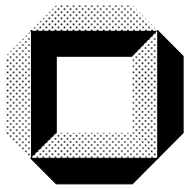
Daten in verteilten Systemen

Ludwig Keller, Dietmar Kottmann

Interner Bericht 30/95



Universität Karlsruhe

Fakultät für Informatik

Vorwort

Heutige verteilte Systeme sind geprägt durch die globale Vernetzung von Rechnersystemen und eine starke Dezentralisierung ihrer Anwendung. Sie erfordern zahlreiche Mechanismen, die die Kommunikation zwischen den kooperierenden Anwendungskomponenten betreffen sowie die Methodiken, um sich in dem verteilten System zurechtzufinden und dessen Dienste effektiv zu nutzen.

Während die Bedeutung von verteilten Systemen heutzutage unumstritten ist, fehlen nach wie vor fundierte Techniken, um die mit einem solchen System einherfallenden Probleme zu lösen. So wurden beispielsweise bislang der Einsatz von Datenbanktechniken in verteilten Systemen nur ungenügend untersucht. Auch ändert sich fluktuativ die Entwicklungsumgebung verteilter Systeme mit dem Fortschritt ihrer Hardware–Technologie. So finden in den letzten Jahren zunehmend mobile Rechnersysteme praktischen Einsatz, deren Berücksichtigung in einem verteilten System neue, innovative Methodiken erfordern.

Mit diesem neuen Aspekten des Entwurfs und der Handhabung verteilter Systeme beschäftigte sich das Seminar „Daten in verteilten Systemen“, welches im Wintersemester 1994/95 an der Universität Karlsruhe vom Institut für Telematik durchgeführt wurde. Es bietet den Studierenden der Fachrichtung Informatik einen Zugang zu dem aktuellen und brisanten Thema, der in einem solchem Umfang und Detaillierungsgrad in den normalen Lehrveranstaltungen nur sehr begrenzt behandelt werden kann.

Das Seminar ist Bestandteil einer Veranstaltungsreihe des Instituts, dessen erstes Seminar im Sommersemester 1992 unter dem Thema „Mechanismen für fehlertolerante verteilte Anwendungen“ stattfand. Während die bisherigen Seminare ihren Schwerpunkt auf Fehlertoleranzaspekten hatte, beschäftigt sich das zuletzt stattfindende Seminar mit allgemeineren Problemstellungen verteilter Systeme. Im einzelnen wurden die folgende Themen behandelt.

Die ersten drei Themen beschäftigen sich mit Grundtechniken verteilter Systeme. Kapitel 1 befaßt sich mit dem Consensus–Problem, d.h. wie mehrere Aktivitätsträger (Prozessoren, Anwedungskomponenten) sich über eine einheitliche Entscheidung einigen können, auch wenn einige von ihnen fehlerhaft arbeiten. Kapitel 2 untersucht Workflow–Ansätze, die zur Steuerung von Abläufen, wie z.B. in der Büroautomatisierung, immer stärker an Bedeutung gewinnen. Kapitel 3 untersucht verteilte Systeme hinsichtlich Echtzeitanforderungen.

Gegenstand der nächsten beiden Themen ist das Mobile Computing. Eine wichtige Voraussetzung für mobile Rechnersysteme ist die Replikation ihrer Datenbestände, die z.T. ein entkoppeltes Arbeiten ermöglichen. In Kapitel 4 werden zunächst der Konsistenzbegriff in verteilten Systemen und anschließend Replikationskontrolltechniken mit schwachen Konsistenzgarantien in verschiedenen existierenden Systemen untersucht. In Kapitel 5 werden sodann die Rahmenbedingungen des Mobile Computing untersucht und anhand des Dateisystems Coda dokumentiert.

Ein dritter Teil beschäftigt sich mit dem Einsatz von Fehlertoleranzmechanismen in verteilten Systemen. In Kapitel 6 werden hierzu im Gegensatz zu Kapitel 4 Replikationskontrolltechniken mit strengen Konsistenzgarantien behandelt und hierbei insbesondere auf die unterschiedliche Behandlung von Datenreplikation und Prozeßreplikation eingegangen. Kapitel 7 behandelt sodann die fehlertolerante Auslegung des UNIX–Betriebssystems ft(UNIX).

In den beiden letzten Themen werden schließlich zwei objektorientierte Systemansätze behandelt. Zunächst beschäftigt sich Kapitel 8 mit objektorientierten Datenbanken; es wird darin gezeigt, wie verteilte und replizierte Daten einer Anwendung transparent bereitgestellt werden kann und welche Probleme hierbei auftreten. Kapitel 9 behandelt danach die Common Object Request Broker Architektur (CORBA), die derzeit zu einem Quasi–Standard verteilter Systeme wird und ihre zukünftige Entwicklung maßgebend beeinflussen wird.

Bevor nun die einzelnen Ausarbeitungen der Seminarbeiträge präsentiert werden, möchten wir allen beteiligten Studierenden für ihre engagierte Mitarbeit danken, ohne die weder der Erfolg des Seminars noch die Qualität des vorliegenden internen Berichts möglich gewesen wäre. Hierzu haben auch die nach den einzelnen Vorträgen stattfindenden z.T. langanhaltenden Diskussionen maßgeblich beigetragen.

Karlsruhe, im Mai 1995

Ludwig Keller Dietmar Kottmann

Inhaltsverzeichnis

Abbildung 1: Verteiltes System

dung. Um diese einheitliche Entscheidung zu erlangen, versuchen die fehlerfreien Prozessoren das sogenannte Consensus-Problem zu lösen. Die Frage der Lösbarkeit und der Komplexität der entsprechenden Protokolle hängt von verschiedenen Parametern ab. Dazu gehören die Topologie des Netzwerkes, die Art und die Anzahl der auftretenden Fehler und der Grad der zeitlichen Asynchronität von Prozessoren und des Kommunikationsflusses.

1.3 Das Consensus-Protokoll

Innerhalb des Consensus-Problems versuchen die Prozessoren zu einer einheitlichen Entscheidung zu kommen. Dabei müssen alle beteiligten Teilnehmer einer Entscheidung, die auf ihren Anfangswerten basiert, zustimmen. Hierbei sind nur zwei Entscheidungswerte erlaubt: '0' und '1'. Haben alle Prozesssoren den Anfangswert 0 (1), so muß die Entscheidung am Ende 0 (1) lauten . Haben einige Prozessoren den Anfanswert 0, und einige den Anfangswert 1, so kann die Entscheidung entweder 0 oder 1 lauten. Die einzelnen Entscheidungswerte repräsentieren verschiedene Aktionen, z.B. kann die '1' einen 'Commit' und die '0' einen 'Abort' in einer verteilten Datenbank bedeuten (siehe Kapitel 1.9).

Formal heißt ein Consensusprotokoll korrekt, wenn folgende Bedingungen erfüllt sind :

1) Consistency: alle Teilnehmer einigen sich auf dengleichen Wert und alle Entscheidungen sind endgültig.

2) Validity: der Wert, auf den man sich geeinigt hat, muß der Anfangswert einiger Teilnehmer gewesen sein.

3) Termination: Jeder Teilnehmer erzielt innerhalb einer begrenzten Zeit einen Entscheidungswert.

Die oben erwähnten Bedingungen gelten im fehlerfreien Fall für alle Teilnehmer. Im Fall, daß Fehler auftreten, gelten sie nur für die fehlerfreien Teilnehmer [2].

1.4 Kommunikations- und Nachrichteneigenschaften

Zuerst wollen wir die verschiedenen Eigenschaften, die ein verteiltes System haben kann vorstellen [2]:

(1) Synchron/asynchrone Prozessoren: Prozessoren heißen synchron, wenn sich ihre Arbeitsweise durch eine Folge von Zeitintervallen, Runden genannt, beschreiben läßt. Zu Anfang jeder Runde kann jeder Prozessor an jeden Nachbarn eine (beliebig lange) Nachricht versenden. Diese Nachricht wird im Laufe der Runde empfangen und verarbeitet, bevor die nächste Runde beginnt. Im Gegensatz hierzu gibt es bei asynchronen Prozessoren keine Schranken für die Zeitabstände, in denen ein einzelner Prozessor einen Schritt ausführt.

(2) Die Kommunikationsverzögerung (communication delay) kann **'begrenzt'** oder **'unbegrenzt'** sein: Falls es eine obere Zeitschranke für die Übertragung von Nachrichten gibt, heißt die Kommunikation begrenzt (bounded), andernfalls unbegrenzt (unbounded).

(3) Die Nachrichtenauslieferung kann **'geordnet'** oder **'ungeordnet'** sein: Die Nachrichtenauslieferung heißt geordnet, wenn die Reihenfolge der Auslieferung von dem Sendezeitpunkt abhängt; andernfalls ungeordnet. Formal ausgedrückt, heißt eine Nachrichtenauslieferung genau dann geordnet, wenn ein Prozessor P1 eine Nachricht m1 zum Zeitpunkt t1 an einen Prozessor Px sendet und ein Prozessor P2 eine Nachricht m2 zum Zeitpunkt t2 an Prozessor Px sendet, und es gilt t1<t2, dann erhält Prozessor Px die Nachricht m1 vor der Nachricht m2.

(4) Die Übertragungsart kann entweder **'point-to-point'** oder **'broadcast'** sein: Man spricht von point-to-point-Übertragung , wenn ein Prozessor in einem atomaren Schritt ein Nachricht höchstens zu einem anderen Prozessor schicken kann. Im Gegensatz hiezu spricht man von Broadcast-Übertragung , wenn ein Prozessor in einem atomaren Schritt eine Nachricht an alle anderen verschicken kann.

(5) Ein Netzwerk kann **'reliable'** oder **'unreliable'** sein: Man spricht von 'reliable', wenn Nachrichten, die irgendwann einmal abgeschickt wurden, auch irgendwann einmal den Empfänger erreichen, d.h. nicht verloren gehen. Andernfalls heißt das Netz 'unreliable' (Im folgenden nehmen wir immer an, daß das zu betrachtende Netzwerk immer reliable ist).

1.5 Bedingungen, unter denen Consensus überhaupt möglich ist

Es existieren 3 Fälle, unter denen Consensus überhaupt möglich ist (Abb. 2).

1. Fall: Die Prozessoren arbeiten synchron und die Kommunikationsverzögerung ist begrenzt.

2. Fall: Die Nachrichtenreihenfolge ist geordnet und die Übertragungsart ist 'broadcast'.

3. Fall: Die Prozessoren arbeiten synchron und die Nachrichtenreihenfolge ist geordnet.

Der erste Fall beschreibt die Situation, in der die Prozessoren durch den Gebrauch von 'Timeouts' entscheiden können, ob ein anderer Prozessor ausgefallen ist.

Der zweite Fall beschreibt eine Situation, in der die Prozessoren auch asynchron sein können.

Abbildung 2: Möglichkeiten eines Consensus

Durch den Gebrauch von geordneten Nachrichten und der broadcast-Nachrichtenübermittlung wird zwar eine gewisse Synchronität erreicht, um aber Consensus zu erreichen, muß jeder Prozessor seinen Anfangswert an alle anderen Prozessoren verschicken. Da die Nachrichtenauslieferung geordnet ist, entscheiden sich alle Prozessoren für den ersten Wert, der auf das Netz gegeben wurde.

Der dritte Fall ist nur der Vollständigkeit wegen angegeben, da der bestbekannteste Algorithmus, der in diesem Fall Consensus erreicht, eine exponentielle Anzahl von Nachrichten benötigt [2].

1.6 Fail-Stop Fehler

Die sogenannnte **Fail-Stop-Fehlersemantik** beinhaltet, daß ein Rechnerknoten, wenn er funktioniert, immer korrekte Ergebnisse liefert. Er erzeugt keine verfälschten Nachrichten und unterdrückt auch keine Nachrichten. Fällt der Rechnerknoten aus, verschickt er gar keine Nachrichten mehr. Mit seinem Ausfall (**fail**) sind alle seine Aktivitäten unterbrochen (**stop**). Während dies kein Problem bei synchronen Prozessoren darstellt, kann im Gegensatz hierzu, die Kombination von asynchronen Prozessoren und Fail-Stop-Fehlern Consensus unmöglich machen, weil für einen Proessor unter Umständen nicht zu entscheiden ist, ob eine Nachricht ausbleibt oder erst zu einem späteren Zeitpunkt ankommt (z.B. bei ungeordneter Nachrichtenauslieferung). Im nächsten Kapitel betrachten wir deshalb asynchrone Systeme, in denen Fail-Stop-Fehler auftreten können.

1.7 Asynchrone Systeme

Betrachten wir nun ein asynchrones System, das über einen gemeinsam genutzten Speicher (**shared memory**), der nur Lese- und Schreibzugriffe unterstützt, kommuniziert. Das verteilte System läßt sich hier folgendermaßen verstehen: Mehrere Prozessoren, die parallel an verschiedenen Aufgaben arbeiten, aber auf denselben Datenbestand zugreifen, kommunizieren über diesen Speicher. Nachrichten (Daten) werden gelesen und eventuell verändert zurückgeschrieben (unter gegenseitigem Ausschluß natürlich). Nachrichten, die in diesem Speicher hinterlegt wurden, sind somit für alle anderen Prozessoren verfügbar.

Auch diese Maßnahme allein reicht nicht aus, um Consensus zu erreichen, und schon gar nicht um Fehler zu tolerieren. Dies läßt sich leicht erklären: Ein gemeinsam genutzter Speicher bietet zwar die gleichen Eigenschaften wie

Abbildung 3: Compare & Swap

die Broadcast-Übertragungsart, aber er garantiert nicht die geordnete Nachrichtenübertragung. Nachdem zwei verschiedene Prozessoren ihre Nachrichten in den Speicher geschrieben haben, gibt es keine Möglichkeit für einen dritten Prozessor zu entscheiden, welcher der beiden ersten seine Nachricht zuerst in den Speicher geschrieben hat (siehe Kapitel 1.5).

Um Consensus zu erreichen, sind also noch weitere Synchronisationmechanismen notwendig. Ein sehr leistungsfähiges Synchronisationsprimitiv ist compare & swap (Abb. 3), bei dem irgendeine Anzahl von Prozessoren ausfallen kann und trotzdem Consensus erzielt wird. Dieser Algorithmus ersetzt den Wert an dem Speicherplatz 'm' genau dann durch 'new', wenn der alte Wert im Speicher gleich 'old' ist. Wir nehmen an, daß ein spezifizierter Speicherplatz 'm', auf den sich alle beteiligten Prozessoren vorher geeinigt haben, den Anfangswert $\perp$ (ein undefinierter Wert) hat [2].

Jeder Prozessor handelt wie folgt:

1. Schreibe Anfangswert nach a[i].
2. Compare & swap (v,$\perp$,i). Versuche $\perp$ an der Stelle v durch Prozessor-ID zu ersetzen.
3. Entscheide a[v].

Es wird nur ein einziger Prozessor Erfolg mit compare & swap haben. Der Wert, den P an der vorher festgelegten Speicherstelle 'v' hinterlegt, wird der Wert sein, auf den sich alle anderen Prozessoren einigen.

Wie wir gezeigt haben, ist es in einem vollständig asynchronen System ohne die Zuhilfenahme von Synchronisationsmechanismen unmöglich Consensus zu erlangen. Für den Fall, daß Fehler auftreten, gilt diese Aussage natürlich erst recht. Folgendes Lemma drückt diesen Sachverhalt aus.

Lemma: In einem vollständig asynchronen System existiert kein deterministisches Consensus-Protokoll, das auch nur den Ausfall eines einzigen Prozessors toleriert (Den Beweis hierzu findet man in [3]).

1.8 Byzantinische Fehler

Eine weitere Fehlerquelle besteht darin, daß die oben erwähnte Fail-Stop Eigenschaft nicht erfüllt ist. Ein Prozessor ist ausgefallen, aber er verhält sich nicht, wie oben gefordert völlig ruhig, sondern produziert weiterhin Nachrichten, die ins Netz fließen und andere Rechner erreichen. Ein fehlerhafter Prozessor kann beliebige falsche Nachrichten generieren oder Nachrichten anderer Prozessoren, die er als Relaisstation übermittelt, verändern oder erfinden. Man bezeichnet derart auftretende Fehler als **Byzantinische Fehler**.

In einer einführenden Arbeit skizzieren Lamport, Shostak und Pease [1] die Situation einer Byzantinischen Armee, die bestehend aus mehreren räumlich getrennten Divisionen einen gemeinsamen Schlachtplan entwerfen muß. Jede Division wird von einem General geführt. Die Generäle können untereinander nur durch den Austausch von Botschaftern kommunizieren. Eine Zusammenkunft aller zur Beratung eines Plans ist aus strategischen Gründen ausgeschlossen. Eine Entscheidung (etwa sofortiger Angriff oder nicht) muß daher von jedem General aus den Vorschlägen seiner Kollegen nach einem vorher festgelegten Verfahren vor Ort getroffen werden. Das Problem dabei ist, daß einige wenige eventuell nicht loyale Generäle, die heimlich mit dem Feind paktieren, durch unterschiedliche Vorschläge an die anderen Generäle

eine einmündige Entscheidung der übrigen verhindern können.

Weiter unterscheidet man noch zwischen Nachrichten mit Authentisierung (jeder Prozessor kann seine Nachrichten mit einer Unterschrift versehen, sodaß Nachrichten von anderen (fehlerhaften) Prozessoren nicht unbemerkt erzeugt oder verändert werden können) und ohne Authentisierung. Werden nämlich sämtliche Nachrichten authentisiert, zum Beispiel dadurch, daß mit Hilfe eines cryptografischen Public-Key Verfahrens eine fälschungssichere digitale Unterschrift erzeugt wird, so kann die Urheberschaft einer Nachricht verifiziert werden.

Im Folgenden betrachten wir nur vollständige Netzwerke, in denen die Prozessoren synchron arbeiten (die Unmöglichkeit des asynchronen Falles folgt aus der Unmöglichkeit für Fail-Stop-Fehler).

(1) Ohne Authentisierung: Byzantinische Übereinstimmung ist möglich, wenn es mindestens 3t+1 Prozessoren in der Anwesentheit von t fehlerhaften Prozessoren gibt. In anderen Worten: Wenn 1/3 oder mehr der Prozessoren fehlerhaft sind, ist Übereinstimmung unter den fehlerfreien Prozessoren nicht möglich [1].

(2) Mit Authentisierung: Ein Algorihtmus zur Lösung des Consensus-Problems ist der unten angegebene SM(t) (wobei das 't' für die Anzahl der fehlerhaften Prozessoren steht).

Beschreibung des Algorithmus SM(t):

Phase 1: Der Sender signiert seinen Wert und sendet ihn an alle Prozessoren.

Phase k: Ein Prozessor P erhält eine Nachricht mit dem Wert v, die von k Prozessoren signiert wurde. P vergleicht den erhaltenen Wert v mit seiner Wertemenge. Ist der Wert schon enthalten, wird der neue Wert ignoriert. Ist er noch nicht enthalten und ist die Anzahl der Elemente in seiner Wertemenge kleiner als 3, so signiert P die Nachricht und schickt sie in der Phase (k+1) an alle Prozessoren, die diese Nachricht noch nicht signiert haben.

Ende: nach (t+1) Phasen.

Entscheidung: Hat ein Prozessor nur einen Wert in seiner Wertemenge, so entscheidet er sich für diesen. Andernfalls entscheidet er sich für einen vorher festgelegten Default-Wert.

Mit diesem Algorithmus ist t-fehlertolerante Übereinstimmung in (t+1) Runden möglich (Den Beweis zum Algorithmus findet man in [1]).

Bei diesem Algorithmus einigen sich die fehlerfreien Prozessoren für jeden Prozessor auf eine Alternative, den sie als dessen Vorschlag ansehen wollen. Für fehlerhafte Prozessoren ist es dabei gleichgültig welche möglicherweise verschiedenartige Vorschläge dieser tatsächlich gemacht hat. Wendet dann jeder fehlerfreie Prozessor auf die Gesamtheit dieser Werte das Entscheidungsverfahren an, so erhalten sie alle zwangsläufig das gleiche Ergebnis. Somit zerfällt das ursprüngliche Problem in unabhängige gleichartige Teilprobleme, bei denen jeweils ein Consensus zu erzielen ist, als Voraussetzung aber nur ein einzelner Wert, nämlich der Vorschlag eines einzelnen Prozessors, gegeben ist.

1.9 Commit

Durch das Auftreten von Fehlern in einer verteilten Datenbank ergeben sich zwei grundlegende Probleme für das Arbeiten mit Datenobjekten und Datensätzen. Zum einem enthalten atomare Aktionen meist mehrere Operationen und eine atomare Aktion gilt erst als abgeschlossen, wenn alle ihre Operationen abgeschlossen sind. Zum zweiten: Um auf dem Gesamtsystem den erwünschten Erfolg zu erzielen, müssen die Orginale nach der Beendigung aller Operationen durch die Arbeitskopien überschrieben werden.

Das Commit-Problem ist eine besondere Art des Consensus-Problem, bei dem die zwei möglichen Entscheidungswerte **'Commit'** und **'Abort'** heißen. Eine Transaktion innerhalb einer verteilten Datenbank ist eine atomare Operation, die auf allen Teilsystemen vollzogen werden muß (commit), oder auf keinem (abort). Wird ein Commit erreicht, werden alle Änderungen in der Datenbank permanent installiert,

Abbildung 4: 2-Phasen-Commit-Protokoll

obwohl alle dem Commit zugestimmt haben, ein Abort aus diesem Grund durchgeführt wurde [4].

Das bekannteste Protokoll ist das **2-Phasen-Commit-Protokoll**:

In der Anfangsphase verschickt ein voher bestimmter Koordinator eine 'Prepare-Message' an jeden Teilnehmer. Die Teilnehmer antworten entweder mit einer Ready-Answer-Message (d.h. sie stimmen einem Commit zu) oder mit einer Abort-Answer-Message. Wenn alle Teilnehmer dem Commit zugestimmt haben (d.h., daß der Koordinator keine einzige Abort-Answer-Message empfangen hat und kein Timeout bei sich ausgelöst wurde), schickt der Koordinator eine Commit-Command-Message an die Teilnehmer zurück, andernfalls eine Abort-Command-Message (Abb. 4). Die Teilnehmer handeln dann entsprechend ihrer Nachricht.

Dieses Protokoll kann blockieren, wenn der Koordinator ausfällt und die Teilnehmerstationen sich in der zweiten Phase befinden, also auf die Antwort des Koordinators warten. In diesem Fall bleibt den Stationen nichts anderes übrig, als auf die Behebung des Koordinatorfehlers zu warten. Betrachten wir zunächst ein-

Abbildung 5: 3-Phasen-Commit-Protokoll

die Transaktion daher gefahrlos abbrechen und die ausgefallene Station folgt dieser Entscheidung im Zuge der Restart-Prozedur.

Die anderen Fälle verlaufen entsprechend dem 2-Phasen-Commit Protokoll. Sind nur Teilnehmer ausgefallen und ist der Koordinator noch intakt, so ist das Vorgehen trivial: Die funktionierenden Stationen verfahren weiter nach den Anweisungen des Koordinators, die ausgefallenen Stationen holen das beim Restart nach, falls sie nicht sowieso schon die Anweisungen des Koordinators ausgeführt haben.

Ist der Koordinator ausgefallen, so muß von den noch funktionierenden Stationen ein neuer Koordinator gewählt werden, der das Protokoll aufnimmt. Dieser neue Koordinator fragt zuerst die restlichen Teilnehmer nach ihrem Systemzustand.

1. Möglichkeit: Hat mindestens ein Teilnehmer den Zustand Prepared_to_Commit eingenommen, kann ein Commit vorgenommen werden, da alle ihre Ready-Message abgesendet haben.

2. Möglichkeit: Hat mindestens ein Teilnehmer den Status Prepared_to_Commit noch nicht

erreicht, kann auch ein Abort gefahrlos angeordnet werden, da der ausgefallene Koordinator noch nicht alle OK's empfangen haben kann, und somit noch keinen Commit vollzogen hat.

Haben einige, aber nicht alle den Status Prepared_to_Commit eingenommen, kann also wahlweise ein Abort oder Commit angeordnet werden.

Die Auswahl eines neuen Koordinators kann dadurch vereinfacht werden, daß die Stationen in einer festen Reihenfolge angeordnet werden, wobei die erste Station anfangs die Rolle des Koordinators übernimmt. Bei einem Ausfall des Koordinators überprüft jede Station, ob noch Stationen kleinerer Ordnungsnummer funktionieren. Ist dies nicht der Fall, so übernimmt die Station die Rolle des Koordinators. Entscheidend für den reibungslosen Ablauf dieses Verfahrens ist natürlich eine einheitliche Sicht des Netzwerkstatus auf allen Stationen.

1.10 Netzwerkpartitionen

Bisher wurden lediglich Ausfälle von Stationen behandelt. Partitionen, also die Trennung eines Kommunikationsnetzes in mehrere Teile, zwischen denen keine Kommunikationsverbindung besteht, stellen ein schwieriger zu behandelndes Problem dar. Zum Beispiel kann das 3-Phasen-Protokoll unter diesen Voraussetzungen fehlerhaft werden.

Generell existiert kein nichtblockierendes Protokoll für Netzwerkpartitionen. Ein Protokoll, das sehr robust im Umgang mit Netzwerkpartitionen ist, ist das **Quorum-based-Commit-Protokoll** (Abb. 6). Dieses Protokoll basiert auf dem 3-Phasen-Commit-Protokoll.

Es entscheidet stets eine Mehrheit von Stationen innerhalb einer Partition über Commit oder Abort einer Transaktion. Dabei können den Stationen unterschiedliche Stimmgewichte zugeteilt werden (Die Stimmenanzahl einer Station kann auch 0 sein, wobei es sich dann um einen passiven Teilnehmer handelt). Die zugrundeliegende Idee ist es, innerhalb einer Partition genug Stimmen für einen Commit zu sammeln. Gelingt es innerhalb einer Partition die dafür notwendige Stimmenanzahl zu erlangen, kann diese Partition mit der Transaktion fortfahren [5]. Die grundlegende Struktur dieses Protokolls stellt sich wie folgt dar:

1. Jede Station hat eine positive ganze Anzahl von Stimmen (votes).
2. Bei einer Gesamtstimmenzahl V im Netzwerk und einer erforderlichen Stimmenanzahl Vc für einen Commit, bzw. Va für einen Abort muß gelten: Vc+Va>V.

   Durch Bedingung 2 wird ein gleichzeitiges Commit und Abort trivialerweise ausgeschlossen. Es kann aber vorkommen, daß weder über ein Commit noch über ein Abort Einigung erzielt werden kann. Daher müssen ausgefallene Stationen im Rahmen einer Restart-Prozedur am Entscheidungsprozeß teilnehmen, falls noch keine Entscheidung erzielt wurde. Dies kann dazu führen, daß sich der Entscheidungsprozeß über längere Zeit hinzieht. Das Bilden eines Quorums findet ausschließlich in der dritten Phase statt, in der eine Transaktion nur vollzogen wird, wenn ein Commit-Quorum von Stationen, die sich im Prepared_to_Commit Status befinden, erzielt wird.

   Dieses Protokoll ist ein sehr pessimistisches Protokoll, wann immer während der ersten Phase entweder eine Seite ausfällt oder eine Partition entsteht, bricht der Koordinator die Transaktion sofort ab.

   Das Ziel des vorgestellten Protokolls ist es natürlich erfolgreich zu terminieren (wenn es sein muß durch einen Abort). Wenn festgestellt wurde, daß eine Netzwerkpartition vorliegt, führen die einzelnen Partitionen ein zweiteiliges Terminierungsprotokoll durch. Im ersten Teil wählen die Partitionen, die von dem Koordinator getrennt wurden, mit Hilfe eines Auswahlverfahrens einen neuen Koordinator. Im zweiten Teil versuchen die einzelnen Partitionen ein Quorum zu bilden. Das Terminierungsprotokoll ist aus zwei Gründen komplexer: Erstens, weil ein neuer Koordinator mit einem geringeren Wissen arbeitet wie der ehemalige Koordinator, und daher nicht weiß, ob sich die Transaktion in einem Zustand befindet, der zu einem

Abbildung 6: Quorum-based-Commit Protokoll

Commit führen kann. Zweitens: Wo vorher ein einziger Koordinator war, arbeiten jetzt mehrere Koordinatoren in verschiedenen Partitionen.

Das erste Problem liegt darin, daß ein Koordinator nur ein Commit-Quorum bilden kann, wenn sich ein Teilnehmer innerhalb der Partition in einem Zustand befindet, der zu einem Commit führen kann (z.B. Prepared_to_Commit). Das zweite Problem liegt darin, daß der Koordinator diesmal explizit ein Abort-Quorum bilden muß. Ein Teilnehmer zeigt seine Absicht in einem Abort-Quorum teilzunehmen, indem er in den Zustand 'Prepared_to_Abort' wechselt (Dieser Zustand entspricht dem Prepared_to_Commit auf der Abort-Seite).

Das Terminierungsprotokoll läßt sich folgendermaßen beschreiben:

In der ersten Phase erfragt der Koordinator die Zustände der einzelnen Teilnehmer, die in den nächsten zwei Phasen die weiteren Handlungen beeinflussen. Wenn irgendein Teilnehmer schon einen Commit (Abort) durchgeführt hat, wird der Commit (Abort) sofort auf allen anderen Seiten durchgeführt; andernfalls versucht der Koordinator ein Quorum zu bilden.

