einem anderen Gerät kausal parallel geändert worden sein kann, kann ein Konflikt auftreten. Ein Konflikt kann dabei auf zwei Ebenen vorliegen. Zum einen kann die fremde Änderung noch nicht empfangen worden sein, als die lokale Änderung gespeichert wurde. In diesem Fall wird das Replikationssystem einen Konflikt erkennen. Dazu ist für das Textfeld jedes Abschnitts die Konfliktgranularität "cell"definiert worden.

Zum anderen kann die Änderung durch das fremde Gerät empfangen worden sein, nachdem der angezeigte Artikel aus der lokalen Datenbank geladen wurde, jedoch bevor die lokale Anwendung aufgefordert wurde, diesen zu speichern. Ein einmal geladener Artikel wird aus Gründen der Benutzbarkeit nicht automatisch aktualisiert, wenn sich seine Daten in der lokalen Datenbank durch Empfang von fremden Transaktionsdaten ändern. Der Konflikt, der sich durch die parallele Änderung der lokalen Datenbank ergeben hat, muss von der Anwendung selbst erkannt werden. Würde der aktuell angezeigte Artikel direkt in die Datenbank geschrieben, so würde SYMORE keinen Konflikt erkennen, da diese Operation als kausal nach der vorher empfangenen entfernten Änderung angesehen würde. Die Anwendung erkennt allerdings, dass die Daten zu dem zu speichernden Abschnitt in der lokalen Datenbank von den zum Anzeigen des Abschnitts geladenen Daten differieren und teilt dieses dem Nutzer über ein Dialogfenster mit. Ihm wird sowohl seine Version als auch die, die nun in der lokalen Datenbank aktuell ist, angezeigt. Er hat die Wahl eine von beiden zu speichern oder den Vorgang abzubrechen.

Reihenfolge von Abschnitten ändern: Die Abschnitte eines Artikels sind geordnet. Deren Reihenfolge kann verändert werden, indem immer zwei Abschnitte miteinander vertauscht werden. Diese Änderung wird sofort auf der Datenbank ausgeführt. Ist der Artikel dabei nicht mehr in der Datenbank vorhanden, gilt das in Anwendungsfall 6.1 beschriebene Verfahren. Wie die Änderung der Reihenfolge von Artikeln implementiert werden kann, wird in Abschnitt 6.2 vorgestellt.

Artikel speichern: Einen Artikel zu speichern bedeutet, dessen Titel sowie alle geänderten Abschnitte zu speichern. Wie bei dem Speichern eines einzelnen Abschnitts können auch in diesem Fall Konflikte entstehen. Je nach Konfliktebene werden diese von dem Replikationssystem festgestellt oder müssen von der Anwendung selbst erkannt werden. Für das Feld `Title` der Tabelle `article` wird deshalb ebenfalls die Konfliktgranularität „cell“ definiert.

Auch hier muss von der Anwendung erkannt werden, wenn die lokale Datenbank geändert wurde, während ein Artikel angezeigt wird und bevor dieser gespeichert wurde. Der Anwender wird für jeden Abschnitt, der nach dessen letztem Speichern sowohl in der Anwendung durch den Anwender als auch in der Datenbank durch fremde Transaktionen geändert worden ist, gefragt, welche Version gespeichert werden soll. Ebenso wird er benachrichtigt, falls der Titel des Artikels zwischenzeitlich geändert wurde.

Status von Transaktionen verfolgen: Alle Operationen auf der Datenbank finden als Transaktionen statt. Ob diese mit anderen, fremden Transaktionen in Konflikt stehen, wird erst verzögert erkannt. Der Anwender hat die Möglichkeit sich über den Status der von ihm initiierten Transaktion zu informieren und angezeigt zu bekommen, ob eine Transaktion definitiv festgeschrieben oder abgebrochen wurde oder ob ihr Ergebnis noch unbestimmt ist.

Artikel suchen: Ein Anwender kann nach Artikeln suchen, die ein Suchwort in ihrem Titel oder in dem Text eines ihrer Abschnitte enthalten. Die so gefundenen Artikel können anschließend ausgewählt und angezeigt werden.

6.2 Reihenfolge von Abschnitten

Wie oben beschrieben soll es möglich sein, die Reihenfolge, in der Abschnitte eines Artikels angezeigt werden, zu ändern. Zu diesem Zweck ist jedem Abschnitt ein Positionsfeld `position` zugeordnet. Dessen Wert bestimmt die Ordnung der Abschnitte.

Wird ein Abschnitt eingefügt, muss dessen Positionsfeld initialisiert werden. Der Wert dieses Feldes wird aus der Hälfte der Summe der Positionswerte des vorherigen sowie des nachfolgenden Abschnitts gebildet: $neu.pos = \frac{vorg.pos+nachf.pos}{2}$. Auf diese Weise ist es nicht nötig, die Positionswerte aller auf den eingefügten Abschnitt folgenden Abschnitte zu erhöhen. Gibt es keinen vorherigen oder nachfolgenden Abschnitt, so wird der

Wert 0 bzw. `MAXINT` für den Positionswert des Vorgängers oder Nachfolgers gewählt. Bei dem hier beschriebenen Verfahren ist nur eine begrenzte Anzahl von Einfügeoperationen zwischen zwei Abschnitten möglich. Wird der Positionswert des ersten Abschnitts auf $\frac{MAXINT}{2} = 2^{30}$ gesetzt, so können im ungünstigsten Fall 30 Abschnitte zwischen zwei gegebenen Abschnitten eingefügt werden. Dieser Wert ergibt sich beispielsweise, wenn immer oberhalb eines Abschnittes ein weiterer Abschnitt eingefügt wird. Der erste Abschnitt erhält Positionsnummer 2^{30}, der zweite die Nummer 2^{29} $\left(\frac{2^{30}-0}{2}\right)$ usw. Dieses wird als ausreichend für das vorliegende System erachtet.

Soll die Reihenfolge zweier Abschnitte vertauscht werden, genügt es in einer Transaktion deren Positionswerte zu vertauschen. Auch hier muss beachtet werden, dass die Abschnitte, die der Nutzer angezeigt bekommt, nicht mehr den Abschnitten bzw. der Reihenfolge entsprechen müssen, wie sie in der lokalen Datenbank vorliegen. Wurde der Artikel oder Abschnitt zwischenzeitlich auf der Hintergrunddatenbank gelöscht, wird dieses dem Benutzer angezeigt.

Unabhängig von dieser Problematik funktioniert das Vertauschen der Positionswerte zweier Abschnitte nur, wenn diese Werte unterschiedlich sind. Diese Bedingung würde verletzt, wenn auf zwei Geräten ein neuer Abschnitt kausal parallel hinter einem gleichen bestehenden Abschnitt eingefügt würde. Um dieses zu verhindern, wird, zusammen mit dem Einfügen eines neuen Abschnitts, eine eigentlich unnötige Schreiboperation auf dessen Vorgängerabschnitt vorgenommen. Auf diese Weise wird ein Konflikt provoziert und das Replikationssystem sorgt dafür, dass nur eine von möglicherweise mehreren parallelen Einfügeoperationen aktiviert und letztendlich festgeschrieben wird. Somit ist gewährleistet, dass jeder Abschnitt eines Artikels eine eindeutige Positionsnummer erhält. Nachteilig ist, dass diese anwendungssemantisch unproblematischen parallelen Einfügeoperationen nun als Konflikt aufgefasst werden. Eine Anwendung muss sich selbst darum kümmern, diese Einfügeoperation erneut auszuführen, wenn die ursprüngliche Einfügetransaktion auf Grund von Konflikten definitiv abgebrochen wurde.

Der alternative Ansatz, parallele Einfügeoperationen zuzulassen und die Eindeutigkeit des Positionswertes eines Abschnitts durch einen global eindeutigen Gerätebezeichner sicherzustellen, funktioniert nicht. Seien beispielsweise kausal parallel zwei Abschnitte desselben Artikels eingefügt worden, die beide die Positionsnummer 10 erhielten. Zur Unterscheidung würden die Bezeichner 2 bzw. 3 herangezogen. Ein Problem besteht nun, wenn auf beiden Geräten zwischen diese beiden Abschnitte je kausal parallel ein weiterer einfügt würde. Beide Abschnitte erhielten die gleichen Positionswerte und die gleichen Bezeichner. Auch eine Verwendung von Zufallszahlen als zusätzlichem Positionswert liefert nicht das gewünschte Ergebnis. Bei jedem weiteren Zwischeneinfügen müsste eine neue Zufallszahl an die Positionsnummer angehängt werden, damit deren globale Eindeutigkeit gewahrt bleibt. Damit würde der Wert immer länger. Auch könnten Kollisionen durch die zufällige Wahl gleicher Werte auftreten.

Auch der Ansatz, einem neuen Abschnitt die um eins erhöhte Positionsnummer seines Vorgängers zuzuweisen und die Positionsnummern aller nachfolgenden Abschnitte zu

erhöhen, ist problematisch. Auch hier stünden parallele Einfügeoperationen eigentlich unnötigerweise miteinander in Konflikt, wenn für Positionsfelder die Konfliktgranularität „cell“ festgelegt ist. Ohne Konflikterkennung könnten inkonsistente Zustände auftreten und verschiedenen Abschnitten könnte die gleiche Positionsnummer zugewiesen werden.

6.3 Beobachter

Die MOBILEWIKI-Anwendung registriert Beobachter in SYMORE um informiert zu werden, wenn für eine Transaktion eine Commit- oder Abbruch-Entscheidung getroffen oder die aktuelle Datenbanksicht geändert wurde. Bevor eine lokale Transaktion ihr lokales Pre-Commit ausführt, wird deren Transaktionsnummer ermittelt. Anhand dieser kann der dem Nutzer angezeigte Zustand der Transaktion aktualisiert werden, wenn SYMORE den Beobachter über die Commit- oder Abbruch-Entscheidung für diese informiert.

Bei der Aktualisierung der `article`-Tabelle wird die Anwendung ebenfalls benachrichtigt. Sie ermittelt daraufhin die neu eingefügten oder gelöschten Zeilen dieser Tabelle und aktualisiert die Verweise in dem gerade angezeigten Artikel. Ist ein Artikel gelöscht worden, so wird ein in einem gerade angezeigten Abschnitt eventuell vorhandener Verweis entfernt. Ist ein Artikel hinzugefügt worden, so wird ein Wort eines gerade angezeigten Abschnittes eventuell als Verweis markiert. Ebenso wird bei geänderten Titeln von Artikeln verfahren.

6.4 Fazit

Anhand dieser Beispielanwendung wird erkenntlich, dass SYMORE dem Entwickler einer dezentralen Anwendung, deren Komponenten nicht ständig miteinander verbunden sind, aber dennoch Zugriff auf gemeinsame Daten haben sollen, viel Arbeit abnimmt. So muss sich ein Entwickler nicht darum kümmern, wie lokale Änderungen an seinem Datenbestand anderen Knoten bekannt gemacht werden und wie mögliche Konflikte erkannt werden. SYMORE kann allerdings nicht einfach wie ein zentralisiertes Datenbanksystem betrachten werden. Die Eigenschaften, die sich durch optimistische, asynchrone Replikation ergeben, können nicht vernachlässigt werden. So müssen Konfliktgranularitäten definiert werden, um zu bestimmen, auf welchen Datenbankstrukturen kausal parallele Änderungen erkannt werden sollen. Wie anhand der Problematik der Artikelreihenfolge ersichtlich wird, ist auch dieses in machen Fällen nicht ausreichend. Die Anwendung muss selbst Vorkehrungen treffen, damit logische Inkonsistenzen nicht auftreten können. Hier geschieht dieses durch eine zusätzliche Schreiboperation.

