Technische Universität Kaiserslautern
Fachbereich Informatik
Lehrgebiet Informationssysteme

Integriertes Seminar zum Thema:
Mobile and Context-aware Database Technologies and Applications

Sommersemester 2007

Konsistenzsicherung für mobile Anwendungen

Volker Hudlet
v_hudlet@informatik.uni-kl.de

Betreuer: Christian Mathis

Zusammenfassung Da mobile Geräte mit diversen Problemen, wie Verbindungsabbrüchen, konfrontiert sind, die herkömmliche Rechner normalerweise nicht haben, können bisherige Verfahren, die die Konsistenz der bearbeiteten Daten sichern sollen, nicht ohne weiteres eingesetzt werden. Diese müssen erweitert oder durch fähigere Verfahren ersetzt werden. Dieser Artikel präsentiert verschiedene dieser Verfahren und zeigt Vor- und Nachteile auf.

1 Einführung

Für klassische Szenarien, die keine Mobilität unterstützen, gibt es bereits vielfältige Lösungen, um die Konsistenz zwischen verschiedenen Rechnern zu wahren (z.B. ACID für Transaktionen). Eine neue Herausforderung ergibt sich durch die rasante Verbreitung mobiler Geräte, wie Notebooks, PDAs oder Mobiltelefone, die zudem auch immer leistungsfähiger werden. Diese kommunizieren in der Regel drahtlos mittels Technologien wie WirelessLAN, Bluetooth oder GPRS. Das Problem drahtloser Technologien ist ihre Abhängigkeit von externen Faktoren. Da die Signale Umwelteinflüssen und Störsignalen ausgesetzt sind und diese zudem nur eine begrenzte Reichweite haben, kann es zu spontanen Verbindungsabbrüchen unbestimmter Länge kommen. Des Weiteren sind mobile Geräte batteriebetrieben, was zur Folge hat, dass das Gerät aufgrund leerer Batterien nicht erreichbar ist. Außerdem sind die Geräte beweglich, was auch berücksichtigt werden sollte, da beispielsweise ein Gerät erreichbar ist, im nächsten Moment aber nicht mehr, weil es durch die Bewegung außerhalb der Reichweite gekommen ist. Dies sind die zentralen Probleme mobiler Geräte und ihrer Kommunikation, weshalb man die klassischen konsistenzsichernden Methoden meistens nicht ohne weiteres in diesem Kontext einsetzen kann.

Betrachten wir folgendes Beispiel: Ein Außendienstmitarbeiter der Firma xy hat einen Kundentermin. Dafür erstellt er auf seinem Laptop eine lokale Replik der von ihm benötigten Firmendaten, da er beim Kunden keine Möglichkeit hat auf den Firmenserver und die Firmendatenbank zuzugreifen. Bei der Produktpräsentation entdeckt der Mitarbeiter einen Rechtschreibfehler, den er natürlich korrigiert, außerdem bestellt er für den Kunden 100 Einheiten des auf Lager liegenden Produkts. Wie kann nun sichergestellt werden, dass die Daten korrekt in der Firmendatenbank hinterlegt werden und tatsächlich Produkte vorhanden sind? Zurück in der Firma will der Mitarbeiter für seine nächste Dienstreise von seinem Notebook aus den Firmenwagen vormerken. Kurz vor Beendigung des Vorgangs bricht die WLAN-Verbindung aufgrund eines Sturmes ab. Was passiert nun mit der Buchungstransaktion?

Wie man an diesen Beispielen sieht, steckt der Teufel im Detail, d. h. man muss Fälle in Betracht ziehen, die bei stationären Hosts und drahtgebundener Kommunikation in dieser Art nicht auftreten. Dafür gibt es verschiedene Ansätze, die versuchen diese Probleme zu lösen, wobei jene meist nur einen Aspekt lösen, da sie auf eine bestimmte Anwendungsdomäne zugeschnitten sind.

Nachfolgend werden in diesem Artikel verschiedene ausgewählte Ansätze zur Konsistenzsicherung vorgestellt, ihre Funktionsweise erklärt und ihre Vor-

bzw. Nachteile aufgezeigt. Zuvor werden jedoch einige Grundlagen und Begriffe erläutert, welche von den Verfahren benutzt werden.

2 Grundlagen

2.1 Struktur des mobilen Umfelds

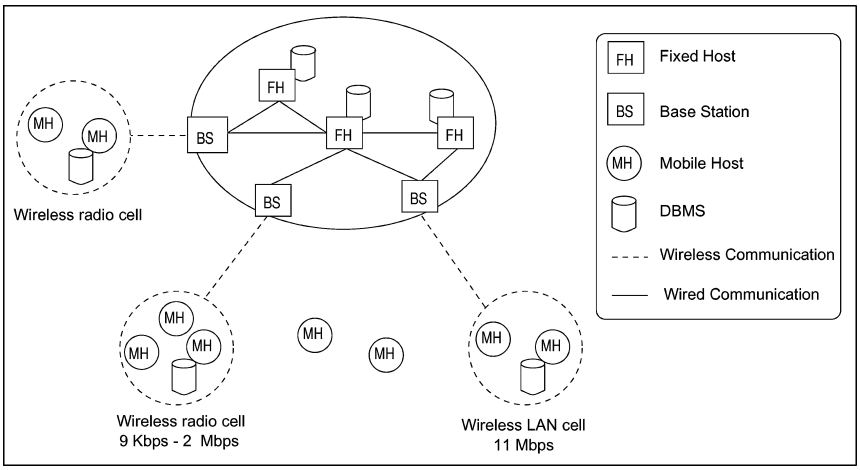

Abbildung 1. Übersicht über die Netzwerkstruktur mit mobilen Knoten

Wie bereits angesprochen, besteht bei mobilen Geräten die Problematik unbestimmter Verbindungsabbrüche, die von verschiedenster Natur sind. Es ist also nicht wie bei klassischen Client-Server-Architekturen, bei denen die Funktionalität jedes Akteurs statisch definiert ist, davon auszugehen, dass die Verbindung zwischen den Geräten (Hosts) mehr oder weniger „fest verdrahtet“ ist, sondern es muss in Betracht gezogen werden, dass sich der Ort der mobilen Einheit und die Verbindung dauernd ändern.

Deswegen unterscheiden wir zwischen *fixed host* und *mobile host* (vgl. Abb. 1). Während fixed hosts Rechner sind, die klassisch drahtgebunden miteinander vernetzt sind, sind mobile hosts mobile Geräte jedweder Art (PDA, Laptop etc.), die über drahtlose Technologien unter Zuhilfenahme einer Basisstation (z.B. Accesspoint bei WLAN) mit den anderen Hosts kommunizieren können. Dabei können allerdings mobile und ortsfeste Hosts sowohl Clients als auch Server sein, bzw. bei dezentralen Lösungen gleichberechtigte Hosts sein.

2.2 Ausführungsmodelle

(Mobile) Anwendungen operieren während ihrer Laufzeit auf Daten, deswegen ist es eine Design-Entscheidung, wo man die Datenhaltung ansiedelt. Hier existiert schon bei der klassischen Client-Server Architektur eine breite Vielfalt zwischen *Thin-Client* (Client speichert lokal keine Daten) und *Fat-Client* (Client hält die Daten lokal vor), da auch jede Variante zwischen diesen beiden Polen möglich ist. Um autonomeres Arbeiten und eine Steigerung der Verfügbarkeit zu ermöglichen, greift man auf replizierte Datenbanken [6] zurück. Dabei werden Kopien der Datenbank verteilt, die unabhängig voneinander operieren können. Nachteilig ist jedoch, dass so schnell Inkonsistenzen entstehen, die abgeglichen werden müssen.

Überträgt man diese Überlegungen auf ein mobiles Umfeld, wie in Abb. 1 gezeigt, ergeben sich nach [7] fünf Ausführungsmodelle, die nachfolgend erläutert werden.