Bedingungen für ein Commit-Quorum: Ein Commit-Quorum ist möglich, wenn mindestens einer der Teilnehmer sich in dem Zustand Prepared_to_Commit befindet und die Summe der Stimmen der Teilnehmer, die sich im Prepared_to_Commit- und im Wait-Zustand befinden, größer oder gleich Vc ist. Trifft dies zu, versucht der Koordinator die Teilnehmer, die sich im Wait-Zustand befinden, in den Prepared_to_Commit-Zustand zu bringen. Gelingt dies alles und treten keine weiteren Fehler auf, wird die Transaktion durch einen Commit beendet. Im Gegensatz hierzu blockiert das Protokoll, wenn weitere Fehler die Teilnehmer an diesem Handeln hindern.

Bedingungen für ein Abort-Quorum: Ein Abort-Quorum ist möglich, wenn die Summe

der Stimmen der Teilnehmer, die sich im Prepared_to_Abort- und im Wait-Zustand befinden, größer oder gleich Va ist. Trifft dies zu, versucht der Koordinator diesmal alle Teilnehmer in den Prepared_to_Abort-Zustand zu bringen. Gelingt dies, wird ein Abort durchgeführt, andernfalls blockiert das Protokoll.

Innerhalb der Partitionen wird also zuerst versucht Stimmen für einen Commit zu sammeln. Gelingt dies nicht, versucht man genug Stimmen für einen Abort zu sammeln. Bleibt dieser Versuch wiederum erfolglos, werden innerhalb der betroffenen Partition alle weiteren Handlungen blockiert. Das Protokoll blockiert solange bis die Fehler wieder behoben sind. Wir nehmen hierzu an, daß das Wiederherstellen von Kommunikationspfaden von den einzelnen Teilnehmern festgestellt werden kann.

1.11 Zusammenfassung

Um eine gemeinsame Entscheidung zu erlangen, ist es notwendig, daß die beteiligten Prozessoren das Consensus Problem lösen. Diese Aufgabe ist einer der wichtigsten Gesichtspunkte bei der Entwicklung verteilter Systeme. Es besteht oft nur eine feine Trennlinie zwischen dem Möglichen und dem Unmöglichen. Das Auftreten von Fehlern (Fail-Stop- oder byzantinische Fehler) kann das Erlangen eines Consensus unmöglich machen. Die Frage der Lösbarkeit und der Komplexität der entsprechenden Protokolle hängt von verschiedenen Parametern ab. Dazu gehören die Topologie des Netzwerkes, die Art und die Anzahl der auftretenden Fehler und der Grad der zeitlichen Asynchronität von Prozessoren und des Kommunikationsflusses.

In verteilten Datenbanken wandelt sich das Consensusproblem zu einem Commitproblem, mit den zwei möglichen Entscheidungen 'Commit' und 'Abort'. Innerhalb dieses Problems kann das Auftreten einer Netzwerkpartition einen Commit unmöglich machen. Vorgestellt wurden dazu Protokolle, die beim Auftreten von Fehlern (Ausfallfehler oder Netzwerkpartitionen), Inkonsistenzen verhindern.

Literatur

[1] [LSP82] L.Lamport,R.Shostak,and M.Pease, *The Byzantine Generals Problem*, ACM Trans. Programming Languages and Systems,Vol.4,No.3,July 1982,pp.382-401.

[2] [TUS82] J.Turek and D.Shasha, *The Many Faces of Consensus in Distributed Systems*, IEEE Computer 26/6 (1992), pp. 8-17.

[3] [LYN89] N.A. Lynch, *A Hundred Impossibility Proofs for Distributed Computing*, Proc. of the 8th Symposium on Principles of Distributed Computing, Edmonton, Canada (1989), pp. 1-27.

[4] [SKE83] D.Skeen and M.Stonebraker, *A Formal model of Crash Recovery in a Distributed System*, IEEE Transactions on Software Engineering SE 9/3 (1983), pp. 219-228.

[5] [SKE82] D.Skeen, *A Quorum-Based Commit Protocol*, Proc. Berkeley Conference on Distributed Data Management (1982), pp. 69-80.

2 Workflow

GUNNAR DITTLOFF

2.1 Einleitung

In der realen Welt treten immer wieder standardisierte Arbeitsabläufe auf, die mehrfach in verschiedenen Situationen von mehreren Mitarbeitern ausgeführt werden. Im Bereich der Bürowelt kommt es zum Beispiel immer wieder vor, daß ein Arbeitsablauf mehrere Mitarbeiter eines Unternehmens einschließt. Im allgemeinen werden dafür alle Dokumente, die zu so einem Büroablauf gehören in einer Akte zusammengefaßt. Diese Akte wandert dann zu den verschiedenen Mitarbeitern. Jeder Mitarbeiter erledigt nun einen Teilschritt des gesamten Ablaufs, bevor er die Akte an den nächsten Mitarbeiter weiter gibt. Ein Ablauf ist erledigt, wenn der letzte Mitarbeiter seinen Teilschritt ausgeführt hat.

In diesem Beitrag soll untersucht werden, wie verteilte Abläufe auf verteilten Systemen automatisch ausgeführt werden können.

Im Kapitel 2.2 soll anhand eines Beispiels zunächst vermittelt werden, was verteilte Abläufe sind. Es wird ein Modell für verteilte Abläufe zugrunde gelegt. Danach wird ein Ansatz beschrieben, wie verteilte Abläufe mit Hilfe von verteilten Systemen bearbeitet werden können. Es handelt sich dabei um den Ansatz IPSO, der in [1] und [2] beschrieben wird. Zu diesem System gehört eine Sprache, die die Spezifikation von verteilten Abläufen und Serverumgebungen ermöglicht. Es werden Laufzeitmechanismen behandelt, die verteilte Abläufe verwalten und ausführen. Dazu gehören zum Beispiel Konzepte zum Binden von Diensten an Server. Außerdem wird eine kleine Einführung in verteilte objektorientiere Systeme gegen, da diese allen Überlegungen zugrunde liegen.

2.2 Verteilte Abläufe

Alle folgenden Überlegungen beziehen sich auf verteilte Abläufe. Deshalb soll hier einleitend ein Modell für verteilte Abläufe beschrieben werden, um eine Vorstellung davon zu vermitteln, was ein verteilter Ablauf ist.

Ein verteilter Ablauf ist ein komplexer Vorgang einer Anwendungswelt. Er läßt sich aufgrund seiner Struktur in viele Teilschritte zerlegen. Zwischen den Teilschritten eines Ablaufs bestehen Abhängigkeiten. Die Art der Abhängigkeiten zwischen den Teilschritten eines Ablaufs, hängt stark mit ihrer Ausführbarkeit zusammen. Teilschritte können parallel, alternativ, oder nur sequentiell ausgeführt werden. Die Ausführung eines verteilten Ablaufs erfolgt durch schrittweises Abarbeiten seiner Teilschritte. Jeder Teilschritt wird durch einen bestimmten Dienst innerhalb des verteilten Systems ausgeführt. Die Koordination der Teilschritte ist Aufgabe des Laufzeitsystems. Die Teilvorgänge können vollautomatisch durch Server ausgeführt werden. Sie können aber auch halbautomatisch, rechnergestützt durch Mitarbeiter ausgeführt werden.

Die Verteilung eines Ablaufs kann verschiedene Ursachen haben. Im Falle des folgenden Beispiels aus der Bürowelt, sind es betriebliche Gründe, die eine Verteilung bewirken. Von einem Büroblauf sind verschiedene Mitarbeiter betroffen, deren Büros sich an unterschiedlichen Orten befinden. Die Verteilung entsteht, weil der Büroablauf während der Ausführung zu den Mitarbeitern wandert. Jeder Teilschritt wird im allgemeinen durch einen anderen Mitarbeiter an einem anderen Ort ausgeführt. Ist eine Verteilung nicht aufgrund einer betrieblichen Situation erforderlich, so gibt es weitere Gründe, die für eine Verteilung sprechen. Sie werden im folgenden aufgezählt:

1. Durch Verteilung läßt sich eine Leistungssteigerung erzielen, indem Teilschritte parallel ausgeführt werden.

2. Eine gleichmäßige Ausnutzung redundanter Ressourcen kann erreicht werden, indem Mechanismen zur Lastverteilung zur Anwendung kommen.

3. Mit Hilfe der Verteilung läßt sich die Ausfallsicherheit erhöhen. Man erreicht dies, indem bestimmte kritische Aufgaben eines Auftrags

auf redundanten Knoten ausgeführt werden. Die Ausführung eines Auftrags ist dann gesichert, sobald einer der redundanten Knoten arbeitet. Es können somit Ausfälle toleriert werden, solange noch ein Knoten arbeitet.

4. Nicht zuletzt läßt sich mit Hilfe der Verteilung die gemeinsame Nutzung teurer Ressourcen ermöglichen, wodurch Kosten eingespart werden.

Als Beispiel soll hier die Abwicklung einer Reisekostenabrechnung innerhalb eines Unternehmens dienen. Abbildung 7 zeigt eine Darstellung dieses Beispiels in Form eines Ablaufgraphen. Der Gesamtvorgang der Reisekostenabrechnung umfaßt dabei das Ausfüllen eines Abrechnungsformulars durch einen Angestellten, das Eintragen von Projekt- und Managementdaten, ggf. das Abzeichnen durch einen Manager, die endgültige Abrechnung und die Überweisung durch die Reisekostenstelle.

Die Knoten entsprechen hier den Teilschritten, die ausgeführt werden müssen. Mit Hilfe der Kanten werden die Abhängigkeiten modelliert. Da die Abhängigkeitsrelation gerichtet ist, sind auch die Kanten gerichtet. Der abhängige Teilschritt befindet sich immer am Ende einer Kante.

In diesem Beispiel kommen alle Abhängigkeiten vor, die hier betrachtet werden sollen. Nachdem das Formular im ersten Teilschritt ausgeführt wurde, können Managementdaten und Projektdaten eingetragen werden. Diese beiden Vorgänge sind unabhängig voneinander und können somit parallel erfolgen. Im Graphen wird die Parallelität spezifiziert, indem mehrere Kanten ohne Bedingung von einem Knoten ausgehen.

Nach Eintragen der Managementdaten hängen weitere Aktionen von der Höhe des Betrags ab. Im Normalfall zeichnet der Manager das Formular ab und bestätigt so die Abrechung (Fall 1). Bei sehr geringen Aufwendungen ist eine Abzeichnung nicht erforderlich (Fall 2). Die Bestätigung aus dem höheren Management muß eingeholt werden, falls die Aufwendung hoch ist (Fall 3). Im Graphen werden Alternativen spezifiziert, indem mehrere Kanten mit Bedingungen von einem Knoten ausgehen.

In dem Ablaufgraphen sind implizit einige Anforderungen enthalten, die eigentlich selbstverständlich sind. Sie werden hier nochmal genannt.

- Der Ablaufgraph muß zusammenhängend sein und einen ausgezeichneten Anfangs- und Endknoten besitzen. Ein ausgezeichneter Anfangsknoten ist notwendig, damit man einen Startpunkt für einen Ablauf hat. Analog ist der Endknoten notwendig, um ein definiertes Ende zu besitzen. Die Eigenschaft *zusammenhängend* ist notwendig, da durch sie gerade die verschiedenen Aktivitäten, zu einem Ablauf gehörend, spezifiziert werden.

- Es muß sich parallele Ausführbarkeit von Aktivitäten und bedingte Verzweigung spezifizieren lassen. Innerhalb eines Ablaufs können Aktivitäten parallel ausgeführt werden, sobald sie voneinander unabhängig sind. Diese Möglichkeit soll auch zur Leistungssteigerung ausgenutzt werden. Die Parallelität läßt sich relativ einfach erreichen. Für die Alternativen ist es zunächst einmal notwendig, Bedingungen zu formulieren, die für die Wahl der Alternativen ausgewertet werden können.

- Der Graph muß hierarchisch strukturierbar sein, damit sich auch komplexe Abläufe übersichtlich modellieren lassen.

2.3 Systemkomponenten

In den folgenden Abschnitten soll die Architektur IPSO zur Ausführung verteilter Abläufe vorgestellt werden. Die gesamte Architektur basiert auf einem verteilten System. Ihre Realisierung erfordert gewisse Anforderungen an das unterliegende System. In diesem Abschnitt werden diese Anforderungen ausgeführt. Dies erfolgt, indem die Anforderungen an das Laufzeitsystem formuliert werden. Die Realisierung eines derartigen Laufzeitsystem erfordert dann Unterstützung durch das verteilte System. Es wird

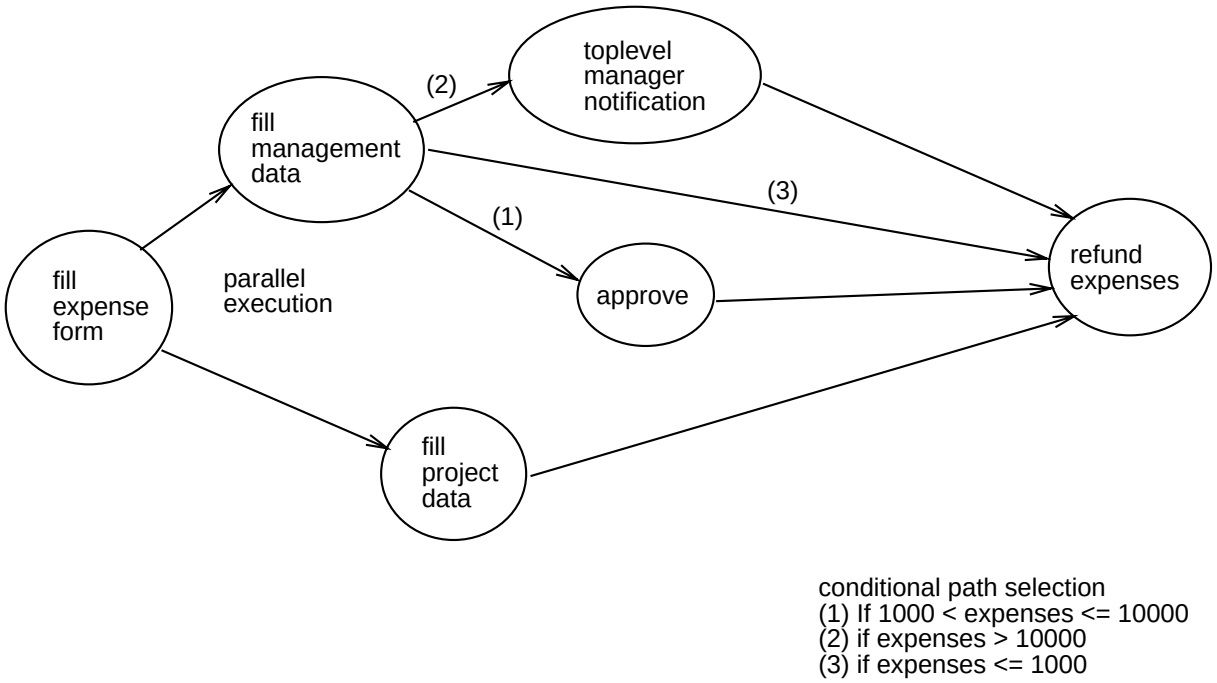


Abbildung 7: Ablaufgraph

sich zeigen, das sich verteilte objektorientierte Systeme sehr gut eignen, um die Ausführung und Repräsentation verteilte Abläufe zu unterstützen.

2.3.1 Anforderungen an das Laufzeitsystem

Das System IPSO enthält ein Laufzeitsystem zur Verwaltung verteilte Abläufe. Dazu gehört im wesentlichen die Ausführung von verteilten Abläufen. In diesem Abschnitt werden nun die folgenden Anforderungen an das Laufzeitsystem ausgeführt: Das Laufzeitsystem soll eine Kontrolle von Abläufen ermöglichen. Ein Ablauf sollte explizit als ein Objekt repräsentiert werden, damit sich verschiedene Abläufe voneinander abgrenzen lassen (Bei paralleler Ausführung von Abläufen wird das zugehörige Ablaufobjekt allerdings repliziert). Es ist somit möglich, Statusanfragen an einen Ablauf zu stellen. Das Binden von Teilschritten eines Ablaufs an Dienste des verteilten Systems ist für die Ausführung notwendig.

Das Laufzeitsystem muß Kontrollmechanismen bereit stellen. Dazu gehört, daß sich der Zustand eines Ablaufs ermitteln läßt. Es muß möglich sein, den Ort eines Ablaufs im verteilten System zu bestimmen. Ein expliziter Eingriff in Abläufe muß möglich sein, um zum Beispiel Fehlerfälle behandeln zu können.

Zu einem Ablaufobjekt gehören im allgemeinen Datenobjekte, die während der Ausführung bearbeitet werden. Das System muß die Möglichkeit bieten, Datenobjekte an Ablaufobjekte zu binden. Die Zusammengehörigkeit von Objekten soll also unterstützt werden können.

Das Laufzeitsystem sollte das statische und dynamische Binden von Teilschritten an Dienste unterstützen. Für das statische Binden eines Teilschritts wird im verteilten Ablauf explizit ein Dienst angegeben. Das Laufzeitsystem muß in diesem Fall den Teilschritt auf den spezifizierten Dienst abbilden. Beim dynamischen Binden werden für einen Teilschritt nur Eigenschaften des Dienstes angegeben. Das Laufzeitsystem wählt unter allen verfügbaren Diensten mit diesen Eigenschaften einen zur Ausführung aus.

Aus diesen Anforderungen an die Laufzeitverwaltung lassen sich nun Anforderungen an das verteilte System ableiten.

2.3.2 Anforderungen an das verteilte System

Die Ausführung von Abläufen erfordert es, daß Server miteinander kommunizieren können. Es muß möglich sein, Daten und Kontrolle eines Ablaufs zwischen Servern zu übergeben. Grundsätzlich muß es möglich sein, Nachrichten einem Ablauf zuzuordnen. Bei der Anfrage von Statusinformationen muß spezifizierbar sein, welche Statusinformationen angefragt werden. Für die Übergabe der Kontrolle eines Ablaufs muß spezifizierbar sein, in welchem Zustand sich der Ablauf befindet, also welche Abarbeitungsphase gerade begonnen werden soll. Es sind systemweite eindeutige Kennungen für Abläufe notwendig, um Abläufe im System auffinden zu können. Bei Statusanfragen werden diese Kennungen ausgenutzt.

Diese grundsätzlichen Anforderungen werden von verteilten objektorientierten Systemen gut unterstützt. Sie besitzen folgende Eigenschaften:

- Alle Einheiten eines verteilten objektorientierten Systems werden durch Objekte modelliert.
- Jedes Objekt bekommt bei der Erzeugung eine systemweit eindeutige Kennung, die sich während der gesamten Lebensdauer nicht ändert.
- Objektaufrufe erfolgen lokationstransparent. Für den Methodenaufruf eines Objekts ist lediglich die Kennung des Objekts erforderlich. Die Suche nach Objekten übernimmt das System.
- Objekte können dynamisch migrieren, also während der Laufzeit zwischen den Rechnern verlagert werden.

2.3.3 Modellierung mit verteilten objektorientierten Systemen

Aufgrund der Eigenschaften objektorientierter verteilter Systeme, läßt sich das Laufzeitsystem realisieren. Bei dem Ansatz IPSO werden Server einer Anwendungsumgebung und Daten als Objekte modelliert. Ein Objekt besitzt immer eine Menge von internen Daten und stellt zugehörige Operationen für die Manipulation von Daten bereit. Die Objekte eines verteilten Systems liegen auf verschiedenen Knoten. Sie können durch Referenzen miteinander in Beziehung gebracht werden.

Weiterhin wird vorausgesetzt, daß es eine verteilte Namensverwaltung gibt. Die Namensverwaltung ermöglicht das Abbilden von logischen Namen auf eindeutige Adressen in einem verteilten System. Dies wird ausgenutzt, um die Adresse eines Servers unter Angabe eines Dienstes zu ermitteln.

Die Repräsentation von Abläufen innerhalb eines verteilten objektorientierten Systems erfolgt mit Hilfe von Objekten. Jeder Ablauf wird in erster Linie durch ein Ablaufobjekt repräsentiert. Das Ablaufobjekt enthält alle Information, die notwendig sind, um den Ablauf ausführen zu können. Auch die Daten, die zu dem Ablauf gehören, werden in Form von Objekten repräsentiert und vom Ablaufobjekt verwaltet. Abbildung 8 zeigt ein Beispiel für ein Ablaufobjekt. Grob betrachtet enthält es ein Segment für die Spezifikation des Ablaufs, den Ablaufplan und ein Segment für die zugehörigen Daten. Das Segment für die Daten läßt sich weiter gliedern in dynamische und statische Datenslots.

2.4 Repräsentation

Für die Ausführung eines Auftrags durch das Laufzeitsystem erfolgt eine Auswertung von Daten des Ablaufobjekts. Dazu gehört insbesondere die Ablaufbeschreibung. Mit Hilfe der dabei gewonnen Informationen wird der Ablauf auf das verteilte System abgebildet. Das bedeutet, das die einzelnen Teilschritte des Ablaufs mit Hilfe von Servern ausgeführt werden müssen. Eine wesentliche Aufgabe des Laufzeitverwaltung ist also die Abbildung von Diensten auf Server. Diesen Vorgang nennt man Binden. Der Vorgang des Bindens wird in späteren Abschnitten genauer behandelt. Hier kommt es lediglich darauf an, zu verstehen, daß dem Laufzeitsystem explizite sprachliche Repräsentationen vorliegen müssen, um das Binden ausführen

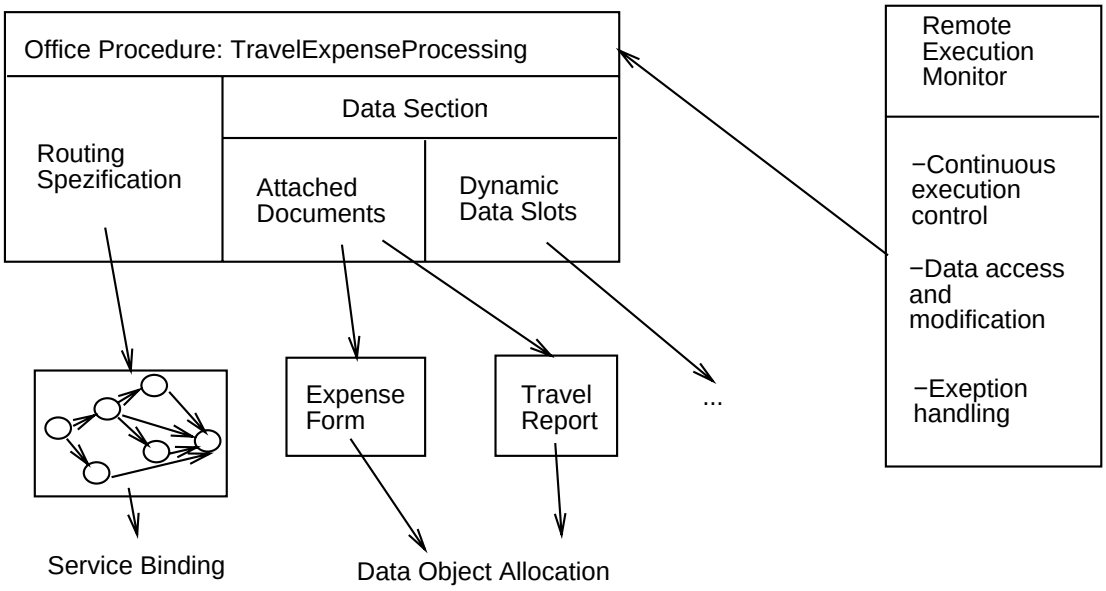


Abbildung 8: Ablaufobjekt

zu können. Dazu gehört sowohl eine Beschreibung des Ablaufs, als auch eine Beschreibung der Serverumgebung. In diesem und im folgenden Abschnitt soll, anhand des Beispiels, eine Sprache eingeführt werden, die eine Repräsentation ermöglicht.

2.4.1 Server Umgebungen

Als elementare Elemente werden Knoten und Server unterschieden. Ein Knoten ist ein physikalisches Element in einem verteilten System. Rechner, Drucker und Scanner sind Beispiele für Knoten.

Knoten und Server werden mit Hilfe einer streng typisierten Notation beschrieben. In dieser Notation wird die Serverumgebung dem Laufzeitsystem bekannt gemacht. Nachfolgend wird die Spezifikation von Knoten und Servern beschrieben.

Die Spezifikation eines Knotens setzt sich aus einem Typennamen und einer Menge von Ressourcen zusammen. Der Typenname ist eine eindeutige Bezeichnung für einen Knotentyp innerhalb des verteilten Systems. Die Menge der Ressourcen legt fest, welche Ressourcen unter diesem Typ vorhanden sein sollen. Folgende Zeilen illustrieren die Typenbeschreibung eines Knotens.

```
NODE_TYPE universalhost
  RESOURCES
    printer,
    scanner;
END_NODE_TYPE
```

Es wird der Knotentyp *universalhost* eingeführt. Knoten von diesem Typ müssen über einen *printer* und über einen *scanner* verfügen.

Ist erst einmal der Typ eines Knotens festgelegt, so können Instanzen von diesem Typ angelegt werden. Für die Spezifikation einer Knoteninstanz muß ein Typ angegeben werden. Eine Instanz eines Knotens muß damit über alle Ressourcen verfügen, die durch den Typ des Knotens festgelegt werden. Es sind jedoch zusätzliche Ressourcen erlaubt. Durch den Typ wird also nur eine Mindestanforderung verlangt. Folgende Zeilen führen Instanzen von Knoten ein.

```
NODES
  officeserver1,
  officeserver2:
    universalhost;
END_NODES
```

Es werden die Knoten *officeserver1* und *officeserver2* eingeführt. Beide sind vom Typ *universalhost* und verfügen somit über einen *printer* und über einen *scanner*.

Nachdem wir Knoten repräsentieren können, wollen wir uns jetzt mit der Spezifikation von Servern befassen. Auch hier wird zwischen Typen und Instanzen unterschieden. Die Spezifikation eines Servertyps umfaßt sowohl eine Dienstbeschreibung, als auch ein Festlegen von Attributen. Zur Dienstbeschreibung gehört die Festlegung der Eingabe- und Ausgabeparameter. Es wird also die Signatur der Schnittstelle festgelegt. Durch Attribute werden bestimmte Eigenschaften eines Dienstes näher beschrieben.

Durch folgende Zeilen wird ein Dienst zum Drucken beschrieben:

```
SERVER_TYPE printserver
  SERVICES
    print ( ATTACH d: document,
            OPTIONAL format )
      ATTRIBUTES speed = regular,
                 cost = d.size
    quickprint ( ATTACH d: document,
                 OUT time )
      ATTRIBUTES speed = fast,
                 cost = d.size
  STATUS_INQUIRY inputqueue,
                 remainingpaper
  REQUIRED_RESOURCES printer
  OTHER_SERVERS alternativeprinters:
    SET printserver
  DELEGATION_TO alternativeprinters
    IF reminigpaper < minamount
END_SERVER_TYPE
```

Hinter dem Servertyp *printserver* verbergen sich somit die Dienste *print* und *quickprint*. Durch die Attribute *speed* und *cost* werden die Druckgeschwindigkeit und die Kosten des Dienstes näher beschrieben.

Mit den Angaben unter dem Schlüsselwort *STATUS_INQUIRY* werden einfache Anfrageoperationen definiert, mit denen der Status des Servers ermittelt werden kann. Diese Möglichkeit wird vom Laufzeitsystem ausgenutzt, um aus einer Menge von Servern, die für die Ausführung eines Dienstes in Frage kommen, ein Auswahlkriterium zu erhalten.

Im Teil *REQUIRED_RESOURCES* werden die notwendigen Ressourcen des Dienstes beschrieben, um eine Platzierung des Dienstes auf Knoten zu unterstützen.

Mit den Abschnitten *OTHER_SERVERS* und *DELEGATION_TO* werden verschiedene Server gleichen Typs miteinander in Beziehung gesetzt. Ein Server kann Referenzen zu anderen Servern mit ähnlichen Eigenschaften besitzen. Es ist somit möglich, daß ein Server seine Aufträge an einen anderen Server weiterleitet, falls er die Bearbeitung nicht vornehmen kann. Durch das Schlüsselwort *SET* wird hier eine Menge von alternativen Servern verwaltet.

Bei den Eingabeparametern an der Dienstschnittstelle können die Schlüsselwörter *ATTACH* oder *COPY* angegeben werden. Sie dienen zur Steuerung der Objektmigration im verteilten objektorientierten System.

- Mit *ATTACH* wird spezifiziert, daß der Parameter zum Server migireren soll. Dadurch wird ein effizienter lokaler Zugriff ermöglicht.
- Mit *COPY* wird spezifiziert, daß eine lokale Kopie des Parameters angefertigt werden soll. Diese Kopie liegt dann beim Server vor.

Mit dem Parameterattribut *OPTIONAL* wird ein optionaler Parameter spezifiziert. Das Attribut *OUT* bei *quickprint* gibt einen Timeout für die Ausführung des Dienstes an.

Die Instanz eines Servers wird folgendermaßen spezifiziert.

```
SERVER printer: printserver;
  ALLOCATION: hostsystem;
  print.speed = fast;
  quickprint.cost = constantcosts;
  alternativeprinters = printer2,
                        printer3;
END_SERVER
```

Diese Notation erlaubt das Überschreiben von Attributen. Es wird hier das Attribut *print.speed* mit dem Wert *FAST* überschrieben.