Das Problem, das sich daraus ergibt, dass sich der Zustand eines Artikels in der lokalen Datenbank verändern kann, während der Artikel angezeigt wird, ist nicht spezifisch für replizierte Datenbanken. Dasselbe Problem ergäbe sich in einer zentralisierten Datenbank, wenn der Artikel während eines Bearbeitungsvorganges nicht gesperrt würde.

Die Bearbeitung eines Artikels im MOBILEWIKI hätte alternativ als langlaufende Transaktion auf der lokalen Datenbank implementiert werden können. Diese umfasste das Laden des Artikels und alle Änderungen an diesem und würde erst beendet, wenn der Artikel gespeichert wird. Solange der Artikel bearbeitet wird, wäre dieser auf der lokalen Datenbank gesperrt. Kausal parallele Änderungen an diesem Artikel würden somit in jedem Fall von dem Datenbanksystem erkannt. In der momentanen Implementierung von SYMORE bleibt die gesamte lokale Datenbank für die Ausführung empfangener Updatetransaktionen gesperrt, solange eine lokale Transaktion stattfindet. Je länger die Transaktion dauert, desto länger ist die lokale Datenbank blockiert. Somit könnten also auch keine fremden Transaktionen ausgeführt werden, die sich nicht auf den momentan bearbeiteten Artikel auswirken. Aus diesem Grund verwendet die hier präsentierte Implementierung des MOBILEWIKI keine langlaufende Transaktion, deren Dauer letztendlich von dem Nutzer abhängt, sondern zwei kurze Transaktionen zum Laden und Speichern eines Artikels mit den beschriebenen Konsequenzen.

7 Skalierbarkeit

Ein wesentliches Kriterium bei der Bewertung eines Systems ist die Skalierbarkeit. Wie verhält sich ein System, wenn bestimmte Einflussgrößen wie z.B. die Anzahl der Teilnehmer erhöht werden? Dieses soll in diesem Abschnitt für SYMORE untersucht werden.

Der wichtigste Faktor in dem vorliegenden optimistischen Replikationssystem ist dabei die Konflikthäufigkeit. Ein Konflikt zwischen zwei Transaktionen führt in SYMORE zwangsläufig zu einem Abbruch einer dieser Transaktionen, also im Normalfall zu einer Zurückweisung einer vom Anwender initiierten Änderungsoperation. Auch wenn auf diese Weise die Datenbankkonsistenz gewahrt bleibt, beeinträchtigen zu viele Transaktionsabbrüche die Benutzbarkeit einer auf diesem Replikationssystem aufsetzenden Anwendung. Werden fehlgeschlagene Transaktionen automatisch von der Anwendung wiederholt, so kann dieses das Problem verschärfen. Noch mehr Transaktionen finden statt und noch mehr Konflikte können auftreten.

SYMORE ist nicht für Szenarien geeignet, in denen viele Knoten sehr häufig die gleichen Datenelemente manipulieren. Sind die große Mehrheit der Operationen dagegen Leseoperationen oder ist durch die Anwendung eine Partitionierung der Schreibzugriffe unterschiedlicher Knoten auf verschiedene Datenelemente gegeben, ist ein erfolgreicher Einsatz von SYMORE zur Replikation und Synchronisation möglich. Im Folgenden soll analysiert werden, wie sich die Konfliktwahrscheinlichkeit bei unterschiedlichen Werten der verschiedenen Systemparameter in SYMORE verhält.

Ein weiterer die Skalierbarkeit beeinflussender Faktor ist die Anzahl der Nachrichten, die benötigt werden, um die verschiedenen lokalen Datenbanken möglichst synchron zu halten. Jede lokale Transaktion wird als eine Nachricht an eine Verteilungskomponente übergeben, die dafür sorgt, diese an alle anderen Knoten der Replikationsgruppe zu verteilen. Wieviele Nachrichten dafür benötigt werden, hängt stark von dem verwendeten Datenverteilungsverfahren ab und wird hier nicht weiter betrachtet.

7.1 Systemparameter

Um eine genauere Aussage über die die Konflikthäufigkeit beeinflussenden Faktoren treffen zu können, wird zunächst analysiert, welche Parameter in SYMORE existieren und wie diese zusammenhängen. Abbildung 7.1 stellt deren Beziehungen untereinander dar. Jeder dieser Parameter wird nun genauer beschrieben:

Größe der Replikationsgruppe: Dieser Parameter bestimmt, wieviele Knoten bzw. Replikationsmanager an der Replikationsgruppe aktiv teilnehmen. Das Peer-to-Peer-

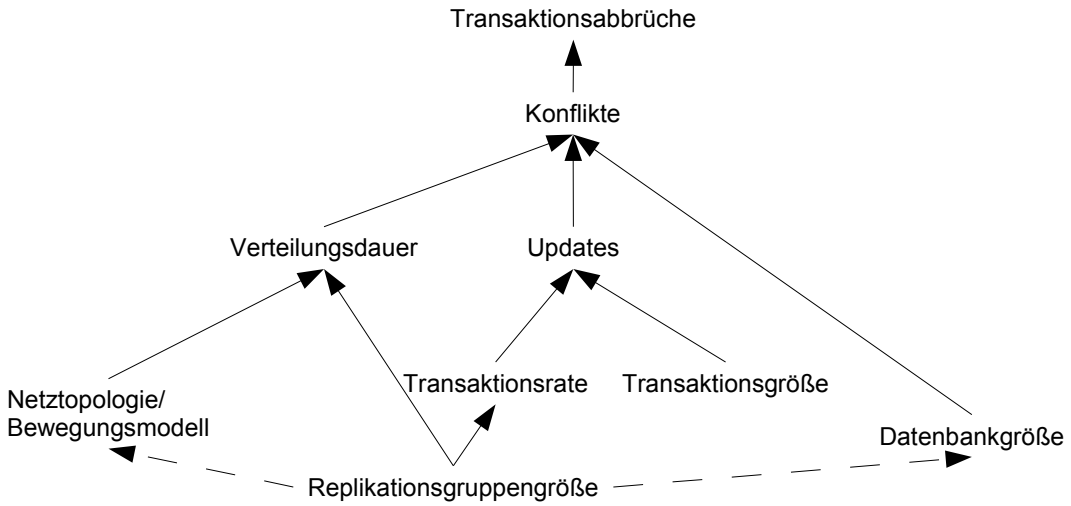

Abbildung 7.1: Beziehungen zwischen Parametern, die die Konflikthäufigkeit beeinflussen.

Replikationssystem Symore soll Anwendungen ermöglichen, die das mobile, kooperative Arbeiten einer Gruppe von Personen unterstützen. Es macht im Normalfall keinen Sinn, mit Hunderten von Teilnehmern mobil an einer gemeinsamen Aufgabe zu arbeiten. Die typische Größe einer Replikationsgruppe der hier anvisierten Szenarien beträgt etwa 10 bis 30 Teilnehmer.

Datenbankgröße: Dieser Parameter gibt die Anzahl der logischen Datenelemente an, die auf jeden Knoten repliziert werden. Er kann durch die Größe der Replikationsgruppe beeinflusst sein, etwa wenn für jeden Knoten spezifische Daten repliziert werden müssen. Im Normalfall ist die Datenbankgröße aber ein unabhängiger Parameter.

Netztopologie/Bewegungsmodell: Je nach Szenario ergeben sich unterschiedliche Netztopologien und Bewegungsmodelle. Betrachtet man Szenarien, in denen Knoten sich in Fahrzeugen relativ schnell fortbewegen, wird man sehr dynamische Änderungen der Topologie erhalten. Betrachtet man dagegen Fußgänger, die sich in einem eng umgrenzten Raum bewegen, so wird man ein anderes Bewegungsmodell und eine sich weniger dynamisch ändernde Topologie vorfinden. Die Dichte eines Netzes hängt unter anderem von der Anzahl der Knoten ab. Je mehr Knoten vorhanden sind, desto dichter ist das Netz.

Verteilungsdauer: Dieser Parameter gibt an, wie lange es im Durchschnitt dauert, bis eine Nachricht von allen Knoten der Replikationsgruppe empfangen wurde. Dieses wird zum einen durch die Größe der Replikationsgruppe beeinflusst. Je mehr Knoten eine Replikationsgruppe enthält, desto mehr Kopien müssen bei Änderungen einer Kopie aktualisiert werden und desto länger dauert diese Aktualisierung.

Zum anderen beeinflussen die Netztopologie und das zugrunde liegende Bewegungsmodell diesen Parameter. Im günstigsten Fall befinden sich alle Knoten im Broadcastradius zueinander. Dann können Daten zwischen diesen mit nur einer einzigen Nachricht sehr schnell verteilt werden. Im ungünstigsten verbundenen Fall befinden sich alle Knoten in einer Reihe und jeder kann nur je einen linken und rechten Nachbarknoten erreichen. Dabei werden $\#Knoten - 1$ Nachrichten benötigt und die Verteilung einer Nachricht dauert entsprechend länger. In MANETs sind normalerweise nicht alle Knoten permanent miteinander verbunden. Hier hängt die Dauer der Datenverteilung von der Bewegung der Knoten und der sich daraus ergebenden Häufigkeit ab, mit der zwei Knoten in Kommunikationsreichweite zueinander gelangen.

Transaktionsrate: Dieser Parameter gibt an, wie hoch die mittlere Frequenz ist, mit der Transaktionen initiiert werden. Die Transaktionsrate hängt von der Art der verwendeten Anwendung bzw. dem Anwendungsfall, in dem das Replikationssystem eingesetzt wird, als auch von der Knotenanzahl ab.

Transaktionsgröße: Dieser Parameter ist durch die Anwendung bestimmt. Er gibt an, aus wievielen (Schreib-)Operationen eine Transaktion im Mittel besteht.

Updates: Die Anzahl der Updateoperationen ergibt sich aus der Größe der einzelnen Transaktionen sowie der Rate, mit der Transaktionen ausgeführt werden.

Konflikte: Ein Konflikt entsteht, wenn ein Datenelement kausal parallel von zwei auf unterschiedlichen Knoten initiierten Transaktionen manipuliert wird. Dieser Parameter wird durch die mittlere Frequenz von Transaktionsausführungen sowie der Geschwindigkeit von deren Verteilung, bzw. der Dauer der Unverbundenheit eines Knotens beeinflusst. Je schneller Transaktionsdaten an alle anderen Teilnehmer der Replikationsgruppe verteilt werden, desto geringer ist die Wahrscheinlichkeit, dass Konflikte auftreten können. Steigt allerdings die Rate an, mit der Transaktionen auf unterschiedlichen Knoten ausgeführt werden, so erhöht sich auch die Wahrscheinlichkeit dafür, dass kausal parallel auf gleiche Datenelemente zugegriffen wird.

Auch die Größe der Transaktionen und die Größe der Datenbank beeinflussen die Wahrscheinlichkeit, dass Konflikte auftreten, wenn angenommen wird, dass Schreiboperationen der Transaktionen gleichverteilt auf Datenelemente der Datenbank ausgeführt werden.

Transaktionsabbruch: Die Anzahl der Konflikte, die zwischen Transaktionen auftreten, bestimmt direkt die Anzahl der Transaktionen, die letztendlich abgebrochen werden müssen. Da von Transaktionen, die an einem Konfliktelement in Konflikt stehen, alle bis auf eine abgebrochen werden, werden insgesamt mindestens $\frac{\#Konflikte}{2}$ Transaktionen abgebrochen.