1. *Komplette Ausführung auf ortsfesten Servern*: Hier agiert der mobile Host nur als Initiator und propagiert dem Server gewünschte Daten(änderungen), bzw. ruft im transaktionalen Kontext ein TA-Programm des TP-Monitors auf; lokal auf dem Host werden keine Daten gespeichert. Der Vorteil hierbei ist, dass die Daten zentral verwaltet werden und dadurch immer aktuell und vor allem in sich konsistent sind. Eine Variante davon wäre, die zentrale Datenbank transparent für den Benutzer zu replizieren. Dies wird beispielsweise beim Verfahren der client-zentrierten Konsistenz genutzt, welches in Abschnitt 3.3 erläutern wird. Ein weiterer Vorteil ist, dass hier auch die Anwendung auf mobilen Geräten mit begrenztem/kleinem Speicher (beispielsweise Navigationsgeräte oder Mobiltelefone) möglich ist. Solange die Verbindung zwischen mobilem Host und Basisstation besteht, gibt es keinerlei Probleme. Allerdings ist dieser Ansatz sehr unflexibel, was Verbindungsabbrüche angeht. Wenn der mobile Host keine Verbindung hat, ist es nicht möglich, Daten zu lesen/ändern oder Transaktionen zu starten, und die Anwendung muss warten, bis wieder eine Verbindung besteht.
2. *Komplette Ausführung auf dem mobilen Host*: Bei diesem Ansatz agiert die mobile Anwendung auf lokalen Daten, bzw. führt lokale Transaktionen durch. Vorraussetzung hierzu ist, dass der mobile Host die Fähigkeit besitzen muss, Daten lokal zu speichern bzw. Transaktionen auszuführen, wofür ein lokales DBMS nötig ist. Dazu wird eine lokale Replik der Daten(bank) oder einer Teilmenge davon auf dem mobilen Host hinterlegt. Der Vorteil besteht darin, dass der mobile Host eine gewisse Autonomie erhält und es so möglich ist, weiter auf den Daten zu arbeiten, selbst wenn keine Verbindung zum Server besteht. Doch wie so oft gibt es keine optimale Lösung und so gibt es auch hier ein gravierendes Problem. Dadurch dass n verschiedene Hosts parallel und unabhängig voneinander auf ihren lokalen Repliken arbeiten, entstehen zwangsläufig Inkonsistenzen zwischen den verschiedenen Repliken. Nachdem wieder eine Verbindung zum Server besteht, muss also ein Abgleich zwischen den verschiedenen Daten geschehen, um eine konsistente Version zu erhalten. Im ungünstigsten Fall müssen dabei lokale Daten wieder zurückgesetzt

werden, da sie nicht konsistent zum Gesamtsystem sind. Dies ist die zentrale Thematik, mit der sich dieser Artikel im Folgenden beschäftigt. Es gilt also die Inkonsistenzen zu vermeiden und unter Kontrolle zu halten, wozu mehrere Verfahren entwickelt worden sind. Diese werden ausgiebig in Kapitel 3 vorgestellt und ihre Funktionsweise wird erläutert.

3. *Verteilte Ausführung zwischen mobilem Host und ortsfesten Servern*: Dieses Modell ist eine Erweiterung der vorhergehenden Idee und basiert auf dem Konzept, dass der mobile Host, um Energie oder Verbindungskosten zu sparen, nur dann auf den Server zugreift, wenn dies von Nöten ist, ansonsten agiert die mobile Anwendung auf den lokalen Daten und es folgt nur ein periodischer Abgleich. Ein Beispiel für dieses Modell ist der in der Einleitung erwähnte Mitarbeiter, welcher dem Kunden das Produkt auf der Basis lokaler Daten präsentiert, aber beim Vertragsabschluß zum Bankserver verbinden muss, um beispielsweise die Bonität des Kunden zu überprüfen.
4. *Verteilte Ausführung zwischen mehreren mobilen Hosts*: Dies ist ein peer-to-peer-Ansatz, bei dem Interaktion zwischen verschiedenen mobilen Hosts ermöglicht werden soll. Geografisch nahe Hosts, die zueinander näher sind als der zentrale Server und sich zum Beispiel in der gleichen WLAN-Zelle befinden, gleichen untereinander ihre lokalen Daten ab.
5. *Verteilte Ausführung zwischen mobilen und ortsfesten Hosts*: Bei diesem Ansatz handelt es sich um ein vollständig verteiltes Szenario, welches mobile und ortsfeste Hosts gleichbehandelt. Dies könnte mit gewichteten Votieren (vgl. Abschnitt 3) erreicht werden.

[4] verallgemeinert von der jeweiligen Ausführung und postuliert mehrere Anforderungen, die ein konsistenzsicherndes Verfahren im mobilen Umfeld erfüllen sollte: Zu allererst sollte das Verfahren während einer Phase ohne Verbindung autonome Operationen auf dem mobilen Host zulassen und unterstützen. Des Weiteren sollte ein Augenmerk auf geringen Verbrauch und eventuelle Einschränkungen der Bandbreite der Verbindung gelegt werden. Darüber hinaus sollte das Verfahren in der Lage sein, sich wechselnden Verbindungsbedingungen anzupassen. Letztlich sollte es auch die Mobilität des Hosts in Betracht ziehen.

2.3 Korrektheit

Wie bei den Ausführungsmodellen ersichtlich wurde, ist Konsistenzsicherung ein wichtiges, aber nicht leicht zu realisierendes Merkmal. Dabei werden an das Verfahren, welches die Konsistenz sichern soll, mehrere Anforderungen gestellt (siehe [6]): Zum Einen soll gegenüber der mobilen Anwendung bzw. dem Benutzer dieser Anwendung Replikationstransparenz herrschen, d. h. nach außen soll nicht erkennbar sein, dass nur auf einer lokalen Replik gearbeitet wird. Dadurch sollen beispielsweise verwirrende Situationen für den Benutzer vermieden werden. Dieser hat Daten geändert, die aber vom System einige Zeit später, ohne dass er es gemerkt hat, zurückgesetzt werden. Der Effekt ist dadurch zu erklären, dass die Änderung nicht in Einklang mit den anderen Repliken zu bringen ist. Dies führt zu einem weiteren Aspekt, den es zu berücksichtigen gilt: Das Verfahren muss

nicht nur dafür sorgen, dass lokal alle Daten konsistent sind, sondern auch dafür, dass Änderungen automatisch auf alle Kopien übertragen werden, damit diese untereinander konsistent bleiben. Um diesen Anforderungen gewachsen zu sein, müssen die Verfahren das Korrektheitskriterium der 1-Kopien-Serialisierbarkeit [1] erfüllen.

Dies wird erreicht, indem nur Schedules (d. h. Ablauffolgen von Transaktionen mit ihren zugehörigen Operationen) zugelassen werden, die äquivalent zu serialisierbaren Schedules auf einer nicht-replizierten (1-Kopien-)Datenbank sind. Anders ausgedrückt soll also die Ausführung nebenläufiger Transaktionen in einer replizierten Datenbank äquivalent zu einer seriellen Ausführung dieser Transaktionen auf einer 1-Kopien-Datenbank sein.

Um diese Äquivalenz zu überprüfen, wird auf Sichtäquivalenz [1] zurückgegriffen. Sichtäquivalenz weicht die Kriterien der Äquivalenz von Historien auf, indem nur nur noch die Liest-von-Beziehungen der Historien in der richtigen Reihenfolge sein müssen und das letzte Schreiben für alle Datenelemente identisch sein muss. Die von zentralen Datenbanken bekannte Konfliktäquivalenz [1] hilft hier nicht weiter, da die Historie bei Replikation (H) andere Operationen hat als die serielle 1-Kopien-Historie (H_{1C}). Diese Operationen sind nicht vergleichbar, also auch nicht zu ordnen, was eine gleiche Anordnung konfliktbehafteter Operationen in beiden Historien (dies entspricht gerade Konfliktäquivalenz) unmöglich macht (vgl. $H =< w_1(x_A), w_2(x_B) >; H_{1C} =< w_1(x), w_2(x)) >$).

Mit dem Korrektheitsbegriff der 1-Kopien-Serialisierbarkeit haben wir nun eine Methode zur Hand, um zu überprüfen, ob der jeweilige Konsistenzsicherungs-Ansatz die Konsistenz auch wirklich sichern kann. Falls 1-Kopien-Serialisierbarkeit nicht vollständig implementiert ist oder nicht vollständig implementiert werden kann, können Schedules entstehen, die nicht 1-Kopien-serialisierbar sind. Dadurch entstehen Inkonsistenzen und Konflikte, die teilweise automatisch behoben werden können (z. B. durch Abort der Transaktion, falls diese noch nicht commitet hat), die ansonsten aber manuell aufgelöst werden müssen.