Am Schluß dieses Abschnitts sollen Vorteile der Typisierung von Servern aufgelistet werden. Der wichtigste Vorteil ist eine Abstraktion von Serverinstanzen, die durch die Typisierung ermöglicht wird. Da mit einem Servertyp alle Eigenschaften entsprechender Instanzen

festliegen, ist es nun möglich, Abläufe zu spezifieren ohne dabei explizit Server benennen zu müssen. Die Typisierung ermöglicht also eine Trennung zwischen Diensten und Servern. Bei der Spezifikation von Abläufen stützt man sich im folgenden auf Typen von Servern und abstrahiert von den Instanzen. Für das Binden ergibt sich dadurch die Möglichkeit der Dynamik. Man spricht von dynamischen Binden was im Gegensatz zum statischen Binden eine flexible Abbildung von Diensten auf Servern zur Laufzeit ermöglicht. Die Vorteile des dynamischen Bindens liegen zum Beispiel im Bereich der Lastverteilung und der Fehlertoleranz. Viele Vorteile, die ein verteiltes System besitzt, werden durch das dynamische Binden erst ausgenutzt.

2.4.2 Verteilte Abläufe

Die Spezifikation eines verteilten Ablaufs gliedert sich in drei Teile. In dem Routingteil werden Dienste und Struktur des Ablaufs beschrieben. In dem Datenteil werden die zu dem Ablauf gehörenden Datenobjekte beschrieben. Schließlich werden in dem Terminierungsteil Aktionen zum Abschluß eines verteilten Ablaufs beschrieben.

```
OFFICE_PROCEDURE
  ROUTING
  DATA_SECTION
  TERMINATION
END_OFFICE_PROCEDURE
```

Der bedeutendste Teil ist der Routingteil. Er wird im folgenden genauer beschrieben.

Im Routingteil werden die erforderlichen Dienste beschrieben, die einen verteilten Ablauf ausmachen. Das erfordert zunächst einmal die sprachliche Umsetzung des Ablaufgraphen. Als Beispiel wird der Graph aus Abbildung 7 betrachtet:

```
SERVICE fillexpenseform START
  LINKS_TO
    fillmanagementdata,
    fillprojectdata;

SERVICE fillmanagementdata
  LINKS_TO
    approve
      IF 1000 <
         expenseform.expenses
         <= 10000 ;
    refundexpenses
      IF expenseform.expenses <= 1000;
    toplevelmanagernotification
      If expenseform.expenses > 10000;

SERVICE toplevelmanagernotification
  LINKS_TO refundexpenses;

SERVICE approve
  LINKS_TO refundexpenses;

SERVICE refundexpenses END
```

Im wesentlichen werden die Schlüsselwörter *SERVICE* und *LINKS_TO* verwendet. Hinter dem Schlüsselwort *SERVICE* werden die Dienste des verteilten Ablaufs genannt. Sie können durch die Attribute *START* und *END* näher beschrieben werden. Der Ablaufgraph besitzt, wie im Kapitel 2.2 gesagt, einen ausgezeichneten Anfangs- und Endknoten. Die entsprechenden Dienste werden mit den Attributen *START* und *END* versehen. Während mit den Schlüsselwort *SERVICE* die Knoten eines Ablaufgraphen beschrieben werden, dient das Schlüsselwort *LINKS_TO* zur Spezifikation von Kanten. Hier müssen sich Parallelität und Alternativen spezifizieren lassen. Bei dem Dienst *fillexpenseform* wird zum Beispiel die Parallelität spezifiziert, indem die parallel auszuführenden Dienste einfach mit *Komma* getrennt angegeben werden. Der Dienst *fillmanagementdata* ist ein Beispiel für alternative Ausführung. Die verschiedenen Folgedienste sind alle mit einer Bedingung versehen. Ein Dienst wird nur dann ausgeführt, wenn seine Bedingung zutrifft.

Die Dienste eines Ablaufgraphen müssen nun noch genauer beschrieben werden. Es müssen die Attribute belegt werden, die auch bei den Servern vorhanden sind. Das Laufzeitsystem wertet diese Attribute aus. Dienste werden so

an Server gebunden, daß eine Übereinstimmung von Attributen vorliegt. Darüber hinaus müssen weitere Angaben gemacht werden, um das Binden zu unterstützen.

```
SERVICE manager.fillmanagementdata
  TIMEOUT:
    3 days; CHECKPOINT;
  KEEP_BINDING;
  BINDING_WITH:
    department = Employee.department;
  ATTRIBUTES:
    managementlevel = secondline;

SERVICE print
  TIMEOUT:
    30 minutes ABORT;
  ATTRIBUTES:
    speed = fast;
  MEASURES:
    cost + 0.5 *
    processorload + 0.1 *
    printqueue;
  INSTANCES:
    printer1, printer2;
```

Die Schlüsselwörter haben folgende Bedeutung:

- Mit *TIMEOUT* kann ein Timeout für jede Dienstanfrage spezifiziert werden.
- Mit *CHECKPOINT* kann die Wiederaufnahme eines Ablaufs nach einem Fehler unterstützt werden. Am Schluß einer Dienstausführung mit Kontrollpunkt, wird der gesamte Zustand des Ablaufs gespeichert. Das Umfaßt sowohl die zugehörigen Daten als auch den aktuellen Ausführungsschritt.
- Mit *KEEP_BINDING* wird spezifiziert, das ein einmal verwendeter Server eines Ablaufs immer wieder verwendet werden muß. Dies ist notwendig, wenn lokale Informationen in einem Server vorhanden sind, die mit einem Ablauf zusammengehören.
- Mit *SAME_BINDING_AS: service_name* kann spezifiziert werden, daß ein Dienst an den gleichen Server gebunden werden soll, an den auch schon der vorher ausgeführte Dienst *service_name* gebunden wurde.

2.5 Laufzeitunterstützung

Aufgrund der Sprachunterstützung, die in den vorherigen Abschnitten vorgestellt wurde, kann das Laufzeitsystem nun die Ausführung von Abläufen auf der Basis einer Serverumgebung unterstützen. Das Laufzeitsystem erstreckt sich über das gesamte verteilte System.

Die Ausführung eines Ablaufs gliedert sich in drei Phasen. Es handelt sich dabei um eine Initialisierungsphase, eine Ausführungsphase und eine Terminierungsphase.

Während der Initialisierungsphase wird eine Instanz eines Ablaufs erzeugt. Dafür steht ein Konstruktor zur Verfügung, der das Ablaufobjekt erzeugt und initialisiert. Als Rückgabe liefert der Konstruktor einen eindeutigen Identifikator für das erzeugte Ablaufobjekt. Diesen Identifikator kann man nun der Namensverwaltung des Systems übergeben, um später dann Zugriff zu diesem Objekt zu haben.

Während der Ausführungsphase wird der Ablaufplan des Ablaufs schrittweise abgearbeitet. Das Ablaufobjekt und zugehörige Daten können dafür zwischen den Knoten migrieren. Hier wird die Situation betrachtet, daß gerade ein Teilschritt eines Ablaufs durch einen Server ausgeführt wurde. Es geht nun darum, den Ablauf an einen oder bei Parallelausführung an mehrere Server zu übergeben, die den nächsten Teilschritt ausführen. Folgende Aktionen sind dafür erforderlich:

1. Als erstes werden die nächsten Teilschritte bzw. Dienste bestimmt, die ausgeführt werden müssen. Dazu wird der Ablaufplan herangezogen. Alle Kanten, die vom aktuellen Teilschritt ausgehen, werden betrachtet. Die evtl. vorhandenen Bedingungen an den Kanten werden ausgewertet. Es resultiert eine Menge von Diensten, die im nächsten Schritt ausgeführt werden muß.
2. Für jeden auszuführenden Dienst wird in diesem Schritt ein Server bestimmt. Dieser Voragang heißt Binden. Er wird hier für einen Dienst beschrieben. Für den auszuführenden Dienst werden zunächst alle Informationen für das Binden ausgewertet. Mit Hilfe dieser Informationen ermittelt die Laufzeitverwaltung eine Menge von

Servern, die in Betracht gezogen werden. Mit Hilfe einer Anfrage werden dann Zustand und Attribute der Server ermittelt. Schließlich wird ein Server mit entsprechenden Attributen ausgewählt, der den Ablauf übernimmt. Falls keine Informationen zum Binden vorhanden sind, so wird aufgrund des Servertyps oder dem Namen der Serverinstanz mit Hilfe des Namensdienstes ein Server ermittelt. Ist ein Binden auch dann nicht möglich, so wird die Ausführung des Ablaufs unterbrochen, es wird ein Kontrollpunkt gesetzt und eine Ausnahmebehandlung gestartet. Es ist in diesem Fall ein expliziter Eingriff eines Benutzers erforderlich. Werden an diesem Dienst verschiedene parallele Abläufe zusammengeführt, so darf er nur ausgeführt werden, wenn alle anderen Abläufe schon ausgeführt wurden.

3. Nachdem der nächste Server ausgewählt wurde, migriert das Ablaufobjekt dorthin. Die Ortsänderung wird der Namensverwaltung des verteilten objektorientierten Systems mitgeteilt, um einen Zugriff der Kontrollstationen zu ermöglichen. Splittet sich ein Ablauf in mehrere parallele Pfade auf, so wird das Ablaufobjekt repliziert. Dies erfolgt so, daß es unter allen Kopien immer eine ausgezeichnete Primärkopie gibt. Änderungen dürfen nur auf der Primärkopie vorgenommen werden.

4. Die Eingabeparameter des Dienstes werden instantiiert. der Auftrag wird in die Warteschlange des Servers eingereiht.

5. Nun startet die Ausführung durch den Server. Bei Überschreiten des Timeouts erfolgt eine Meldung an den Server. Der Server muß mit geeigneten Maßnahmen reagieren. Das heißt der Ablauf wird entweder abgebrochen oder auf den letzten Kontrollpunkt zurückgesetzt und unterbrochen.

6. Nach Ausführung des Dienstes, werden die Ausgabeparameter in die Datenslots des Ablaufobjekts geschrieben. Der Ausführungszustand wird im Ablaufplan festgeschrieben.

2.6 Ablaufsteuerungsmodelle

Nachdem in den vorherigen Abschnitten ein Systemansatz zur automatischen Ausführung von verteilten Abläufen beschrieben wurde, werden in diesem und den folgenden Abschnitten einige Aspekte herausgenommen, alternative Lösungen vorgeschlagen und Vergleiche bezüglich gewissen Kriterien angestellt.

2.6.1 Ablaufsteuerung

In diesem Abschnitt soll die Ablaufsteuerung betrachtet werden. In dem beschriebenen System IPSO erfolgt diese dezentral durch das Laufzeitsystem. Jeder Server, der gerade einen Teilschritt eines Ablaufs bearbeitet, muß die Ausführung gemäß des Ablaufplans an den nächsten Server weitergeben. Er wird somit zum Client.

Alternativ kann man die Ablaufsteuerung auch zentral gestalten. Es gibt dann einen Ablaufkoordinator, der alleine für die Synchronisation der Teilschritte verantwortlich ist. Die zentrale Ablaufsteuerung hat den Vorteil, daß nicht jeder Server in der Lage sein muß, Ablaufpläne zu interpretieren. Diese Aufgabe führt nämlich der Ablaufkoordinator durch. In einem System mit zentraler Ablaufsteuerung müssen die Server nur Aufträge ausführen und Fertigmeldungen senden.

Der Vergleich zwischen zentraler und dezentraler Ablaufsteuerung erfolgt anhand des Kommunikationsaufwands und anhand der Verteilung der Kommunikation im System.

Kommunikationsaufwand

Im Falle der dezentralen Ablaufsteuerung bei IPSO erfolgt für jeden Dienstaufruf eine Kommunikation, indem das Ablaufobjekt zum Server hin migriert. Von Statusanfragen, die zuvor erfolgen, wird hier abstrahiert, da sie bei der zentralen Ablaufsteuerung auf die gleiche Art und Weise durchgeführt werden müssen und somit für den Vergleich uninteressant sind. Die

Anzahl der Kommunikationsvorgänge hängt somit direkt von der Anzahl der auszuführenden Teilschritte ab.

Das gilt für sequentiellen, parallelen und alternativen Kontrollfluß. Im Falle von sequentiellem Kontrollfluß ist das offensichtlich. Ein Server, der gerade einen Dienst erbracht hat, liest die Ablaufbeschreibung und ermittelt somit den folgenden Dienst. Nach Auswahl des Servers migriert das Ablaufobjekt dorthin, was einer Kommunikationsaktion entspricht. Im Falle alternativer Ausführung ist dies genau so, wobei lediglich zusätzlich Bedingungen der Kanten ausgewertet werden müssen, was aber kein Kommunikationsaufwand erfordert. Bei paralleler Ausführung muß man zwischen Verzweigung und Verschmelzung unterscheiden. Beim Verzweigen haben wir wieder den Fall, daß für jeden Serveraufruf eine Migration erfolgt. Das Verschmelzen erfolgt so, daß jeder Server die Vollendung seines Schritts im Ablaufobjekt vermerkt und gleichzeitig überprüft, ob die anderen Schritte schon ausgeführt wurden. Ist das der Fall, so ist er für den Aufruf des nächsten Dienstes verantwortlich. Bei dezentraler Ablaufsteuerung ist der Kommunikatoinsfluß also durch die Anzahl der Teilschritte des Auftrags gegeben.

Bei zentraler Ablaufsteuerung hängt der Kommunikationsaufwand ebenfalls von der Anzahl der Teilschritte eines Auftrags ab. Er ist exakt doppelt so hoch, wie bei der dezentralen Ablaufsteuerung. Das ergibt sich durch folgende Überlegung: Für jeden Teilschritt ist eine Startanweisung notwendig, die der Ablaufkoordinator an den Server sendet. Nach Ausführung eines Teilschritts sendet der Server eine Fertigmeldung an den Auftragskoordinator. Der Auftragskoordinator wertet diese Meldungen aus und sorgt für die Ausführung der folgenden Schritte.

Verteilung der Kommunikation

Bei der zentralen Ablaufsteuerung tritt das Problem auf, daß alle Meldungen eines Ablaufs beim zentralen Koordinator eintreffen und daß alle Dienste von diesem angestoßen werden. Damit ist ein Kommunikationsengpaß beim Koordinator verbunden. Bei der dezentralen Ablaufsteuerung tritt dieses Problem nicht auf. Die dezentrale Ablaufsteuerung sorgt für eine räumliche Entzerrung der Kommunikation, während bei der zentralen Ablaufsteuerung ein Kommunikationsengpaß beim Auftragskoordinator auftritt.

2.6.2 Fehlertoleranz

Ein weiterer Aspekt ist der Umgang mit Fehlern. Im beschriebenen System führen Fehler zu einer Ausnahmesituation, die interaktiv aufgelöst werden muß. Es ist jedoch auch möglich, eine Fehlerbehandlung ohne Eingriff des Benutzers durchzuführen.

Man unterscheidet statische und dynamische Fehlerbehandlung. Bei der statischen Fehlerbehandlung wird der Ablaufplan um die Fehlerbehandlung erweitert. Dafür muß es möglich sein, Fehlerfälle zu spezifizieren und entsprechende Behandlungsschritte vorzusehen. Diese Vorgehensweise ist für den Programmierer sehr aufwendig, da im allgemeinen viele Fehlerfälle zu betrachten sind. Er muß obendrein Systemverständnis mitbringen, um Fehlersituationen einschätzen zu können. Systemtechnisch gesehen, also zum Beispiel für das Laufzeitsystem, ergibt sich keine neue Situation. Deshalb wollen wir die statische Fehlerbehandlung auch nicht weiter betrachten.

Ein anderer, interessanterer Ansatz ist die dynamische Fehlerbehandlung. Die Idee ist, daß der Server, der einen Fehler bemerkt, Fehlerbehandlungsschritte in den Ablauf einfügt. Er modifiziert dafür den Ablaufplan. Diese Art der Fehlerbehandlung ist insbesondere bei dezentraler Ablaufsteuerung vorteilhaft, da Fehler dort korrigiert werden, wo sie auch auftreten. Ein fehlerhafter Server kennt am ehesten die Fehlerursache und kann deshalb auch am besten Korrekturschritte in den Ablaufplan einfügen.

Die dynamische Fehlerbehandlung erfordert zwei Schritte. Erstens muß der Ablaufplan um die Korrekturschritte erweitert werden. Zweitens müssen die Server, die die Korrekturschritte ausführen, angestoßen werden.

2.6.3 Lastverteilung

Bei der Lastverteilung geht es darum, eine möglichst gleichmäßige Ausnutzung von Servern zu erreichen. Eine Lastverteilung ist immer dann möglich, wenn es bei der Auswahl eines Servers mehrere Alternativen gibt.

Das vorgestellte System IPSO sieht eine Lastverteilung vor, indem zunächst alle Server ermittelt werden, die einen Ablauf übernehmen können. Mit Hilfe von Statusanfragen wird nun der Zustand jedes Servers ermittelt. Nach einem Optimierungskriterium wird schließlich ein Server ausgewählt.

Genau wie bei der Ablaufsteuerung läßt sich auch bei der Lastverteilung ein zentraler Ansatz realisieren. Bei der zentralen Lastverteilung gibt es eine Zentrale, die die Zustände aller Server verwaltet. Es ist dafür notwendig, daß jeder Server seine Zustandsänderungen an die Zentrale sendet, damit diese informiert ist. Die Zustandsanfragen zur Auswahl eines Servers können nun an die Zentrale gerichtet werden. Anders als bei der zentralen Ablaufsteuerung läßt sich bei der zentralen Lastverteilung durch Bündelung der Anfragen eine Reduzierung des Kommunikationsaufwands erreichen. Kommen für die Übernahme eines Ablaufs n Server infrage, so sind n Anfragen und n Antworten, also $2 \cdot n$ Sendungen notwendig. Bei der zentralen Lastverteilung umfaßt die Kommunikation mit der Zentrale 2 Sendungen. Durch den Aufruf eines Servers entstehen ebenfalls 2 Sendungen, womit sich insgesamt 4 Sendungen ergeben, um einen Auftrag zu übergeben.

2.7 Conclusio

Dieser Beitrag gab einen Überblick über Anforderungen, Probleme und Lösungsansätze zur rechnergestützten Ausführung verteilter Abläufe. Es wurde ein Modell für verteilte Abläufe entwickelt. Eine Sprache zur Repräsentation von verteilten Abläufen und verteilten Systemumgebungen wurde vorgestellt. Es wurden Laufzeitmechanismen zur Verwaltung verteitelter Abläufe in verteilten Systemumgebungen vorgestellt. Im Kapitel 2.6 wurden alternative Lösungsansätze zum System IPSO vorgestellt und diskutiert.

Literatur

[1] Alexander Schill: *Strukturierung und Kontrolle verteilter Büroabläufe.* HMD, Heft 164, 1992, S. 128 - 146.

[2] Alexander Schill: *Distributed system and execution model for office environments.* Computer Communications Journal, Vol. 14, No. 8, Okt. 1991, S. 478 - 488.

[3] Andreas Winckler: *Dezentrale Ablaufsteuerung in verteilten Systemen.* Kommunikation in verteilten Systemen (KiVS93), München, 1993.

3 Fehlertoleranz in verteilten Realzeitsystemen

ANKE OTTO

3.1 Einleitung

Diese Ausarbeitung beschäftigt sich mit Fehlertoleranzmechanismen in verteilten Realzeitsystemen. Es werden zu Beginn die Probleme aufgezählt, die in verteilten Realzeitsystemen in Bezug auf Fehlertoleranz entstehen, und anschließend Lösungsansätze bzw. -mechanismen vorgestellt.

Bevor in den kommenden Abschnitten Probleme bzw. Lösungen von Fehlertoleranzmechanismen in verteilten Realzeitsystemen vorgestellt werden, ist es wichtig, sich über die Bedeutung von Verteilung bzw. Realzeit im Klaren zu sein. Hier wird von folgenden Definitionen ausgegangen:

Verteiltes System Ein verteiltes System ist eine Kopplung von mehreren, möglicherweise unterschiedlichen Rechnern. Das Verhalten des Systems wird durch Algorithmen bestimmt, die so geschaffen wurden, daß sie in der Lage sind, mit verschiedenen Kontollorten und nebenläufigen asynchonen Berechnungen zu arbeiten.

Realzeitsystem In Realzeitsystemen müssen Berechnungen innerhalb festgelegter Zeiten abgeschlossen werden.

3.2 Probleme

Bereits bei den Definitionen von Verteilung und Realzeit wird deutlich, daß Fehlertoleranzmechanismen, die die Eigenschaften dieser unterschiedlichen Systeme gleichzeitig beachten müssen, auf Probleme stoßen. Selbst wenn Realzeitanforderung und Verteilung getrennt betrachtet werden, entstehen Schwierigkeiten bei der Fehlertoleranz.

- Da in Realzeitsystemen die Antwortzeiten innerhalb eines bestimmten Zeitraums liegen müssen, ist es erforderlich, die Fehlerbehandlung so zu minimieren, daß diese Zeiten eingehalten werden können.

- Bei verteilten Systemen ist auf Grund der Kooperation nebenläufiger Prozesse die ursprüngliche Form des Rücksetzens der Prozesse zur Fehlerbehebung zu aufwendig, da diese zum sogenannten `Dominoeffekt` führen würde, bei dem das Rücksetzen eines Prozesses zum Rücksetzen des nächsten etc. führt.

- Aus der Zusammenführung von Verteilung und Realzeitanforderung ergibt sich nun zusätzlich, daß die verteilten Prozesse nicht mehr beliebig blockieren oder abbrechen dürfen, da dies wiederum die Realzeitbedingungen verletzen könnte. Es müssen folglich Methoden der Fehlertoleranz gefunden werden, die diese Probleme berücksichtigen.

3.3 Lösungen

Die Lösungen dieses Ausarbeitung können in zwei Gruppen unterteilt werden.

- Die eine befaßt sich mit der äußeren Betrachtung des Problems, d.h. wann Fehlertoleranzmechanismen wie stark eingesetzt werden können, ohne die Funktionalität des Rechnersystems zu sehr zu beeinträchtigen. Dazu dient die Methode der *Adaptiven Fehlertoleranz*.

- Die andere Gruppe behandelt die inneren Betrachtung, d.h. welche Fehlertoleranzmechanismen in verteilten Realzeitsystemen realisierbar sind. Beispiele hierfür sind:

  1. *Compensating transactions scheme*
  2. *Distibuted recovery block scheme* und die darauf aufbauenden Schemata
  3. *Programmer-transparent coordination*
  4. *Parallel dynamic action model.*

3.3.1 Adaptive fault tolerance (AFT)

In vielen verteilten Realzeitsystemen in denen Fehlertoleranzmechanismen benötigt werden, verändern sich die Anforderungen an das Rechnersystem dynamisch. Diese Veränderungen der äußeren Bedingungen bewirken eine Veränderung der Berechnung, sowie der funktionellen Ziele und Handlungsstrategien. Man stelle sich ein Flugzeug vor, das nach ruhigem Flug einen Luftraum betritt, in dem sich bereits mehrere Flugzeuge befinden und stark wechselnde Lichtverhältnisse herrschen. Dies bewirkt einen extremen Anstieg der Sensordaten, die der Boardcomputer verarbeiten muß, eine Verschärfung der Realzeitanforderungen, um schnell genug reagieren zu können, und verstärkte Kommunikation des Boardcomputers mit entfernten Computersystemen. Wenn nun die Anforderungen an die Fehlertoleranzmechanismen sehr hoch sind, reichen die Ressourcen des Rechnersystems nicht aus diese Mechanismen zu unterstützen, ohne Teile der restlichen Funktionalität zu beschränken. Nun stellt sich allerdings die Frage in welchem Bereich Ressourcen gespart werden können. Soll die allgemeine Funktionalität d.h. die Anzahl der zur Verfügung stehenden Funktionen, beschränkt werden, um so Konsistenz und kurze Antwortzeiten zu erhalten, oder ist es besser Konsistenzbedinungen zugunsten von Funktionalität und kurzen Antwortzeiten zu lockern. Der `Systemressourcenmanager` muß deshalb die Fehlertoleranzmechanismen dynamisch angleichen, was zum Prinzip der *Adaptiven Fehlertoleranz (AFT)* führt, bei der die Fehlertoleranzmechanismen dynamisch an verringerte und sich dynamisch verändernde Ressourcen angepaßt werden.

3.3.2 Anforderungsmodi

Zum besseren Verständnis der*AFT* seien die Anforderungen, die an Fehlertoleranz gestellt werden, in mehrere Modi aufgeteilt. Dem `Fehlertoleranz-Managementsystem (FTMS)`, das die Fehlertoleranzmechanismen auswählt, stehen verschiedene Methoden der Fehlertoleranz für die verschiedenen Modi zur Verfügung. Beim Übergang von einem Anforderungsmodus in den nächsten, gleicht das `FTMS` seine Handlungsstrategie an und geht selbst in einen neuen, für diesen Fall effektivsten Arbeitsmodus über. Man nennt das `FTMS` deswegen auch `adaptive, multi mode FTMS`. Ein Vorteil dieses `multi mode FTMS` ist, daß seine modulare Struktur einfacher zu implementieren ist, als eine monolithische. Die verschiedenen Handlungsmodi, die von einem `adaptiven FTMS` erkannt werden, setzen sich jeweils aus den zwei Komponenten *Umgebungsmodus* und *Fehlerhäufigkeitsmodus* zusammen.

1. **Fehlerhäufigkeitsmodus** Das Auftreten von Fehlern kann mehrere Ursachen haben. Einige Fehler werden intern vom Rechnersystem verursacht, andere durch äußere Gegebenheiten. Bei diesem *Fehlerhäufigkeitsmodus* wird nun die Häufigkeit des Auftretens von Fehlern betrachtet. Man unterscheidet dann zwischen einem Modus mit hohem Fehleraufkommen und einem mit einem geringen.

2. **Umgebungsmodus** Bei dem *Umgebungsmodus* können zwei Modi unterschieden werden: `hard-real-time` und `soft-real-time`. Diese unterscheiden sich in ihren Anforderungen an maximal erlaubte Antwortzeiten.

   `hard-real-time Modus`: hier sind sehr strenge Antwortzeiten gesetzt, deren Einhaltung ständig überwacht werden muß. Deswegen werden im `hard-real-time Modus` `pessimistische` Ansätze verfolgt, d.h. es wird versucht Fehler a priori zu vermeiden. Ein Beispiel hierfür wäre ein Semaphorkonzept, bei dem vor jeder Bearbeitung eine Sperre gesetzt, bzw. nach jeder Bearbeitung wieder gelöscht wird.

   `soft-real-time Modus`: bei diesem Modus ist die Einhaltung der Antwortzeiten nicht so kritisch. Deshalb können in diesem Modus zeitaufwendige Diagnosen und Fehlerbehebungstechniken angewandt werden. Aus diesem Grund werden hier verstärkt `optimistische` Ansätze verfolgt. Das bedeutet die Fehlerbehandlung tritt a posteriori auf. Ein Beispiel für optimistische

Ansätze ist, die Bearbeitung in eine Lesephase (Vorbereitung der Schreibdaten), eine Validationsphase (Test auf Konsistenzverlust) und eine Schreibphase zu unterteilen.

Ein weiteres Beispiel für unterschiedliche Methoden bei `hard bzw. soft-real-time` ist z.B. die Anzahl redundanter Kopien einer Datenbank, die je nach Modus angepaßt werden kann.

Von *mechanisch, parametrischer Adaption (M/P)* spricht man, wenn ein System von einem Arbeitsmodus zum anderen wechselt bzw. in einem bestimmten Mechanismus Parameter verändert, um steigende Handlungskontinuität zu erlangen. Dies kann sowohl durch eine gesteigerte Fehlerrate, als auch durch Übergang der äußeren Bedingungen von einem Modus zum anderen erfolgen. Ein *M/P Adaptionssystem* könnte zum Beispiel aus einem `optimistischen` und einem `pessimistischen` Mechanismus bestehen, dann kann bei einem Übergang von `hard` auf `soft-real-time` wie bereits oben beschrieben verfahren werden. Ebenso könnte bei einem Wechsel der Fehlerrate vorgegangen werden, wobei in einem Zustand geringer Fehlervorkommen der `optimistische`, bei einer hohen Fehlerrate der `pessimistische Mechanismus` besser arbeitet. Bei Steigerung der Fehlerrate wäre irgendwann der Punkt erreicht, an dem das System den Mechanismus wechselt. Da `optimistische Ansätze` gewöhnlich a posteriori arbeiten, müssen sie sowohl über Fehlererkennungs als auch über Fehlerbehebungsmechanismen verfügen. `Pessimistische Ansätze` dagegen arbeiten a priori und versuchen fehlererzeugende Situationen im Voraus zu erkennen und zu vermeiden. Dies ist zwar aufwendiger in der Berechnung, bei hoher Fehlerrate wäre die häufige Fehlerbehebung jedoch teurer. Sowohl der optimistische als auch der pessimistische Ansatz erhalten gleichermaßen die Konsistenz. Es gibt jedoch auch Ansätze bei denen ein Wechsel in der Konsistenzbehandlung stattfindet. Folglich muß der *Ressourcenmanager* auch bei *M/P Adaptivität* entscheiden, ob er bei Funktionalität, Konsistenz oder der Zeitanforderung Ressourcen spart. Dazu muß er wissen, wie wichtig jeder der drei Punkte in den verschieden Phasen ist.