Parameter	Beschreibung
E(U)	Erwartetes Intervall der Übertragungsvorgänge
E(T)	Erwartetes Intervall, in dem ein Knoten Transaktionen ausführt
E(T')	Erwartetes Intervall zwischen zwei potentiell in Konflikt stehenden Transaktionen
e	replizierte Elemente der Datenbank
o	Anzahl der Operationen jeder Transaktion (auf disjunkte Elemente)
k	Anzahl der Teilnehmer der Replikationsgruppe

Tabelle 7.1: Parameter des Systemmodells

7.2 Analyse

Es soll nun eine Formel entwickelt werden, die die grundlegenden Parameter *Größe der Replikationsgruppe*, *Datenbankgröße*, *Verteilungsdauer*, *Transaktionsrate* und *Transaktionslänge* quantitativ in Beziehung setzt, um eine Aussage darüber zu ermöglichen, wie sie jeweils die Wahrscheinlichkeit für Konflikte beeinflussen.

Modell

Um die Wahrscheinlichkeit für einen Konflikt zu bestimmen, muss zunächst ein Systemmodell definiert werden, das die komplexe Realität vereinfacht widerspiegelt. Dieses System bestehe aus einer Menge von k Knoten. Jeder Knoten initiiere Transaktionen, wobei die Zufallsvariable T die Zeitspanne zwischen den Pre-Commits zweier aufeinanderfolgender lokaler Transaktionen beschreibe. Diese sei exponentialverteilt mit einem bekannten Erwartungswert E(T). Die Exponentialverteilung wird typischerweise bei der Modellierung zufälliger Zeitintervalle, wie z.B. Ankunftsereignissen verwendet ([Hü00, S. 35]). Lokale Transaktionen werden zu anderen Knoten übertragen, sobald Verbindungen zu diesen bestehen. Vereinfachend wird angenommen, dass sich bei solch einem Übertragungsereignis alle Knoten der Replikationsgruppe synchronisieren. Die Zeitspanne zwischen zwei Übertragungsereignissen sei durch die ebenfalls exponentialverteilte Zufallsvariable U angegeben, wobei ihr Erwartungswert E(U) als gegeben angenommen wird. Jede Transaktion bestehe aus o Operationen. Jede dieser Operation schreibe ein anderes Datenelement. Diese Datenelemente werden gleichverteilt aus der Menge aller e Datenelemente der Datenbank gewählt. Die Dauer der Transaktionsausführung sowie der Übertragungsvorgänge wird vernachlässigt. Außerdem werden ausschließlich Konflikte betrachtet, die sich durch kausal parallele Schreiboperationen ergeben. Schreib-Lese-Konflikte werden nicht berücksichtigt. In Tabelle 7.1 sind die Parameter dieses Modells noch einmal zusammengefasst.

Eine Transaktion t_A eines Knotens A steht demnach mit mindestens einer anderen Transaktion in Konflikt, wenn in der Zeitspanne zwischen dem letzten Übertragungsvor-

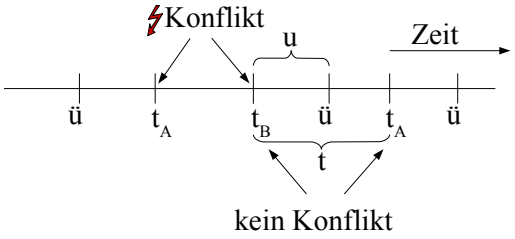

Abbildung 7.2: Auftreten von Konflikten im Modell (ü – Übertragung; t – Transaktion)

gang vor t_A und dem nächsten Übertragungsvorgang nach t_A mindestens eine weitere Transaktion t_B, initiiert von einem anderen Knoten, stattfindet und beide Transaktionen mindestens ein gleiches Datenelement manipulieren. Dieses ist in Abbildung 7.2 dargestellt.

Eine Formel, die aus diesen Eingabegrößen $E(T)$, $E(U)$, k, o und e die Wahrscheinlichkeit bestimmt, dass eine Transaktion mit mindestens einer anderen in Konflikt steht, wird nun schrittweise hergeleitet.

7.2.1 Bestimmung der Konfliktwahrscheinlichkeit

Basisformel

Vorerst wird angenommen, dass die Datenbank nur aus einem einzigen Element besteht und jede Transaktion von einem anderen Knoten initiiert wird. Jede Transaktion kann also potentiell mit jeder anderen in Konflikt stehen. Die Zufallsvariable T' gibt das Intervall zwischen zwei potentiell in Konflikt stehenden Transaktionen an. Wie sich dieses aus den Parametern $E(T)$, e, o und k bestimmt, wird in den folgenden Schritten gezeigt.

Um die Wahrscheinlichkeit für einen Konflikt $P(K)$ zu bestimmen, wird zunächst die Gegenwahrscheinlichkeit $P(\overline{K})$, dass eine Transaktion mit keiner anderen in Konflikt steht, ermittelt. $P(\overline{K})$ ist die Wahrscheinlichkeit, dass in der Zeitspanne zwischen zwei Übertragungsereignissen nur eine Transaktion stattfindet. Ausgehend von einer Transaktion t folgt dabei als nächstes Ereignis eine Übertragung und erst danach eine weitere Transaktion. Ebenso ist das letzte Ereignis vor t eine Übertragung. Die letzte Transaktion vor t fand vor dieser Übertragung statt.

Auf Grund der Gedächtnislosigkeit der Exponentialverteilung kann die Wahrscheinlichkeit, dass nach einer Transaktion eine Übertragung vor einer weiteren Transaktion stattfindet, als $P(T' > U)$ ausgedrückt werden. Es wird für jede mögliche Zeitspanne u zwischen einer Transaktion und dem folgenden Übertragungszeitpunkt die Wahrscheinlichkeit berechnet, dass die Zeitspanne t zwischen dieser Transaktion und der nächsten Transaktion größer als u ist:

$$P(T' > U) = \int_0^\infty f(u) \left(\int_u^\infty f(t)\, dt \right) du$$

Setzt man in diese Formel die Dichtefunktion der Exponentialverteilung ein, so erhält man:

$$\begin{aligned}
P(T' > U) &= \int_0^\infty \lambda_1 e^{-\lambda_1 u} \left(\int_u^\infty \lambda_2 e^{-\lambda_2 t} dt \right) du \\
&= \int_0^\infty \lambda_1 e^{-\lambda_1 u} (1 - F(u)) du \\
&= \int_0^\infty \lambda_1 e^{-\lambda_1 u} (1 - (1 - e^{-\lambda_2 u})) du \\
&= \int_0^\infty \lambda_1 e^{-\lambda_1 u} e^{-\lambda_2 u} du \\
&= \int_0^\infty \lambda_1 e^{u(-\lambda_1 - \lambda_2)} du \\
&= \left[-\frac{\lambda_1 e^{u(-\lambda_1 - \lambda_2)}}{\lambda_1 + \lambda_2} \right]_0^\infty \\
&= 0 - \left(-\frac{\lambda_1}{\lambda_1 + \lambda_2} \right) = \frac{\lambda_1}{\lambda_1 + \lambda_2}
\end{aligned}$$

$E(U) = \frac{1}{\lambda_1}$ ist der Erwartungswert für die Intervalllänge zwischen zwei Übertragungen, $E(T') = \frac{1}{\lambda_2}$ der Erwartungswert für die Intervalllänge zwischen zwei potentiell in Konflikt stehenden Transaktionen. Setzt man diese Erwartungswerte in die obige Formel ein, so erhält man $\frac{\lambda_1}{\lambda_1+\lambda_2} = \frac{E(T')}{E(T')+E(U)}$. Dieses ist die Wahrscheinlichkeit, dass nach einer Transaktion eine Übertragung stattfindet, bevor eine weitere Transaktion initiiert wird.

Um die Wahrscheinlichkeit zu bestimmen, dass eine Transaktion mit keiner anderen in Konflikt steht, darf auch vor dieser und nach der letzten vorherigen Übertragung keine weitere Transaktion initiiert worden sein. Diese Wahrscheinlichkeit ist gleich der eben berechneten. Die Wahrscheinlichkeit, dass eine Transaktion mit keiner anderen in Konflikt steht, beträgt somit:

$$P(\overline{K}) = P(T' > U)^2 = \left(\frac{E(T')}{E(T') + E(U)} \right)^2 = \left(1 - \frac{E(U)}{E(U) + E(T')} \right)^2$$

Die Wahrscheinlichkeit für einen Konflikt ist die Gegenwahrscheinlichkeit, also

$$P(K) = 1 - \left(1 - \frac{E(U)}{E(U) + E(T')} \right)^2. \tag{7.1}$$

Berücksichtigung der Knotenanzahl

Bisher wurde nicht beachtet, dass Transaktionen von unterschiedlichen Knoten initiiert werden und nur Transaktionen unterschiedlicher Knoten miteinander in Konflikt stehen können. Nun soll betrachtet werden, wie sich die Konfliktwahrscheinlichkeit verhält, wenn Transaktionen von k unterschiedlichen Knoten initiiert werden. E(T) ist der Erwartungswert des Intervalls, mit dem ein Knoten Transaktionen erzeugt. Da bei k Knoten

insgesamt k-mal so viele Transaktionen initiiert werden, beträgt der Erwartungswert für die Dauer zwischen der Ausführung zweier beliebiger Transaktionen nun $\frac{E(T)}{k}$.

Um die Konfliktwahrscheinlichkeit mit Berücksichtigung der Knotenanzahl zu bestimmen, wird eine weitere Zufallsvariable K verwendet. K gibt an, wieviele Transaktionen des gleichen Knotens hintereinander vorkommen, bis eine Transaktion eines anderen Knotens auftritt. Da T und K stochastisch unabhängig sind, errechnet sich die erwartete Zeitspanne zwischen zwei Transaktionen unterschiedlicher Knoten folgendermaßen: $E(T') = E(KT) = E(K) \cdot E(T)$.

E(K) berechnet sich wie folgt:

$$\mathrm{E}(K) = \sum_{n=1}^{\infty} n \cdot P(K = n) = \sum_{n=1}^{\infty} n \cdot \frac{k\,(k-1)}{k^{n+1}} = \frac{k}{k-1}$$

Die Wahrscheinlichkeit $P(K = n)$ ergibt sich daraus, dass die Anzahl der günstigen Fälle ermittelt wird und durch die Anzahl aller Fälle geteilt wird. Insgesamt gibt es $k^{(n+1)}$ Möglichkeiten k Transaktionstypen (Transaktionen unterschiedlicher Knoten) $(n+1)$-mal hintereinander anzuordnen. Die günstigen Fälle sind die Fälle, bei denen n-mal hintereinander eine Transaktion des gleichen Knotens vorkommt und anschließend eine andere. Für die n gleichen Transaktionen gibt es k mögliche Transaktionstypen. Das folgende, andere Element kann dann aus den verbleibenden $(k-1)$ Transaktionstypen gewählt werden.

Berücksichtigung von Datenbank- und Transaktionsgröße

Dass Transaktionen auf unterschiedlichen Datenelementen arbeiten können und dann nicht miteinander in Konflikt stehen, wurde bisher vernachlässigt. Wenn jede Transaktion ein Datenelement gleichverteilt aus der Datenbank wählen kann, verringert sich die Konfliktwahrscheinlichkeit mit zunehmender Datenbankgröße. Transaktionen müssen auch nicht nur aus einer Operation bestehen. Manipuliert eine Transaktionen mehrere unterschiedliche Datenelemente, so erhöht sich die Wahrscheinlichkeit, dass sie an mindestens einem Datenelement mit einer anderen Transaktion in Konflikt steht. Es sei o die Anzahl der Operationen jeder Transaktion, die auf unterschiedlichen Datenelementen arbeiten und e die Größe der Datenbank.