Nachfolgend werden nun mehrere Verfahren vorgestellt, die 1-Kopien-Serialisierung ansteben und auf verschiedene Art und Weise zur Konsistenzsicherung eingesetzt werden.

3 Ansätze zur Konsistenzsicherung

3.1 Mobile Transaktionen

Transaktionen unterliegen im mobilen Umfeld gewissen Eigenschaften, die sie von klassischen Transaktionen unterscheiden. Deswegen wird zuerst erklärt, was berücksichtigt werden muss, wenn man das Transaktionsmodell auf ein mobiles Umfeld übertragen will. Anschließend wird das Clustering-Verfahren vorgestellt, welches mobile Transaktionen unterstützt.

Allgemein Da mobile Hosts, wie bereits mehrfach erwähnt, Besonderheiten aufweisen, die beispielsweise bei einem zentralen DBMS nicht auftreten, müssen

diese auch bei (mobilen) Transaktionen berücksichtigt werden. Bei klassischen DBMS hat sich zur sicheren und konsistenten Ausführung von Transaktionen das ACID-Paradigma (vgl. [2]) als korrekt erwiesen. Nun stellt sich die Frage, man dieses Konzept ohne Probleme und Einschränkungen auch auf das mobile Umfeld übertragen kann.

Um dies genauer zu überprüfen, sollte zunächst der Begriff der mobilen Transaktion erläutert werden. Unter einer mobilen Transaktion versteht man nach [7] eine Transaktion, an der mindestens ein mobiler Host beteiligt ist, welcher den zuvor skizzierten Ausfallszenarien unterliegt. Dies ist eine allgemeine Definition, die hier aber vollkommen ausreicht.

Zusätzlich zu den Anforderungen an konsistenzsichernde Verfahren, die in Kapitel 2 aufgelistet wurden, kommen bei mobilen Transaktionen weitere Eigenschaften hinzu: Transaktionen können aufgrund der variablen Bandbreite sehr verschiedene Ausführungszeiten haben und die Ausführung kann je nach Zugangsmöglichkeit des Netzes unterschiedliche Kosten haben. Außerdem können aufgrund plötzlicher Verbindungsabbrüche oder leerer Batterien des mobilen Hosts Transaktionsfehler auftreten.

Will man nun das ACID-Paradigma auf ein Umfeld mit mobilen Teilnehmern übertragen, muss man mehrere Prämissen beachten, die nachfolgend erläutert werden. Das A (Atomarität) des ACID-Paradigma wird durch Commit-Protokolle gelöst. Im mobilen Umfeld sind Commits einer Transaktion aber von verbindungslosen Phasen unbestimmter Länge betroffen. Während dies bei Ausführungsmodell 1 (vgl. Kapitel 2.2) noch relativ unproblematisch ist (die Transaktion kann commiten, während der mobile Host offline ist; dieser bekommt das Ergebnis beim nächsten Verbinden mitgeteilt), kann dies bei der lokalen Ausführung auf dem mobilen Host (Ausführungsmodell 2) zu Problemen führen. Hier ist es nötig, bereits lokal geänderte Daten in den Server zu integrieren und zu commiten. Da es sein kann, dass keine Verbindung besteht, muss das Commit verzögert – bei Wiederverbindung – geschehen. Da zwischenzeitlich andere Host schon Änderungen propagiert haben können, die zu den eigenen Änderungen unverträglich sind, kann dies zu einem Abort der Transaktion führen.

Bei verteilten Transaktionen hat sich bei ortsfesten Hosts das Two-Phase-Commit-Protokoll als Standard durchgesetzt. Dieses hat allerdings mehrere Eigenschaften, die es für das mobile Umfeld nicht qualifizieren. Zum einen ist Offline-Verarbeitung nicht möglich. Zum anderen ist es ein blockierendes Protokoll, was zur Folge hätte, dass alle Beteiligten für unbestimmte Zeit blockiert wären, wenn bei einem Knoten während des Commit-Vorgangs die Verbindung abbricht. Darüber hat das Protokoll einen großen Nachrichten-Overhead, was sich bei eingeschränkter Bandbreite negativ auswirkt.

Auch I (Isolation) muss überdacht werden. Für klassische Szenarien werden meist pessimistische Verfahren wie „Two-Phase-Locking“ angewendet. Da aber auch hier (ähnlich wie bei 2PC) Nachrichten ausgetauscht werden müssen und plötzliche Verbindungsabbrüche zu unbestimmt langen Locks führen können, sind diese nicht für den mobilen Einsatz geeignet. Hier empfiehlt [7] optimis-

tische Ansätze zu verwenden. Zur weiteren Erklärung von optimistischen und pessimistischen Synchronisationsverfahren siehe [2].

Um die Konsistenz zu sichern, aber verbindungsloses Arbeiten zu ermöglichen, wird meist auf Repliken zurückgegriffen, deren Protokoll aber 1-Kopien-Serialisierbarkeit (siehe Kapitel 2) implementiert. Protokolle, die dieses nicht oder nur teilweise tun, müssen mit Inkonsistenzen rechnen und können keine harten Garantien für Transaktionssicherheit geben. In der Forschung wurden zahlreiche Verfahren entwickelt, die mobile Transaktionen ermöglichen. Nachfolgend wird nun stellvertretend der Ansatz des Clustering vorgestellt.

Clustering Clustering ist ein konsistenzsicherndes Verfahren im mobilen Umfeld, das auf Replikation beruht und auf einem vollständig verteilten System arbeitet, d. h. Transaktionen können von einem mobilen oder einem ortsfesten Host initiiert werden. Das Verfahren unterstützt die Ausführungsmodelle 2 und 3 (siehe Abschnitt 2.2). Bei Clustering hat jeder Host lokal auf seinem Gerät eine Replik der Datenbank vorliegen, die jedoch keine volle Kopie sein muss, sondern auch nur einen Teil der Daten beinhalten kann.

Der Name Clustering rührt daher, dass die Datenbank in Cluster unterteilt wird. Ein Cluster wird jeweils von nahe beieinander gelegenen Daten gebildet und besteht aus einem oder mehreren Hosts, die untereinander konsistent sind. Um Cluster zu ermitteln, bietet sich das Kontextmerkmal des Ortes an, d. h. das Clustering basiert auf der physischen Örtlichkeit der Daten. Dabei bilden Daten auf dem gleichen Host, Nachbarn und stark verbundene (drahtgebundene oder eine hohe Bandbreite besitzende) Hosts zusammen ein Cluster. Der große Vorteil dieser Cluster ist, dass ihre Konfiguration dynamisch ist. So wird unter anderem die Bewegung der Hosts unterstützt. Wenn ein mobiler Host beispielsweise in eine neue WLAN-Zelle kommt, kann der Host vom alten Cluster in ein neues Cluster wechseln, indem es mit den bereits in dieser Zelle befindlichen Hosts ein Cluster bildet. [4] bietet jedoch noch weitere Alternativen, aufgrund dessen Cluster gebildet werden können. So können Cluster auch auf Basis semantisch verwandter Daten oder aufgrund von Benutzerprofilen gebildet werden. Bei letzterem wird das Profil genutzt, um zum Beispiel die am häufigsten verwendeten Daten des Benutzers zu clustern. Letztlich kann Clustering auch auf der Basis von Daten-Anforderungen von Anwendungen geschehen, da Anwendungen meist nur auf einem Teil der Gesamtdaten agieren.

Wie man sieht, gibt es vielfältige Möglichkeiten Cluster zu definieren und so ist es möglich, je nach Anwendung die dafür geeignetste Variante zu wählen. Erwähnenswert ist an dieser Stelle noch, dass mobile Hosts selbst zu einem Cluster werden, wenn keine Verbindung mehr besteht.

Jede (mobile) Anwendung kann bei diesem Verfahren ihre eigenen Konsistenzanforderungen spezifizieren. So ist es z.B. nicht immer notwendig auf den aktuellsten konsistenten Zustand zurückzugreifen. Will der Mitarbeiter aus der Einleitung eine Verkaufstrendanalyse von Produkt A der letzten 5 Jahre durchführen, ist es nebensächlich, ob für die beiden letzten Wochen bereits alle

endgültigen Verkaufszahlen eingetragen sind oder nur (mittlerweile inkonsistente) vage Vorabschätzungen.