3.3.3 Schemata zur Fehlerbehebung

Im folgenden Abschnitt werden einige Methoden der Fehlerbehebung vorgestellt. Wichtigstes Merkmal dieser Methoden ist die Unterscheidung in *vorwärtslaufende Fehlererkennung (forward recovery)* bzw. *rückwärtslaufende Fehlererkennung (backward recovery)*. *Backward recovery* ist in verteilten Realzeitsystemen nur sehr beschränkt anwendbar. In dem verteilten Realzeitsystem darf weder die Möglichkeit bestehen, daß Fehler von einem Prozeß an den nächsten weitergegeben werden können, noch darf es sich im `hard-real-time Modus` befinden. Diese Bedingungen sind praktisch nie erfüllt. Da einige der *forward recovery Schemata* jedoch auf *backward recovery Schemata* aufbauen, werden auch dieses Schemata hier vorgestellt.

3.3.4 Compensating transaction scheme

Bevor im Folgenden das *compensating transaction scheme* vorgestellt wird, noch eine kurze Erinnerung an die Bedeutung einer Transaktion. Eine Transaktion wird durch folgende Eigenschaften bestimmt:

Atomizität: Bis zum erfolgreichen Abschluß der Transaktion gibt es nach außen keine Wirkung, und entweder alle oder keine Auswirkungen der Transaktion sind permanent

Konsistenz: Ein konsistenter Zustand wird durch eine Transaktion wieder in einen konsistenten Zustand überführt.

Isolation: Alle nebenläufigen Prozesse laufen, als ob sie alleine wären.

Dauerhaftigkeit: Ein durch `commit` bestätigtes Ergebnis einer Transaktion geht nicht mehr verlohren.

Falls während der Berechnung Fehler auftreten, bewirkt ein `abort` ein Rücksetzen der Transaktion. Da die Auswirkungen der Transaktion

anderen Transaktionen / Prozessen erst nach dem`commit` Befehl zugänglich gemacht werden, kann dies bei langen Transaktionen zu einem Verlust von Nebenläufigkeit führen, weshalb das *cancelling transaction scheme* entwickelt wurde. Bei diesem Schema werden die Auswirkungen der Transaktion bereits vor dem `commit` Punkt zugänglich gemacht, d.h. die Eigenschaft der Atomizität wird augegeben. Im Falle eines `abort` müssen folglich alle Transaktionen, die Resultate der abbrechenden Transaktion für ihre Berechnungen benutzt haben, ebenfalls zurücksetzen, was auch zu einer Aufgabe der Isolation führt. Wie oben bereits erwähnt, ist diese Art von `backward recovery` in verteilten Realzeitsystemen nicht sinnvoll. Das *cancelling transaction schema* wurde deswegen dahingehend modifiziert, daß eine Transaktion nach ihrem Rücksetzen den anderen Transaktionen eine *kompensierende Aktion* zur Verfügung stellt, die die Effekte ihrer Berechnungen rückgängig macht. Obwohl diese kompensierende Aktion sowohl `forward` als auch `backward recovery` benutzen kann, findet aus der Sicht des Gesamtsystems `forward recovery` statt. Deswegen ist die Methode für verteilte Systeme im `hard-real-time Modus` geeignet.

3.3.5 Distributed recovery block scheme (DRB)

Ein *recovery Block* wird durch folgende Funktion definiert :

```
ensure T by B1  else by B2.. else by Bn
       else error;
```

Wobei gilt:

1. T: Akzeptanztest
2. B1: primärer Versuchsblock
3. Bk, (2<k<n+1): alternative Versuchsblöcke

- Alle `Versuchsblöcke` werden so entworfen, daß sie das gleiche oder ähnliche Ergebnis liefern.
- Der `Akzeptanztest` besteht aus einem logischen Ausdruck, mit dem die Akzeptanz der, von den Versuchsblöcken gelieferten Ergebnisse, erkannt werden kann.

Bei der Methode des *distributed recovery block* werden verschiedene Algorithmen eines *recovery blocks* auf verschiedenen Prozessoren ausgeführt. Ein Prozessor wird als primär bestimmt. Falls der primäre Prozessor seine Berechnungen nicht innerhalb eines gegebenen Zeitraums erfolgreich beenden kann, wird das Ergebnis desjenigen Prozessors genommen, der seine Berechnungen als erster erfolgreich beendet hat. Auf Grund der verteilten Berechnung kann *DRB* zwar im `hard-real-time Modus` verwendet werden, bei kooperierenen, verteilten Prozessen ist es jedoch nicht anwendbar, da Softwarefehler nicht erkannt werden und sich so im System fortpflanzen können. DRB dient jedoch als Grundlage der folgenden Schemata.

3.3.6 Programmer transparent coordination scheme

Das *programmer transparent coordination scheme (PCT)* ist eine Erweiterung des *DRB*, bei der eine Weitergabe von Fehlern zwischen verschiedenen Prozessen vermieden werden soll. Es arbeitet mit `recovery lines`(Menge von Wiederherstellungspunkten) und `recoverable regions` (Wiederherstellungsregionen).

- Eine `recoverable region` besteht aus einem Berechnungssegment mit der Fähigkeit der Validation und der Wiederherstellung. Eine tt recoverable region sendet Nachrichten an andere Prozesse und widerruft diese, falls die Validation der Berechnung nicht möglich war.

PCT wurde hauptsächlich für den Gebrauch in WAN entwickelt. Es erleichtert Laufzeitkoordination unabhängig voneinander entwickelter Prozesse, in Bezug auf ihre gemeinsame

Fehlerentdeckung und behebung, mit Hilfe eines **intelligenten Systemkerns**. Dieser **intelligente Kern** muß in der Lage sein, die Prozesse so zu unterstützen, daß bei der Fehlerbehebung kein **Dominoeffekt** entsteht. Solch ein Kern muß in der Lage sein, geeignete Wiederherstellungspunkte (**recovery lines**) interagierender Prozesse herzustellen. Protokolle für die Kooperation solcher intelligenter Kerne wurden bereits entwickelt. Wenn nun allerdings die **recoverable regions** verschiedener Prozesse ohne Koordination untereinander entwickelt wurden, besteht die Möglichkeit, daß sich ein Fehler fortpflanzt und nicht innerhalb eines tolerierbaren Zeitlimits entdeckt und behoben werden kann. Deshalb muß bei zeitkritischen Funktionen ein gewisser Grad an Koordination im Design der **recoverable regions** garantiert werden. Eine andere Möglichkeit ist, dem die Nachrichten empfangenden Prozeß mehr Autonomität einzurichten. Nach Erhalt einer widerrufenden Nachricht kann der Prozeß entscheiden, ob er :

1: den Widerruf ignorieren möchte.

2: an den letzten vorher bestätigten Wiederaufsetzpunkt zurücksetzen möchte.

3: eine kompensierende Aktion zum Wiederherstellen des alten Zustands benutzen möchte (siehe Kapitel 3.3.4).

Im Basisschema ist nur Fall 2 enthalten. Es heißt deswegen *PCT/OR* (with Obedient Receivers) wohingegen die zweite Version *PCT/AR* (with Adaptive Receivers) genannt wird. *OR* ist ein **backward recovery** Ansatz, wohingegen bei *PCT/AR* **backward** und **forward** Ansätze möglich sind.

3.3.7 Parallel dynamic aktion scheme

Die Hauptidee dieses Modells ist, genügend Flexibilität zu erhalten, um für die Fehlerbehandlung und Nebenläufigkeitskontolle systemeigene Hilfsmittel, die den jeweiligen Anwendungen angepaßt sind, benutzen zu können. Zur Vermeidung des **Dominoeffektes** basiert dieses Modell auf der Existenz von **recovery lines** und **recovery regions**. Dabei bezeichnet **recovery lines** wieder eine Menge von Wiederherstellungspunkten, während **recovery region**im Unterschied zu den in Abschnitt 3.3.6 beschriebenen **recoverable regions**, für die Aktivität steht, die rückgangig gemacht werden muß, um eine **recovery line** wiederherzustellen. Dies führt zu folgender Definition:

Dynamische Aktion: Eine *Dynamische Aktion* ist eine **Transaktion**, mit dynamischen **recovery regions**, d.h. sie können zur Laufzeit verändert werden. Normalerweise sind die**recovery regions** statisch, d.h. a priori an das Prozeßmodell gebunden. Da die Anpassung an ein enges Prozeßmodell jedoch sehr aufwendig ist, wurden die *Dynamischen Aktionen* entwickelt.

Weitere Eigenschaften sind:

- *Eine Dynamische Aktion* wird durch das **commit**-Protokoll (siehe Kapitel 3.3.4) festgeschrieben.
- Eine abbrechende Aktion beeinflußt die Umgebung durch ein **abort** nicht.
- Außerdem können *DAs* verschachtelt sein, was insbesondere bei langandauernden Aktionen von Vorteil ist, da innere *DAs* unabhängig von den äußeren fehlschlagen.
  - Die Abhängigkeiten von *DAs* werden, in jedem site, in einem **lokalen recovery Graphen** aufgezeichnet, die Menge aller lokalen Graphen ergibt den **recovery Graph (RG)**. Dieser ist eine verteilte Datenstruktur, die die Abhängigkeiten der **recovery regions** der einzelnen *DAs*, wiedergibt. Er wird während des **Reliable Broadcast Protokolls** (alle sites erhalten eine gesendete Nachricht) berechnet.
  - Die Knoten des Graphen heißen **recovery units (RU)**. Sie beschreiben die Wirkung einer DA in einem bestimmten site, wobei für jeden von einer *Dynamischen Aktion* betroffenen site, ein **RU** existiert.

– Insgesamt unterscheidet man zwei Arten von Abhängigkeit zwischen RUs.

1. `DA Abhängigkeit`, sie verbindet alle `RUs` verschiedener sites, die zur gleichen *DA* gehören.
2. `Informations Abhängigkeit` besteht zwischen den `RUs` eines lokalen Graphen und verdeutlicht, daß die Auswirkungen auf eine `RU` von Informationen über andere `RUs` (und somit anderen `DA`) abhängt .

Wenn nun eine bestimmte *DA* ein `abort` sendet, können anhand des `RG` alle *DAs* gefunden werden, die von dieser *DA* abhängen und ebenfalls zurückgesetzt werden müssen. Die zurückgesetzten *DAs* werden in der sogenannten `chase message` bekannt gegeben. Sobald ein site die `chase message` erhält, setzt er alle darin enthaltenen DAs, sowie die davon abhängigen, zurück und erzeugt gegebenenfalls eine neue `chase message`. Im commit Fall muß eine ähnliche Behandlung stattfinden, wobei zu beachten ist, daß entweder alle oder keiner ein commit absetzen kann. Zu diesem Zweck wird ein zentraler Koordinator benötigt, was jedoch wieder zum Problem eventueller Verklemmungen führt.

In Bezug auf Realzeitanforderungen ist es notwendig, zeitliche Redundanz zu vermeiden.

- Zeitredundanz kann insbesondere durch Optimierung der commit-Phase minimiert werden. Hauptprobleme des bisher vorgestellten `commit- Protokolls`, ist die Absprache der DAs im commit und abort Fall, ebenso wie die Möglichkeit des Blockierens, auf Grund zurücksetzender *DAs*. Dies führt zum Entwurf des `redundanten RG (RRG)`, in dem jede verteilte *DA*, alle *DAs* von denen sie abhängt kennt (auch *DAs* anderer Knoten), ebenso wie alle bisher besuchten sites. Die Absprache der *DAs* bei commit / abort ist somit nicht mehr nötig.

- Um die Möglichkeit des Blockierens auszuschließen, wird dem `commit-Protokoll` ein `prepared` Zustand zugefügt. Wenn eine *DA* ihre Berechnungen abgeschlossen hat und sicher ist diese speichern zu können, schickt sie den anderen sites ein `prepared` Signal. Wenn sich alle anderen *DAs* ebenfalls im `prepared` Zustand befinden, antworten sie ihrerseits mit `prepared`, ansonsten mit `failed`. Nur wenn sich alle *DAs* im `prepared` Zustand befinden , kann ein `commit` gesendet werden. Um ein Blockieren zu verhindern, wird allen *DAs* mitgeteilt, falls ein site fehlerhaft ist, so daß sie nicht auf diesen warten müßen. Aus Konsistenzgründen muß allerdings verhindert werden, daß eine *DA* eines fehlerhaften sites, die sich im `prepared` Zustand befand, in `commit` übergeht. Dazu dient eine Erweiteung des Protokolls, in der eine *DA* nur dann ein `commit` senden kann, wenn sie ihre eigene `prepared` Meldung erhält.

- Eine weitere Möglichkeit die Effektivität zu steigern, besteht aus einer Verbindung von *DRB* und*DA*. Im *DRB* werden dazu alle Berechnungen parallel ausgeführt, Ergebnisse werden nur dann sichtbar, wenn alle Akzeptanztests erfolgreich beendet werden konnten. Zur Verbesserung des *DRB* werden alle Berechnungen durch DAs ausgeführt, dies führt zu einem Maximum an Nebenläufigkeit und einem Minimum an Fehlerfortpflanzung. Dieses Schema ist als **Parallel dynamic action scheme** bekannt. Dieses Schema ist sowohl `hard-real-time` als auch `verteilungs`fähig.

3.4 Schlußwort

Bei fast allen Verfahren zur Fehlertoleranzbehandlung in verteilten Realzeitsystemen, die in dieser Ausarbeitung vorgestellt wurden, traten Probleme auf, die entweder die Verteilung oder die Realzeitfähigkeit betrafen. Abschließend kann festgestellt werden, daß lediglich die Verfahren PTC/AR und Parallel dynamic action scheme in verteilten Systemen im hard-real-time Modus anwendbar sind. Dabei sollte allerdings nicht vergessen werden, daß die meisten, der in diesem Artikel vorgstellten Methoden, bisher nur Modelle sind, deren Implementierung noch Gegenstand der Forschung ist. Sie sollten deswegen nur als plausible Konzepte gesehen werden.

Literatur

[1] Kim, KH. und Lawrence. TF: *Adaptive fault-tolerance in complex real-time distributed computer system applications*, Computer communicatios Vol. 15, No. 4 May 1992

[2] Le Lann, G.: *Designing real-time dependable systems*, Computer communicatios, Vol. 15 No. 4 May 1992

[3] Nett, E. and Schuhmann, R.: *Supporting fault-tolerant distributed computations under real-time requirements*, Computer communicatios, Vol 15 No. 4 May 1992

4 Replikation mit schwachen Konsistenzgarantien

CHRISTIAN BECKMANN

4.1 Einführung

Durch die Einführung von Replikationen in einem verteilten System,d.h. die Verteilung von Kopien derselben Daten auf verschiedene Knoten im Rechnernetz, erreicht man eine erhöhte Zuverlässigkeit und Verfügbarkeit der Daten. Trotz dieser Vorteile handelt es sich, insbesondere wenn die Anzahl der Kopien steigt, um eine teure Angelegenheit. Die verwendeten Konsistenzprotokolle, wie z.B. das Zwei-Phasen-Commit, sind relativ aufwendig und verursachen ein hohes Nachrichtenaufkommen auf dem Netz, so daß bei vielen Replikaten sogar die Verfügbarkeit vermindert sein kann.

Nun hat man festgestellt, daß es Anwendungen gibt, die keine perfekte Konsistenz benötigen, wie sie von bisherigen Protokollen gewährleistet wird. Bei diesen Anwendungen reicht es aus, wenn ihnen bekannt ist, wie weit die jeweilige Kopie sich von der aktuellen Version unterscheidet.

Dieser Sachverhalt soll an folgendem Beispiel [1] aus dem Bankwesen verdeutlicht werden: Die einzelnen Bankfilialen führen ihren eigenen Bargeldbestand, der regelmäßig aktualisiert wird. Die Zentrale erhält davon Kopien, bestimmt am Tagesende den Gesamtbestand und investiert ihn über Nacht. Je früher die Bank den Gesamtbestand abschätzen kann, desto höhere Gewinne kann sie durch Investitionen erzielen. Solange die Zentrale also die Summe der Filialbestände in etwa kennt, muß nicht bei jeder Änderung eines einzelnen Filialbestands ein Update durch die Filiale durchgeführt werden. Erst wenn der Filialbestand sich um einen bestimmten Betrag vom letzten der Zentrale bekannten Bestand unterscheidet, muß diese informiert werden, da sich das Investitionsvorgehen ändern würde, falls diese Differenz bei allen Filialen auftreten sollte. Dies kann durch schwach konsistente Protokolle erreicht werden, bei denen nur garantiert wird, daß die lokale Kopie sich innerhalb tolerierbarer Grenzen bewegt.

Im nächsten Kapitel werde ich auf die zugrundeliegenden Konzepte eingehen, die für Replikation mit schwachen Konsistenzgarantien benötigt werden. Danach stelle ich in Kapitel 4.3 die existierenden Protokolle vor und erläutere sie. In Kapitel 4.4 werden sie verglichen und klassifiziert.

4.2 Zentrale Konzepte

Die Kenntnis folgender grundlegender Begriffe und Konzepte ist für das Verständnis des nächsten Kapitels erforderlich.

Konsistenz: Unter dem Begriff der Konsistenz versteht man in verteilten Systemen das Übereinstimmen sämtlicher Kopien derselben Daten zu jedem Zugriffszeitpunkt. Das bedeutet, daß im Falle des Auftretens eines Updates bei einer Kopie alle anderen Kopien zu sperren sind und der Zugriff auf sie erst wieder möglich ist, nachdem der Update auch bei ihnen durchgeführt wurde.

Nachrichtenübermittlung: Es gibt verschiedene Nachrichtenübermittlungsverfahren. Bei der zuverlässigen Übermittlung sichert das zugrundeliegende Übertragungsprotokoll zu, daß eine gesendete Nachricht garantiert beim Empfänger ankommt, was bei einer unzuverlässigen Übermittlung nicht der Fall ist. Beim Atomic Broadcast (Rundruf) wird die Nachricht vom sendenden Knoten an alle anderen Knoten weitergeleitet.

Nachrichtenordnung: Unter der Nachrichtenordnung versteht man die zugesicherte Reihenfolge, in der Nachrichten beim Empfänger ankommen. Bei einer totalen Ordnung erfährt der Empfänger von den ausgeführten Updates in genau der Reihenfolge, in der sie stattfanden. Er kann also einer Nachricht entnehmen, ob er ein Update noch nicht empfangen hat. Bei einer kausalen Ordnung ist nur garantiert, daß der Empfänger einer Nachricht über die Updates informiert ist, die unbedingt vor der empfangenen Nachrichten ausgeführt sein müssen.

Prozeßgruppe: Bei einer Prozessgruppe handelt es sich um den Zusammenschluß von Prozessen, die Zugang zu den selben Daten haben und die über das Netz miteinander kommunizieren können. Dabei kennt jeder Prozeß alle Mitglieder der Gruppe. Bei einigen Implementierungen ist ihm sogar die Topologie des Netzesbekannt.

4.3 Existierende Protokolle

4.3.1 Zentralisierte Protokolle

Bei dieser am einfachsten zu implementierenden Art von Diensten, die in weiträumig verteilten Netzen angeboten werden, erlaubt der Server den Clients, sich anzuschließen und mit ihm zu kommunizieren (jede Nachricht wird gesendet oder empfangen). Dieses System ist aber nur so schnell wie der Rechner die Nachrichten verarbeiten kann und so verfügbar wie das Netz zwischen ihm, den Servern und den Clients. Bereits vorhandene Systeme sind unter anderem das WAIS Text-Retrieval System, das verteilte Hypertext System im World Wide Web und der Archie FTP Location Dienst.

Um die Fehlertoleranz und Verfügbarkeit dieser Server zu erhöhen, gibt es den *primary copy* oder *master slave* Ansatz. Bei diesem Ansatz wird die Technik der *quasi copy* [1] benutzt. Hierbei führen die Kopien eine Übereinstimmungsbedingung (*coherency condition*); bei diesem Prädikat kann es sich um eine Verzögerungs-, Versions- oder arithmetische Bedingung handeln. Das System sorgt dafür, daß dieses Prädikat nicht verletzt wird – entweder durch den zentralen Knoten, der im Falle eines Updates für dessen Verbreitung verantwortlich ist, oder durch periodisches Auffrischen von Seiten der Clients. Die Überwachung durch den zentralen Knoten ist erforderlich, da nur ihm (und nicht den Clients) bekannt ist, bei welchen Knoten das Update benötigt wird.

`Beispiel`: Der Knoten X verwalte die Daten A, der Knoten Y die Daten B. Deren Konsistenzbeziehung sei $A \leq B$. X halte eine quasi-copy B' von B, Y eine quasi-copy A' von A. Die Konsistenzbedingung habe die Form $\mid A \Leftrightarrow A' \mid \leq 5$ und $\mid B \Leftrightarrow B' \mid \leq 5$. Die Transaktion T_1 auf X lese A und B', ändere A in A_{new}. Es muß gesichert sein, daß immer $A_{new} \leq B$ gilt. Daher führt man die Limits A_l und B_l ein mit:

$$A \leq A_l \leq B_l \leq B$$

Sollte X A_l erniedrigen oder Y B_l erhöhen, handelt es sich um *sichere* Änderungen, um unsichere, falls X A_l erhöht oder Y B_l erniedrigt. Die Limits seien gesetzt als

$$A_l = A + 0.5(B \Leftrightarrow A) = B_l$$

. Zu Beginn seien $A, A' = 0$, $B, B' = 20$ und demnach $A_l, B_l = 10$. Erst wenn auf X A größer als 5, aber nicht größer als 10 ($A < B_l$) wird, muß Y informiert werden (wegen $\mid A \Leftrightarrow A' \mid \leq 5$).

4.3.2 Konsistente Replikationsprotokolle

Einem Client stehen ein oder mehrere Server bzw. Replikate gegenüber. Jede Lese-/Schreibe-Operation ist atomar [2] und konsistent. Die möglichen Server-zu-Server-Operationen sind das Hinzufügen und Entfernen von Kopien sowie die Handhabung von Kopieverlust und -fehlern.

Es gibt folgende Klassen von Replikationen:

- `available copy` : Ein Client kann von einer beliebigen Kopie lesen, während Update-Operationen allen erreichbaren Kopien mitgeteilt werden müssen. Voraussetzung hierfür sind ein unteilbares Netz, damit alle Kopien von einem Update erfahren, sowie eine zuverlässige Nachrichtenübermittlung.

- `voting` : Jeder Kopie ist eine bestimmte Anzahl von Stimmen (*votes*) zugewiesen. Eine Operation sammelt Stimmen von Kopien und kann bei Erreichen einer bestimmten Zahl ausgeführt werden. Ein voting-Protokoll ist zum Beispiel das majority-consensus-voting Protokoll

- `hybrid` : Hierbei handelt es sich um eine Mischform aus den beiden obigen, wie zum Beispiel das virtual-partition Protokoll

Diese Protokolle werden bei einer großen Anzahl von Kopien ineffizient und benötigen einen gewaltigen Kommunikationsoverhead auf Wide-Area-Netzen.

4.3.3 Orca

Hierbei handelt es sich um eine Sprache zur verteilten Programmierung [3]. Replikation wird als primäres Programmierparadigma in eng gekoppelten Sytemen verwendet. Die verteilten Anwendungen bestehen aus Termen von geteilten Datenobjekten. Deren Update-Operationen sind serialisierbar. Es gibt drei Implementierungen von Laufzeitsystemen. Bei der ersten ist jeder Prozeß *multithreaded*, bestehend aus einem Object Manager Thread und einigen Anwendungsthreads. Muß ein Thread eine Update-Operation ausführen, sendet er eine Nachricht an jeden Prozeß und blockiert. Der Object Manager führt die Operationen in der Reihenfolge des Empfangens durch. Sobald die Operation bei dem Prozeß ausgeführt wurde, der sie aufrief, wird der Thread wieder aufgeweckt. Diese Implementierung baut auf einem zuverlässigen Atomic-Broadcast-Protokoll auf.

Bei der zweiten Implementierung führt jeder Prozeß einen Zähler von selbstausgesendeten Nachrichten sowie einen Vektor der Nachrichtenzähler aller Prozesse. Ein Prozeß hängt an jede ausgehende Nachricht seinen Nachrichtenvektor. Ein Prozeß, der eine Nachricht empfängt, erhöht den Zählerwert des sendenden Prozesses und vergleicht die Vektoren. Bei Unterschieden kann er mit den Prozessen, bei denen der Wert differiert, kommunizieren, um die verlorengegangenen Nachrichten zu erfragen. Da nur durch den Empfang von Nachrichten der Verlust von anderen Nachrichten festgestellt werden kann, sendet das Laufzeitsystem periodisch *dummy messages*. Damit ist ein zuverlässiger Nachrichtenempfang gewährleistet, obwohl sich diese Implementierung nur auf ein unzuverlässiges Multicast-Protokoll stützt.

Die dritte Variante,die Amoeba RPC Implementierung, erreicht Serialisierbarkeit durch ein Primary Copy Protokoll.

4.3.4 Isis

Bei Isis gehören Prozesse als *member* oder als *client* zu einer oder mehreren Prozeßgruppen. Innerhalb einer solchen Gruppe existiert eine konsistente Sicht. Das Gruppen-Multicast benutzt entweder ein totalgeordnetes oder ein kausalitätsgeordnetes Protokoll. Isis ist für kleinere Systeme geeignet, die häufig einen konsistenten, wechselwirkenden (interaktiven) Dienst anbieten müssen.

4.3.5 Epsilon Serialisierbarkeit (ESR)

Dieses Korrektheitskriterium zur Transaktionskontrolle eignet sich nicht nur für verteilte Systeme. Das ESR Kontrollprotokoll erlaubt den Anwendungen, die Inkonsistenzhöhe zu begrenzen, die eine Transaktion wahrnehmen kann. Jede Update-Operation entspricht einer vollständigen Transaktion. Die einzelnen Transaktionen können eine bestimmte Anzahl von Sperrvorrichtungen (*locks*) auf ein Objekt verlangen. Sollte eine Transaktion mehr locks verlangen als der momentane Grenzwert des Objekts beträgt, wird sie blockiert. Es gibt folgende Update-Variationen:

Ordered: Die Updateoperationen werden bei jedem Prozeß in der gleichen Reihenfolge ausgeführt (Grundlage ist eine totalgeordnete, zuverlässige Nachrichtenübermittlung).

Read independend timestamped: nur für Operationen, die Information anhängen oder alte Versionen überschreiben (ungeordnete, zuverlässige Nachrichtenübermittlung).

Optimistic: alle Operationen werden sofort ausgeführt; dann versucht man mittels Kompensation die Folgen der Transaktionen zu beheben, die eine zu große Inkonsitenz bemerkten oder verursachten.

4.3.6 Psync

Dieses System ist ein vollständiges Gruppenkommunikationssystem, das auf dem Psync Kommunikationsprotokoll aufbaut. Es bietet

konsistente, atomare Nachrichtenübermittlung und stellt Operationen zur Gruppenmitgliedschaft zur Verfügung.

Prozesse beginnen die Kommunikation, indem sie sich einer Gruppe anschließen. Dies nennt man *Konversation*. Da ein Prozeß mehreren Gruppen angehören darf, ist nur innerhalb einer Gruppe (Kausal-)Konsistenz zugesichert. Jeder Nachricht, die mit einem Identifikator versehen ist, hängt das Protokoll Kausalinformation an. Diese besteht aus den Identifikatoren der Nachrichten, von denen die gesendete Nachricht abhängt. Daher benötigt jede Nachricht O(n) Speicherplatz. Die einzelnen Nachrichten werden mittels Best-Efford-Multicast gesendet. Mit Hilfe der Kausalinformation erstellt jedes Gruppenmitglied einen Beziehungsgraphen der Nachrichten. Ein Empfänger kann den Verlust einer Nachricht feststellen, da einige andere sie als Vorgänger angeben werden, und vom Sender eine Kopie anfordern. In einem variierten Weak-Consistency Protokoll wird der Overhead der Kausalinformation beseitigt. Hier ist jede Nachricht nur mit einer einzigen Zeitmarke versehen, die gleichzeitig ihr Identifikator ist. Eine Nachricht hängt dann von all den Nachrichten ab, deren Zeitmarke kleiner ist als die eigene.

4.3.7 Zuverlässiges Multicast Protokoll

Dieses von Garcia-Molina und Kogan entwickelte Protokoll ist dem Orca Protokoll sehr ähnlich. Die Nachrichten enthalten Sequenzinformationen,durch die ein Empfänger den Verlust von früheren Nachrichten feststellen kann. Zur Hilfe werden *dummy-Nachrichten* gesendet. Um nach dem Feststellen des Verlusts einer Nachricht eine Kopie davon von einem anderen Server, auch *Principal* genannt, anzufordern, nutzt diese Protokoll die Netzwerktopologie aus. Es erstellt eine Hierarchie der Principals auf. Kriterien hierfür sind die Kommunikatonsentfernung zwischen Hosts oder die Anzahl der Netzwerk-Links. Bei einem gut aufgebauten Baum wird so der Netzverkehr minimalisiert. Das Problem hierbei besteht darin, daß entweder ein zentraler Server oder die Einbeziehung der gesamten Gruppe nötig ist, um den Baum zu aktualisieren, wenn ein Principal die Gruppe verläßt oder sich ihr anschließt.