Seien t_1, t_2 zwei Transaktionen, die mindestens ein gleiches Datenelement schreiben und somit potentiell in Konflikt miteinander stehen. Die Zufallsvariable D gebe an, wieviele Transaktionen zwischen t_1 und t_2 vorkommen, die nicht potentiell mit t_1 in Konflikt stehen, also kein Datenelement manipulieren, das auch von t_1 manipuliert wird. Um nun den Erwartungswert der Anzahl der Zwischenräume zwischen t_1 und t_2 zu erhalten, muss 1 zu $E(D)$ addiert werden.

Analog zu der Berücksichtigung der Knotenanzahl kann somit die erwartete Zeitspanne zwischen zwei Transaktionen, die mindestens ein gleiches Datenelement manipulieren, errechnet werden: $E(T') = E(DT) = (E(D) + 1) \cdot E(T)$.

$$E(D) = \sum_{n=1}^{\infty} n\, P(D = n) = \sum_{n=1}^{\infty} n \frac{\binom{e}{o} \cdot \left(\binom{e}{o} - \binom{e-o}{o}\right) \cdot \binom{e-o}{o}^n}{\binom{e}{o}^{n+2}} = \frac{\binom{e-o}{o}}{\binom{e}{o} - \binom{e-o}{o}}$$

Um $P(D = n)$ zu bestimmen, werden zunächst alle Möglichkeiten bestimmt, die es für die $n + 2$ betrachteten Transaktionen gibt, so dass die oben genannten Bedingungen der Zufallsvariable D erfüllt sind. Die erste und letzte Transaktion müssen mindestens ein gleiches Datenelement manipulieren und die Transaktionen dazwischen dürfen nicht potentiell mit der ersten in Konflikt stehen. Für die erste Transaktion gibt es $\binom{e}{o}$ Möglichkeiten o Elemente aus e Elementen auszuwählen. Die Schreibmenge der (n+2)ten Transaktion soll sich mit dieser überlappen. Es gibt $\binom{e-o}{o}$ Möglichkeiten, dass sie dieses nicht tut, also $\binom{e}{o} - \binom{e-o}{o}$ Möglichkeiten, dass sie sich mit dieser überlappt. Für die n Transaktionen zwischen der ersten und der letzten gibt es jeweils $\binom{e-o}{o}$ Möglichkeiten, so dass sich ihre Schreibmengen nicht mit der ersten überlappen.

Dividiert man die so erhaltene Zahl durch die Anzahl aller Möglichkeiten $(n + 2)$-mal o Elemente aus e Elementen auszuwählen ($\binom{e}{o}^{n+2}$), so erhält man die gesuchte Wahrscheinlichkeit.

Die gesuchte Wahrscheinlichkeit beträgt also:

$$P(D = n) = \frac{\binom{e}{o} \cdot \left(\binom{e}{o} - \binom{e-o}{o}\right) \cdot \binom{e-o}{o}^n}{\binom{e}{o}^{n+2}}$$

Für den Spezialfall $o = 1$ vereinfacht sich der Erwartungswert zu $E(D) = \frac{(e-1)}{e-(e-1)} = e - 1$

Gesamtformel

Aus den oben ermittelten Teilformeln ergibt sich unter Berücksichtigung der Knotenanzahl, der Datenbankgröße und der Transaktionslänge folgende Formel für die Konfliktwahrscheinlichkeit:

$$\begin{aligned} P(K) &= 1 - \left(1 - \frac{E(U)}{E(U) + \frac{E(T)}{k} \cdot \frac{k}{k-1} \cdot \left(\frac{\binom{e-o}{o}}{\binom{e}{o} - \binom{e-o}{o}} + 1\right)}\right)^2 \\ &= 1 - \left(1 - \frac{E(U)}{E(U) + \frac{E(T)}{k-1} \cdot \left(\frac{\binom{e-o}{o}}{\binom{e}{o} - \binom{e-o}{o}} + 1\right)}\right)^2 \end{aligned} \tag{7.2}$$

Diskussion

Natürlich spiegelt das hier vorgestellte Modell nur annähernd ein reales Replikationssystem wider. Reales Verhalten von Teilnehmern wurde mit Wahrscheinlichkeitsverteilungen angenähert. So ist die Annahme, dass sich alle Knoten in exponentialverteilten

Intervallen synchronisieren, vereinfacht. In der Realität werden Daten schrittweise über mehrere Hops verteilt und nicht zwischen allen Knoten auf einmal, wie in diesem Modell angenommen. Einzelne Knoten können über längere Zeiträume nicht mit dem Netz verbunden sein, während andere sehr schnell Daten mit benachbarten Knoten austauschen. Auch werden Nutzer nicht exponentialverteilt Transaktionen initiieren. In realen Systemen können sich Nutzer absprechen und versuchen, Konflikte zu vermeiden. Ebenfalls wird in vielen Anwendungen nicht gleichverteilt auf Datenelemente zugegriffen, sondern es ergibt sich eine natürliche Partitionierung unter den verschiedenen Nutzern, was zu einer Verminderung der Konflikte führt. Auf der anderen Seite sollte vermieden werden, dass bestimmte Datenelemente sich zu Hotspots entwickeln, auf die sehr häufig von verschiedenen Nutzern zugegriffen wird. Dieses führt zu einer erhöhten Konfliktwahrscheinlichkeit. Die Konflikterkennungs- und Lösungsmechanismen SYMORES dienen hauptsächlich dazu, zu verhindern, dass inkonsistente Datenbankzustände auftreten können. Durch Einsatzzweck, Anwendung und Nutzerverhalten muss dafür gesorgt werden, dass potentiell konfliktverursachende Aktionen so selten wie möglich sind.

Die hier hergeleitete Formel ist nicht spezifisch für SYMORE, sondern gilt für jedes optimistische, asynchrone peer-to-peer Replikationssystem. Viele Spezifika von SYMORE wie die Art der Konflikterkennung und Konfliktlösung oder die Verwendung von Echtzeit spielten in der Analyse keine Rolle und beeinflussen grundsätzlich nicht, dass Konflikte auftreten. SYMORE bietet mit der flexiblen Definition von Konfliktelementen die Möglichkeit, die Granularität der Datenelemente, auf denen Konflikte erkannt werden sollen, möglichst gering zu wählen und so die Wahrscheinlichkeit für Konflikte zu minimieren.

Trotzdem liefert diese Formel eine Richtlinie, anhand derer sich die Konfliktwahrscheinlichkeit und damit Konflikthäufigkeit in konkreten Anwendungen mit bestimmten Parametern abschätzen lässt. Für die MOBILEWIKI-Anwendung sei beispielsweise angenommen, dass jeder der 15 Teilnehmer einer kollaborierenden Gruppe im Mittel 10 Änderungen pro achtstündigem Arbeitstag vornimmt. Für jede Änderung wird eine Transaktion ausgeführt, die 2 disjunkte Elemente manipuliert. Im Mittel synchronisieren sich alle Teilnehmer zwei mal pro Stunde. Die Datenbank enthalte 2000 Elemente (Artikel und Abschnitte). Es ergeben sich also das erwartete Übertragungsintervall mit $\mathrm{E}(T) = 0,8h$ und das erwartete Transaktionsintervall mit $\mathrm{E}(U) = 0,5h$. Damit ergibt sich die Wahrscheinlichkeit, dass irgendeine Transaktion mit mindestens einer anderen in Konflikt steht, von 3,4%. Das bedeutet, dass von allen $15 * 10 = 150$ an einem Tag ausgeführten Transaktionen 5,1 im Mittel mit anderen in Konflikt stehen. Da der Konfliktlösungsalgorithmus in SYMORE nicht alle in Konflikt stehenden Transaktionen abbricht, sondern an jedem Konfliktelement eine Transaktion festgeschrieben werden kann, müssen im günstigsten Fall von diesen nur 2,55 Transaktionen abgebrochen werden.

Da Transaktionen zunächst asynchron und lokal ausgeführt werden und erst nach einem Pre-Commit verteilt werden, können selbst im optimalen Fall, wenn sich alle Knoten in Kommunikationsreichweite zueinander befinden, Konflikte auftreten. Die Wahrscheinlichkeit für einen Konflikt ist dann allerdings sehr gering. Verringert sich E(U) bei-

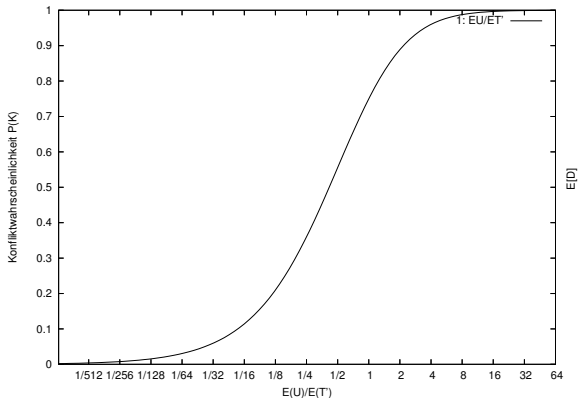

Abbildung 7.3: P(K) in Bezug auf E(U)/E(T′)

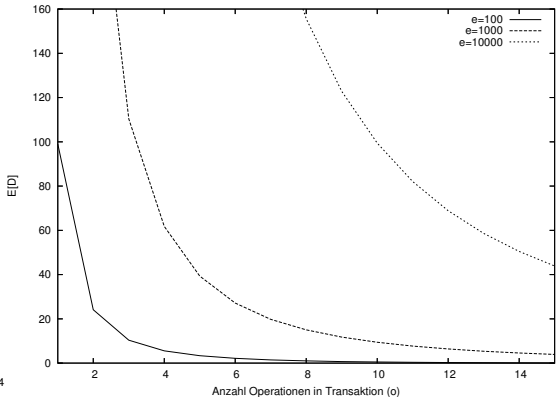

Abbildung 7.4: E(D) in Bezug auf Datenbankgröße und Transaktionslänge

spielsweise auf 10s, so beträgt die Konfliktwahrscheinlichkeit im obigen Fall weniger als 0,02%.

Wie an der Herleitung von Formel 7.1 deutlich wird, hängt die Konfliktwahrscheinlichkeit von dem Verhältnis von erwartetem Übertragungsintervall E(U) zu erwartetem Intervall zwischen zwei potentiell in Konflikt stehenden Transaktionen E(T') ab. Je kleiner E(U) im Verhältnis zu E(T') ist, desto geringer ist die Konfliktwahrscheinlichkeit. Wie genau sich diese beiden Größen E(U) und E(T') auf die Konfliktwahrscheinlichkeit auswirken, zeigt Abbildung 7.3. Die Kurve dort zeigt das Verhältnis von E(U) zu E(T') bei einer logaritmischen Skalierung der x-Achse zur Basis 2. So kann die Konfliktwahrscheinlichkeit auch für Werte kleiner als 1 gut abgelesen werden. Außerdem lässt sich einfach vergleichen, wie sich die Konfliktwahrscheinlichkeit bei Verdopplung oder Halbierung des Verhältnisses $E(U)/E(T')$ verhält. Bei gleichem Verhältnis von E(U) zu E(T') beträgt die Konfliktwahrscheinlichkeit 75%. Konfliktwahrscheinlichkeiten von weniger als 10% ergeben sich erst bei Verhältnissen von weniger als 1/16. Bei weiterer Verdopplung von E(T′) nimmt die Konfliktwahrscheinlichkeit allerdings immer geringer ab.