Der Kern des Verfahrens steckt darin, zwei Arten von Transaktionen zu unterscheiden: *schwache* und *strikte Transaktionen*. Während strikte Transaktionen auf stark verbundenen Hosts ausgeführt werden und alle Hosts erreichbar sein müssen, werden schwache Transaktionen ausgeführt, wenn der Host nur schlecht verbunden ist oder keine Verbindung besitzt. Strikte Transaktionen hinterlassen einen global (d. h. auf allen Repliken/Clustern) konsistenten Zustand; schwache Transaktionen hingegen können bis zu einem gewissen Grad global inkonsistent sein. Sie müssen aber auf jeden Fall in dem Cluster, zu dem sie gehören, bzw. bei Clustern, die nur aus einem Host bestehen, lokal auf dem Host, konsistent sein. Es gibt mehrere Möglichkeiten den Grad der Inkonsistenz zu beschränken, unter anderem durch die Anzahl der lokalen Commits oder durch die Anzahl an Transaktionen, die auf inkonsistenten Daten arbeiten dürfen. Dieser Grad kann abhängig von der Verbindungsbandbreite variieren.

Um zwischen beiden Transaktionsarten zu unterscheiden, gibt es von jedem Datenobjekt zwei Kopien: eine schwache und eine strikte Version mit den Operationen schwaches bzw. striktes Lesen und Schreiben. Während schwache Transaktionen nur auf der schwachen Version arbeiten, wird bei Änderungen der strikten Version auch die schwache aktualisiert. Zudem sind schwache Schreiber nicht dauerhaft, solange sie nicht global commitet wurden. Schwache Transaktionen müssen deshalb zweimal commiten: Einmal lokal bzw. im Cluster und einmal global, wenn die Änderungen zum Server propagiert werden sollen. Um innerhalb eines Clusters konsistent zu bleiben, reicht es bei diesem Verfahren, dass die Historie des Clusters konfliktserialisierbar ist, während die Projektion auf strikte Transaktionen, wie in Kapitel 2 eingeführt, 1-Kopien-serialisierbar sein muss.

Wenn nun bei einer Wiederverbindung oder einer stärkeren Bandbreite Cluster zusammengefasst werden, bzw. lokale schwache Datenänderungen global commitet werden sollen, wird ein Abgleich gemacht. Dazu werden möglichst viele schwache Schreibvorgänge akzeptiert, solange die 1-Kopien-Serialisierbarkeit der strikten Transaktionen nicht verletzt ist. Probleme ergeben sich, wenn schwache Schreiber mit strikten Transaktionen in Konflikt stehen. Die zugehörigen schwachen Transaktionen müssen in diesem Fall zurückgesetzt werden, was in manchen Fällen kaskadierende Rücksetzungen anderer schwacher Transaktionen nach sich zieht. [4] argumentiert aber, dass dies für die meisten Anwendungen nicht von Relevanz ist, da schwache Transaktionen nur an ungefähren Daten interessiert sind. Letztendlich kann man sagen, dass es Clustering schafft, dem mobilen Umfeld gerecht zu werden. Clustering unterstützt dynamische Clusterkonfiguration und erlaubt es mobilen Hosts, zeitweise ohne Verbindung autonom zu arbeiten. Außerdem nimmt das Verfahren Rücksicht auf die Bandbreite und definiert so den maximalen Grad der Inkonsistenz. Allerdings ist es möglich, dass beim Abgleich mehrerer Cluster und/oder globaler Propagierung bereits lokal commitete Daten wieder zurückgesetzt werden müssen, was ein Nachteil ist.

Betrachten wir unseren Mitarbeiter aus Abschnitt 1 in Bezug auf dieses Verfahren: Da er beim Kunden offline ist, arbeitet er dort mit schwachen Transaktionen. Er kann die Daten (Bestellung und Rechtschreibkorrektur) eingeben, muss diese allerdings, wenn er zurück in der Firma ist, abgleichen. Im besten Fall geht alles gut, das andere Extrem wäre das alles zurückgesetzt werden muss. Ein Problem wird hier nicht gelöst: Wenn der Mitarbeiter die Bestellung aufgibt, sind in seiner lokalen Replik noch genügend Teile vorhanden. Leider sind diese Daten mittlerweile hochgradig inkonsistent, da alle Teile verkauft wurden. Da dem Mitarbeiter dies aus seinen inkonsistenten Daten aber nicht ersichtlich wird, verkauft er trotzdem. Eine Idee, die dieses Problem behebt, ist der nachfolgend vorgestellte Ansatz der Reservierung.

3.2 Reservierungsprotokolle

Um härtere Garantien für Datenänderungen zu geben, die auch beim Serverabgleich erhalten bleiben und damit konsistenter sind, bietet es sich an, die Daten auf denen gearbeiten werden soll, für sich zu reservieren. Ein Verfahren, das dies anbietet ist Mobisnap, welches nachfolgend vorgestellt wird.

Mobisnap erweitert mobile Transaktionen um Garantien, die Inkonsistenzen und Konflikte vermeiden sollen. Die zentrale Idee von Mobisnap ist, Garantien anzubieten, die es ermöglichen, den Ausgang einer Transaktion auch schon zu wissen, wenn der Host gerade nicht verbunden ist. Hierzu bietet Mobisnap vier Typen von Reservierungen an (vgl. [5]), die dies ermöglichen: Mit *Escrow* kann man eine Anzahl einer teilbaren Ressource reservieren. Hierunter fallen etwa Sitzplätze in einem Flugzeug. Bei *Slot* reserviert man sich das Recht einen Datensatz mit festvorgegebenen Werten hinzuzufügen. Beispielsweise könnte unser Mitarbeiter aus der Einleitung damit das Recht reservieren, sich an einem festgelegten Tag den Firmenwagen auszuleihen. *Value-Change* reserviert die Berechtigung gewisse Werte in der Datenbank zu ändern und *Value-Use* die Berechtigung Transaktionen auszuführen, die bestimmte Werte nutzen. Letzteres verbürgt zum Beispiel das Recht, dass der Mitarbeiter das Produkt zu dem bei seiner Replik gegebenen Preis verkaufen darf, auch wenn zwischenzeitlich der Preis zentral erhöht wurde. Beim Kommunizieren mit dem Server können die Anwendungen/Hosts ihre gewünschten Reservierungen anfordern.

Um Blockaden des gesamten Systems zu vermeiden, werden die Reservierungen zeitlich limitiert (geleast). Diese Blockaden könnten dadurch entstehen, dass mobile Hosts, die Reservierungen halten, permanent keine Verbindung mehr haben. Solange ein (mobiler) Host dem Server seine Änderungen auf reservierten Werten propagiert, bevor der Lease abgelaufen ist, ist somit garantiert, dass die Transaktion auf jeden Fall positiv abschließt, also commiten kann. Da es meistens keine optimale Lösung gibt, findet man auch bei diesem Ansatz einen Haken. Die Reservierung von bestimmten Werten/Tupeln durch einen Host zieht nach sich, dass andere Hosts nicht auf diesen Werten operieren dürfen, solange der reservierende Host nicht auf dem Server commitet hat oder seine Reservierung nicht abgelaufen ist.

Damit man überhaupt reservieren kann, muss dies von der Datenbank unterstützt werden. Dazu spezifiziert der Datenbank-Designer beim Entwurf ein zugehöriges Reservierungsskript. In diesem wird festgelegt, welche Daten reserviert und welche Reservierungstypen dafür benutzt werden können. Darüber hinaus wird definiert, wem welche Reservierung für wie lange gewährt wird. Bei Escrow-Reservierungen wird zudem ein Maximum an reservierbaren Einheiten angegeben.

Ein gut durchdachtes Reservierungsskript ist ausschlaggebend für die Leistungsfähigkeit des Gesamtsystems, denn die Effizienz ist an die passende Wahl der eben erwähnten Parameter gekoppelt. Beispielsweise muss eine Abstimmung bei der zeitlichen Dauer der Reservierung gefunden werden. Bei zu kurz gewählter Dauer benachteiligt man mobile Knoten, die eventuell aufgrund der Infrastruktur keine Chance hatten, ihre Commits zu propagieren und die so wieder optimistisch hoffen müssen, dass die Datenänderungen übernommen werden. Bei zu lang gewählter Dauer wiederum werden alle Hosts benachteiligt, die auf Daten arbeiten möchten, welche momentan aber reserviert sind. Wenn eine Anwendung auf nicht-reservierten Daten arbeitet (Vorraussetzung hierfür ist, dass kein anderer Host diese Daten reserviert hat) oder die Reservierung abgelaufen ist, bevor dem Server die Änderungen propagiert werden konnten, hat sie weiterhin keinerlei Garantie, dass die Änderungen vom Server akzeptiert werden und konfliktfrei sind; die Anwendung arbeitet also potenziell auf inkonsistenten Daten. Beim Arbeiten auf reservierten Daten (und rechtzeitiger Propagierung) werden global konsistente Daten garantiert.