4.3.8 OSCAR

Die Architektur OSCAR (Open System for Consistency And Replication) [4] für schwachkonsistente Replikation bietet eine zuverlässige,schließlich eintretende Nachrichtenübermittlung an mit einer Vielfalt von Nachrichtenreihenfolgen. Ihre Besonderheit besteht darin, daß jedes Replikat mit einem *Replikator* und einem *Mediator* ausgstattet ist. Bei jedem Update sendet der Replikator, an dem sie ausgeführt wird, den anderen Replikatoren eine Nachricht, die eine Versionsnummer und eine Zeitmarke enthält. Dadurch werden die Updates in der richtigen Reihenfolge bei allen Replikaten ausgeführt. Periodisch holt sich ein Master Mediator von den Replikatoren die Versionsvektoren und kombiniert sie. Das Ergebnis wird an alle Replikatoren gesendet, damit diese von möglicherweise verloren gegangen Updates erfahren und sich auf den aktuellen Stand bringen können.

Falls das Netz geteilt wird, wird ein Mediator Master in jedem Teil. Da die Mediators geordnet sind, wird ein weniger-priorisierter Mediator aktiv, wenn er in einem bestimmmten Zeitabschnitt keine Meldung von einem höherpriorisierten erhält. Ist der Netzzusammenhang wieder hergestellt, kehren die Mediatoren zur alten Ordnung zurück.

Der Nachteil dieser Architektur ist der hohe Netzverkehr, der dadurch entsteht, daß die Replikatoren bei jeder Nachricht multicasten und die Mediatoren ihnen regelmäßig Nachrichten senden müssen.

4.3.9 Lazy Replication

Dieses System [5] benutzt Nachrichtenzähler und -vektoren ähnlich wie das Orca Protokoll. Es stelt drei Arten von Update-Operationen zur Verfügung:

1. Client-ordered Methode (für z.B. senden, lesen) Diese Methode bietet zwei Operationen an: *update* und *query*. Bei jedem Aufruf einer Update-Operation liefert der Service einen *unique identifier*, uid, zur Bezeichnung des Aufrufs. Jede Operation führt als Argument einen *label*, eine Liste von uids, in der die Update-Operationen stehen, die der Ausführung der Operation vorangehen müssen. Die Frage-Operation liefert einen aktuellen Label sowie einen Wert (*value*).

Der Client erfährt von Operationen anderer Clients durch direkte Kommunikation oder durch Ansehen des Service Status. Eine Möglichkeit zur Handhabung der Labels ist das Anhängen des Labels an alle Nachrichten durch das System und anschließendes Verschmelzen bei Empfang. Die Clients können sie auch explizit kontrollieren, wodurch eine bessere Performance erreicht wird. Hierbei entscheidet der Client selber, welchen Label er dem Service sendet. Weiß er zum Beispiel von einem Update u und möchte eine Operation ausführen, die nicht nach u stattfinden muß, enthält sein Label die uid von u nicht. Der Label bietet also die Möglichkeit der expliziten Angabe von Abhängigkeiten. Man bezeichnet diese Art des Updating auch als Ausführen von Operationen in der Vergangenheit.

Zunächst führe ich einige Abkürzungen ein , bevor ich auf die Spezifikation eingehe: q sei eine query; $q.prev$ der Eingabelabel; $q.op$ die Frageoperation; $q.value$ der Ergebniswert ; $q.newl$ der Ergebnislabel, u ein update; $u.op$ die Updateoperation; S(e) die Menge der Ereignisse, die dem Ereignis e in der Ausführungssequenz E vorangehen; $S(e).label$ die Menge der u in S(e). Nun die Spezifikation:

1. $q.prev \subseteq q.newl$: Der zurückgegebene Label umfaßt alle dem Client bekannten Updates sowie noch möglicherweise hinzugefügte.

2. $u.uid \in q.newl \Longrightarrow \forall$ updates v mit $dep(u,v)$: $v.uid \in q.newl$: Der zurückgegebene Label ist abhängigkeitsvollständig, d.h. ist u durch den Label identifiziert, dann auch alle Update v, von denen u abhängt. $dep(u,v) = (v.uid \in u.prev)$

3. $q.value = q.op(Val(q.newl))$: Der Wert muß in einem Zustand berechnet werden, der nach der Ausführung der durch den Label gegebenen Updates erreicht wird, konsistent zur Abhängigkeitsrelation: $dep(u,v) \Longrightarrow v.op$ wird vor $u.op$ ausgeführt.

4. $q.newl \subseteq S.label$: Alle identifizierten Updates haben sich ereignet.

Die Architektur sieht wie folgt aus: Jeder Client betreibt einen *front end* an seinem Knoten. Dieser sendet eine Nachricht an einige benachbarte Replikate, wenn der Client eine Serviceoperation aufruft. Das Replikat führt die Operation durch und sendet eine Antwort. Untereinander kommunizieren die Replikate mittels langsamen (lazy) Austausch von *gossip*. Da der front end jede Nachricht mit einem *cid*, unique call identifier, versieht, kann das System Fehler durch verlorengegangene, verspätete, duplizierte oder in falscher Reihenfolge eintreffende Nachrichten verhindern.

Zum Test dieser Methode wurden Probeläufe einer Anwendung mit drei Repliken durchgeführt und mit derselben Anwendung ohne Repliken verglichen. Dabei zeigte es sich, daß das häufigere Senden von gossip die Dauer einer Operation senkte. Dadurch wird außerdem die Zuverlässigkeit erhöht, da das Auftreten von Fehlern verhindert wird, die durch eine Teilung des Netzes hervorgerufen werden könnten. Der Nachteil ist der erhöhte Netzverkehr, der durch vielen Gossip hervorgerufen wird.

2. Server-ordered-Methode und

3. globally-ordered Methode (für z.B. das Hinzufügen oder Löschen eines Benutzers) Gegenüber der client-ordered Spezifikation ändert sich nur die erste Klausel sowie die Abhängigkeitsbeziehung. L(e) enthalte die neuesten global-geordneten Operationen, die dem Ereignis e vorausgehen.

1.

$$q.prev \cup L(q).label \subseteq q.newl :$$

Anfragen spiegeln die neuesten global-geordneten Updates wieder

$$dep(u,v) : IF globally \Leftrightarrow ordered(u) THEN v \in S(u)$$

$$ELSE (v.uid \in u.prev) \vee$$

$$(server \Leftrightarrow ordered(u) \& server \Leftrightarrow ordered(v)$$

$$\& v.uid < u.uid)$$

4.3.10 Epidemie-Replikation

Die einzelnen Knoten (*sites*) der verteilten Datenbasis befinden sich immer in einem der drei Zustände s, susceptible (anfällig), i, infective (infiziert), r, removed (entfernt). Hat ein Knoten noch nichts von der Updateoperation u erfahren, ist er noch anfällig. Ist er im Besitz von u und möchte u weiterleiten, ist er infiziert. Verläßt er diesen Zustand, ist er entfernt [6].

Der spezielle Wert einer Kopie k ist eine Funktion, die von dem vorherigen Wert $v \in$ V und einer Zeitmarke $t \in$ T (T totalgeordnet) abhängt: $k.ValueOf = (v,t)$

Grundtechniken

Direct Mail: Führt ein Knoten einen Update durch, sendet er die Nachricht (update,(v,t)) an alle anderen Knoten. Der Empfänger s einer solchen Nachricht setzt seinen Wert auf (v,t), falls $s.ValueOf.t < t$ ist. Diese Methode hat einen sehr hohen Netzverkehr zur Folge.

Anti-Entropy: In bestimmten Abständen nimmt ein Knoten s_1 mit einem zufällig bestimmten Knoten s_2 Kontakt auf und vergleicht sich mit ihm. Es gibt drei Methoden für die Updateoperation $ResolveDifference[s_1, s_2]$:

1. `push` : Ist die Zeitmarke von s_1 größer als die des Kommunikationspartners, dann wird dessen Wert geändert und er übernimmt den Stand von s_1.

   $$IF s_1.ValueOf.t > s_2.ValueOf.t THEN$$

   $$s_2.ValueOf.t \leftarrow s_1.ValueOf.t$$

2. `pull`: Ist die Zeitmarke von s_1 kleiner als die des Kommunikationspartners, dann wird der eigene Wert geändert und s_1 übernimmt den Stand von s_2.

   $$IF s_1.ValueOf.t < s_2.ValueOf.t THEN$$

   $$s_1.ValueOf.t \leftarrow s_2.ValueOf.t$$

3. `push-pull` : Verbindung der Operationen push und pull

Sämtliche Knoten sind in Abhängigkeit von der Methode proportional zur Knotenzahl n in bestimmter Zeit infiziert (push: $log_2(n) + ln(n) + O(1)$ für große n).

Häufig wird diese Technik mit der ersten gemischt (einige Knoten werden direkt angemailt, einige infiziert). Die Wahrscheinlichkeit, daß ein Knoten nach anti-entropy Zyklus i noch nicht infiziert ist, konvergiert schnell gegen 0.

Um den Netzverkehr zu senken, gibt es folgende Variation: Anstelle der ganzen Kopie werden zunächst nur Checksummen, häufig mit einem zusätzlichen Zeitfenster versehen, verglichen. Wenn diese nicht übereinstimmen, wird die Kopie ausgetauscht.

Rumor Mongering: Zu Beginn sind die Kopien alle *inactive*. Sobald eine Knoten n von einer Updateoperation weiß, wird sie *active* und versucht, dieses Gerücht (*Rumor*) an einen anderen Knoten l weiterzugeben. Kennt l das Gerücht schon, wird n mit Wahrscheinlichkeit 1/k inaktiv. Messungen haben ergeben, daß 20% der Replikate bei einer Abbruchwahrscheinlichkeit 1/k = 1 am Ende der Epidemie nicht über den Update informiert sind, bei einer Wahrscheinlichkeit von 1/2 sogar nur 6%. Es besteht auch die Möglichkeit, das Interesse `blind` zu verlieren, d.h. unabhängig davon, ob der Kontaktknoten das Gerücht kennt oder nicht.

Der prinzipielle Unterschied zu Anti-Entropy besteht darin, daß beim Rumor Mongering Garantie gegeben ist, daß jeder Knoten von dem Gerücht erfährt.

Es gibt drei interessante Werte:

- `Rest`: Wert von s, wenn keine infizierten Knoten mehr existieren, wenn also die Epidemie beendet ist.

- `Verkehr` :

$$m = \frac{Gesamtverkehr}{Knotenzahl}$$

- `Verzögerung` : Zeit vom Beginn eines Updates bis zur Ankunft bei einem Knoten. Es werden zwei Zeiten berücksichtigt: t_{avg}, die Durchschnittsverzögerung, und t_{last}, die Zeit, bis der letzte Knoten vom Update erfährt.

Anstelle der Wahrscheinlichkeit 1/k kann auch ein Zähler k eingeführt werden. Nach k erfolglosen Versuchen wird das Gerücht nicht mehr vom Knoten verbreitet.

Desweiteren kann man mit einem Verbindungslimit arbeiten sowie mit `Hunting`. Hierbei versucht ein Knoten nach einer Verbindungsabweisung noch im selben Zyklus einen anderen Knoten zu erreichen.

Löschen von Kopien Das Löschen eines Stückes einer Kopie gestaltet sich schwieriger als das Updating. Das Entfernen des Stückes der lokalen Kopie reicht nicht aus, da durch die anti-entropy Sitzung möglicherweise alte Kopien zu dem Knoten gelangen, an dem sie bereits vernichtet wurden. Daher werden diese Stücke mit einem *death certificate* versehen und verbreitet. Nach einer gewissen Zeitspanne τ_1 werden die Stücke der lokalen Kopie gelöscht. τ_1 sollte mindestens die Ausbreitungsgeschwindigkeit der Epidemie betragen, um die Wahrscheinlichkeit zu erhöhen, daß alle Stücke gelöscht werden. Je größer τ_1 gewählt wird, desto geringer ist das Risiko, daß eine gelöschte Kopie wieder in Umlauf gerät, aber umso mehr Speicherplatz wird für sie verbraucht. Ein weiteres Problem ist die Möglichkeit, daß ein Knoten für Tage oder Wochen nicht am Netz hängen kann. Um dadurch entstehenden Fehlern entgegenzuwirken, werden einige death certificates nicht nach τ_1 gelöscht, sondern sie werden *schlafend* (dormant) und erst nach einer größeren Zeitspanne τ_2 (30 Tage bis mehrere Jahre) gelöscht. Welche Kopien in den Zustand schlafend übergehen, ist einer der delete-Nachricht angehängten Liste zu entnehmen. Sobald ein veraltetes Update auf ein dormant death certificate trifft, wechselt dieses vom Zustand schlafend in aktiv und wird wieder verbreitet. Zusätzlich werden die death certificates mit einer Aktivierungszeit versehen, um zu verhindern, daß durch sie ein Update vernichtet wird, das vor dem Löschen der Kopie auszuführen ist.

Räumliche Verteilung Um ein günstiges Zeit- und Netzverkehrverhalten zu erreichen, ist es nötig, die Netztopologie zu kennen, um den Partner für den anti-entropy Austausch auszusuchen (Bei einem in Amerika und Europa gleichverteilten Netz und nur ein einer Transatlantikverbindung sollte diese möglichst entlastet werden, d.h. nur ihre Endknoten und deren nächste Nachbarn sollten sie benutzen). Man wählt bei linearen Netzen einen Knoten der Entfernung d mit der Wahrscheinlichkeit proportional zu d^{-a}. Der Netzverkehr variiert dann zwischen O(n) für $a < 0$ und O(1) für $a > 2$. Die Konvergenzzeiten sind polynomial zu n für $a > 2$ und zu log n für $a < 2$. So wählt man eine d^{-2}-Verteilung zum Weiterleiten.

In einem realen Netz ist die Generalisierung der d^{-2} Verteilung schwierig. Zunächst ging man zu einer d^{-2D} Verteilung über, wobei D die Dimension der Maschen *mesh* ist. Der Nachrichtenverkehr fällt auf O(log n) bei einer Verteilung von $d^{-(D+1)}$. So kam man darauf, jedem Knoten s die Kommunikationspartner in Abhängigkeit einer Verteilungsfunktion $Q_s(d)$ zu wählen, wobei $Q_s(d)$ die Summe der Sites mit einer Entfernung kleiner gleich d ist. In einem D-dimensionalen Netz ist $Q_s(d)$ gleich $O(d^D)$, so daß eine Verteilung $1/Q_s(d)^2$ gleich $O(d^{-2D})$ ist.

In Tabelle 1 ist die Leistung für 1000 Knoten ohne Berücksichtigung der räumlichen Verteilung unter Verwendung des Zählers k und Abbruch bei erfolgreichem Austausch abgebildet, in Tabelle 2 für die selbe Realisierung mit blinden Abbruch.

Beim Rumor Mongering hängen die idealen Werte für verschiedene a unter Berücksichtigung der räumlichen Verteilung auch von dem Wert des Counters k ab, wie in Tabelle 3 zu sehen ist. Je nach Art der räumlichen Verteilung ändert sich der Zählerwert k, um die besten Werte für t_{last}, t_{avg}, ... zu erzielen.

spatial dist.	k	t_{last}	t_{avg}	compare		update	
				avg	bushey	avg	bushey
uniform	4	7.83	5.32	8.87	114.0	5.84	75.87
$a = 1.2$	6	10.14	6.33	3.20	18.0	2.60	17.25
$a = 1.4$	5	10.27	6.31	2.86	13.0	2.49	14.05
$a = 1.6$	8	11.24	6.90	2.94	9.80	2.27	10.54
$a = 1.8$	7	12.04	7.24	2.40	5.91	2.08	7.69
$a = 2.0$	6	13.09	7.74	1.99	3.44	1.90	5.94

Tabelle 3: push-pull mit rumor mongering

Zähler k	Rest s	Verkehr m	Verzögerung t_{avg}	t_{last}
1	0.176	1.74	11.0	16.8
2	0.037	3.30	12.1	16.9
3	0.011	4.53	12.5	17.4
4	0.0036	5.64	12.7	17.5
5	0.0012	6.68	12.8	17.7

Tabelle 1: Abbruch bei erfolgreichem Austausch

Zähler k	Rest s	Verkehr m	Verzögerung t_{avg}	t_{last}
1	0.960	0.04	19	38
2	0.205	1.59	17	33
3	0.060	2.82	15	32
4	0.021	3.91	14.1	32
5	0.008	4.95	13.8	32

Tabelle 2: blinder Abbruch

4.3.11 Timestamped Anti-Entropy

Eine Weiterentwicklung der Epidemie-Replikation ist das Timestamped Anti-Entropy Protokoll [7] von Golding für Gruppenkommunikation. Dieses Protokoll ermöglicht einem Prozeß unter anderem festzustellen, wann alle Gruppenprozesse eine Nachricht erhalten haben. Um das zu realisieren, hält jeder Prozeß drei Datenstrukturen:

- `Nachrichten Log` : Er enthält alle Nachrichten, die der Prozeß empfangen hat. Eine Nachricht wird daraus gelöscht, wenn sie von allen Prozessen der Gruppe empfangen wurde.
- `Summary Timestamp Vektor` : In ihm sind die Zeitmarken enthalten, bis zu denen der Prozeß die Updates der anderen mitbekokmmen hat.
- `Acknowledgement Timestamp Vektor` : In ihm sind die Zeitmarken enthalten, bis zu denen die eigenen Updates von den anderen Prozessen mitbekommen wurden.

Will ein Prozeß p Mitglied einer Gruppe werden, sucht er sich einen Sponsorprozeß s, von dem er die Daten, Sichten und den Gruppenstatus übernimmt. Zum Verlassen einer Gruppe ändert p seinen Status in `leaving` und wartet, bis alle Gruppenmitglieder die Änderung mitbekommen haben. Während dieser Zeit nimmt p nur an anti-entropy Sitzungen teil, sendet aber selbst keine weiteren Nachrichten.

4.4 Diskussion

Ein Vergleich der verschiedenen Protokolle gestaltet sich schwierig, da sie meistens für bestimmten Anwendungen in vorgegebenen Umgebungen entwickelt wurden. Eine Möglichkeit besteht im Betrachten der Garantien, die von den Protokollen zugesichert sind. Diese sind:

1. `Nachrichtenzustellung`: Es gibt zuverlässige und bestmögliche.
2. `Zustellungsordnung`: Die Ordnungen sind total, kausal, nach Prozeß oder überhaupt nicht geordnet.

Nachrichten Ordnung	Zustellung zuverlässig wechselwirkend	zuverlässig schließlich	unzuverlässig
ungeordnet	reliable multicast	anti-entropy OSCAR	direct mail rumor mongering
kausal	lazy replication ISIS CBCAST Psync	lazy replication	
total, nichtkausal	ϵ-Serialisierbarkeit	OSCAR	
total, kausal	ISIS ABCAST ORCA zentralisierte konsistente		

Abbildung 9: Protokoll-Klassifikation

Atomic Broadcast	Nachrichtenübermittlung zuverlässige	unzuverlässige
Orca (1) Psync	konsistente Replikation Isis ϵ Serialisierbarkeit	zuverlässiges Multicast Orca (2) OSCAR Lazy Replication Epidemie Timestamped Anti-Entropy

Abbildung 10: Systemvoraussetzungen

3. `Zustellungszeit`: Sie lassen sich einteilen in synchron, gebunden und sich ergebend.

   Abbildung 9 zeigt diese Klassifizierung. Einige Quadrate bleiben leer, da die entsprechneden Protokolle nicht realisierbar sind. Es kann zum Beispiel bei unzuverlässiger Zustellung nur eine ungeordnete Nachrichtenreihenfolge geben, da eine totale oder kausale nicht gewährleistet werden kann. So ist auch bei einer zuverlässigen schließlichen Zustellung keine total kausale Ordnung möglich.

   Desweiteren sind Klassifizierungen möglich nach den Systemvoraussetzungen, wie in Abbildung 10 ,oder nach den Aufräumverfahren. Hierbei wird danach unterschieden, wann ein Knoten die Information über einen ausgeführten Update vergessen kann. Es gibt die Möglichkeiten, sofort nach Erhalt der Nachricht zu vergessen (wie bei den meisten Protokollen, z.B. bei Isis, Orca oder Psync), nach einem bestimmten Zeitraum – wenn deterministisch zugesichert ist, daß dann der Update bei allen Knoten bekannt ist (wie beim Timestamped Antiu-Entropy Protokoll) –, oder überhaupt nicht – da keine Zusicherung möglich ist (wie beim Rumor Mongering).

   Je nach Implementierung lassen sie sich auch in Bezug auf die Partnerwahlstrategien unterscheiden. Es ist möglich, einfach nur mit dem nächsten Nachbarn zu kommunizieren, den Partner zufällig auszuwählen oder in Abhängigkeit von der (bekannten) Netztopologie, wie es bei anti-entropy Protokollen geschieht.

4.5 Zusammenfassung

Abschließend läßt sich sagen, daß es ein breites Spektrum an Protokollen für Replikation mit schwachen Konsitenzgarantien gibt. Für welches man sich entscheidet, hängt eng mit der Anwendung zusammen, für die es benötigt wird. Die zuzusichernde Konsistenz (bzw. die mögliche Inkonsistenzhöhe) sowie die Netztopologie beeinflussen die Auswahl. Es bietet sich an, für die vorgegebene Anwendung ein möglichst günstiges Protokoll auszuwählen und dann anwendungsspezifisch zu modifizieren.

Literatur

[1] Barbara,Garcia-Molina: *The Case for Controlled Inconsistency in Replicated Data*,1st Workshop on Management of Replicated Data, S.35-38, 1990.

[2] Lockemann,Krüger,Krumm: *Telekommunikation und Datenhaltung*, S.596, Hauser 1993.

[3] R.A.Golding: *Weak Consitency Group Communication and Membership*, S.38, 1992.

[4] Downing,Greenberg,Peha:*OSCAR: An Architecture for Weak-Consitency Replication*, S.350 ff., International Conference on Databases 1990.

[5] Liskov,Shrira:*Lazy Replication: Exploiting the Semantics of Distributed Services*, S.43 ff., 9th Symposium on Principles of Distributed Computing 1990.

[6] Demers,Greene,Hauser :*Epidemic Algorithms for Replicated Database Maintenance*, 6th Symposium onPrinciples of Distributed Computing 1987.

[7] R.A.Golding: *Weak Consistency Group Communication for Wide-Area Systems*,S.13 ff., 2nd Workshop on the Management of Replicated Data 1992.

5 Daten im Mobile Computing

GERNOT STENZ

5.1 Einführung - Was ist Mobile Computing

Die Anfänge des Computerzeitalters kannten nur einzelne, isolierte Rechenmaschinen. Doch schon bald wurden die ersten dieser Einheiten miteinander verknüpft, um Datenaustausch und Kommunikation zu ermöglichen. Ende der 60er Jahre entstand das Arpanet, einer der Vorläufer des heutigen Internets. Doch gerade die Entwicklung, die das Internet in jüngster Zeit vollzogen hat, weisen den Weg vom rein textorientierten System für Spezialisten zum Multi–Media–Forum für breite Bevölkerungsschichten. Wenn der Umgang mit dem heimischem Computer und Modem heute gang und gäbe ist, so folgt jetzt der nächste Schritt: In Analogie zur Revolution in der drahtlosen Telefontechnik (man bedenke nur die die lawinenartige Verbreitung von D–Netz Handys) werden auch die Computer immer größere Unabhängigkeit von fest installierten Netzwerkzugangspunkten erlangen. Aktuell geworden ist *Mobile Computing* durch die Einführung einer neuen Rechnerklasse, deren wohl bekanntester Vertreter der Apple Newton Message Pad ist. Diese *Personal Digital Assistants* (PDA) erfüllen durch ihr Gewicht und ihre Größe die Anforderungen dieses neuen Anwendungsspektrums.
Wenn Mobile Computing im Moment auch noch auf einige kleine Anwendungsbereiche beschränkt ist, werden in absehbarer Zukunft breite Bevölkerungskreise die neuen Dienstleistungen in Anspruch nehmen ([FZ94], [IB94] und [DH94]). Zum Beispiel

- kann sich der Besucher eines Einkaufszentrums auf seinem PDA über Angebote und Dienstleistungen der ansässigen Geschäfte informieren.
- kann man sich auf dem Weg zum Flughafen oder Bahnhof über Verzögerungen bei der Abreise informieren.
- kann man in fremden Städten und Gegenden Orientierungshinweise empfangen.
- kann man sich unabhängig vom Ort die lokalen Nachrichten oder Wetterberichte, oder auch die aus der eigenen Heimat, überspielen lassen.
- können sich Arbeitskollegen oder Geschäftspartner zu jeder Zeit und von jedem Ort aus zu ad hoc Konferenzen und Besprechungen zusammenfinden.

Des weiteren können auch viele schon vorhandene Dienste fester Netze auf drahtlose Netzwerke erweitert werden.

5.2 Verschiedene Konzepte für drahtlose Netze

Die Verwirklichung der Vision vom allgegenwärtigen Mobile Computing erfordert das Vorhandensein einer Netzwerkinfrastruktur mit ausreichenden Übertragungskapazitäten und genügend tragbare Rechnerleistung. Gerade aber die Infrastruktur für das Mobile Computing existiert bis jetzt erst ansatzweise, wird aber, wenn sie realisiert wird, vermutlich auf einem zellularen Ansatz basieren. Im folgenden werden einige drahtlose Netzwerke beschrieben, die zur Zeit schon benutzt werden (Vergleiche [IB94]).

- *Zellulare Netze.* Netze auf zellularer Basis, wie analoge und digitale Mobiltelefon–Netze (z.B. C–Netz und D–Netz) können ihren Benutzern Audio- und Datenkommunikation bieten. Ein Problem ist jedoch die nicht flächendeckende Bedienung, siehe dazu auch Abschnitt 5.3.3. Außerdem sind die gegenwärtigen zellularen Netze nicht für wirklich große Zahlen von Benutzern ausgelegt. Man hat nur wenig Erfahrung mit der Übertragung von Daten, wobei die gegebenen Bandbreiten für datenintensive Anwendungen gegenwärtig nicht ausreichend sind.
- *Drahtlose Local Area Networks (LAN).* Hierbei handelt es sich um ein LAN im traditionellen Sinne, das um eine Schnittstelle zur Bedienung kleiner Portabler Terminals erweitert wird. Das

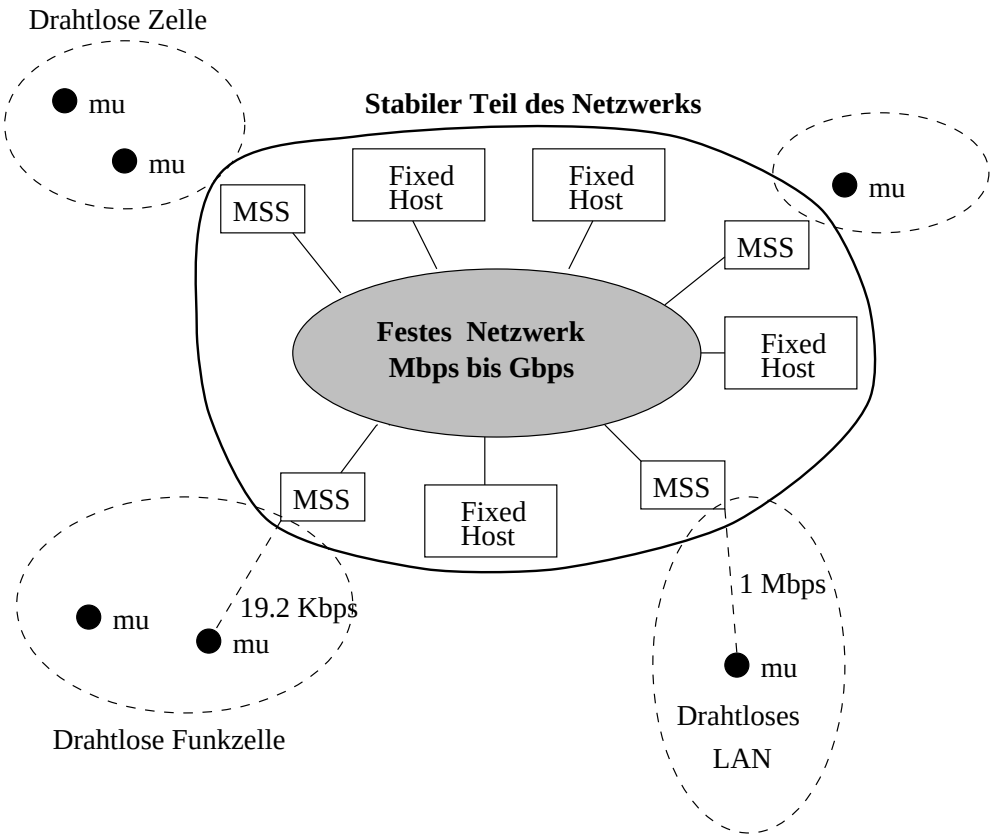


Abbildung 11: Ein Netzwerk für Mobile Computing. MSS steht für Mobile Support Station, d. h. einen Rechner, über den die mobilen Rechner Kontakt mit dem festen Netzwerk halten, und der auch die Kommunikation innerhalb einer Zelle gewährleistet. mu steht für eine mobile Einheit (Mobile Unit).

drahtlose LAN wird auf der anderen Seite an ein größeres festes Netzwerk wie z.B. ein festes LAN, ein WAN (Wide Area Network) oder das Internet angeschlossen. Drahtlose LAN stehen heute schon zur Verfügung: Wavelan von NCR, ALTAIR von Motorola, Range LAN von Proxim oder ARLAN von Telesystem. Das Kernstück des drahtlosen LAN ist die antennenbestückte Interfacekarte. Eine solche Karte kann sowohl am drahtlosen Terminal wie auch am festen Teil des Netzes angeschlossen werden. Der Nachteil dieser Netze ergibt sich aus der geringen Reichweite, sie sind eher für die Benutzung innerhalb von Gebäuden ausgelegt und unterstützen bis jetzt noch keine weiteren Ortswechsel der Benutzer.