Interessant ist nun zu untersuchen, wie die Parameter Knotenanzahl, Datenbankgröße und die Transaktionslänge den Wert E(T') beeinflussen.

Da die Formel für den Erwartungswert der Zuvallsvariablen D nicht ganz einfach zu interpretieren ist, ist diese für verschiedene Datenbankgrößen und Transaktionslängen in Abbildung 7.4 dargestellt. Man erkennt, dass der erwartete Abstand zweier in Konflikt stehender Transaktionen mit zunehmender Transaktionslänge rapide abnimmt. Damit steigt die Konfliktwahrscheinlichkeit überproportional an.

In den Abbildungen 7.5 und 7.6 ist die Konfliktwahrscheinlichkeit für feste Verhältnisse von E(U) zu E(T), einer Transaktionslänge von 1 und verschiedenen Werten für die Knotenanzahl und die Größe der Datenbank dargestellt. Auf diese Weise wird ersichtlich, mit welchen Parametern welche Konfliktwahrscheinlichkeiten erreicht werden. Ist das Verhältnis E(U) zu E(T) gleich 1, so ist die Konfliktwahrscheinlichkeit selbst bei 50

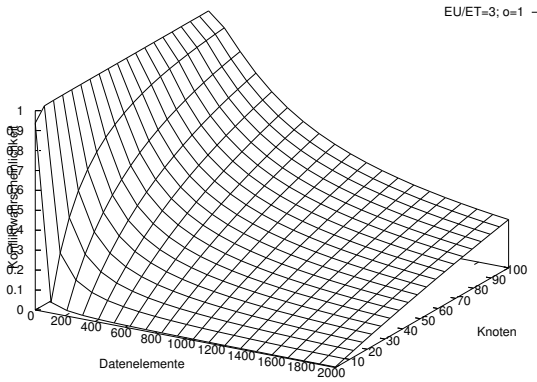

Abbildung 7.5: P(K) mit $\frac{E(U)}{E(T)} = 3$

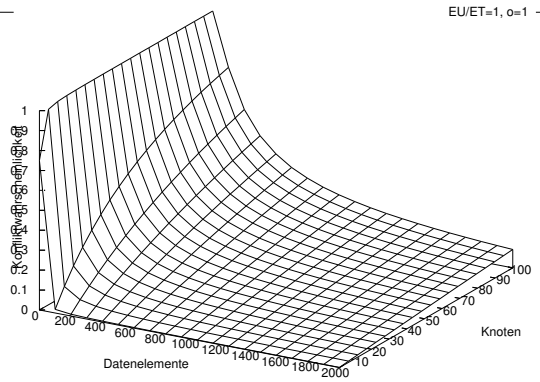

Abbildung 7.6: P(K) mit $\frac{E(U)}{E(T)} = 1$

Knoten und 1000 Datenelementen kleiner als 10% und bei 100 Knoten und 1000 Datenelementen noch kleiner als 20%. Bei einem Verhältnis von E(U) zu E(T) von 3 werden bei diesen Werten bereits etwa 24% bzw. 40% erreicht.

7.3 Gray: Dangers of Replication

Eine andere Abschätzung der Konfliktwahrscheinlichkeit in optimistischen und asynchronen Peer-to-Peer-Replikationssystemen nimmt Gray in [GHOS96, S. 178] vor.

Das Systemmodell dort ist ähnlich dem für die obige Analyse angenommenen. Jeder Knoten initiiert mit der Frequenz TPS lokal Transaktionen, die aus je o Updateoperationen bestehen. Diese werden auf Kopien von Datenelementen ausgeführt, die gleichverteilt aus einer Datenbank mit e Elementen gewählt werden. In seinem Modell hat ein Knoten für eine bestimmte Zeitspanne $Disconnected\_Time$ keine Verbindung zu anderen Knoten. In dieser Zeit werden dort $Outbound\_Updates \approx disconnect\_time \cdot TPS \cdot o$ Operationen ausgeführt. Da jeder andere Knoten ebenfalls etwa diese Anzahl an Operationen ausführt, empfängt der Knoten, wenn er sich wieder mit dem Netz verbindet, $Inbound\_Updates \approx (k-1) \cdot disconnect\_time \cdot TPS \cdot o$ Operationen auf unterschiedlichen Datenelementen.

Konflikte entstehen, wenn $Inbound\_Updates$ Datenelemente enthält, die sich auch in $Outbound\_Updates$ befinden. Die Wahrscheinlichkeit dafür wird von Gray näherungsweise folgendermaßen angegeben:

$$P(conflict) \approx \frac{Inbound\_Updates \cdot Outbound\_Updates}{e} \tag{7.3}$$

Bei dieser Formel können Werte größer als 1 auftreten. Es handelt sich also nicht um eine Dichtefunktion. Diese Näherung gilt somit nur, wenn $e > |Inbound\_Updates \cdot Outbound\_Updates|$ ist. Setzt man die obigen Formeln für $Inbound\_Updates$ und

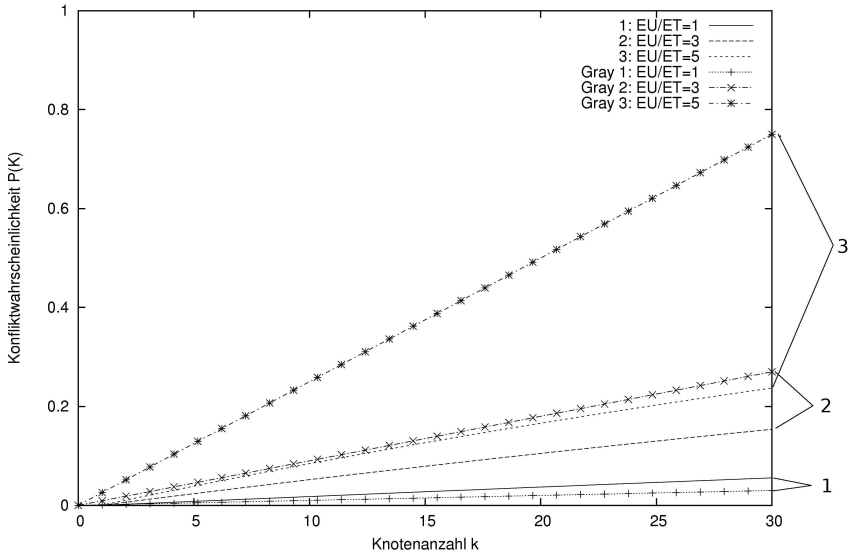

Abbildung 7.7: Vergleich der Konfliktwahrscheinlichkeiten nach Gray und nach der Analyse in 7.2.

Outbound_Updates in diese Abschätzung ein, so erhält man:

$$P(conflict) \approx \frac{k \cdot (Disconnect\_Time \cdot TPS \cdot o)^2}{e}$$

Aus dieser Formel leitet Gray die Rate ab, mit der Konflikte im gesamten System behoben werden müssen. Diese beträgt etwa:

$$\begin{aligned} Lazy\_Group\_Reconciliation\_Rate &\approx P(conflict) \cdot \frac{k}{Disconnect\_Time} \\ &\approx \frac{Disconnect\_Time \cdot (TPS \cdot o \cdot k)^2}{e} \end{aligned}$$

Man erkennt hier, dass diese Rate mit dem Quadrat der Knoten und Transaktionen in dem System wächst, aber nur mit der einfachen Zeit, in der Knoten unverbunden sind. Wird die Anzahl der Knoten also beispielsweise um den Faktor 10 erhöht, so erhöht sich die Rate, mit der Konflikte behoben werden müssen bereits um den Faktor 100.

Für einen Vergleich der Analyse Grays mit der Formel 7.2 der obigen Analyse in Abschnitt 7.2 sind in Abbildung 7.7 die Werte, die sich aus beiden Formeln bei gleicher Eingabe ergeben, für einige Eingabewerte gegenübergestellt. Die *Disconnect_time* in Grays Formel entspricht etwa dem Erwartungswert des Übertratungsintervalls $E(U)$ der Formel 7.2. TPS entspricht etwa $\frac{1}{E(T)}$. Als Parameter sind $e = 1000$ und $o = 1$ sowie unterschiedliche Verhältnisse von $E(U)/E(T)$ gewählt. Auf der x-Achse wird über die Knotenanzahl variiert. Bis auf Eingaben mit kleinem Verhältnis $E(U)/E(T)$ liefert Grays Abschätzung wesentlich höhere Konfliktwahrscheinlichkeiten als die obige Analyse. Diese steigen bei zunehmendem k zudem schneller an. Dieses liegt daran, dass Gray ein etwas anderes Modell verwendet. Im Gegensatz zur Analyse in Abschnitt 7.2 geht Gray

von einer festen Zeit der Unverbundenheit aus, in der eine feste Anzahl an Transaktionen stattfinden. Er verwendet keine Wahrscheinlichkeitsverteilungen. Auch ist seine Abschätzung für die Anzahl der Objekte, die sich in den Mengen *Inbound_Updates* und *Outbound_Updates* überlappen, in Gleichung 7.3 nicht genau, insbesondere wenn diese Mengen im Verhältnis zur Datenbankgröße viele Elemente enthalten. Ist die Datenbank dagegen groß und die Anzahl der Elemente in diesen Mengen klein, liefert die Abschätzung recht gute Werte.

Abschließend lässt sich feststellen, dass die in dieser Arbeit hergeleitete Formel in ihrer Genauigkeit über die Graysche Formel hinausgeht. Sie ist somit besser geeignet, die Konfliktwahrscheinlichkeit bei optimistischer Peer-to-Peer-Replikation zu bestimmen. Sie zeigt zudem, dass die Konfliktwahrscheinlichkeit in vielen Fällen wesentlich geringer ist als durch Grays Abschätzung behauptet wird.

8 Zusammenfassung und Ausblick

In diesem Kapitel wird zusammengefasst, was in der vorliegenden Diplomarbeit erreicht wurde. Anschließend wird in einem Ausblick aufgezeigt, welche Aspekte ergänzt werden sollten, um von dem im Rahmen der Arbeit entwickelten Prototyp des Replikationssystems zu einem Produktivsystem zu gelangen und in welche Richtungen das hier vorgestellte Replikationskonzept weiterentwickelt werden könnte.

8.1 Zusammenfassung

Im Rahmen der vorliegen Arbeit wurde das System SYMORE, ein optimistisches und asynchrones Peer-to-Peer-Datenbankreplikationssystem für mobile Geräte, entworfen und in Java prototypisch für J2ME CDC implementiert. Aufgabe dieses Systems ist es, Datenelemente, die auf jedes mobile Gerät repliziert sind, global konsistent zu halten. Auf Grund des optimistischen Synchronisationsansatzes sind Konflikte zwischen Transaktionen auf unterschiedlichen Geräten und zwischenzeitliche Abweichungen eines lokalen Datenbankzustandes von einem globalen Zustand möglich. Eventual Consistency wird aber erreicht. Anders als in vielen anderen optimistischen Replikationssystemen können Gruppen von Operationen zu Transaktionen zusammengefasst werden. Letztere werden atomar und in Isolation zu anderen lokalen Transaktionen ausgeführt. Die Semantik der von SYMORE verwendeten Transaktionen orientiert sich an der von Transaktionen klassischer zentralisierter Datenbanksysteme. Die klassische Serialisierbarkeitstheorie kann demnach auch in SYMORE angewendet werden. Es wurde gezeigt, dass 1-Kopien-Serialisierbarkeit aller definitiv festgeschriebenen Transaktionen garantiert wird.