Zur Laufzeit funktioniert das System folgendermaßen: Nachdem sich der Host Reservierungen zugesichert hat und gegebenenfalls Daten vom Server lokal repliziert hat, kann er auf diesen arbeiten. Dabei werden lokal Transaktionen gestartet, die die Daten bearbeiten. Will der Host die Änderungen propagieren, wird dieselbe Transaktion nochmals auf dem Server ausgeführt und das endgültige Ergebnis zurückgegeben. Dies ist der Vorgang, falls Host und Server miteinander kommunizieren können. Ist dies nicht möglich, da der Host keine Verbindung hat, wird die Transaktion zuerst lokal ausgeführt und hat dann einen der folgenden Zustände:

Reserved commit – die Transaktion konnte lokal commiten und wird dies aufgrund der Reservierungen auch global tun.

Tentative commit – die Transaktion konnte lokal commiten, da aber keine Reservierungen vorlagen, ist ungewiss, ob die Transaktion auch global commiten kann.

Unknown – konnte nicht lokal ausgeführt werden, da beispielsweise Datensätze, die benötigt werden, lokal nicht zu Verfügung stehen. Auch hier ist der globale Commit ungewiss.

Zusammenfassend lässt sich sagen, dass der Mobisnap-Ansatz den Vorteil bietet, dass man durch Einsatz von Reservierungen auch Aussagen über den Transaktionsausgang machen kann, wenn der (mobile) Host zeitweise offline war, wie im mobilen Umfeld anzutreffen. Als nachteilig ist allerdings zu bewerten, dass Hosts nicht auf Daten arbeiten können, wenn diese gerade reserviert sind. Es ob-

liegt dem Datenbank-Designer ein gutes Reservierungsskript zu entwerfen, das ein effizientes Arbeiten möglich macht. Leider wird auch keine Aussage darüber gemacht, wie momentan nicht verbundene Hosts erkennen, welche Reservierungen gerade aktuell sind.

Wenn wir nun den Mitarbeiter aus der Einleitung betrachten, kann man folgende Schlüsse ziehen: Unter der Vorraussetzung, dass er vor Verlassen der Firma beziehungsweise vor Verbindungstrennung die betreffenden Daten reserviert hat, kann man garantieren, dass die Änderungen erfolgreich und konsistent propagiert werden. Es kämen zum Beispiel folgende Reservierungstypen in Frage: Um sich beim Bestellen abzusichern, dass garantiert eine geforderte Menge noch vorhanden ist, könnte sich der Mitarbeiter via Escrow-Reservierung 100 oder mehr (aber auch nicht unbegrenzt viele, da andere Mitarbeiter auch Einheiten verkaufen möchten) Einheiten zusichern und per Value-Use seinen Preis garantieren. Da der Tippfehler (siehe Einleitung) im Vorraus nicht bekannt ist, wird dafür auch nichts reserviert und weiterhin optimistisch propagiert. Dies ist in diesem Falle völlig unproblematisch, da davon auszugehen ist, dass die Änderung durchgeht. Falls nicht, wurde der Fehler zwischenzeitlich bereits korrigiert und propagiert, was auch keine Inkonsistenz nach sich zieht. Beim Vormerken des Firmenwagens würde sich Slot anbieten. Hier reserviert der Mitarbeiter das Recht den Firmenwagen an einem bestimmten Tag auszuleihen. Falls während dem Zuteilungsschritt die Verbindung abbricht, bekommt er den Firmenwagen trotzdem zugesichert.

3.3 Client-zentrierte Konsistenz (Sitzungsgarantien)

Bisher haben wir uns Verfahren angesehen, bei denen die (mobilen) Hosts lokal eine Replik auf dem Gerät vorhalten und darauf arbeiten. Dies hat den Vorteil, dass auch offline gearbeitet werden kann. Da allerdings auch andere Replikationsformen existieren, die für den mobilen Kontext interessant sind, sollten auch diese dafür sorgen, dass die Konsistenz gesichert ist.

Ein solches Schema ist die *Read-Any/Write-Any-Replikation*. Bei dieser gibt es eine Anzahl von Servern, auf die die komplette Datenbank repliziert wurde. Die Besonderheit bei dieser Art von Replikation ist, dass man einen beliebigen Server für Lese- und Schreibvorgänge nutzen kann. Diese Eigenschaft macht das Schema für mobile Anwendungen interessant, da der mobile Host einen Server nach Eigenschaften wie Verfügbarkeit oder Zugriffskosten auswählen kann.

Aus diesem Zugriffsmodell ergibt sich ein hohes Maß an Inkonsistenz, da der gleiche Host bei jedem Zugriff einen anderen Server wählen kann (vgl. Abbildung 2). Man nennt die Form der (In-)Konsistenz bei diesem Schema auch eventuelle Konsistenz. Es ist also möglich, dass ein mobiler Host auf einer Replik Änderungen propagiert und beim nächsten Zugriff auf einer Replik agiert, die diese Änderungen noch nicht mitgeteilt bekommen hat. Für den Client erscheint es also so, als seien seine Änderungen nicht von der Datenbank übernommen worden.

Um dies zu verhindern, bedient man sich der Idee der Client-zentrierten Konsistenz, welche von [10] eingeführt wurde. Diese setzt sich aus den vier Mo-

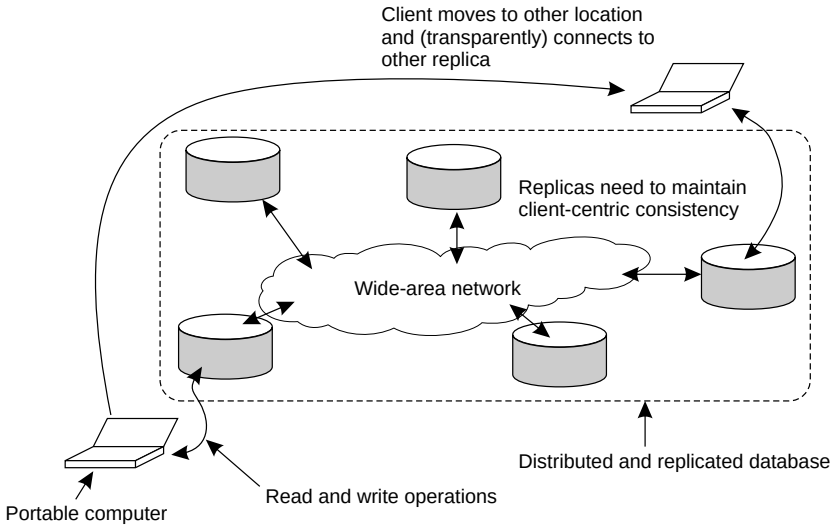

Abbildung 2. Das Read-Any/Write-Any-Replikationsschema und eine mögliche Bewegung des mobilen Hosts

dellen *Read-your-Writes*, *monotones Lesen*, *Writes-follow-Reads* und *monotones Schreiben* zusammen, deren Ideen nachfolgend erläutert werden.

Bei *Read-your-Writes* wird garantiert, dass die Wirkung einer Schreiboperation eines Clients von nachfolgenden Leseoperationen desselben Clients gesehen wird. Um dies zu ermöglichen, sind Leser nur auf Repliken erlaubt, die alle vorhergehenden Schreibvorgänge enthalten. Ein Beispiel für die Erfordernis von Read-your-Writes ist ein Login-System. Wenn ein Benutzer sein Passwort ändert, könnte es ohne die Garantie passieren, dass er sich nach der Änderung wegen einem angeblich falschen Passwort nicht einloggen kann. Dies ist dadurch zu erklären, dass für den Passwortabgleich eine Replik verwendet wird, die noch das alte Passwort enthält. Mit Read-your-Writes tritt dieses Problem nicht mehr auf.