- *Drahtlose Wide Area Networks.* Das sind spezielle Mobilfunknetze von privaten Anbietern oder TETRA wie Modacom. Ihre Infrastruktur bietet landesweite Unterstützung (in Amerika) für Dienste niedriger Bandbreite, wie Electronic Mail oder Terminalfunktion für Programme, die auf festinstallierten Rechnern ablaufen. Es jedoch keine allgegenwärtige, flexible Netzanbindung unterstützt und es ist unklar, wie sich ein solches System bei steigenden Benutzerzahlen verhitem.

- *Paging Networks.* Z.B. Cityruf. Die Flächendeckung ist hier gewährleistet, aber diese Dienste funktionieren nur unidirektional und mit sehr niedriger Bandbreite.

- *Neue Satellitensysteme.* In der Planungsphase befinden sich einige Systeme mit bis zu 70 erdnahen *Low Earth Orbit (LEO)* Satelliten, bereits im Betrieb sind einige Systeme mit geostationären Satelliten.

5.3 Spezifische Probleme beim Mobile Computing

Mobile Computing bringt eine Reihe von Problemen mit sich, die zum größten Teil mit althergebrachten Methoden nicht zu lösen sind, auch, da sie auf den Schwächen von Technologien beruhen, die sehr neu sind. Dieser Abschnitt wird einige dieser Probleme und die für sie erdachten Lösungsansätze aufzeigen.

5.3.1 Schwache Verbindungen und niedrige Bandbreite

Das Problem bei Datenkommunikation über drahtlose Netze sind die niedrigen Übertragungsraten zwischen mobilem Computer und fester Empfangsstation: Feste Netzwerke wie Ethernet erreichen Übertragungsraten von bis zu 10 Mbps, Fast Ethernet sogar bis zu 100 Mbps. Dagegen schaffen drahtlose LAN (ALTAIR) höchstens 5.7 Mbps, andere drahtlose Netze nur noch 1 Mbps mit Infrarot- und 2 Mbps mit Funktechnik, zellulare Telefonnetze sogar nur noch 9-14 Kbps. Da mobile Rechner durch ihre Ortswechsel auch ständig ihre Umgebung verändern, sind die möglichen Übertragungsraten auch starken Schwankungen unterworfen. So kann ein Computer zuerst im Büro an das feste Ethernet angeschlossen sein, später, im Konferenzraum über ein Infrarot–Link and das drahtlose LAN, auf der Fahrt nach Hause an das Funktelefon und am heimischen Schreibtisch an einen ISDN–Anschluß. Ein mobiler Rechner muß der unterschiedlichen Betriebsbedingungen gewärtig sein und seine Funktionalität und die Übertragungsprotokolle bei gleichzeitiger Kostenminimierung laufend anpassen (Siehe auch Abschnitt 5.3.4 und [FZ94]).

Heutige Rechnersysteme sind zumeist auf ein dauerhaft stabil funktionierendes Netzwerk angewiesen und bieten nur sehr geringe Ausfalltoleranz. Da in drahtlosen Netzen Verbindungsprobleme alltäglichen Charakter haben, müssen mobile Computer dafür gerüstet sein, mit häufigen Verbindungsabbrüchen fertig zu werden. Man kann dabei versuchen das Netzwerk sicherer zu machen oder den mobilen Geräten zu größerer Autonomie zu verhelfen. Auch hier muß die Recheneinheit zwischen verschiedenen Umgebungen unterscheiden können; in einem Büro mit einem festen Anschluß kann sie sich weitgehend auf die Sicherheit der Verbindung verlassen, in einer störungsanfälligen Umgebung soll sie so autonom wie möglich operieren. Ein System, das gute Funktionalität trotz häufiger Verbindungsunterbrechungen bietet, ist das Coda File System, das in Abschnitt 5.5 näher erläutert wird. Aufgrund des natürlichen Ungleichgewichts zwischen mobilen und stationären Computern ist es angebracht, die Last von Kommunikation und Rechenleistung ungleich zwischen den beiden Seiten zu verteilen, so daß der mobile Computer meistens eine passive Rolle als Empfangsgerät für fertig verarbeitete Daten einnimmt. Da mobile Einheiten stets knapp an Strom sind und das Empfangen von Informationen weniger Energie kostet als das Senden, muß der Datentransfer stark asymmetrisch gestaltet werden. Ein interessanter Ansatz ist das Konzept des Information Broadcasting. Da sich gezeigt hat, das bei Anfragen an Datenbanken sich 80 oder mehr Prozent der Anfragen auf höchstens 20 Prozent der Daten beziehen, liegt es nahe, besonders häufig verlangte Informationen zyklisch auszusenden, so daß überhaupt keine direkte Kommunikation zwischen dem mobilen und dem festen Partnern stattfindet. Wenn der Benutzer Teile dieser Daten abfragt, wartet das Mobile System, bis es die Daten aus dem Broadcast aufnehmen kann. Die Wartezeit beträgt hier maximal einen Broadcast–Zyklus. Daten werden hier quasi „im Äther“gehalten. Nur noch wenn eine Anfrage nicht aus dem Broadcast beantwortet werden kann, startet der mobile Computer eine reguläre Interaktive Anfrage ([IB94]).

5.3.2 Inhärente Probleme von mobilen Rechnern

Portable Rechner waren, sind und werden stationären Rechenanlagen immer in einer Reihe von Punkten unterlegen sein. Diese Probleme sind inhärent, da jedem Fortschritt in der Technologie portabler Systeme ein äquivalenter Fortschritt in der Technologie stationärer Compu-

System	Leistung
Display-Beleuchtung	35%
CPU / Speicher	31%
Festplatte	10%
Floppy Disk	8%
Display	5%
Tastatur	1%
Funktelefon	133%(!)

Tabelle 4: Anteiliger Stromverbrauch einiger wichtiger Komponenten. Der Wert für das Funktelefon wurde proportional zu den anderen Werten aus Tabelle 2 in [FZ94] ermittelt.

ter gegenüber stehen wird.Im folgenden sind ein paar dieser Punkte genannt ([Sat93], [FZ94] und [IB94]).

Leistungsschwache Stromversorgung. Im Gegensatz zu stationären Anlagen, die mit Strom- und Platzverbrauch, sowie mit der Verkabelung großzügig umgehen können, werden tragbare Systeme auf geringen Leistungsverbrauch hin optimiert. Die Batterien stellen in solchen Geräten gewichtsmäßig den größten Anteil. Folgerichtig kann geringerer Stromverbrauch durch geringeres Batteriegewicht und/oder längere Betriebszeiten die Portabilität des Rechners stark verbessern. Der Stromverbrauch elektronischer Komponenten ist von drei Faktoren abhängig: Kapazitiver Widerstand, Betriebsspannung und Taktfrequenz. Man reduziert

- den kapazitiven Widerstand durch höhere Integration,
- die Spannung durch die Entwicklung von Bauteilen, die mit weniger als den standardmäßigen 5 Volt auskommen, z.B. mit 2.5 bis 3.3 Volt,
- die Taktfrequenz durch optionalen Verzicht auf Rechengeschwindigkeit.

Im weiteren läßt sich Energie auch durch intelligenten Betrieb der Komponenten sparen, indem man z.B. eine Festplatte sehr schnell nach dem Ende eines Zugriffs wieder abschaltet. Da das Senden von Daten um den Faktor 10 mehr Energie verbraucht als das Empfangen, ergibt sich ein weiteres Argument dafür, die Kommunikation mehr zu Lasten des stationären Servers zu verschieben [FZ94].

Da aber in absehbarer Zeit kaum bahnbrechende Fortschritte in der Batterientechnologie zu erwarten sind (in absehbarer Zeit wird sich Kapazität von Batterien höchstens um 20% steigern!), wird die Energieknappheit auf Dauer ein schwieriges Problem für mobile Systeme bleiben. Die Zahlen in den Tabellen 4 bis 6 sind [FZ94] entnommen.

Geringe Speicherkapazität. Der Speicherplatz von portablen Computern ist durch Größe und Energiebedarf begrenzt. Festplatten, die normalerweise den Massenspeicher zur Verfügung stellen, werden in PDAs nicht verbaut, auch weil sie der rohen Behandlung, der sie im portablen Einsatz ausgesetzt wären, kaum standhalten könnten. Da diese Probleme nicht neu sind, existiert bereits eine Anzahl von Lösungen, wie automatische Dateikompression, Remote File Access oder die Kompression virtuellen Speichers. Diese Lösungen lassen sich aber nur zum Teil auf mobile Computer übertragen, da diese mit der Unsicherheit schlechter Verbindungen konfrontiert sind, und nicht mit der Sicherheit fester verteilter Dateisysteme.
Ein geeigneter Ansatz ist sicherlich der Ersatz von Objektcodeprogrammen durch Skriptsprachen, die auf Interpretern laufen. Ein angenehmer Seiteneffekt dieser Lösung wäre neben der Reduzierung der Programmgrößen sicher auch die höhere Systemunabhängigkeit der Software.

Geringe Datensicherheit. Gerade tragbare Computer sind besonders anfällig für Datenverlust und den unberechtigten Zugriff auf Daten. Dieses Problem tritt besonders deutlich zutage, wenn der mobile Computer nicht nur als Remote Terminal, sondern ebenso als selbständige Einheit benutzt wird. Als Lösung bietet es sich an, Daten auf nichtflüchtigen Speichermedien zu verschlüsseln. Dies macht sich jedoch nur bezahlt, wenn der Benutzer ein Mindestmaß an Sorgfalt walten läßt. So sollte er es unterlassen, den Rechner nach seiner Autorisierung unbeobachtet zu lassen, Gegen den Verlust von Daten kann man sich am besten schützen, indem man

Produkt	RAM	Takt	CPU	Batterien		Gewicht	Display	
				(Betr.h.)	Typ		(Pixel)	(sq.in)
Amstrad Pen Pad PDA600	128 KB	20	Z-80	40	3 AA	0.9	240 ×320	10.4
Apple Newton MessagePad	640 KB	20	RISC	6-8	4 AAA	0.9	240 ×336	11.2
Apple Newton MessagePad 110	1 MB	20	RISC	50	4 AA	1.25	240 ×320	11.8
Casio Z-7000 PDA	1 MB	7.4	8086	100	3 AA	1.0	320 ×256	12.4
Sharp Expert Pad	640 KB	20	RISC	20	4 AAA	0.9	240 ×336	11.2
Tandy Z-550 Zoomer PDA	1 MB	8	8086	100	3 AA	1.0	320 ×256	12.4
AT&T EO 440 Personal Communicator	4-12 MB	20	Hobbit	1-6	NiCad	2.2	640 ×480	25.7
Portable PC	4-16 MB	33-66	486	1-6	NiCad	5-10	640 ×480 1024 ×768 (in Farbe)	40

Tabelle 5: Charakteristika verschiedener PDA–Produkte im Vergleich mit einem Laptop

von wichtigen Dateien Kopien auf einer stationären Maschine hält. Erfahrungsgemäß werden Backups aber häufig vernachlässigt. Abhilfe schaffen hier automatische Replikationsstrategien (siehe dazu auch 5.4.2).

Beschränkte Benutzerschnittstelle. Der heutige Standard für Benutzerschnittstellen, d. h. Fenstertechnik, Maus und Tastatur, eignet sich kaum für den Einsatz in kleinformatigen PDAs. Auf den üblichen, kleinen Bildschirmen wird niemand mehrere Fenster nebeneinander haben wollen, zumal durch Fensterbegrenzungen wertvoller Platz vergeudet wird. Alternativ bieten sich folgende Eingabetechniken an ([FZ94]):

- *Analoge Eingabe statt Tasten.* Der Platzmangel auf portablen Rechnern hat eine Hinwendung zu neuartigen Eingabetechniken wie Handschrift–, Gesten– oder Spracherkennung bewirkt. Praktisch alle derzeit im Handel erhältlichen PDAs bedienen sich der ersten Methode. Doch obwohl Handschrifterkennung in speziellen Anwendungen eine Zuverlässigkeit von 96-100 % erreicht, ist sie im täglichen Einsatz eher ein Ärgernis. Dies liegt vor allem daran, daß im allgemeinen wenig bis keine Informationen über den Kontext der Eingabe bekannt sind, die dann fehlen, wenn das System mehrdeutige Eingaben richtig interpretieren soll. So kann ein Kreis, den der Anwender malt, ein Objekt oder ein Gebiet auswählen, einen Kreis, das Zeichen für Grad oder einfach den Buchstaben *o* darstellen. Spracherzeugung und -erkennung, ein an sich idealer Weg der Kommunikation, der Hände und Augen freiließe, scheitert vor allem an Speicherbedarf und Rechenleistung. Außerdem kann Sprach-Ein- und Ausgabe in ruhigen Umgebungen (Bibliotheken o.ä) störend sein, dagegen in lauten Umgebungen nahezu unmöglich, abgesehen von Datenschutzproblemen, die sich dabei zwangsweise ergeben.

- *Zeigemechanismen.* Da Mäuse für Kleinstrechner nicht zu gebrauchen sind, hat sich in dieser Klasse allgemein der Stift als Zeiger durchgesetzt. Der verlangt aber nach ganz anderer Soft- und Hardware, da man mit einem Stift übergangslos zu anderen Bildpositionen springen und im Gegensatz zur Maus auch direkt schreiben kann.

Heterogene Netzwerkumgebungen.
Während sich ein stationärer Rechner auf eine

Gerät	Leistung
Basissystem (2 MB, 25 MHz CPU)	3.650
Basissystem (2 MB, 10 MHz CPU)	3.150
Basissystem (2 MB, 5 MHz CPU)	2.800
Bildschirmbeleuchtung	1.425
Festplattenmotor	1.100
Coprozessor	0.650
Floppy Laufwerk	0.500
Externe Tastatur	0.490
LCD Schirm	0.315
Aktive Festplatte	0.125
IC-Kartenschacht	0.100
Zusatzspeicher (pro MB)	0.050
Parallelschnittstelle	0.035
Serielle schnittstelle	0.030
Zubehör:	
1.8 Zoll PCMCIA Festplatte	0.7-3.0
Mobiltelefon (aktiv)	5.400
Mobiltelefon (standby)	0.300
Infrarotnetzwerk 1 Mbit/s	0.250
PCMCIA Modem 14,400 bps	1.365
PCMCIA Modem 9,600 bps	0.625
PCMCIA Modem 2,400 bps	0.565
GPS Empfänger	0.670

Tabelle 6: Stromverbrauch von Komponenten und Zubehör portabler Computer

stabile, gleichbleibende Umgebung stützen kann (z.B. ein LAN), ist ein mobiles System gezwungen, unter verschiedensten Bedingungen korrekt zu funktionieren (z.B. bei der Kommunikation über Modem/ISDN oder unter den besonderen Bedingungen instabiler Funkverbindungen).

5.3.3 Probleme durch die zellulare Struktur mobiler Netze

Neben der bereits erwähnten niedrigen Übertragungsrate stellen sich bei zellularen Netzen noch weitere Probleme. Die Bandbreiten in solchen Netzen sind sehr begrenzt und müssen unter allen zu einem Zeitpunkt in einer Zelle befindlichen Nutzern aufgeteilt werden. Da die Anwender nicht gleichmässig über die Gesamtfläche eines Netzes verteilt sind, müssen die Zellen in stark besiedelten Gebieten dichter und kleiner gestaltet werden. Dies kann auf verkleinert man die Größe einer Zelle, so daß in ein gegebenes Gebiet mehr disjunkte Zellen passen.
In nicht–mobilen Netzen sind ortsabhängige Daten, wie die Netzadresse, oder der Ort des nächsten Servers statisch, verändern sich also nicht. In mobilen Netzen sind diese Angaben häufigen Veränderungen unterworfen. Wenn sich Benutzer bewegen, ändert sich ihre Netzadresse, ihr ursprünglicher Server muß nicht mehr der nächstgelegene sein und übertragene Daten müssen zunehmende Distanzen über das Netz zurücklegen. Abschnitt 5.4.4 befaßt sich mit der Problematik der veränderlichen Umgebung (Vergleiche [FZ94]).

Wechselnde Netzwerkadressen. Die Problematik der wechselnden Netzwerkadressen läßt sich mit einer Reihe von Verfahren in den Griff bekommen, die hier kurz aufgezählt werden.

- *Selektiver Broadcast.* Bei der Broadcast–Methode wird an alle Zellen des Netzwerks eine Aufforderung an den gesuchten Rechner gesendet, sich mit seiner momentanen Adresse zu melden. Wenn diese Methode auch für große Netzwerke ungeeignet ist, da eine Flut von Suchmeldungen das Netz belasten würde, ist der selektive Broadcast für kleine Gebiete durchaus geeignet, wenn der gesuchte Teilnehmer sich bekanntermassen nur in einer kleinen Menge von Zellen aufhalten kann.

- *Zentraldienst.* Bei diesem Verfahren wird die Netzadresse jedes Teilnehmers in einer zentralisierten Datenbank gespeichert. Wann immer ein Teilnehmer seinen Ort verändert, sendet er eine Nachricht, die der Datenbank seinen neuen Aufenthaltsort mitteilt. Auch wenn die Datenbasis sich logisch an einem festen Ort befindet, können die üblichen Methoden der Leistungssteigerung, Verteilung, Replikation und Caching trotzdem angewendet werden.

- *Heimatbasis.* Heimatbasen stellen den Grenzfall eines verteilten Zentraldienstes dar, wo die Ortsinformation für einen Teilnehmer genau einem Server bekannt ist. Da in diesem Schema keine Replikation vorgesehen ist, kann die Verfügbarkeit der Information stark eingeschränkt sein: Wenn ein Server ausfällt sind die in seiner Zuständigkeit befindlichen Teilnehmer nicht erreichbar. Ebenso problematisch, wenn auch wesentlich seltener, ist der Wechsel eines Teilnehmers zu einer anderen Heimatbasis. Mehr noch, die Suche nach Teilnehmern kann zu unnötig hohem Kommunikationsaufwand innerhalb des Netzes führen.

- *Weitergabezeiger* Der Teilnehmer hinterläßt bei jedem Ortswechsel seine neue Adresse an seinem alten Aufenthaltsort. Jede Nachricht an diesen Teilnehmer wandert die Reihe der Weitergabezeiger entlang, bis sie den Teilnehmer erreicht. Damit die Wege für die Daten nicht zu lang werden, findet in regelmäßigen Abständen ein Update dieser Zeiger statt. Obwohl diese Methode mit eine der schnellsten ist, ist sie doch an jedem Punkt der Weitergabe für Fehler anfällig, und in ihrer simplen Form verlangt sie vor dem Löschen der Weitergabezeiger, daß alle möglichen Nachrichtenquellen ein Update erhalten haben. So wird sie in der Regel nur zur Beschleunigung anderer Verfahren benutzt. Um die Nachrichten weiterleiten zu können, muß an jedem Ort eines Zeigers ein aktiver Partner vorhanden sein, das aber deckt sich nicht mit den Standard–Netzwerkmodellen, die vorsehen, daß eine Netzadresse entweder einen passiven Teil beschreibt, wie ein Ethernet–Kabel, oder daß sie an den mobilen Rechner gebunden ist, der ja nicht zurückbleiben kann, um die an ihn gerichteten Nachrichten weiterzuleiten. Diese Unstimmigkeiten resultieren in Schwierigkeiten bei der effizienten Implementierung der Weitergabe.

  Handover. Wenn sich Teilnehmer in einem zellularen Netzwerk bewegen, überschreiten sie dann und wann die Grenzen ihrer ursprünglichen Zelle und treten in eine neue ein. Die Gesamtheit der Maßnahmen, die notwendig sind, um den Wechsel eines Users in eine andere Zelle abzuwickeln, wird als Handover bezeichnet (Wenn ein System einem Benutzer erlaubt, sich beliebig im Netz zu bewegen, bezeichnet man das als *Roaming*). Es gibt vier Möglichkeiten, wie sich feste und mobile Einheiten über den Ortswechsel verständigen:

  - *active on* – der mobile Rechner verständigt seine neue zuständige MSS über seinen vorigen Aufenthaltsort.

  - *active off* – der mobile Rechner verständigt seine gegenwärtige MSS über seinen zukünftigen Aufenthaltsort.

  - *passive on* – der mobile Rechner betritt die neue Zelle, ohne der lokalen MSS seinen vorigen Aufenthaltsort mitzuteilen.

  - *passive off* – der mobile Rechner verläßt die alte Zelle, ohne der lokalen MSS seinen zukünftigen Aufenthaltsort mitzuteilen.

  Die Art und Weise des Übergangs hängt auch mit der Art der Algorithmen, die in dem mobilen Netz verwendet werden, zusammen. So kann ein *active off* dazu führen, daß in der alten MSS ein Weitergabezeiger auf die neue Zelle

gespeichert wird. Die Bewegungen eines Teilnehmer in einem mobilen Netz sind nicht transitionslos, d. h. , zwischen dem Verlassen der alten Zelle und dem Eintritt in die neue kann eine Zeitspanne liegen, in der der Teilnehmer im Netz nicht erreichbar ist. Dies kann zu Problemen in Zusammenhang mit gewissen Techniken der Kommunikation zwischen festen und mobilen Computern führen, wie z.B. beim selektiven Broadcast. Nachrichten können prinzipiell von einem Teilnehmer an beliebig viele andere Teilnehmer gesendet werden, und ein Broadcast von einer MSS erreicht alle mobilen Teilnehmer in ihrer Zelle. Aufgrund verschiedener Verzögerungszeiten erreicht eine ausgestrahlte Nachricht von einem festen Rechner aber nicht alle Zellen gleichzeitig. Es kann also vorkommen, daß eine, in einer Zelle ausgestrahlte, Mitteilung einen mobilen Empfänger nicht erreicht, da die Nachricht vor seinem Eintritt in die Zelle ausgestrahlt wurde oder erst nach seinem Austritt in eine andere Zelle. Andererseits ist auch möglich, daß ein Teilnehmer jeweils vor seinem Austritt und nach seinem erneuten Eintritt zweimal dieselbe Nachricht erhält. Siehe [AIB93].

5.3.4 Sicherheit, Datenschutz und Kostenstruktur

Sicherheit und Datenschutz. Wie bereits beschrieben, verursacht die Einfachheit des Netzzugangs ernste Sicherheitsprobleme. Prinzipiell sind drahtlose Verbindungen weniger sicher als solche über feste Leitungen. Die Situation wird noch komplizierter, wenn Grenzen von Sicherheitsbereichen nicht strikt eingehalten werden müssen, d. h. , wenn verschiedene Benutzergruppen ähnlichen, aber nicht gleichen Zugriff auf Ressourcen haben.
Es müssen also besondere, diesem Umstand angemessene Sicherheitsvorkehrungen getroffen werden. Die normale Methode, Sicherheit in wenig vertrauenswürdigen Umgebungen zu erzeugen, ist die Verschlüsselung von Daten. Ver- und Entschlüsselung sind mit Public Key Systemen schnell und effizient zu realisieren; das Problem dabei sind die Schlüssel, die nur autorisierten Parteien zur Verfügung stehen sollen und vor allen Anderen geheimgehalten werden müssen.
Ein System, das Schlüssel automatisch verwaltet, ist Kerberos vom MIT. Dieses System ermöglicht die automatische Autorisierung von Netzwerkbenutzern auch in für sie fremden Netzbereichen, ohne daß ihre Passwörter unverschlüsselt über unsichere Verbindungen gesendet werden müssen. Dennoch ist auch Kerberos nur bedingt sicher. Gerade die Tatsache, daß Passwörter nicht über das Netz gehen müssen, macht Kerberos anfällig für Passwortcracker.

Kostenstruktur und Abrechnung. Mobile Rechner werden in äußerst verschieden drahtlosen Umgebungen operieren. Es kann sich dabei um drahtlose LANs handeln oder auch um mobile Telefonnetze, wobei diese Dienste jeweils völlig unterschiedliche Gebührensysteme haben können. Der Nutzer wird selbstverständlich daran interessiert sein, seine Rechnung möglichst niedrig zu halten, indem er Dienste in einer Art und Weise in Anspruch nimmt, die auf seine Bedürfnisse und die lokalen Kommunikationsverhältnisse zugeschnitten sind.
Falls zum Beispiel die Gebühren, wie im Datex-J, per Online-Zeit gezählt werden, wird man jeden Tastendruck sofort zum Server schicken, um die Antwortzeiten kurz zu halten, eine Technik, die sich auch für menügesteuerte Systeme eignet. Wenn die Daten dagegen in Paketen weitergeleitet werden (um die Analogie weiter zu treiben, wie in Datex-P), dann könnte diese Taktik aber zu zu vielen und/oder halbleeren Paketen führen; in diesem Fall werden Daten gepuffert und en Block weitertransportiert ([IB94]).

5.4 Verteilung und Netzwerke

In diesem Abschnitt werden wir näher auf die Thematik der Datenhaltung in verteilten Systemen und Netzwerken, speziell im Hinblick auf das Mobile Computing, eingehen: Replikation von Daten, die Topologie von Netzwerken und die lokale Umgebung von Benutzern.

5.4.1 Replikation – optimistisch oder pessimistisch?

Daten replizieren heißt, von diesen Daten an getrennten Orten Kopien anzulegen. Dabei sol-

len diese Daten aber weiterhin als eine Entität behandelt werden, man muß sich sich also um die Konsistenz der verschiedenen Replikate kümmern. Dafür gibt es zwei grundlegend verschiedene Strategien:

- *Pessimistische Replikation.* Diese Art der Replikation ist strikt darum bemüht, alle Kopien jederzeit so weit wie möglich konsistent zu halten. Das wiederum hat häufige Schreib-/Lese–Operationen zur Folge, um die Replikate so oft wie möglich auf den gleichen Stand zu bringen, sowie eine sehr restriktive Vergabe von Zugriffsrechten auf die replizierten Objekte, um unlösbare Konflikte von vorneherein auszuschliessen. Pessimistische Replikation ist für Mobile Computing Anwendungen ungeeignet, da die häufige Trennung von Verbindungen die Konsistenzerhaltung unmöglich machen würde.

- *Optimistische Replikation.* Diese Strategie geht davon aus, daß auch wenn theoretisch beliebig viele Benutzer gleichzeitig auf dieselben Daten zugreifen können, dieses in der Praxis aber fast nie vorkommt, weswegen das System sehr viel freieren Zugang zu Daten gestatten darf, ohne tatsächlich die Konsistenz dieser Daten zu gefährden, die auch nur noch im Bedarfsfall überprüft wird. Wie in [KSM+93] treffend bemerkt wird: „Optimistische Replikation beruht auf der Annahme, daß es effektiver ist, sich zu entschuldigen als vorher zu fragen."Optimistische Replikation ist die Strategie, die sich für Mobile Computing eindeutig besser eignet, wie [KSM+93] belegt. Probleme ergeben sich aber in den Fällen, in denen durch die liberale Handhabung der Zugriffsrechte Inkonsistenzen auftreten (siehe Abschnitt 5.5).

5.4.2 Datenreplikation

Die Fähigkeit von Teilnehmern in einem mobilen Netz, mitsamt ihren Dienstleistungen an andere Orte weiterzuwandern, wirft neue Fragen bezüglich der Replikation und Verteilung von Daten auf:

- Unter welchen Bedingungen müssen Daten auf einem mobilen Computer repliziert werden?

- Soll ortsabhängige Information, die sich jetzt dynamisch ändert, auch repliziert werden?

- Wie beeinflussen die Bewegungen eines Benutzers das Replikationsschema, und wie sollen ihm die kopierten Daten folgen?

Betrachten wir ein Beispiel für die Replikation von Daten auf einen mobilen Rechner. Seien im folgenden c und s ein Client und ein Server (beide mobil), x sei ein Objekt, auf das der Server schreibenden und der Client lesenden Zugriff hat. Betrachten wir hier nur den einfachen Fall eines Clients und eines Servers, die sich jeweils nur in ihrer Zelle bewegen. Der Client liest die Daten vom Server. Die möglichen Orte für eine Kopie von x sind: beim Server, bei der MSS des Servers oder bei der MSS des Clients. Alternativ kann mittels Caching das Objekt auch beim Client repliziert werden, in diesem Fall sendet der Server, wenn er x modifiziert, eine Nachricht an den Client, daß seine Kopie ungültig geworden ist, worauf sich c bei s eine neue Kopie von x besorgt, sobald ein erneuter Zugriff auf die jetzt veralteten Daten ansteht. Wenn sich die Kopie nicht auf einem festen, sondern auf einem mobilen Rechner befindet, hängt die Leistungsfähigkeit eines Replikationsschemas von verschiedenen Faktoren, wie der jeweiligen Aktivität von Schreiber und Leser oder den Kosten für die Suche nach einem Partner ab. Gegebenenfalls müssen Client und Server den momentanen Ort des jeweils anderen in Erfahrung bringen. Wenn sich die Kopie dagegen auf einem stationären Computer befindet, dessen Adresse allgemein und auf Dauer bekannt ist, sind solche Suchaktionen nicht nötig, d. h. , die Replikationsschemata müssen beim Deponieren von Kopien dieses Ungleichgewicht im Kommunikationsaufwand berücksichtigen.