SYMORE bietet Flexibilität im Umgang mit Konflikten. So können nicht nur die kausal parallelen Manipulationen von Kopien gleicher Datenelemente, sondern optional auch Lese-Schreib-Konflikte erkannt und Phantome vermieden werden. Außerdem kann ein Anwender des Replikationssystems Datenelemente zu größeren Einheiten, den Konfliktelementen, zusammenfassen. So kann zwischen Konflikterkennung und der Menge der zu übertragenen Daten abgewogen werden. Je feingranularer die Konflikterkennung eingestellt ist, desto höher ist der Datenaufwand. Auch ist es möglich, auf manchen Strukturen keine Konflikterkennung durchzuführen und so auch kausal parallele Transaktionen zuzulassen.

Beziehungen zwischen Transaktionen werden in einem Vorgängergraphen dargestellt. Die in SYMORE verwendete syntaktische Konflikterkennung ist nicht so flexibel wie eine semantische Konflikterkennung. Sie ist allerdings effizienter, was für leistungsschwä-

chere mobile Geräte wichtig ist. Zur syntaktischen Konflikterkennung reicht eine Bestimmung der Vorgängerbeziehungen zwischen Transaktionen aus. Im Gegensatz dazu müssten bei semantischer Konflikterkennung bei jeder Änderung des vorläufigen Datenbankzustandes die Vorbedingungen von Transaktionen erneut ausgewertet werden. Dieses erfordert viel Rechenzeit. Es gibt sicher Anwendungsfälle, für die eine semantische Konfliktlösung vorteilhaft ist. Für viele Problemstellungen reicht die syntaktische Konflikterkennung jedoch aus, wie nicht zuletzt an der Beispielanwendung des mobilen, Peer-to-Peer-Wikisystems demonstriert wurde. Ein weiterer Vorteil SYMORES ist seine Orientierung an der Handhabung zentralisierter Datenbanksysteme. Dadurch wird die Verwendung des Systems erleichtert. Ein Anwendungsentwickler kommuniziert wie gewohnt über die JDBC-Schnittstelle mit dem Replikationssystem und verwendet SQL, um Daten abzurufen und zu manipulieren. Ein entsprechender SQL-Parser wurde für den Prototypen entwickelt.

Durch die Verwendung von Echtzeit können Konfliktlösungsalgorithmen entwickelt werden, die berücksichtigen, in welcher Reihenfolge auch kausal parallele Transaktionen ausgeführt wurden. Dieses führt zu einer für Nutzer intuitiveren Auswahl der Transaktionen, die im Konfliktfall abgebrochen werden. Außerdem wurde ein Konfliktlösungsalgorithmus vorgestellt, der Prioritätswerte von Transaktionen bei dieser Auswahl berücksichtigt. Das System ermöglicht zudem benutzerdefinierte Konfliktlösungsalgorithmen zu verwenden. Diese können ohne großen Aufwand entwickelt werden. Da eine Synchronisation der Uhren der mobilen Geräte nicht vollständig erreicht werden kann, wurde das Konzept der „Rasterzeit“ eingeführt. Dabei wird die Auflösung der Zeitpunkte von Ereignissen künstlich herabgesetzt. Sind Ereignisse in dieser Rasterzeit „gleichzeitig“, werden andere Ordnungskriterien herangezogen. Auf diese Weise ist sichergestellt, dass alle Ereignisse auf allen Knoten in der gleichen Reihenfolge geordnet werden. Dieses ist essentiell, um überall den gleichen Datenbankzustand zu erreichen. Die Korrektheit der verwendeten Verfahren wie Konfliktlösung, Zeitsynchronisation und das Erreichen von Eventual Consistency wurden ebenso wie die Garantie von 1-Kopien-Serialisierbarkeit nachgewiesen.

Nachrichten über lokale Transaktionen werden über eine für MANETs optimierte Verteilungskomponente effizient in der Replikationsgruppe verteilt. Außer diesen Nachrichten sind keine weiteren nötig, um Konflikte zu erkennen oder zu entscheiden, welche Transaktionen festgeschrieben werden müssen. Diese Entscheidungen trifft jeder Knoten lokal und autonom. Implizite Informationen dazu werden aus den die Transaktionen verteilenden Nachrichten gewonnen. Auf diese Weise müssen nur wenige Nachrichten versandt werden. Da jeder Sendevorgang mit einem hohen Energieverbrauch verbunden ist, wird so effizient mit den begrenzten Energiereserven mobiler Geräte umgegangen.

Schließlich wurde analysiert, wie sich die Wahrscheinlichkeit für Konflikte in einem optimistischen Peer-to-Peer-System in Bezug auf verschiedene Parameter verhält. Auch wenn bei der Analyse von einem vereinfachten Systemmodell ausgegangen wurde, ermöglicht die entwickelte Formel eine Abschätzung, für welche Anwendungsfälle diese

Art der Replikation geeignet ist und mit welcher Konfliktrate gerechnet werden muss.

Alles in allem bietet SYMORE eine neuartige, optimistische und asynchrone Peer-to-Peer-Datenreplikation. Durch die Wahl effizienter und leichtgewichtiger Verfahren für die einzelnen Teilbereiche des Systems ist ein besonders für MANETs geeignetes System entstanden, das mit einem transaktionsorientierten Ansatz über die rein zustandsorientierten Ansätze anderer Systeme hinausgeht.

Die erfolgreiche Demonstration an einem Prototypen zeigt, dass das System für die Datenreplikation in kooperativen, ad-hoc miteinander kommunizierenden Gruppen geeignet ist. Dieses lässt darauf hoffen, das System nach weiterer Verfeinerung in interessanten Anwendungen einsetzen zu können.

8.2 Ausblick

Das Ziel der Diplomarbeit war es, einen Replikations- und Synchronisationsmechanismus für relationale Datenbanken zum Einsatz auf mobilen Geräten in MANETs zu entwickeln. Der Fokus lag dabei auf der Konflikterkennung und -lösung. Es wurde ein Prototyp entwickelt, in dem das in dieser Arbeit vorgestellte Replikationskonzept erfolgreich umgesetzt wurde. Zu einem produktiv einsetzbaren System gehören noch weitere Aspekte, deren Behandlung nicht Aufgabe der hier vorliegenden Arbeit war, wie die Sicherstellung von Konsistenz auch bei Systemabstürzen und dem darauf folgenden Wiederanlauf (recovery). Bei einem Wiederanlauf einer lokalen Datenbank muss sichergestellt werden, dass keine Inkonsistenzen auftreten. Viele Konsistenzaspekte lassen sich an die eingebettete Datenbank delegieren, es müssen aber interne Datenstrukturen wie der Vorgängergraph oder die SkewMatrix wiederhergestellt werden.

Mögliche Protokolle zum Gruppenbeitritt und -austritt wurden vorgestellt. Die Methode, mit der ausgefallene Knoten sicher und konsistent aus der Replikationsgruppe ausgeschlossen werden können, muss noch weiter verfeinert werden.

Das Verfahren zur Konflikterkennung kann ebenfalls weiterentwickelt werden. Trotz der oben genannten Vorteile syntaktischer Konflikterkennung kann ihr Einsatz in manchen Fällen zu eigentlich unnötigen Transaktionsabbrüchen führen. Gerade in einer mobilen Umgebung mit unter Umständen länger andauernden Phasen der Unverbundenheit sollten nebenläufige Änderungen an logischen Datenelementen so wenig wie möglich eingeschränkt werden. Die hergeleitete Formel zur Abschätzung der Konfliktwahrscheinlichkeit einer Transaktion gilt allgemein für optimistische und asynchrone Peer-to-Peer-Replikationssysteme. Soll der Ansatz eines vollständig dezentralen Systems mit gleichberechtigten Teilnehmern beibehalten werden, sind hier keine größeren Verbesserungen möglich. Wenn auch die Wahrscheinlichkeit potentieller Konflikte nicht verringert werden kann, so kann verhindert werden, dass mindestens jede zweite der in Konflikt stehenden Transaktionen abgebrochen werden muss. Für manche Anwendungsfälle sind starke Garantien der Isolationseigenschaften von Transaktionen nicht nötig. Nicht immer stellen kausal parallele Operationen aus Anwendungssicht einen Konflikt

dar. SYMORE bietet die Möglichkeit, für manche Datenelemente keine Konflikterkennung vorzunehmen und so (kommutative) Operationen auch kausal parallel auszuführen. Dieses Verfahren ist einfach und effizient. Manchmal wäre es aber wünschenswert, komplexere Vorbedingungen definieren zu können, anhand derer festgestellt wird, ob eine Transaktion erfolgreich angewendet werden kann. In MOBILEWIKI wäre eine solche Bedingung z.B., dass eine Positionsnummer in allen zu einem bestimmten Artikel gehörenden Abschnitten eindeutig ist. Neben der syntaktischen Konflikterkennung ist es also wünschenswert, auch eine semantische Konflikterkennung zu ermöglichen. IceCube geht diesen Weg, allerdings auf sehr komplexe Weise. Im Gegensatz zu IceCube ist SYMORE ein echtes Peer-to-Peer-System, so dass die Lösungen, die IceCube bietet, nicht einfach übernommen werden können. In SYMORE könnte zunächst erkannt werden, ob zwei Transaktionen parallel ausgeführt worden sind. Nur in diesem Fall ist es nötig, die semantischen Vorbedingungen dieser Transaktionen zu überprüfen. Damit kann entschieden werden, ob beide Transaktionen trotzdem angewendet werden können oder eine von beiden abgebrochen werden muss. Ist der kausal parallele Zugriff auf ein Datenelement immer ein Konflikt, wie bei der Änderung eines Abschnitts eines Artikels, soll die semantische Konflikterkennung auch deaktiviert werden können.

Des Weiteren wurde in der vorliegenden Arbeit eine autonome, abgeschlossene Replikationsgruppe betrachtet. In vielen realen Anwendungen wird man dagegen eine Lösung vorziehen, bei der mobile Geräte im Normalfall mit einem Server verbunden sind und auf einen zentralen Datenbestand zugreifen. Diese mobilen Geräte können Daten des Servers replizieren und in Fällen, in denen keine Netzwerkverbindung zu der zentralen Datenbank vorhanden ist, mit diesen Kopien arbeiten. Interessant wäre es zu untersuchen, wie dieses erweiterte Client-Server-Szenario mit dem hier vorgestellten Peer-to-Peer-Replikationssystem kombiniert werden kann. Befinden sich mehrere mobile Geräte in Kommunikationsreichweite und besteht keine Verbindung zum Server, sollte es möglich sein, dass mobile Geräte Änderungen auch untereinander austauschen.

Abschließend lässt sich feststellen, dass SYMORE sowohl konzeptionell als auch praktisch in Form des vorliegenden Prototyps eine solide Grundlage bildet, um das Replikationskonzept in den in diesem Abschnitt skizzierten Richtungen weiterzuentwickeln.

Literaturverzeichnis

[AAS97] D. Agrawal, A. El Abbadi, and R. C. Steinke. Epidemic algorithms in replicated databases (extended abstract). In *PODS '97: Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 161–172, New York, NY, USA, 1997. ACM Press.

[BHG87] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[CDK00] George F. Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. Addison-Wesley, Boston, MA, USA, 2000.

[CM99] S. Corson and J. Macker. Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations. RFC 2501 (Informational), January 1999. http://www.ietf.org/rfc/rfc2501.txt.

[Cri89] F. Cristian. A probabilistic approach to distributed clock synchronization. *Distributed Computing*, 3:146–158, 1989.