Monotones Lesen ermöglicht, dass der Wert eines Datenelements, das von einem Client gelesen wurde, beim nächsten Lesen denselben oder einen aktuelleren Wert zurückgibt. Dafür dürfen die Leser nur auf Repliken ausgeführt werden, die alle Schreibeffekte enthalten, die schon von vorhergehenden Lesern gesehen wurden. Ein Bespiel dafür wäre ein Datenbankgestützter Terminkalender, der dem Benutzer seine Termine anzeigt. Ohne monotones Lesen könnte es passieren, dass kürzlich gelöschte Termine auf einmal wieder auftauchen, obwohl sie vorher nicht mehr im Kalender enthalten waren.

Um Aktualisierungen als Resultat vorhergehender Leseoperationen weiterzugeben, wird *Writes-follow-Reads* benötigt. Hier wird dem Client zugesichert,

dass eine Schreiboperation auf einem Datenelement, die einem vorhergehenden Leser folgt, garantiert auf demselben gelesenen oder einem aktuelleren Wert stattfindet. Damit wird eine Ordnung der Schreiber impliziert, da diese dadurch aufeinander aufbauen. Beispielsweise kann dies bei einem Newsgroup-System genutzt werden. Mit Writes-follow-Reads wird dabei garantiert, dass eine Antwort auf einen Artikel nur sichtbar ist, wenn dies auch der ursprüngliche Artikel ist. Eine Replik darf also nur die Antwort speichern, wenn dort zuvor der Originalartikel gespeichert wurde. Ohne diese Garantie könnte es sein, dass die Antwort sichtbar ist, ohne dass man sieht, worauf sie antwortet.

Das letzte Modell ist *monotones Schreiben*. Dieses gewährleistet, dass Schreiber den vorausgegangenen Schreibern folgen und deren Wirkungen reflektieren. Damit wird impliziert, dass Schreiber auf allen Repliken in der richtigen Reihenfolge weitergegeben werden.

Der große Vorteil von Client-zentrierter Konsistenz ist, dass jede Anwendung ihre Konsistenzbedürfnisse definieren kann (alle vier oder eine beliebige Teilmenge daraus) und diese dann während einer Sitzung, die die Abfolge von Lese- und Schreibeoperationen während der Programmausführung darstellt, sichergestellt werden. Bei jedem Client agiert im Hintergrund ein so genannter Sitzungsmanager, dessen Aufgabe es ist, die Konsistenzkriterien einzuhalten, d. h. Repliken auszuwählen, die alle Erfordernisse erfüllen. Auch kann er eine Replik anweisen, sich zu aktualisieren, um Änderungen zu reflektieren und so auch fähig zu sein, den benötigten Konsistenzgrad zu bieten.

Wie ersichtlich wird, kann mit Hilfe von Client-zentrierter Konsistenz bzw. Sitzungsgarantien sichergestellt werden, dass auch in Read-Any/Write-Any--Replikationsschemata die Konsistenz gewahrt bleibt.

Vergleicht man den Ansatz mit den von Gray definierten und in den SQL-Standard übernommenen Konsistenzstufen (vgl. [2]), ist erkennbar, dass die Konsistenzstufe Serializable erfüllt ist, solange eine Sitzung alle vier Modelle erfordert, da dadurch keine Anomalien auftreten können. Werden nicht alle vier Modelle benutzt, ist dies mit einer niedrigeren Konsistenzstufe vergleichbar, da beispielsweise ohne monotones Lesen Phantomprobleme möglich sind.

3.4 Gewichtetes Votieren (Quorum consensus)

Beim vorhergehenden Verfahren wurde ein Modell präsentiert, das Konsistenz in einem Read-Any/Write-Any-Replikationsschema zusichert. Wie bereits erwähnt, gibt es mehrere Replikationsarten, die für mobile Anwendungen interessant sind.

Ein Weiteres ist das Read-Some/Write-Some-Replikationsschema. Bei diesem ist zwar, wie schon bei Read-Any/Write-Any, keine Offlinearbeit möglich, allerdings müssen die Änderungen nicht allen Replikaten mitgeteilt werden, sondern nur einem Teil. Hierbei können die Repliken auf den (mobilen) Hosts gespeichert werden und so kann die Idee eines Peer-to-Peer-Netzwerks umgesetzt werden. Der Fakt, dass Änderungen nicht allen Repliken mitgeteilt werden müssen, unterstreicht die Einsatzfähigkeit im mobilen Umfeld, da mobile Hosts, wie bereits mehrfach geschildert, nicht immer erreichbar sind.

Um nun auch hier die Konsistenz zwischen Repliken zu sichern, wurde das Verfahren des gewichteten Votierens eingeführt. Die Idee dahinter ist, von mehreren Hosts Berechtigungen einzuholen, bevor ein repliziertes Datum gelesen oder geschrieben wird. Dadurch ist sichergestellt, dass immer die aktuellsten Daten verarbeitet werden und somit keine Inkonsistenzen entstehen können. Um dies zu erreichen, wird mit Schreib- und Lese-Quorum gearbeitet, welches bei einem Zugriff erfüllt sein muss. Bei N Repliken ist ein Lese-Quorum eine beliebige Menge aus N_R oder mehr Hosts, analog dazu ein Schreib-Quorum mit N_W oder mehr Hosts. Es existiert außerdem eine Versionsnummer, die beim Schreiben erhöht wird und beim Lesen mit zurückgegeben wird. Zusätzlich müssen folgende Bedingungen zutreffen, um ein gültiges Quorum zu haben (vgl. [9]):

(1) $N_R + N_W > N$

(2) $N_W > \frac{N}{2}$

Die erste Bedingung garantiert, dass Lese/Schreib-Konflikte verhindert werden und beim Lesen immer mindestens eine aktuelle Version dabei ist. Die zweite Bedingung verhindert Schreib/Schreib-Konflikte und sichert zu, dass nur bei Mehrheit geschrieben wird.

Will man also eine Änderung propagieren, benötigt man dazu ein Schreib-Quorum, um dies ausführen zu dürfen. Analog dazu braucht man bei Lesern zuerst ein Lese-Quorum, welches dem Initiator eventuell verschiedene Versionen liefert, da mehrere Repliken gelesen wurden. In diesem Fall nimmt man die aktuellste davon, die, durch Bedingung 2 abgesichert, auch global die aktuellste Version ist.

Ein Vorteil dieses Verfahrens ist, dass die Größen für Lese- und Schreibquorum (N_R und N_W) beliebig gewählt werden können, solange die beiden oben genannten Bedingungen eingehalten werden. Nachteilig ist allerdings der große Kommunikationsaufwand, da N_R bzw. N_W viele Repliken adressiert werden müssen. Außerdem braucht man für jeden Lesevorgang mehrere Leser auf verschiedenen Repliken. Letztlich müssen auch alle Hosts bzw. ihre Adressierungsmöglichkeiten bekannt sein.

Um das Verfahren in Aktion zu sehen, betrachten wir das Beispiel aus Abbildung 3. Bei (a) bestand das letzte Schreib-Quorum aus den Hosts C bis L. All diese Repliken enthalten die neueste Version und folglich auch die neueste Versionsnummer. Beim nächsten Lese-Quorum, das sich in diesem Falle auf die Repliken A bis C stützt, ist wegen Bedingung 2 auf jeden Fall die aktuellste Version dabei. Durch Vergleich der Versionsnummern kann diese ermittelt werden. Beispiel (b) veranschaulicht das Scheitern des Verfahrens, wenn die Bedingungen (in diesem Fall Bedingung 2) nicht eingehalten werden. Hierbei tritt ein Schreib/Schreib-Konflikt auf, da ein Host {A, B, C, E, F, G} als Schreib-Quorum hat und seine Änderungen propagiert, ein anderer {D, H, I, J, K, L}. Beide Aktualisierungen werden akzeptiert, allerdings tritt beim nachfolgenden Lese-Quorum ein Konflikt auf.