Replikationsschemata. Angenommen, Server und Client bewegen sich jeweils nur innerhalb ihrer Zellen, dann bieten sich für die Replikation von Datenobjekten vier verschiedene Schemata an ([BI92]). In den ersten beiden wird kein Caching angewendet, während die letzten drei Kopien der Daten an verschiedenen Orten cachen und das Updaten der Daten solange zurückstellen, bis ein Zugriff auf nicht mehr aktuelle Daten

geschieht. Die verschiedenen Replikationsschemata sind:

1. Der Server repliziert die Daten auf dem mobilen Client. Bei jedem Schreibzugriff müssen die Daten neu zum Client geschrieben werden, was eine Suche nach der Adresse des Clients erfordert. Der Client liest die Daten aus seiner lokalen Kopie.

2. Die replizierte Kopie liegt auf der MSS des Clients oder der des Servers. Der Client liest die Daten also von einer MSS, Lese– und Schreibzugriffe finden auf einer statischen Kopie statt.

3. Der Server cached eine Kopie seiner Daten auf seine MSS oder die des Clients. Diese Kopie wird beim ersten Schreibzugriff seit dem letzten Lesezugriff des Client für ungültig erklärt. Bei einem Lesezugriff auf eine ungültige Kopie muß der Server ausfindig gemacht werden, ist die Kopie gültig, werden die Daten von der MSS des Servers gelesen.

4. Eine Kopie liegt im Cache des Clients. Wenn seit dem letzten Update ein Lesezugriff geschah, sendet der Server bei jedem Schreibzugriff eine Invalidierungsnachricht an den Client. Wenn der Client veraltete Daten lesen will, wird Kontakt zum Server hergestellt und das Cache aktualisiert.

Das Replikationsschema muß dynamisch entsprechend den jeweiligen Umständen ausgesucht werden, wobei das System die Trade–Off–Points erkennt und seine Strategie anpasst. Siehe dazu auch [BI92]. In seinem Papier [Wol93] umreißt Ouri Wolfson ein System, bestimmte Anforderungen an die Replikation formal zu spezifizieren. Die Arbeit konzentriert sich auf den Begriff des *Data Allocation Scheme.* Allgemein soll dem Benutzer ermöglicht werden, seine Anforderungen dem System bekanntzugeben, indem er bestimmte Datenbestände abonniert, die ihm das System dann kontinuierlich zur Verfügung stellt. Er kann dabei auch abschwächende Parameter formulieren, etwa daß die Daten bis zu einem Grad veraltet sein dürfen. Auf etwas höherer Ebene kann festgelegt werden, wieviele Kopien eines Objekts jeweils im Netz existieren müssen oder wer von einem Objekt immer die aktuellste Version erhält. Abgesehen von solchen manuellen Anpassungen führt das System die Data Allocation aber automatisch durch, da hier für eine Leistungsoptimierung eine globale Sicht nötig ist, die der einzelne User in der Regel nicht kennt.

5.4.3 Dynamik der Netzwerktopologie

Durch die Beweglichkeit von Rechnern in einem Netzwerk wird eine Reihe von neuartigen Mechanismen notwendig, um die weitgehend auf statischen Informationen beruhenden Netzwerkstrukturen an die Bedürfnisse des Mobile Computing anzupassen.

Anpassung der Topologie. Viele Algorithmen für verteilte Systeme bauen auf Kenntnissen über die zugrundeliegende logische Struktur zwischen den Elementen eines Netzwerks auf. Der Hauptzweck einer solchen Struktur ist es, die Interaktionen in einem Netz zu einem gewissen Grad geordnet und vorhersehbar zu gestalten. Nachrichten von einem Netzteilnehmer an einen anderen folgen bestimmten logischen Pfaden durch das Netz. Ein solcher Pfad ist die Grundeinheit komplexerer Strukturen, wie Ringe, Bäume, Sterne oder Netze. Da sich mobile Rechner jedoch frei in dem Netz bewegen können, müssen diese logischen Pfade kontinuierlich neu bestimmt werden. Man betrachte dazu das Szenario in Abbildung 12(a) mit den drei mobilen Rechnern *h1*,*h2* und *h3*, die durch einen logischen Ring miteinander verbunden sind ([AIB93]). In unserem Modell besteht ein logischer Pfad zwischen zwei Elementen des Rings wie *h1* und *h2* aus drei Komponenten:

1. Der drahtlosen Verbindung zwischen *h1* und seiner gegenwärtigen MSS, genannt MSS1,

2. einer logischen Verbindung zwischen MSS1 und MSS2, der MSS von Rechner *h2*, und

3. einer drahtlosen Verbindung zwischen MSS2 und *h2*.

Wenn jetzt *h1* seinen Ort wechselt und sich in die zu MSS2 gehörige Zelle begibt, muß der logische Ring zwischen den drei Teilnehmern neu

auf das Netz abgebildet werden (siehe Abbildung 12(b)).
Die Notwendigkeit der Umstrukturierung der logischen Ordnung ergibt sich auch in einem weiteren Fall: In einem Netzwerk mit mobilen Teilnehmern werden bestehende Verbindungen zwischen Teinehmern viel häufiger getrennt als in einem Festnetz, sei es unfreiwillig durch einen Abriß der Verbindung (Siehe Abschnitt 5.3.1), oder freiwillig, bedingt durch Maßnahmen zur Energieeinsparung seitens eines mobilen Computers (siehe auch Abschnitt 5.3). Um Strom zu sparen, gehen portable Rechner in Phasen längerer Inaktivität in einen *Doze*–Modus über, in dem sie nicht mehr aktiv an den Netzinteraktionen teilnehmen. Betrachten wir wieder den logischen Ring aus Abbildung 12(a). Typischerweise funktioniert der Nachrichtentransport in solchen Ringen durch das weitergeben von Tokens (*Token Ring*) unter den Teilnehmern entlang des Rings. Nur der Teilnehmer, der im Besitz des Tokens ist, kann Aktionen auslösen. Aber selbst wenn ein Teilnehmer keine Aktionen initiieren will, muß er dennoch bereitstehen, um den Token weiterzureichen. Das heißt, wenn Rechner *h2* sich zur Zeit im Doze–Modus befindet, wird er gezwungen, diesen zu verlassen, wenn der Token von *h1* eintrifft und zu *h3* weitergeleitet werden muß.
Algorithmen, die auf logischen Strukturen basieren, müssen diesem Verweilen im Doze–Modus Rechnung tragen, mehr aber noch den besonderen Gegebenheiten bei der Trennung einzelner Komponenten vom Netz. Insbesondere muß sichergestellt werden, daß die erforderliche logische Struktur unter den im Netz verbleibenden Teilnehmern wiederhergestellt werden kann. Nehmen wir an, daß sich, wie in Abbildung 12(c), der Rechner *h2* vom Netz trennt. Bei einer freiwilligen Trennung kann *h2* vorher seinen Vorgänger im Netz, *h1*, über den bevorstehenden Austritt in Kenntnis setzen. *h1* kann darauf eine direkte Verbindung zum Nachfolger von *h2*, in diesem Fall *h3*, aufbauen, und danach *h2* die Erlaubnis zur Trennung erteilen. Der logische Ring bleibt dann mit den beiden verbleibenden Teilnehmern erhalten. Eine Trennung bewirkt also das Entstehen und Auflösen logischer Verbindungen, indirekt die Neukonfigurierung physischer Verbindungen. Im Gegensatz dazu bewirkt der Ortswechsel von Teilnehmern, wie in Abbildung 12(b) gezeigt, nur eine Umordnung auf physischer Ebene, weder die logischen Verbindungen noch die Menge der Teilnehmer sind betroffen.

5.4.4 Lokale Umgebung und Dienste

Mobile Computing erzeugt eine neue Art von lokaler Umgebung, eine, die sich mit den Bewegungen der User mitverändert. Selbst wenn ein mobiler Rechner weiß, wo er den nächstgelegenen Server für eine bestimmte Dienstleistung findet, kann sich diese Information doch mit der Zeit ändern. Da die räumliche Entfernung zweier Punkte nicht notwendigerweise ihrer Entfernung im Netz entspricht, können schon kleine Ortsveränderungen einen überproportionalen Anstieg der Kommunikationswege im Netzwerk zur Folge haben, beispielsweise wenn eine geringfügige physische Bewegung den Wechsel in eine andere Netzwerkdomäne bewirkt. In diesem Fall muß jede Kommunikation mit der ursprünglichen Umgebung mehr Zwischenstationen durchlaufen, was in längeren Antwortzeiten und häufigeren Verbindungsunterbrechungen resultiert. Durch die größere Zahl beteiligter Elemente steigt auch die Belastung des Netzwerks an, wenn auch oft für den Benutzer transparent. Um diesen Schwierigkeiten aus dem Wege zu gehen, kann und sollte die lokale Umgebung dynamisch auf die jeweils nächstgelegenen Diensterbringer ausgelegt werden. Jedoch stellen die vorgenannten Aspekte nicht den einzigen Grund für eine „wandernde lokale Umgebung“ dar. In Zukunft werden an mobile Anwendungen dieselben Ansprüche in Bezug auf Leistungsfähigkeit und Komfort gestellt werden, wie an Anwendungen auf festinstallierten Systemen. Ein Benutzer eines mobilen Rechners möchte Unabhängig von seinem Aufenthaltsort auf dieselbe Menge von Ressourcen zugreifen können. Hierzu kann man dafür sorgen, daß für den Benutzer wichtige Prozesse oder Daten ihm durch das Netz folgen, so daß Netzbelastung und Antwortzeiten klein gehalten werden und der Ortswechsel dem User transparent erscheint. Des weiteren sollte ein User auch in

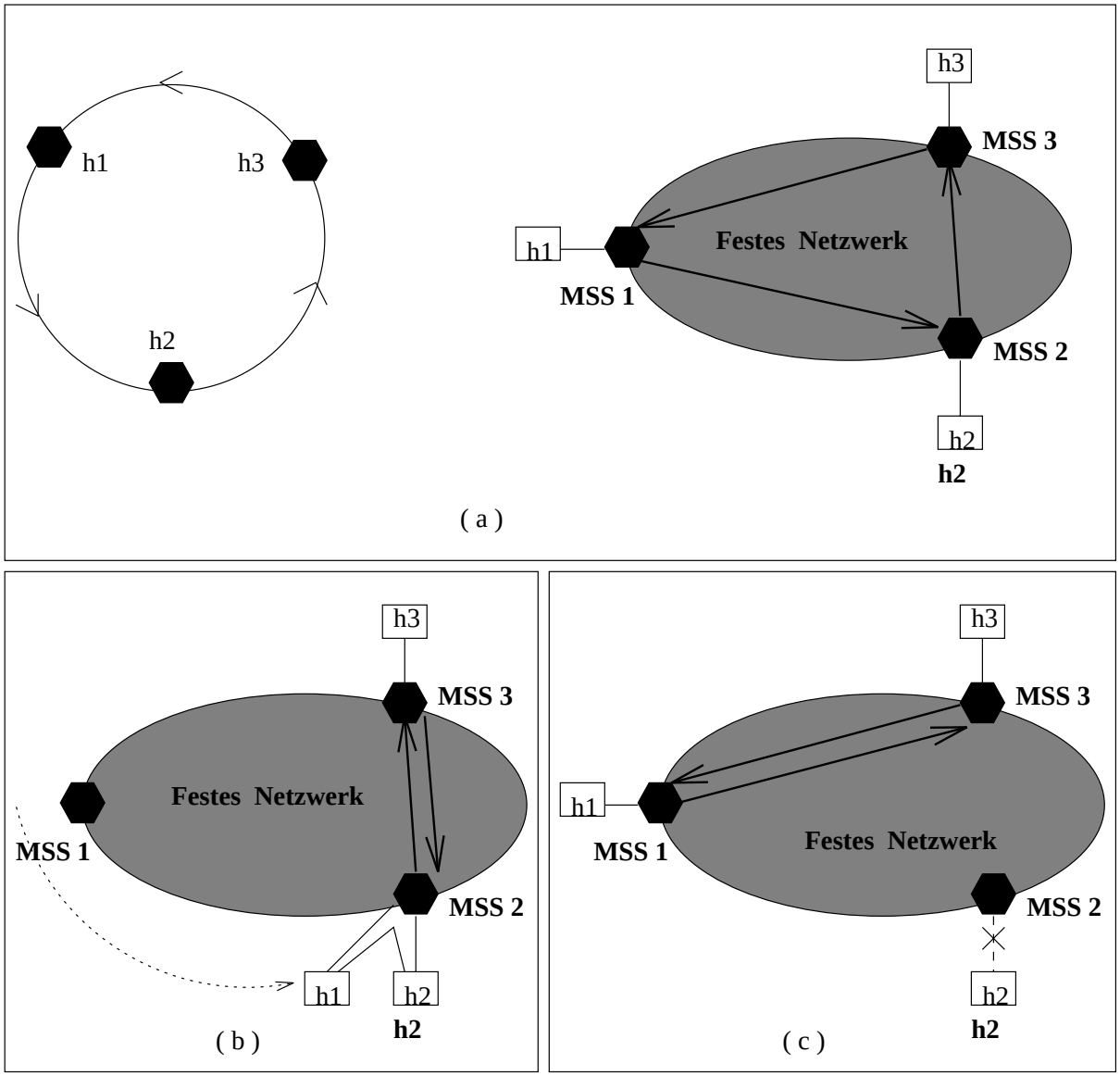


Abbildung 12: Ein Beispiel zu Veränderungen in der Netzwerktopologie

der Lage sein, in Zusammenhang mit seinem Aufenthaltsort lokal vorhandene Ressourcen zu nutzen, wie etwa den örtlichen Drucker oder Rechenkapazität auf Rechnern vor Ort ([DH94]).

5.5 Ein Beispiel: Das Coda File System

In den vorigen Abschnitten wurden Problemstellungen und Lösungsansätze des Mobile Computing auf eine meist sehr abstrakte Art vorgestellt. In diesem Abschnitt werden wir ein tatsächlich realisiertes Software– Produkt betrachten, in dem Probleme wie Replikation und das Behandeln von Verbindungsunterbrechungen gelöst wurden.

5.5.1 Verteilte Dateisysteme

Bevor wir uns mit einem Beispiel für ein Filesystem für Mobile Computing befassen, sehen wir uns kurz zwei konventionelle verteilte Dateisysteme an: SUN NFS und Andrew von der Carnegie–Mellon Universität. [LS90] stellt in übersichtlicher Art und Weise die Charakteristiken einiger verteilter Dateisysteme dar.

NFS ist ein fester Bestandteil des SUNOS Be-

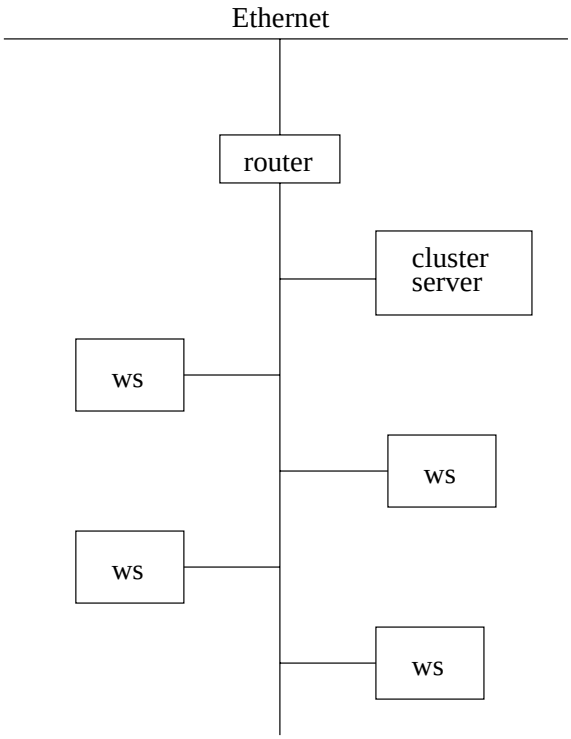


Abbildung 13: Ein Beispiel für ein Andrew–Cluster

triebssystems. Es ist weniger ein verteiltes Filesystem, als vielmehr ein Netzwerkdienst für lokale LANs. NFS wurde dafür gedacht, in einer heterogenen Umgebung Zugriff auf Remote Files zu erlauben. Es ist wohl das bekannteste verteilte Dateisystem.

Andrew dagegen ist als ein wirkliches homogenes verteiltes Dateisystem konzipiert. Andrew wurde speziell auf große Benutzermengen hin entwickelt – in der Größenordnung von 5000 Workstations.

Das System ist hierarisch geordnet. Es ist in einzelne Cluster unterteilt, in denen jeweils ein Server für mehrere Clients zuständig ist. Diese Cluster sind untereinander mit einem Backbone Netzwerk verbunden.

Jeder Client hat einen zweigeteilten Namensraum, der sich in den lokalen Namensraum und den gemeinsamen Namensraum gliedert. Den lokalen Namensraum bildet das lokale root-directory, während der für alle Maschinen gleiche und ortstransparente gemeinsame Namensraum von einer Gruppe von Servern, genannt *Vice*, zur Verfügung gestellt wird (siehe Abbildung 14). Die Unterteilung in Cluster erfolgt hauptsächlich aus Leistungsgründen: Die Clients eines Clusters sollen – wenn möglich – Zugriffe nur auf ihren lokalen Server durchführen. Der gemeinsame Namensraum gliedert sich in sogenannte *Volumes*, eine kleine Einheit, die etwa die Dateien eines einzelnen Benutzers beinhalten kann. Die Volumes werden durch einen dem Mount ähnlichen Mechanismus zusammengefasst.

Im Gegensatz zu NFS werden aber Zugriffe auf Remote Files nicht beim Server durchgeführt, jeder Client hat einen lokalen Task, genannt *Venus*, der beim Öffnen einer Remote File diese komplett in ein lokales Cache holt. Alle weiteren Lese- und Schreiboperationen finden dann nur noch lokal statt. Dabei geht Venus von einer optimistischen Update–Philosophie aus. Venus betrachtet seine lokale Kopie solange als gültig, bis der Server ihm mitteilt, daß die Kopie ungültig geworden ist (Callback–Methode). Beim Schliessen der Datei wird diese dann auf den Server zurückgeschrieben. Auf diese Weise wird die Last auf dem Netz drastisch gesenkt. Diese Strategie funktioniert sehr gut, da sich in der Praxis gezeigt hat, daß ein gemeinsamer Zugriff mehrerer User auf dieselbe File sehr selten vorkommt.

5.5.2 Hoarding, Emulation und Reintegration

Das Coda File System (siehe [KSM+93]) basiert auf dem oben beschriebenen Andrew File System. Auch Coda Clients haben einen Venus–Task, der jedoch noch zuätzliche Aufgaben übernehmen muß. Da Coda für das Mobile Computing entwickelt wurde, enthält es Strategien, um freiwilliger oder unfreiwilliger Trennung vom Netzwerk begegnen zu können. Bei Venus unterscheidet man hier drei Phasen: Hoarding, Emulation und Reintegration.

Hoarding. Dies ist die Phase während des normalen Betriebs am Netz. Venus versucht während dieser Zeit laufend, alle aktuell gebrauchten Dateien zu cachen, um für den Fall einer Verbindungsunterbrechung gewappnet zu sein. Dabei wird laufend überprüft, ob der momentan vorhandene Bestand an replizierten Dateien den Anforderungen entspricht. Dabei wer-

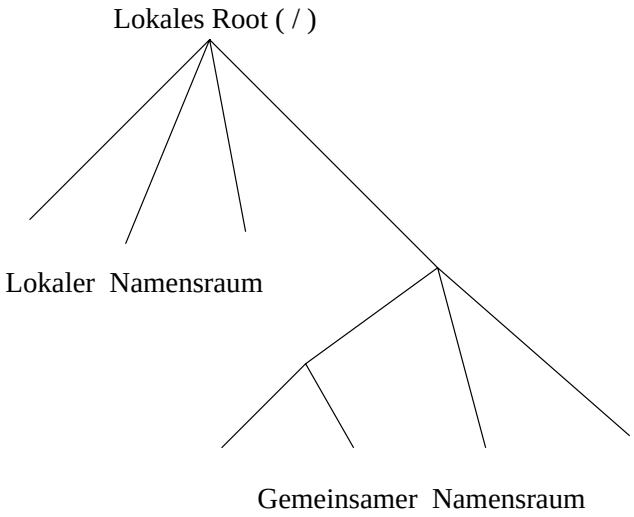


Abbildung 14: Der Namensraum von Andrew

den sowhl implizite als auch explizite Informationen benutzt. Implizite Informationen stellen z.B. Logfiles über Filezugriffe in der Vergangenheit dar, es wird hier durchaus mit gängigen Algorithmen gearbeitet (LRU, LFU). Explizite Informationen kann der User mittels Hoarding Profiles spezifizieren.

Emulation. Im Falle eines Verbindungsabbruchs stellt Venus auf den Emulatiosmodus um, in dem versucht wird, das System für den Benutzer transparent weiterzubetreiben, was auch mit sehr gutem Erfolg gelingt. Der Abbruch der Verbindung kann sowohl freiwillig oder auch unfreiwillig erfolgen; tatsächlich trennten die Test–Benutzer des Systems die Verbindung meistens willentlich (etwa, bevor sie mit ihrem Computer ins Wochenende fuhren). Entscheidend für den reibungslosen Ablauf der Emulationsphase ist das geschickte Caching während der Hoard–Phase und die Kontinuität der Benutzeranforderungen während der Emulation. Wenn Venus einen Cache–Miss entdeckt, kehrt es mit einer Fehlermeldung zum Aufrufenden Prozeß zurück, worauf der Benutzer die Ausführung des Prozesses bis zur Wiederherstellung der Netzverbindung anhalten kann. Die Persistenz von Änderungen an Dateien in dieser Phase wird durch das Führen einer Transaktionsprotokolldatei erreicht. Um diese Logfile nicht zu groß werden zu lassen, bedient sich Venus mehrerer Optimierungstechniken, mit überaus gutem Erfolg – wie sich im folgenden Abschnitt zeigt.

Reintegration. Wenn es gelingt, die Verbindung zum Remote Server wiederherzustellen, tritt Venus in die Phase der Reintegration ein. Dabei wird versucht, die in der Emulationsphase modifizierten Cache–Files auf den Server zurückzuschreiben. Diese Reintegration dauert durchschnittlich nur zwischen 5 und 20 Sekunden und wird, da sie im Background abläuft, vom Benutzer oft überhaupt nicht registriert. Kompliziert wird es jedoch, wenn bei der Reintegration ein Konsistenz–Konflikt auftritt. Da die Intelligenz eines Dateisystems nicht ausreicht, solche semantischen Probleme zu lösen, fällt dem Benutzer die Aufgabe zu, die verschiedenen Versionen einer Datei miteinander in Einklang zu bringen.

5.5.3 Quantitative Auswertung

Bei der Betrachtung der Leistung von Coda sind drei Aspekte von besonderem Interesse:

- Wie groß muß die lokale Platte (auf die die Remote Files gecacht werden) sein?
- Wie stark macht sich die Reintegration bemerkbar?
- Wie wichtig ist die Optimierung der Logfiles?

Lokaler Speicherbedarf. Zur Auswertung des benötigten Plattenplatzes wurden verschiedene Testläufe gefahren, während derer jede Referenz auf lokale, remote oder gecachte Dateien gezählt wurde. Die Zugriffsprofile wurden sorgfältig auf hohe Aktivität ausgesucht. Es wurden 10 Profile von Workstations ausgesucht, 5 davon stellen 12–Stunden Arbeitstage dar, die anderen 5 repräsentieren jeweils die Aktivität einer ganzen Woche. Tabelle 7 zeigt die Profile.

Die Auswertung der Daten zeigte deutlich, daß die Cache–Belegung am Anfang der Simulationen stark anstieg, sich jedoch bald in der nähe eines Maximums stabilisierten und nicht mehr weiter wuchsen. Beispielsweise erreichten die meisten der Ein–Tages–Läufe binnen weniger Stunden 80 % der Spitzenwerte. Ähnlich erreichten die meisten Wochen–Läufe den größten Teil der Cache–Belegung am Ende des zweiten Tages. Der Speicherverbrauch der Ein–Tages–Läufe belief sich im Maximum auf 25 MB und im Mittel auf 10 MB. Bei der Wochen–Simulation betrugen diese Werte knapp 100 MB bzw. unter 50 MB. Diese Werte zeigen deutlich, daß mit den heute zur Verfügung stehenden Plattenkapazitäten problemlos lange Abschnitte der Emulation gefahren werden können, selbst auf kleinen Laptops.

Reintegrationsdauer. Die Dauer der Reintegration ist Abhängig von der Anzahl der Modifikationen während der Emulationsphase. Im praktischen Gebrauch des Systems dauerte die Reintegration nach einer Trennungszeit von einem Tag eine Minute oder weniger. Tabelle 8 zeigt die Dauer, die Anzahl der Records in der Logfile und die Menge der Daten, die während der Reintegration vom Client zum Server geschrieben wurde. Lediglich in einem Fall dauerte das Zurückspielen nach einem Wochen–Durchlauf ungefähr 20 Minuten, was jedoch auf einen Software–Fehler zurückgeführt werden konnte.

Zweierlei sollte noch zur Reintegration bemerkt werden.
Zum Einen ist die vom Benutzer wahrgenommene Reintegrationsdauer nahezu null, da dieser Vorgang zumeist von einem Daemon, nicht vom Benutzer, ausgelöst wird und so vollständig im Hintergrung abläuft und die Arbeit des Benutzers nicht stört. Zum Anderen stellen die Simulationen Aktivitäten dar, die eine ganze Anzahl von Volumes betrifft (5–10 für Arbeitstage und 10-15 für Wochenläufe), wobei jedoch die Reintegration in einem einzelnen Volume stattfand. In Wirklichkeit würden statt einer großen Reintegration mehrere kleinere ablaufen, was durch Parallelitätseffekte eine Verbesserung vom Faktor 3 oder 4 brächte.

Optimierung der Logfiles. Aus mehreren Gründen ist eine Optimierung des Cache und Loggingverhaltens erstrebenswert. Einerseits ist Speicherplatz während der Emulationsphase eine knappe Ressource, andererseits kann so bei der Reintegration die Dauer und Netzlast niedrig gehalten werden. Venus bedient sich zu diesem Zwecke verschiedener Optimierungsmethoden, und das mit erstaunlichem Erfolg. Tabelle 9 zeigt den Platzverbrauch für den optimierten und den nichtoptimierten Fall in den verschiedenen Simulationen.

Die hohen Unterschiede ergeben sich dadurch, daß, während der Verbrauch bei beiden Varianten in der Anfangszeit ungefähr gleich steigt, sich die optimierte Version nach einiger Zeit stabilisiert, indessen die nichtoptimierte Version einen konstant wachsenden Speicherbedarf hat. Dies macht bei einer Wochenlauf Simulation im Durchschnitt mehr als 145 MB aus. Im Extremfall betrug das Verhältnis im Platzbedarf nichtoptimiert/optimiert 28.9:1, ein Unterschied von 850 Megabyte! Auch bei der Reintegrationsdauer lassen sich signifikante Einsparungen erhalten, selbst wenn die Algorithmen ihre Schwachstellen verlieren sollten, werden die Verhältnisse zwischen nichtoptimierter und optimierter Leistung immer noch zwischen 4.5:1 für Tagesprofile und 7.6:1 für Wochenprofile betragen.

5.5.4 Mögliche Verbesserungen

Ausnutzung schwacher Verbindungen. Obwohl sich Coda als verteiltes Filesystem für das Mobile Computing gut eignet, hat es doch einige Schwächen:

- Cache Misses sind für den Benutzer nicht transparent.

Profil	Maschinenname	Typ	Simulationsbeginn	Zugriffsrecords
Arbeitstag #1	brahms.coda.cs.cmu.edu	IBM RT–PC	25.03.91, 11:00	195289
Arbeitstag #2	holst.coda.cs.cmu.edu	DECstation 3100	22.02.91, 09:15	348589
Arbeitstag #3	ives.coda.cs.cmu.edu	DECstation 3100	05.03.91, 08:45	134497
Arbeitstag #4	mozart.coda.cs.cmu.edu	DECstation 3100	11.03.91, 11:45	238626
Arbeitstag #5	verdi.coda.cs.cmu.edu	DECstation 3100	21.02.91, 12:00	294211
Wochenlauf #1	concord.coda.cs.cmu.edu	SUN 4/330	26.07.91, 11:41	2948544
Wochenlauf #2	holst.coda.cs.cmu.edu	DECstation 3100	18.08.91, 23:21	3492335
Wochenlauf #3	ives.coda.cs.cmu.edu	DECstation 3100	03.05.91, 12:15	4129775
Wochenlauf #4	messiaen.coda.cs.cmu.edu	DECstation 3100	27.09.91, 00:15	1613911
Wochenlauf #5	purcell.coda.cs.cmu.edu	DECstation 3100	21.08.91, 14:47	2173191

Tabelle 7: Statistik der Tages- und Wochenzugriffsprofile

Task	Log Records Gesamt	Back–Fetch MB Gesamt	Dauer sec.
Andrew Benchmark	203	1.2	10
Venus Make	146	10.3	38
Arbeitstag #1 Rücksp.	1422	4.9	72
Arbeitstag #2 Rücksp.	316	.9	14
Arbeitstag #3 Rücksp.	212	.8	8
Arbeitstag #4 Rücksp.	873	1.3	32
Arbeitstag #5 Rücksp.	99	4.0	22
Wochenlauf #1 Rücksp.	1802	15.9	176
Wochenlauf #2 Rücksp.	1664	17.5	160
Wochenlauf #3 Rücksp.	7199	23.7	1217
Wochenlauf #4 Rücksp.	1159	15.1	90
Wochenlauf #5 Rücksp.	2676	35.8	273

Tabelle 8: Dauer der Reintegration. Die Spalte Back–Fetch beschreibt die Menge der Daten, die beim Wiederaufbau der Verbindung zurückgespielt wurden.