[Dan00] Peter H. Dana. Global positioning system overview. http://www.colorado.edu/geography/gcraft/notes/gps/gps_f.html, 2000.

[Die04] Diehl and Associates, Inc. Mckoi SQL Database. http://mckoi.com/database/, 2004.

[EMRP01] Todd Ekenstam, Charles Matheny, Peter L. Reiher, and Gerald J. Popek. The bengal database replication system. *Distributed and Parallel Databases*, 9(3):187–210, 2001. http://citeseer.ist.psu.edu/ekenstam01bengal.html.

[GHOS96] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. pages 173–182, 1996. brief description of different replication models and their probabilities concerning waits, deadlocks and reconciliations; furthermore a 2-tier-replicationmodel is described.

[Gra06] Alex Graveley. Tomboy. http://www.beatniksoftware.com/tomboy/, 2006.

[Ham] Brad Hammond. Wingman: A replication service for microsoft access and visual basic. Microsoft White Paper.

[HTKR05] Hagen Höpfner, Can Türker, and Birgitta König-Ries. *Mobile Datenbanken und Informationssysteme: Konzepte und Techniken*. dpunkt, 2005.

[Hü00] Gerhard Hübner. *Stochastik. Eine anwendungsorientierte Einführung für Informatiker, Ingenieure und Mathematiker*. Vieweg, Braunschweig/Wiesbaden, 2nd edition, 2000.

[iAn04] iAnywhere. *Adaptive Server Anywhere Datenbankadministration*. A Sybase Company, 10 2004.

[IBM04] IBM Corporation. *IBM DB2 Everyplace Application and Developement Guide Version 8.2*, 8 2004.

[Jav06] Java Compiler Compiler. https://javacc.dev.java.net/, 2006.

[KE04] Alfons Kemper and André Eickler. *Datenbanksysteme - Eine Einführung, 5. Auflage*. Oldenbourg, 2004.

[KRSD01] Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, and Peter Druschel. The icecube approach to the reconciliation of divergent replicas. In *PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 210–218, New York, NY, USA, 2001. ACM Press.

[LKBH+88] Jr. Leonard Kawell, Steven Beckhardt, Timothy Halvorsen, Raymond Ozzie, and Irene Greif. Replicated document management in a group communication system. In *CSCW '88: Proceedings of the 1988 ACM conference on Computer-supported cooperative work*, page 395, New York, NY, USA, 1988. ACM Press.

[LL84] Jennifer Lundelius and Nancy A. Lynch. An upper and lower bound for clock synchronization. *Information and Control*, 62(2/3):190–204, 1984.

[Mat89] Friedemann Mattern. Virtual time and global states of distributed systems. In Cosnard M. et al., editor, *Proc. Workshop on Parallel and Distributed Algorithms*, pages 215–226, North-Holland / Elsevier, 1989. (Reprinted in: Z. Yang, T.A. Marsland (Eds.), "Global States and Time in Distributed Systems", IEEE, 1994, pp. 123-133.).

[MBT04] Lennart Meier, Philipp Blum, and Lothar Thiele. Internal synchronization of drift-constraint clocks in ad-hoc sensor networks. In *MobiHoc '04: Proceedings of the 5th ACM international symposium on Mobile ad hoc networking and computing*, pages 90–97, New York, NY, USA, 2004. ACM Press.

[Mil92] D. Mills. Network time protocol (version 3) specification, implementation and analysis. RFC 1305, March 1992. http://www.ietf.org/rfc/rfc1305.txt.

[Mil06] D. Mills. Simple network time protocol (sntp) version 4 for ipv4, ipv6 and osi. RFC 4330 (Informational), January 2006. http://www.ietf.org/rfc/rfc4330.txt.

[OM05] James O'Brien and Marc. An application framework for collaborative, nomadic applications. Rapport de recherche RR-5745, Inria, Rocquencourt (France), November 2005. http://www.inria.fr/rrrt/rr-5745.html.

[PB99] S. Phatak and B. Badrinath. Multiversion reconciliation for mobile databases. In *ICDE '99: Proceedings of the 15th International Conference on Data Engineering*, page 582, Washington, DC, USA, 1999. IEEE Computer Society.

[PB04] S. Phatak and B. Badrinath. Transaction-centric reconciliation in disconnected client-server databases. *Mob. Netw. Appl.*, 9(5):459–471, 2004.

[PBM+00] Nuno M. Preguica, Carlos Baquero, Francisco Moura, J. Legatheaux Martins, R. Oliveira, Henrique Joao L. Domingos, J. O. Pereira, and Sergio Duarte. Mobile transaction management in mobisnap. In *ADBIS-DASFAA*, pages 379–386, 2000. http://citeseer.ist.psu.edu/preguica00mobile.html.

[PSTT96] Karin Petersen, Mike Spreitzer, Douglas Terry, and Marvin Theimer. Bayou: Replicated database services for world-wide applications. In *7th ACM SIGOPS European Workshop*, Connemara, Ireland, 1996. http://citeseer.ist.psu.edu/petersen96bayou.html.

[Rö01] Kay Römer. Time synchronization in ad hoc networks. In *MobiHoc '01: Proceedings of the 2nd ACM international symposium on Mobile ad hoc networking & computing*, pages 173–182, New York, NY, USA, 2001. ACM Press.

[Rö06] Holger Rösch. Implementierung und experimentelle Untersuchung einer Replikationskomponente für mobile Endgeräte in mobilen ad-hoc Netzwerken. Master's thesis, Freie Universität Berlin (Germany), 2006.

[Sch05a] Manuel Scholz. Mobile conflict tree: A data structure for replication in mobile networks. 2005.

[Sch05b] Yark Schröder. Implementierung einer Datenverteilungskomponente für PDAs in Java. Master's thesis, Freie Universität Berlin (Germany), 2005.

[SH99] Gunter Saake and Andreas Heuer. *Datenbanken: Implementierungstechniken*. MITP-Verlag, 1999.

[SS05] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

A Benutzung von Symore

Wie bereits in Kapitel 5 beschrieben wurde, sieht ein Anwender SYMORE wie einen normalen Java-JDBC-Datenbanktreiber und interagiert mit diesem über die normale JDBC-API.

Um ein Datenbankexemplar zu starten, muss ein Exemplar der Klasse `SymoreDataSource` erzeugt werden. Diese Klasse implementiert `java.sql.DataSource` und bietet zusätzlich eigene Methoden, um die Datenbank zu konfigurieren. Diese werden im Folgenden erläutert. Die eigentliche Datenbank wird mit dem ersten Aufruf von `getConnection()` gestartet.

setDatabaseName(String dbname): Setzt den Namen der Datenbank, auf die diese DataSource zugreift.

setCreateDatabase(String create): Teilt der Datenbank mit, dass die mittels `setDatabaseName` angegebene Datenbank erzeugt werden soll, falls sie noch nicht existiert.

setShutdownDatabase(String shutdown): Die Datenbank soll beim nächsten Versuch, ein Connection-Objekt zu erzeugen, heruntergefahren werden.

setPropertiesFile(String file): Gibt die Datei an, aus der Parameter zur Konfiguration des Replikationssystems bei dessen Start geladen werden.

setProperty(String key, String value): Setzt Parameter „key“ auf Wert „value“. Wird ausgewertet, nachdem die Property-Datei eingelesen wurde und überschreibt ggf. den Werte der dort gesetzten Parameter.

registerChangedViewListener: Registriert einen Beobachter, der benachrichtigt wird, wenn sich die vorläufige Datenbanksicht ändert.

registerTACompletedListener: Registriert einen Beobachter, der benachrichtigt wird, wenn eine lokal initiierte und lokal committete Transaktion auch global committet oder abgebrochen wird.

A.1 Konfigurationsparameter

Hier werden alle Konfigurationsparameter aufgelistet, mit denen SYMORE konfiguriert werden kann. Wird ein Parameter nicht über die Konfigurationsdatei oder direkt mittels

setProperty gesetzt, wird ein Default-Wert verwendet. Die verwendete Verteilungskomponente muss ggf. separat konfiguriert werden.

Parameter	Beschreibung	Defaultwert
sendPort	Port, an den die Synchronisationskomponente sendet	33729
receivePort	Port, an dem die Synchronisationskomponente auf Pakete lauscht	33729
sendPort-Disseminator	Port, an den die Dissemination-Komponente Daten sendet	34001
receivePort-Disseminator	Port, an dem die Dissemination-Komponente auf Pakete lauscht	34001
SiteId	Global eindeutige Id des RM	Zufallswert
MemberNumber	Anzahl der Teilnehmer an der Replikationsgruppe (außer eigenem RM)	0
Member$i	RM-Id des $i-ten Replikationsteilnehmers. 0 < $i < MemberNumber	–
Delta	Rasterzeitintervall in Millisekunden	50
ConflictSolving-Algorithm	Vollständiger Bezeichner einer Klasse, die `ConflictSolvingAlgorithm` implementiert und zur Konfliktlösung eingesetzt werden soll.	repldb.conflictsolving.First-WinsHeadConflictSolving-Algorithm
ReadSet	Leseoperationen werden bei der Bestimmung von Konflikten mit berücksichtigt	false
Phantoms	Phantome werden durch Umschreiben von Unteranfragen vermieden	false
Skew-Determination-Algorithm	Die Zeitdifferenz zwischen zwei RM wird über den Zeitbestimmungsalgorithmus ermittelt (Alternativ wird die Differenz von Sendezeitstempel zu Empfangszeitstempel verwendet (garantiert kein maxDelta!)).	false
MaxSkewDeterminationRequests-Retries	Anzahl Versuche, um die Zeitdifferenz zu einem bestimmten RM zu bestimmen, bevor aufgegeben wird.	5
Skew-Determination-Timeout	Dauer, die auf eine SkewDeterminationRequest-Antwort gewartet wird, nachdem eine Anfrage abgeschickt worden ist	5000
PeriodiCommit	ScheduledCommits finden statt	false
CommitInterval	Intervall in dem die ScheduledCommits stattfinden (in Millisekunden); falls PeriodicCommit=true	20000
DummyMsg-Interval	Intervall, in dem Dummy-Nachrichten verschickt werden, falls ein RM inaktiv ist (in Millisekunden).	120000

A.2 Eigene Konfliktlösungsalgorithmen

Wie in Kapitel 4 beschrieben wurde, kann aus verschiedenen Algorithmen zur Konfliktlösung gewählt werden. Außerdem können eigene Konfliktlösungsalgorithmen implementiert werden.

Ein eigener Algorithmus muss von der abstrakten Klasse `repldb.conflictsolving.ConflictSolvingAlgorithm` erben und deren abstrakte Methode `solveConflict` implementieren. Diese Methode erhält als Eingabe ein Exemplar des `ISentinelManager`, über den der Zugriff auf alle Knoten des Vorgängergraphen möglich ist, sowie die Zeitpunkte `start` und `end` in Form von `Event`-Objekten. Ein Event enthält den global eindeutigen Zeitstempel, bestehend aus RM-Id, Rasterzeit und Sequenznummer und implementiert `Comparable` so, dass Event-Exemplare anhand ihrer Zeitstempel geordnet werden können. Da auch alle anderen Ereignisse, wie `Transaction` oder `ForcedCommit` von Event erben, können somit alle Ereignisse einfach geordnet werden.