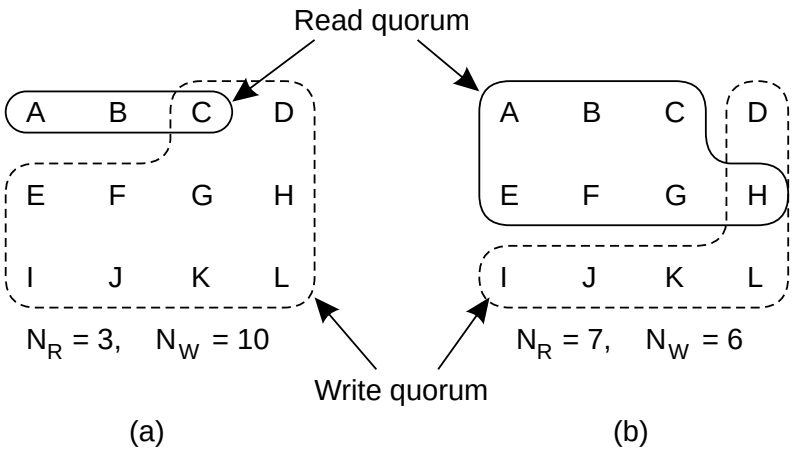

Abbildung 3. Beispiele für eine gültige und eine ungültige Quorum-Konstellation

Auch dieses Verfahren ist mit der Konsistenzstufe Serializable vergleichbar, solange die beiden Bedingungen für eine Quorum-Konfiguration eingehalten werden. Ist (wie in vorhergehendem Beispiel) eine Regel verletzt, ist auch Serializable nicht mehr gegeben, da Phantome (zwei Quoren mit der gleichen Versionsnummer geben verschiedene Werte zurück) auftreten können.

3.5 Operational Transformation

Bislang haben wir uns Verfahren angeschaut, deren Anwendungsdomäne auf Applikationen zutrifft, die ihre Daten in Datenbanken (oder deren Repliken) speichern. Um zu zeigen, dass Konsistenzprobleme allerdings vielfältiger sind und auch in anderen Anwendungsdomänen auftreten (können), wird nachfolgend die Idee der „Operational Transformation“ vorgestellt, die in mehreren Verfahren Anwendung findet.

Die Verfahren wurden für Groupware-Systeme, genauer Echtzeit-Gruppen-Editoren, entwickelt. Mittels dieser Editoren soll es einer Gruppe von Benutzern ermöglicht werden, verteilt zur gleichen Zeit am gleichen Dokument zu arbeiten. Dabei hat jeder Benutzer lokal eine Kopie des Dokumentes vorliegen, auf der er arbeitet. Die vom Benutzer vorgenommenen Operationen werden in seinem lokalen Dokument direkt ausgeführt und zu einem nicht näher spezifizierten Zeitpunkt an die anderen Dokumentrepliken propagiert. Dieser Fakt macht das Verfahren auch für den mobilen Kontext interessant, da implizit Offline-Arbeiten möglich sind und die Änderungen nicht synchron propagiert werden müssen. Eine Besonderheit bei diesen Verfahren ist, dass nicht, wie bei den bisher vorgestellten Verfahren, der Änderungszustand (also der Zustand nach Datenänderung) propagiert wird, sondern der Übergang, d. h. die Operation, die das Dokument

von einem in den nächsten Zustand überführt. Als Beispiel soll hier Insert[z,2] dienen, welches eine Operation ist, die im Dokument das Zeichen „z“ an Position 2 einfügt.

Die Grundidee von Operational Transformation ist, die ankommenden Operationen so zu transformieren, dass zwischen den Dokumentrepliken Konsistenz herrscht, auch wenn Operationen in unterschiedlicher Reihenfolge ankamen. So könnte bspw. aus Insert[z, 2] Insert[z, 3] werden, weil in dieser Replik zuvor schon ein Zeichen an Stelle 2 eingefügt wurde.

Da für den ersten Ansatz dieser Art namens GROVE [8] (Group Outline Viewing Editor) nachgewiesen werden könnte, dass Fälle existieren, in denen die Konsistenz nicht sichergestellt ist, wird nachfolgend das Verfahren REDUCE [8] vorgestellt, das auf GROVE aufbaut, dieses aber verfeinert und das Problem behebt.

Um zu garantieren, dass die verschiedenen Dokumente konsistent zueinander sind, muss zuerst definiert werden, was Konsistenz in diesem Zusammenhang bedeutet. Hierzu werden die Korrektheitskriterien für Konsistenz für das REDUCE-Verfahren zu einem Konsistenzmodell zusammengefasst, welches drei Merkmale hat:

Erstens muss Konvergenz herrschen, was bedeutet, dass Kopien des Dokumentes identisch sind, wenn die gleichen Operationen darauf ausgeführt wurden.

Darüber hinaus gilt Kausalitätserhaltung, d. h. für jedes Paar (O_a, O_b) wird O_a auf allen Kopien vor O_b ausgeführt, falls $O_a \rightarrow O_b$ gilt. Das Zeichen $\rightarrow$ ist eine Kausalordnungsrelation und besagt folgendes: O_b ist kausal abhängig von O_a ($O_a \rightarrow O_b$) wenn eine der drei Eigenschaften zutrifft:

1. O_a und O_b wurden auf der gleichen Kopie erzeugt, aber O_a wurde vor O_b generiert.
2. O_a und O_b wurden auf verschiedenen Kopien (i und j) erzeugt, aber die Ausführung von O_a auf Kopie j erfolgte vor der Erzeugung von O_b.
3. Es gibt eine Operation O_x und es gilt: $O_a \rightarrow O_x$; $O_x \rightarrow O_b$

Wenn zwei Operationen kausal voneinander abhängen, werden sie abhängig genannt, ansonsten unabhängig.

Als drittes Konsistenzkriterium wurde bei REDUCE das Kriterium der Absichtserhaltung eingeführt, welches bei GROVE nicht enthalten war und es deshalb zu Ausnahmen bei der Konsistenz kam. Absichtserhaltung besagt, dass für eine beliebige Operation O die Effekte der Ausführung auf allen Kopien die gleichen sind wie die Absicht von O. Weiterhin sollen die Effekte von unabhängigen Operationen nicht durch die Ausführung von O geändert werden. Um die Notwendigkeit der Absichtserhaltung zu veranschaulichen, soll folgendes Beispiel helfen.

Angenommen das replizierte Dokument beinhaltet anfangs die Zeichenfolge „UVWXYZ“. Betrachten wir weiterhin zwei Operationen: O_1 = Insert[ABC, 3] hat die Absicht „ABC“ an Stelle 3 (also zwischen V und W) einzufügen. O_2 = Delete[4, 3] möchte ab Stelle 4 drei Zeichen löschen, also „XYZ“. Wenn man beide Absichten in Betracht zieht, müsste das Ergebnis nach der Ausführung

„UVABCW“ lauten. Ohne Absichterhaltung würde es (die Reihenfolge O_1 O_2 vorausgesetzt) aber „UVAXYZ“ lauten. Selbst wenn auf allen Kopien diese Reihenfolge eingehalten wird (und die Kopien dadurch zueinander konsistent sind), ist das Ergebnis trotzdem inkonsistent zu den Absichten der Operationen.

Um zu wissen, welche Operationen ausgeführt wurden, werden diese im so genannten History Buffer (HB) gespeichert, der alle Operationen in der Reihenfolge der Ausführung enthält. Die Kausalität wird erhalten, indem alle Operationen mit Timestamps versehen werden.

Konvergenz und Absichtserhaltung werden durch zwei Transformationsschemata eingehalten. Ein undo/do/redo-Schema hält die Konvergenz aufrecht. Dies ist aber nur eine implizite Transformation, da sie sich auf den Dokumentenzustand bezieht und nicht auf die Operation selbst. Das undo/do/redo-Schema setzt eine totale Ordnung zwischen den Operationen voraus; die Operationen selbst dürfen aber in einer beliebigen Reihenfolge ausgeführt werden, solange die Kausalität eingehalten wird. Wenn also eine Operation O ausgeführt werden soll, werden zuerst alle Operation im HB rückgangig gemacht (undo), die in der totalen Ordnung O nachfolgen, um den Zustand des Dokuments vor deren Ausführung zu erhalten. Danach wird O ausgeführt (do) und anschließend werden wieder alle vorher rückgängig gemachten Operation erneut ausgeführt (redo). [8] rät, das undo/do/redo-Schema so zu implementieren, dass nur das Endergebnis, nicht jedoch die Zwischenschritte, für den Benutzer sichtbar sein sollte, da es sich um interne Operationen handelt.

Die Transformation im eigentlichen Sinne, nämlich die Operational Transformation, wird benötigt, um Absichtserhaltung zu gewährleisten. Es konnte gezeigt werden, dass die Absicht erhalten werden kann, wenn sichergestellt ist, dass der Definitionskontext einer Operation O gleich dem Ausführungskontext (DC(O) = EC(O)) ist. Kontext bedeutet hier die Abfolge von Operationen, die ausgeführt wurden, um zum gegenwärtigen Dokumentenzustand zu gelangen. Der Definitionskontext ist der Kontext des Dokumentenzustands als O generiert wurde; der Ausführungskontext ist der Kontext des Dokumentenzustands auf dem O ausgeführt werden soll.

Im REDUCE-Verfahren sorgt ein Algorithmus namens GOT (Generic Operational Transformation) dafür, jede Operation (unter Zuhilfenahme des Ausführungskontextes) so zu transformieren, dass ihre Absicht erhalten bleibt, also DC(O) = EC(O) gilt. Der GOT-Algorithmus bedient sich dabei intern zweier Transformationsfunktionen: Der Inklusions- und der Exklusionstransformation.

Betrachten wir folgendes Szenario: Operation O_b wurde bereits ausgeführt, Operation O_a soll ausgeführt werden. Die Inklusionstransformation transformiert Operation O_a so, dass die Auswirkung von O_b mit einbezogen wird. Die Exklusionstransformation bewirkt das Gegenteil; sie transformiert O_a so, dass die Auswirkungen nicht eingezogen werden.

Während der GOT-Algorithmus (wie der Name schon andeutet) generisch, also Anwendungsunabhängig, ist, muss für die Transformationsfunktionen bei jeder Anwendung festgelegt werden, wie diese arbeiten sollen.

Um nun alle Aspekte des Konsistenzmodells zu vereinen, sollte man das undo/do/redo-Schema, welches die Konvergenz sichert, mit dem GOT-Algorithmus integrieren, um ein undo/transform-do/transform-redo-Schema zu erhalten, welches die Konsistenz sichert.

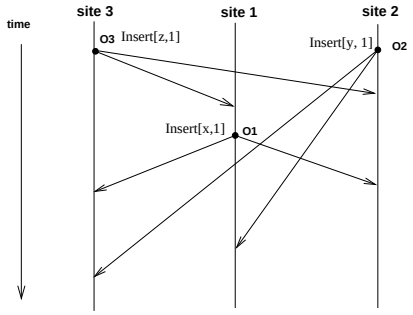

Abbildung 4. Ein Beispiel, bei dem Operational Transformation benötigt wird

Um zu zeigen, wie der REDUCE-Ansatz funktioniert, betrachten wir folgendes Beispiel, welches in Abbildung 4 dargestellt ist. Die Punkte bedeuten die Generierung und lokale Ausführung der Operation, die Pfeile die Propagierung und Ausführung auf den anderen Dokumentkopien. Des Weiteren wird die totale Ordnung in der Reihenfolge O_3 O_1 O_2 angenommen. Die Inklusionstransformation wird hier als Verschieberegel implementiert, d. h. wenn O_b und O_a an der selben Stelle eingefügt werden sollen, wird die Einfügeposition von O_a verschoben. Auf Kopie 3 wird zuerst „z“ eingefügt. Anschließend wird Operation 1 ausgeführt, welche „x“ an Position 1 einfügt (da O_1 abhängig von O_3 ist) und somit die Zeichenfolge „xz“ ergibt. Kommt nun O_2 an, muss diese transformiert werden, da sie unabhängig von O_3 und O_1 ist. Es wird O_3 inkludiert, was in $O_2' = Insert[y, 2]$ resultiert. Anschließend wird O_1 inkludiert und es ergibt sich $O_2'' = Insert[y, 3]$ und als Endresultat „xzy“. Analog funktioniert der Vorgang bei Kopie 1. Bei Kopie 2 wird zuerst wegen O_2 zuerst „y“ eingefügt. Wegen der Reihenfolge der totalen Ordnung muss O_2 zurückgesetzt werden, wenn Operation O_3 ausgeführt werden soll. O_3 wird ausgeführt und deren Effekt in O_2 inkludiert, welche dann als $O_2' = Insert[y, 2]$ wieder ausgeführt wird. Wenn O_1 ausgeführt werden soll, wird O_2' wieder wegen der totalen Ordnung zurückgesetzt. O_3 bleibt erhalten, da sie in der totalen Ordnung vor O_1 steht und O_1 abhängig von O_3 ist. Nach der Ausführung von O_1 wird deren Ergebnis wieder in O_2' inkludiert und es ergibt sich $O_2'' = Insert[y, 3]$. Nach der Ausführung aller Operationen haben alle Kopien den Inhalt „xzy“, d. h. sie sind zueinander konsistent.

Wie man in den vorangegangenen Ausführungen sieht, ist REDUCE ein gutes Verfahren, dass in der Domäne der Gruppen-Editoren Konsistenz sichern kann. In [8] wird das Verfahren trotzdem kritisch hinterfragt und es wird erkennbar, dass durchaus noch Verbesserungspotenzial besteht. Es wird empfohlen, einen

größeren Fokus auf semantische Konsistenz zu legen, was zwar in REDUCE mit dem Kriterium der Absicht erstmals beachtet wurde, was aber trotzdem noch steigerungsfähig ist. Weiterhin wird kritisiert, dass bisherige Ansätze nur einfache Operationen wie Einfügen und Löschen zulassen, nicht aber komplexere Operationen wie beispielsweise Verschieben oder Ersetzen. Es wird also auch in Zukunft noch Aspekte geben, die gründlich erforscht werden müssen.

4 Fazit

Wie man durch die vorhergehenden Ausführungen gesehen hat, gibt es vielfältige Ansätze, um im mobilen Umfeld die Konsistenz zu sichern. Dabei obliegt es meist der Anwendung, zu bestimmen, wie konsistent die Daten sein müssen, auf denen sie arbeitet, beziehungsweise welcher Grad der Inkonsistenz zugelassen wird. Weiterhin ist die Wahl des Konsistenzsicherungsverfahrens meist davon abhängig, welches System dahinter steht oder welches Schema bei Replikation gewählt wurde. Die Verfahren selbst gehen dabei über verschiedene Ansätze den Fall der mobilen Nutzung an und unterstützen ihn mehr oder weniger gut. Beispielsweise wird nicht von allen Verfahren die Möglichkeit geboten, auch ohne Verbindung weiterzuarbeiten. Andere unterstützen dafür zum Beispiel besser die Eigenschaft der Mobilität. Beim Entwerfen einer mobilen Anwendung sollte man also genau abwägen, welche Art der Konsistenz eingehalten werden soll und welche Eigenschaften mobiler Hosts abgedeckt werden sollen. Ein optimales Verfahren gibt es mit Sicherheit nicht. Jedes Verfahren erkauft sich meist bestimmte Vorteile mit dadurch entstehenden Nachteilen. Die Forschung auf diesem Gebiet hat das Potenzial bei Weitem noch nicht ausgeschöpft. Gerade bei Ideen wie Operational Transformation gibt es noch genügend Teilaspekte, bei denen sich eine weitergehende gründliche Erforschung lohnt.

Literatur

1. Bernstein, P. A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.
2. Härder, T., and Rahm, E. *Datenbanksysteme: Konzepte und Techniken der Implementierung.* Springer, 1999.
3. Liu, M. L., Agrawal, D., and Abbadi, A. E. On the Implementation of the Quorum Concensus Protocol. In *PDCS* (1995).
4. Pitoura, E., and Bhargava, B. K. Maintaining Consistency of Data in Mobile Distributed Environments. In *ICDCS* (1995), pp. 404–413.
5. Preguiça, N. M., Baquero, C., Moura, F., Martins, J. L., Oliveira, R. C., Domingos, H. J. L., Pereira, J. O., and Duarte, S. Mobile Transaction Management in Mobisnap. In *ADBIS-DASFAA* (2000), pp. 379–386.
6. Rahm, E. *Mehrrechner-Datenbanksysteme - Grundlagen der verteilten und parallelen Datenbankverarbeitung.* Addison-Wesley, 1994.
7. Serrano-Alvarado, P., Roncancio, C., and Adiba, M. E. A Survey of Mobile Transactions. *Distributed and Parallel Databases 16*, 2 (2004), 193–230.

8. Sun, C., and Ellis, C. A. Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements. In *CSCW* (1998), pp. 59–68.
9. Tanenbaum, A. S., and Steen, M. V. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, 2001.
10. Terry, D. B., Demers, A. J., Petersen, K., Spreitzer, M., Theimer, M., and Welch, B. B. Session Guarantees for Weakly Consistent Replicated Data. In *PDIS* (1994), pp. 140–149.