Die Kriterien für einen korrekten Konfliktlösungsalgorithmus sind in Abschnitt 4.4 beschrieben. Die abstrakte Klasse `ConflictSolvingAlgorithm` bietet zwei Methoden `checkAllDependentTransactionsAreActive` und `checkNoTwoNodesAreInConflict`, mit denen geprüft werden kann, ob nach Ausführung des eigenen Konfliktlösungsalgorithmus alle Vorgängerknoten eines aktivierten Knotens aktiviert sind und ob es bis zum Zeitpunkt `end` keine Konflikte in dem Vorgängergraphen zwischen zwei aktivierten Knoten gibt.

Genutzt wird der eigene Konfliktlösungsalgorithmus, indem dem Parameter ConflictSolvingAlgorithm in der Konfigurationsdatei der vollständige Klassenpfad der Klasse, die diesen Algorithmus implementiert, angegeben wird.

A.3 Eigene Datenverteilung

Auch verschiedene Komponenten zu Datenverteilung sind einsetzbar. Die Datenverteilung besteht aus den beiden Diensten *Dissemination* und *Synchronisation*. Der Dienst Dissemination hat als Aufgabe, ihm übergebene Daten an alle momentan erreichbaren Knoten in seiner Umgebung zu verteilen. Der Dienst Synchronisation verwendet eine eigene Pufferung der ihm übergebenen Daten und sorgt dafür, dass alle Daten irgendwann von allen Knoten der Replikationsgruppe empfangen werden, auch wenn diese zum Zeitpunkt der Übergabe an den Synchronisationsdienst gerade nicht in Kommunikationsreichweite sind.

Eine eigene Komponente muss eine Factory-Klasse implementieren, die von der abstrakten Klasse `AbstractMsgSpreadFactory` erbt und Methoden bereitstellt, um auf die beiden genannten Dienste zugreifen zu können. Diese beiden Dienste implementieren die Schnittstellen `ISymoreDisseminator` und `ISymoreSynchronizer`, die jeweils Methoden anbieten um dem Dienst Daten zur Verteilung zu übergeben und um

empfangene Nachrichten abzurufen.

A.4 Erweiterungen von SQL

Es sind einige Erweiterungen an der Sprache SQL vorgenommen worden, um so bestimmte Replikationsaspekte zu steuern. Auf diese Weise konnte darauf verzichtet werden, die JDBC-Schnittstellen zu modifizieren.

A.4.1 Datendefinitionssprache

Bei der Definition einer Tabelle müssen Konfliktgranularitäten definiert werden. Dazu wurde die Syntax der Anweisung `CREATE TABLE` erweitert. Zusätzlich zu den gewöhnlichen Spalteneinschränkungen (constraints) können die Werte `CELLSENTINEL` oder `COLSENTINEL` angegeben werden, um eine Konfliktgranularität von Zelle bzw. Spalte für diesen Spaltenbezeichner zu wählen. Konfliktgranularitäten von Reihe oder Tabelle werden als Tabelleneinschränkung `SENTINEL ROW` bzw. `SENTINEL COL` angegeben.

A.4.2 Automatische UUID-Generierung

Jede Datenbankreihe muss eine Spalte `rowId` als Primärschlüssel erhalten. Wird bei Erzeugung einer Reihe durch die `INSERT`-Anweisung dieser nicht explizit angegeben oder an der entsprechenden Stelle der Werte-Liste der Anweisung das Schlüsselwort `DEFAULT` verwendet, so wird er automatisch erzeugt. Für den dazu verwendeten Wert wird die aktuelle Uhrzeit und die IP-Adresse des Gerätes einbezogen, so dass er global eindeutig ist.

A.4.3 Interne Prozeduren

Es gibt einige Funktionen, die zusätzlich zu den Standardfunktionen in `SELECT`- und `VALUES`-Anweisungen benutzt werden können:

`getTaId`: liefert die Transaktions-Id der aktuell laufenden Transaktion (als CHAR(36))

`getLastInsertedRowId`: liefert den letzten automatisch generierten Wert für eine rowId.

`getMySiteId`: liefert die eindeutige Id des lokalen RM (als CHAR(36))

B SQL-Grammatik

Die verwendete SQL Grammatik in BNF:

Statement ::= ((Select | Insert | Update | Values | Create | Drop | Delete | Synchronize | Priority | ForcedCommit | CompleteTransaction) („;“ | <EOF>))

CompleteTransaction ::= (<COMMIT> | <ROLLBACK>)

Synchronize ::= <SYNCHRONIZE>

Priority ::= <SET> <PRIORITY> <NUMBER_LITERAL>

ForcedCommit ::= <FORCEDCOMMIT>

Delete ::= (<DELETE> <FROM> TableName (<WHERE> Expression)?)

Create ::= (<CREATE> (CreateTable))

CreateTable ::= (<TABLE> TableName ColumnDeclarationList)

ColumnDeclarationList ::= „(“ ColumnOrConstraintDefinition („,“ ColumnOrConstraintDefinition)* „)“

ColumnOrConstraintDefinition ::= (ColumnDefinition | TableConstraintDefinition | SentinelDefinition)

SentinelDefinition ::= <SENTINEL> (<ROW> | <TABLE>)

ColumnDefinition ::= (Identifier ColumnDataType (<SQLDEFAULT> Expression)? (ColumnConstraint)*)

ColumnConstraint ::= (<NOT> <NULL_LITERAL> | <NULL_LITERAL> | <PRIMARY> <KEY> | <UNIQUE> | <CELLSENTINEL> | <COLSENTINEL>)

TableConstraintDefinition ::= ((<CONSTRAINT> ConstraintName)? (<PRIMARY> <KEY> „(“ BasicColumnList „)“ | <UNIQUE> „(“ BasicColumnList „)“ | <FOREIGN> <KEY> „(“ BasicColumnList „)“ <REFERENCES> TableName („(“ BasicColumnList „)“)? ((<ON> <DELETE> ReferentialTrigger (<ON> <UPDATE> ReferentialTrigger)?) | (<ON> <UPDATE> ReferentialTrigger (<ON> <DELETE> ReferentialTrigger)?))?) (ConstraintAttributes)?)

ConstraintName ::= SQLIdentifier

ReferentialTrigger ::= (<NO> <ACTION> | <RESTRICT> | <CASCADE> | <SET> <NULL_LITERAL> | <SET> <SQLDEFAULT>)

ConstraintAttributes ::= ((<INITIALLY> (<DEFERRED> | <IMMEDIATE>) (<NOT> <DEFERRABLE> | <DEFERRABLE>)?) | ((<NOT> <DEFERRABLE> | <DEFERRABLE>) (<INITIALLY> (<DEFERRED> | <IMMEDIATE>))?))

PositiveIntegerConstant ::= <NUMBER_LITERAL>

GetStringSQLType ::= (<CHARACTER> <VARYING>) | (<LONG> <CHARACTER> <VARYING>) | (<LONG> <VARCHAR>) | (<CHAR> | <CHARACTER>) | <VARCHAR> | <CLOB>

GetNumericSQLType ::= (<INT> | <INTEGER>) | <TINYINT> | <SMALLINT> | <BIGINT> | <FLOAT> | <REAL> | <DOUBLE> | <NUMERIC> | <DECIMAL>

GetBooleanSQLType ::= (<BOOLEAN> | <BIT>)

GetDateSQLType ::= <TIMESTAMP> | <TIME> | <DATE>

GetBinarySQLType ::= (<BINARY> <VARYING>) | (<LONG> <BINARY> <VARYING>) | <LONGVARBINARY> | <VARBINARY> | <BINARY> | <BLOB>

GetTType ::= (GetStringSQLType („(“ PositiveIntegerConstant „)“)? | GetNumericSQLType („(“ PositiveIntegerConstant („,“ PositiveIntegerConstant)? „)“)? | GetBooleanSQLType | GetDateSQLType | GetBinarySQLType („(“ PositiveIntegerConstant „)“)?)

ColumnDataType ::= GetTType

Drop ::= (<DROP> (DropTable))

DropTable ::= (<TABLE> TableName)

Update ::= (<UPDATE> TableName <SET> AssignmentList (<WHERE> Expression)?)

AssignmentList ::= (Identifier <ASSIGNMENT> Expression („,“ AssignmentList)?)

Insert ::= (<INSERT> (<INTO>)? TableName ((„(“ BasicColumnList „)“)? (<VALUES> InsertDataList | GetTableSelectExpression)))

BasicColumnList ::= IdentifierWithStar („,“ IdentifierWithStar)*

InsertDataList ::= „(“ InsertExpressionList „)“ („,“ „(“ InsertExpressionList „)“)*

InsertElement ::= (<SQLDEFAULT> | Expression)

InsertExpressionList ::= InsertElement („,“ InsertElement)*

Values ::= <VALUES> InsertDataList

Select ::= (GetTableSelectExpression (<ORDERBY> SelectOrderByList)?)

GetTableSelectExpression ::= (<SELECT> (SetQuantifier)? SelectColumnList (<FROM> SelectTableList)? (<WHERE> Expression)?)

SetQuantifier ::= (<DISTINCT> | <ALL>)

SelectColumnList ::= SelectColumn („,“ SelectColumn)*

SelectColumn ::= (Expression (<AS> <IDENTIFIER>)?)

SelectTableList ::= TableDeclaration („,“ TableDeclaration)*

TableDeclaration ::= ((TableName | „(“ GetTableSelectExpression „)“) ((<AS>)? TableName)?)

SelectOrderByList ::= Identifier (<ASC> | <DESC>)? („,“ Identifier (<ASC> | <DESC>)?)*

Operand ::= (Function | IdentifierWithStar | „(“ Expression „)“ | SubQueryExpression | <NOT> Expression | <SUBTRACT> Literal | Literal | <EXISTS> SubQueryExpression | <GETTAID> | <LASTGENERATEDKEY> | <MYSITEID>)

SubQueryExpression ::= „(“ (GetTableSelectExpression | ExpressionList) „)“

ExpressionList ::= (Expression („,“ Expression)*)?

Function ::= (<IDENTIFIER> | <COUNT>) „(“ Expression („,“ Expression)* „)“

Expression ::= Operand (Operator Operand)*

Operator ::= (SubQueryOperator | (BooleanOperator | NumericOperator | StringOperator) | <BETWEEN> | <NOT> <BETWEEN>)

SubQueryOperator ::= ((<IN> | <NOT> <IN>) | (SubQueryBooleanOperator ((<ANY> | <ALL> | <SOME>))?))

TableName ::= (<IDENTIFIER> | <DOT_DELIMINATED_REF>)

Identifier ::= (SQLIdentifier | <DOT_DELIMINATED_REF>)

IdentifierWithStar ::= (<STAR> | <GLOBVARIABLE>) | Identifier

SQLIdentifier ::= (<IDENTIFIER> | <OPTION> | <ACCOUNT> | <PASSWORD> | <PRIVILEGES> | <GROUPS> | <LANGUAGE> | <NAME> | <JAVA> | <ACTION> | <TEXT>)

Literal ::= (<STRING_LITERAL> | <BOOLEAN_LITERAL> | <NULL_LITERAL> | <NUMBER_LITERAL> | <HEX_STRING>)

BooleanOperator ::= (<ASSIGNMENT> | <EQUALS> | <GR> | <LE> | <GREQ> | <LEEQ> | <NOTEQ> | <IS> <NOT> | <IS> | <LIKE> | <NOT> <LIKE> | <AND> | <OR>)

SubQueryBooleanOperator ::= (<ASSIGNMENT> | <EQUALS> | <GR> | <LE> | <GREQ> | <LEEQ> | <NOTEQ>)

NumericOperator ::= (<DIVIDE> | <ADD> | <SUBTRACT> | <STAR>)

StringOperator ::= (<CONCAT>)