- Längere Emulationsphasen bergen das Risiko der Ressourcenerschöpfung durch neue Daten im Cache und lange Logfiles.
- Längere Emulationsphasen vergrößern auch die Wahrscheinlichkeit, daß bei bei der Reintegration Konflikte auftreten, die der Benutzer selbst lösen muß.

Als Verbesserung bietet sich hier die Nutzung von Mobilfunknetzen oder auch normalen Telefonleitungen an. Wenngleich diese auch wesentlich langsamer als LANs sind, so ermöglichen sie doch auch während der Emulationsphase begrenzten Kontakt zum Server, den Venus nutzen kann, um benötigte, nicht lokal vorrätige, Daten zu cachen oder durch zwischenzeitliches Back–Fetching die Anzahl der Konflikte bei der Verbindungswiederherstellung zu verringern.

Verbesserung des Hoarding. In der momentanen Implementierung muß der Benutzer noch weitgehend selber festlegen, welche Daten für ihn gehoardet werden sollen. In Zukunft soll diese Arbeit durch bessere Werkzeuge, aber auch durch ein brauchbares Maß für die Güte des Hoarding, vereinfacht werden. Im Moment wird das Hoarding nur nach der Cache–Miss Rate bewertet, wobei alle Cache–Misses als gleich schwerwiegend betrachtet werden. Neue Bewertungskri-

Profil	Container Space		RVM Space		Total Space		
	Unopt	Opt	Unopt	Opt	Unopt	Opt	Ratio
Arbeitstag #1	34.6	15.2	2.9	2.7	37.5	17.8	2.1
Arbeitstag #2	14.9	4.0	2.3	1.5	17.2	5.5	3.1
Arbeitstag #3	7.9	5.8	1.6	1.4	9.5	7.2	1.3
Arbeitstag #4	16.7	8.2	1.9	1.5	18.6	9.7	1.9
Arbeitstag #5	59.2	21.3	1.2	1.1	60.4	22.3	2.7
Wochenlauf #1	872.6	25.9	11.7	4.8	884.3	30.6	28.9
Wochenlauf #2	90.7	28.3	13.5	5.9	104.0	34.2	3.0
Wochenlauf #3	119.9	45.0	46.2	9.1	165.9	54.0	3.1
Wochenlauf #4	222.1	23.9	5.5	3.9	227.5	27.7	8.2
Wochenlauf #5	170.8	79.0	9.1	7.7	179.8	86.5	2.1

Tabelle 9: Optimierter und nichtoptimierter Platzverbrauch. RVM ist eine Art von Transaktionsverwaltung, auf der das Führen der Logfiles beruht.

terien, die untersucht werden, sind z.B.:

- Die Zeit bis zum ersten Cache–Miss.
- Die Zeit bis ein (aus Sicht des Benutzers) kritischer Cache–Miss auftritt.
- Die Zeit, bis der Gesamteffekt mehrerer Cache–Misses eine besimmte Schwelle überschreitet.
- Die Zeit, bis dieVerbindungstransparenz verlorengeht.
- Der Anteil der Cache–Daten, die während einer Trennung vom Netzwerk tatsächlich benutzt werden.
- Die Verbesserung der Cache–Miss Rate in der Hoarding–Phase durch besseres Hoarding.

Zur Untersuchung dieser Sachverhalte gehören auch die Analyse von Filezugriffen – zeitgleich oder post mortem – der situationsabhängige Wert von Hoarding–Hilfswerkzeugen und die bereitstellung graphischer Benutzeroberflächen für die Erstellung von Hoardprofilen.

Konfliktauflösungen. Konflikte bei der Reintegration von Daten müssen zur zeit noch von Hand gelöst werden, doch kann dieser Vorgang in vielen Fällen abhängig vor der Anwendung automatisch erfolgen. So können z.B. verschiedene Dateien für einen Terminkalender dadurch in Einklang gebracht werden, daß man die Einträge vergleicht und bei Konflikten nach belieben einen davon aussucht und die andere Seite von dieser Maßnahme in Kenntnis setzt. Auch für kompliziertere Fälle kann dem Benutzer eine Hilfestellung gegeben werden, so etwa mit graphischen Werkzeugen, die die Unterschiede zwischen verschiedenen Versionen einer Datei am Bildschirm darstellen und die Angleichung mittels Cut–and–Paste erlauben.

5.6 Zusammenfassung und Ausblick

Mobile Computing ist eine Technik, die jederzeit und überall den Zugriff auf digitale Ressourcen gestatten soll. Es stellt eine komfortable Erweiterung heutiger verteilter Netzwerke dar. Im weiteren Sinne könnte man sagen, daß die Vision des Mobile Computing die Aufhebung aller Beschränkungen von Zeit und Ort bedeutet, die der Menschheit durch Tischrechner und festverlegte Netzwerke auferlegt wurden. Wenn man Auswirkungen von Mobile Computing vorhersagen möchte, tut man gut daran, neuere Trends in der Nutzung fester Netze, genauergesagt des Internets, zu analysieren. Hier hat die Entwicklung komfortabler Programme zum Abrufen von Informationen (wie z.B. Mosaic), einen sprunghaften Anstieg in der Belastung der Ressourcen bewirkt. Ihre Zugänglichkeit durch mobile Rechner wird ihre Einbindung in alle

Aspekte des täglichen Lebens ermöglichen und somit eine schnell wachsende Nachfrage nach den Dienstleistungen mobiler Netze zur Folge haben. Es ist die Aufgabe der Informatik, die Techniken und Verfahrensweisen, die sich in eher traditionelleren Anwendungen bewährt haben, so weiterzuentwickeln, daß sie die Herausforderungen des Mobile Computing bewältigen können. Die Datenverarbeitung in mobilen Netzen erfordet neue Forschungsanstrengungen in einer Reihe von Gebieten. Es wird sicher nicht genügen, auf bestehende Festnetze einfach ein drahtloses Kommunikationssystem aufzupfropfen. Ich habe in den vorangegangenen Abschnitten einige dieser Gebiete angesprochen, Probleme präsentiert und stellenweise Lösungsansätze vorgestellt. Die Problemstellungen erstrecken sich vom Niveau globaler Netzwerke bis hinunter zu Fragestellungen bezüglich der Anwendung von PDA–Computern.

Sicherlich ist eines klargeworden: Mobile Computing wird die Art und Weise, wie Menschen Computer benutzen, grundlegend verändern.

Literatur

[AIB93] A. Acharya, T. Imielinski und B.R. Badrinath. Impact of Mobility on Distributed Computations. *Operating Systems Review* **27**(2), April 1993, Seite 15–20.

[BI92] B.R. Badrinath und T. Imielinski. Replication and Mobility. Monterey California, 1992. IEEE Workshop on the Management of Replicated Data, Seite 9–12.

[DH94] N. Diehl und A. Held. Systemintegration bei Mobile Computing Anwendungen. In Bernd Wolfinger (Hrsg.), *Innovationen bei Rechen- und Kommunikationssystemen*, Hamburg, 28. August–2. September 1994. 24. GI Jahrestagung im Rahmen des 13th World Computer Congress IFIP Congress '94, Seite 294–301.

[FZ94] G.H. Forman und J. Zahorjan. The Challenges of Mobile Computing. *IEEE Computer* **27**(4), April 1994, Seite 38–47.

[IB94] T. Imielinski und B.R. Badrinath. Mobile Wireless Computing: Changes in Data Management. *Communications of the ACM* **37**(10), October 1994, Seite 18–28.

[KSM+93] J.J. Kistler, M. Satyanarayanan, L.B. Mummert, M.R. Ebling, P. Kumar und Q. Lu. Experience with Disconnected Operation in a Mobile Computing Environment. Cambridge, MA, August 2–3 1993. USENIX Symposium on Mobile & Location–Independent Computing, Seite 11–28.

[LS90] E. Levy und A. Silberschatz. Distributed File Systems: Concepts and Examples. *ACM Computing Surveys* **22**(4), December 1990, Seite 321–374.

[Sat93] M. Satyanarayanan. Mobile Computing. *IEEE Computer* **26**(9), September 1993, Seite 81–82.

[Wol93] O. Wolfson. Data Allocation in Mobile Computing: A Project Description. Princeton, N.J., October 6th 1993. IEEE Workshop on Advances in Parallel and Distributed Systems, Seite 89–94.

6 Replikationskontrollverfahren

HELMUT FUCHS

6.1 Einführung

In vielen Anwendungen von Rechnersystemen sind Ausfälle nicht tolerierbar. Als Beispiele hierfür seien z.B. Telefonvermittlungs- und Buchungssysteme genannt. Solange die Anwendungen nur auf einem einzigen Rechner betrieben werden, führt ein Ausfall dieses Rechners, oder der Kommunikationseinrichtungen, die zu diesem führen, unvermeidlich zu einem Ausfall des gesamten Systems. Es ist naheliegend die Ausfallsicherheit durch *Replikation*, d.h. durch Vervielfältigung der Daten und Prozesse der Anwendung auf mehrere Rechner, zu erhöhen. Diese Vervielfältigung darf natürlich nicht unkontrolliert geschehen, die Semantik der Anwendung muß erhalten bleiben. Diese Kontrolle überträgt man *Replikationskontrollalgorithmen*, mit denen sich der Rest dieses Beitrags beschäftigt.

6.1.1 Grundsätzliches

Einige der hier aufgeführten Beobachtungen gelten für Daten und Prozesse gleichermaßen. Schließlich läßt sich ein Prozess zu definierten Zeitpunkten vollständig durch seine Zustands- und Programmdaten beschreiben. Die Problematik der Datenreplikation unterscheidet sich von der Prozeßreplikation hauptsächlich dadurch, daß replizierte Daten passiv durch Einwirkung von außen beeinflußt werden, während Prozesse von sich aus aktiv sind und auch selbsttätig weitere Komponenten des Systems beeinflussen können. An den Stellen, an denen die Ausführungen für Daten genauso wie für Prozesse gelten, wird in diesem Papier für Daten und Prozesse der Begriff 'Objekt' gebraucht, der allerdings in diesem Kontext nicht mit den Objekten der objektorientierten Programmierung zusammenfällt. Der Begriff 'Operation' wird in diesem Text ebenso mehrdeutig benutzt. Im Zusammenhang mit Datenobjekten steht er für alle Formen von Anfragen an dieses Datenobjekt (d.h. Lese-, Aktualisierungsanfragen usw.). Bei Prozessobjekten steht 'Operation' für den ganzen Zyklus von Aufruf des Prozesses bis zum Empfang des Ergebnisses.

In den Betrachtungen dieses Beitrags wird grundsätzlich von *fail-stop* Fehlern ausgegangen. Dabei wird angenommen, daß ein Fehlverhalten (fail) eines Knotens immer erkannt wird, und dieser daraufhin seine Arbeit einstellt (stop). Wenn auch fehlerhaft arbeitende Knoten zugelassen sind, wird darauf gesondert hingewiesen.

Da es der Zweck der Replikation ist, fehlerhaftes Verhalten von Komponenten des Systems zu verdecken, soll die Replikation für den Benutzer und die auf das Rechensystem aufsetzenden Prozesse nur in Form größerer Systemverfügbarkeit in Erscheinung treten. Eine Operation mit einem einzelnen Objekt wird auf Kopien des Objekts abgebildet. Dabei muß sichergestellt sein, daß eine Folge von eventuell konkurrierenden Operationen mit replizierten Objekten äquivalent zu derselben Folge von Operationen mit nicht replizierten Objekten ist. Diese Bedingung wird als *Ein-Kopie-Serialisierbarkeits-Kriterium*, im Folgenden kurz EKS-Kriterium, bezeichnet. Das EKS-Kriterium impliziert wechselseitige Konsistenz der replizierten Objekte. Methoden, die die Vervielfältigung von Daten und Prozessen in der oben beschriebenen Weise verwalten, werden *Replikationskontrollalgorithmen* genannt.

Zwei Typen von Fehlern müssen von einem Replikationskontrollalgorithmus behandelt werden: Ausfälle von Rechnerknoten und Ausfälle von Kommunikationsmedien. Der Ausfall eines Rechnerknotens führt dazu, daß die Daten auf diesem Knoten nicht mehr verfügbar sind und auf diesem Knoten gestartete Prozesse nicht mehr zu einem Ergebnis führen. Replikationskontrollalgorithmen haben dafür zu sorgen, daß andere Knoten die Aufgaben des ausgefallenen Knotens übernehmen und fortführen. Viel gravierender, als der Ausfall einzelner Knoten, sind Ausfälle von Kommunikationsmedien, die eine Aufspaltung des Rechnernetzwerks verursachen. Dabei versagen Knoten und Kommu-

nikationsmedien derart, daß das Netzwerk in kleinere Gruppen von Knoten bzw. Partitionen zerfällt. Knoten innerhalb einer Partition können weiterhin miteinander kommunizieren, während die Knoten in anderen Partitionen unerreichbar werden. In dieser Situation ist das EKS-Kriterium nur unter Einschränkung des Zugriffs auf die Objekte aufrechtzuerhalten.

6.2 Replikation von Daten

Bei der Replikation von Daten wird zwischen optimistischen und pessimistischen Ansätzen unterschieden.

Optimistische Verfahren zeichnen sich dadurch aus, daß sie im Falle einer Netzwerkpartitionierung den Zugriff auf die Daten nicht beschränken. Dabei wird von der vereinfachenden Annahme ausgegangen, daß die verschiedenen Gruppen von Knoten keine miteinander kollidierenden Änderungen des Datenbestandes durchführen, so daß bei der Fusion der Gruppen die Datenbestände auf eine einfache Weise wieder zusammengeführt werden können. Unter dieser optimistischen Annahme gibt es natürlich keine Garantie für die Konsistenz des Datenbestandes.

Bei den pessimistischen Verfahren wird dagegen sichergestellt, daß in den verschiedenen Partitionen keine miteinander kollidierenden Änderungen durchgeführt werden. Es wird von einer Worst-Case-Annahme ausgegangen und daher bei einer Netzwerkpartitionierung der Zugriff auf die Daten derart eingeschränkt, daß obige Inkonsistenzen verhindert werden.

6.2.1 Optimistische Ansätze

Wie schon erwähnt wird bei optimistischen Ansätzen der Zugriff auf die Daten nicht eingeschränkt, wenn eine Netzwerkpartitionierung eintritt. Die Aufgaben eines optimistischen Replikationskontrollalgorithmus liegen dementsprechend darin, die Veränderungen des Datenbestandes in irgend einer Form zu protokollieren und im Falle der Wiedervereinigung des Netzwerkes die Datenbestände abzugleichen, wobei das Protokoll dazu dient eventuell aufgetretene Inkonsistenzen zu erkennen.

Beispiel: Optimistische Replikationskontrolle mit Versionsvektoren Der hier vorgestellte Algorithmus bedient sich sogenannter Versionsvektoren. Die einzelnen Datenobjekte (z.B. Dateien oder Datensätze) werden auf jeden Knoten des Netzwerks repliziert. Wenn irgendein Prozeß ein Datenobjekt aktualisiert, wird diese Aktualisierung auf jedem erreichbaren Knoten durchgeführt. Dabei ist jede Kopie des Datenobjekts D mit einem Vektor V assoziiert, in dessen n-ter Komponente die Anzahl der Aktualisierungen, die von Knoten n des Netzwerkes ausgingen, vermerkt wird.

Solange alle Knoten miteinander verbunden sind, werden alle Versionsvektoren gleich aussehen. Tritt jedoch eine Partitionierung des Netzwerkes auf, so können diese divergieren. Ein Versionsvektor V *dominiert* einen Vektor V', wenn jede Komponente $v_n \geq v'_n$ ist. Das heißt: wenn V' von V dominiert wird, ist das Datenobjekt D' ein "Vorfahre" des Datenobjekts D.

Wenn kein Vektor den anderen dominiert, handelt es sich um einen Zugriffskonflikt, der nach einem von der Art der Daten abhängigen Schema oder manuell behoben werden muß.

	K_1		K_2	
Start	$(0,0)$	$\Leftrightarrow$	$(0,0)$	
K_1 aktualisiert	$(1,0)$	$\Leftrightarrow$	$(1,0)$	
Netz zerfällt	$(1,0)$	$\not\Leftrightarrow$	$(1,0)$	
K_1 aktualisiert	$(2,0)$	$\not\Leftrightarrow$	$(1,0)$	
Vereinigung	$(2,0)$	$\Leftrightarrow$	$(2,0)$	K_2 wird akt.
Netz zerfällt	$(2,0)$	$\not\Leftrightarrow$	$(2,0)$	
K_1 aktualisiert	$(3,0)$	$\not\Leftrightarrow$	$(2,0)$	
K_2 aktualisiert	$(3,0)$	$\not\Leftrightarrow$	$(2,1)$	
Vereinigung	$(3,0)$	$\Leftrightarrow$	$(2,1)$	Konflikt!!!

Abbildung 15: Beispiel mit zwei Knoten und einem Datenobjekt

Der Ablauf dieses Verfahrens soll an einem Beispiel mit zwei Knoten und einem einzelnen Datenobjekt erläutert werden (Abbildung 15). Zu Beginn sind beide Knoten noch miteinander verbunden (im Bild dargestellt durch $\Leftrightarrow$). Die erste Aktualisierung, die von K_1 ausgeht, erhöht

die erste Komponente beider Versionsvektoren auf den Wert 1. Dann zerfällt das Netz in zwei Gruppen, bestehend aus den Knoten K_1 und K_2. K_1 aktualisiert das Datenobjekt. Aufgrund der Partitionierung wird nur der Versionsvektor von K_1 geändert. Bei der Wiedervereinigung der Gruppen wird K_2 dann auf den Stand von K_1 aktualisiert. Erneut zerfällt das Netzwerk. Die weiteren Aktualisierungen erfolgen unabhängig voneinander in den einzelnen Gruppen und die Versionsvektoren divergieren. Bei der Wiedervereinigung wird dann ein, von diesem Verfahren nicht behebbarer, Zugriffskonflikt erkannt.

Mit diesem sehr einfachen Verfahren lassen sich nur Aktualisierungskonflikte erkennen. Wenn eine Transaktion auf mehrere Datenobjekte zugreift muß neben dem Write-Write-Konflikt auch der Read-Write-Konflikt behandelt werden. In [Jal94] wird eine Erweiterung dieses Verfahrens angesprochen, die Lese- und Schreibzugriffe protokolliert.

6.2.2 Pessimistische Ansätze

Primary Site Verfahren Zunächst soll die Betrachtung auf den Fall des Ausfalls einzelner Knoten beschränkt bleiben, d.h. alle Knoten sollen erreichbar sein. Des weiteren wird davon ausgegangen, daß die Kommunikationskanäle zuverlässig sind und die Übertragungsreihenfolge beibehalten wird. Das Ziel soll es sein, den Zugriff auf die Datenobjekte bis zu k Ausfällen aufrechtzuerhalten, d.h. *k-fehlermaskierende* (k-resilient) Datenobjekte zu unterstützen.

Um *k-fehlermaskierende* Datenobjekte zu realisieren, müssen diese auf mindestens $k+1$ verschiedene Knoten repliziert werden. Einer der Knoten wird zum *primären* Knoten bestimmt, die anderen zu *Backup*-Knoten. Die $k+1$ Knoten werden logisch linear angeordnet, mit dem primären Knoten am Anfang. Alle Anfragen an ein Datenobjekt werden an den primären Knoten gesandt. Wenn eine Anfrage an einen Backup-Knoten gelangt, wird sie an den primären Knoten weitergeleitet.

Falls es sich um eine Lese-Anfrage an das Datenobjekt handelt, wird die Anfrage einfach vom primären Knoten bearbeitet und das Ergebnis zurückgemeldet.

Handelt es sich aber um eine Aktualisierungs-Anfrage, wird diese Anfrage erst an alle Backups übertragen. Wenn die Backups die Anfrage erhalten haben, führt der primäre Knoten die Operation durch und meldet das Ergebnis zurück.

Fällt der primäre Knoten aus, muß ein Backup dessen Rolle übernehmen. Es gibt verschiedene Methoden, diesen Backup-Knoten zu bestimmen. Ein einfaches Verfahren bedient sich der linearen Anordnung der Knoten. Es wird einfach der erste noch funktionsfähigen Backup-Knoten in der Anordnung zum neuen primären Knoten gemacht.

Wenn mehr als k Knoten ausfallen, kann der Fehler nicht mehr verdeckt werden, da der Grad der Replikation nur $k+1$ beträgt. Fallen nur Backup-Knoten aus, hat dies keine Auswirkungen für den Benutzer, da nur der primäre Knoten mit dem Benutzer kommuniziert.

Alle Anfragen gehen direkt an den primären Knoten. Noch bevor dieser selbst die Anfrage beantwortet wird diese an die Backups weitergegeben. Folglich erhalten alle Backups alle Anfragen in derselben Reihenfolge wie der primäre Knoten und haben somit dieselben Daten wie dieser. Das heißt, daß das EKS-Kriterium immer erfüllt ist, selbst dann, wenn der primäre Knoten ausfällt und ein Backup seine Position einnimmt.

Diese Verfahren kann einfach auf die Behandlung von Netzwerkpartitionen erweitert werden, wenn es möglich ist, den Ausfall von Knoten von einer Partitionierung zu unterscheiden. Da alle Anfragen über den primären Knoten geleitet werden, ist es offensichtlich, daß nur die Gruppe von Knoten Anfragen bearbeiten kann, die im Besitz eines primären Knotens ist. Wenn der primäre Knoten ausfällt, muß ein Backup in dessen Gruppe zum neuen primären Knoten bestimmt werden. Wenn der primäre Knoten aber nur "unsichtbar" wird, darf kein Backup zum primären Knoten erhoben werden, da es sonst zu mehreren primären Knoten kommen kann.

Falls die Anfragen mit aufwendigen Opera-

tionen verbunden sind, empfiehlt es sich die Backups die Anfragen nur mitprotokollieren zu lassen und in gewissen Abständen den Zustand des primären Knotens an die Backups zu übertragen. Ein Backup, der zum primären Knoten wird, muß dann nur die Anfragen bearbeiten, die seit der letzten Zustandsübertragung bei ihm angekommen sind.

6.2.3 Fehlermaskierung mit aktiven Replikaten - Der State Machine Ansatz

Beim Primary Site Verfahren ist nur der primäre Knoten für die Bearbeitung von Benutzeranfragen verantwortlich. Die Backups sind passive Kopien, die nur die Operationen des primären Knotens mitprotokollieren. Dadurch ist es sehr einfach das EKS-Kriterium zu erfüllen. Der hier beschriebene State Machine Ansatz ist nicht zur Behandlung von Netzwerkpartitionen geeignet. Daher soll im Folgenden nur von fehlerhaften Knoten ausgegangen werden. Bei diesem Verfahren sind alle Replikate simultan aktiv. Das System wird als aus Dienstgebern und Dienstnehmern bestehend angesehen: die Knoten, die Kopien der Daten besitzen sind dabei die Dienstgeber, und die Knoten, die Daten anfordern die Dienstnehmer.

Es sollen *k-fehlermaskierende* Datenobjekte zur Verfügung gestellt werden, d.h. Datenobjekte auf die bis zum Fehlverhalten von k Knoten zugegriffen werden kann. Diese werden, falls nur fail-stop Fehler zugelassen sind, durch die Replizierung der Dienstgeber auf $k+1$ Knoten realisiert (Soll der Algorithmus Knoten integrieren, die fehlerhafte Ergebnisse produzieren, dann müssen die Datenobjekte auf $2k+1$ Knoten verteilt werden und die Ergebnisse von Anfragen mit Mehrheitsentscheidung bestimmt werden. Hierzu sei auf [Sch90] verwiesen).

Anstatt die Anfrage an einen ausgezeichneten Knoten zu senden, wird die Anfrage an alle Knoten versandt. Alle Kopien sind gleichwertig: jeder Knoten kann Anfragen bearbeiten und Antworten liefern. Um das EKS-Kriterium zu erfüllen, ist es wichtig, daß alle Knoten identische Kopien der Datenobjekte tragen, wenn sie die Anfrage erhalten, da jeder Knoten die Anfrage beantworten kann.

Wenn im Ausgangszustand alle Knoten identische Kopien des Datenobjekts tragen und jeder Knoten die Anfragen in der gleichen Reihenfolge erhält, werden alle Knoten auch weiterhin identische Kopien des Datenobjekts behalten. Daraus folgt, daß zwei Schlüsselbedingungen erfüllt werden müssen:

Übereinstimmung Alle Anfragen müssen an alle fehlerfreien Replikate übertragen werden.

Ordnung Alle Replikate müssen die Anfragen in der gleichen Reihenfolge erhalten.

Die Übereinstimmungsbedingung lässt sich durch die Benutzung eines geeigneten Übertragungsprotokolls erfüllen.

Die Ordnungsbedingung kann folgendermaßen erfüllt werden: jeder Anfrage eines Dienstnehmers an einen Dienstgeber wird eine einmalige Kennung zugeordnet. Die Dienstgeber bearbeiten die Anfragen entsprechend einer Totalordnung auf der Menge der Kennungen. Der Dienstgeber muß dementsprechend vor Bearbeitung einer Anfrage sicherstellen, daß keine Anfragen mit niedrigeren Kennungen mehr eintreffen können. Eine Anfrage, der keine Anfragen mit niedrigeren Kennungen nachfolgen können wird als *stabil* bezeichnet. Mit anderen Worten: Ein Replikat bearbeitet als nächstes die stabile Anfrage mit der kleinsten Kennung. In [Sch90] werden verschiedene Methoden zur Erzeugung von Kennungen und für den Stabilitätstest erläutert.

Mögliche Optimierungen Falls es sich nur um fail-stop Fehler handelt, kann man bei Anfragen, die den Zustand des Replikats nicht verändern, die (aufwendige) Übereinstimmungsbedingung außer Acht lassen. Es genügt dann, die Anfrage an einen einzigen noch funktionsfähigen Knoten zu übertragen, da die Antwort eines nicht fehlerhaften Knotens sicher richtig ist.

Zwei Anfragen r und r' an ein Datenobjekt heißen kommutativ, wenn die Ergebnisse der

Anfragen und der Endzustand des Datenobjekts von der Reihenfolge in der r und r' bearbeitet werden unabhängig sind. Für kommutative Anfragen kann auf Erfüllung der Ordnungsbedingung verzichtet werden.

6.2.4 Replikationskontrolle mit Abstimmung (Voting)

Statische Abstimmverfahren: Gewichtetes Abstimmen Beim gewichteten Abstimmen erhält jede Kopie eines Datenobjektes eine festgelegte Anzahl von Stimmen. Jeder Knoten, der eine Lese-Anfrage an dieses Datenobjekt richten will, muß zunächst mindestens v_l Stimmen von den Kopien des Datenobjekts erhalten, um die Lese-Anfrage durchführen zu können. Entsprechend erfordert eine Schreib-Anfrage an ein Datenobjekt mindestens v_s Stimmen. Die Wert v_l und v_s werden als Lese- bzw. Schreibquorum bezeichnet.

Sei die totale Anzahl der Stimmen v. Die Quoren müssen die beiden Folgenden Bedingungen erfüllen:

1. $v_l + v_s > v$
2. $v_s > \frac{v}{2}$

Die erste Bedingung stellt sicher, daß sich Lese- und Schreibquorum überschneiden. Das garantiert nicht nur, daß Lese- und Schreiboperationen nicht gleichzeitig stattfinden können, sondern auch, daß jedes Lesequorum ein Datenobjekt enthält, das aktuelle Daten trägt.

Die zweite Bedingung verhindert Write-Write Konflikte, da keine zwei Schreiboperationen zur selben Zeit stattfinden können. Außerdem wird verhindert, daß nach einer Netzwerkpartitionierung in mehr als einer Partition Schreiboperationen stattfinden können.

Die Schreib- und Leseoperationen funktionieren wie folgt: Jedes Replikat ist mit einer *Versionsnummer* versehen, die auf 0 initialisiert wird. Wenn eine Transaktion eine Schreib- oder Leseoperation durchführen will, wird zunächst ein Rundruf durchgeführt, der die Knoten dazu auffordert, Stimmen für diese Anfrage abzugeben. Alle Knoten, die diese Anfrage erreicht antworten darauf mit der Versionsnummer und der Anzahl der Stimmen des Datenobjektes. Falls die für die Anfrage nötige Stimmanzahl erreicht worden ist, führt der Knoten die angefragte Operation durch.

Im Falle einer Leseoperation werden die Daten von dem Knoten mit der höchsten Versionsnummer angefordert. Da Lese- und Schreibquorum sich immer überschneiden trägt das Datenobjekt mit der höchsten Versionsnummer sicherlich die aktuellen Daten.

Damit die Leseoperation in der beschriebenen Art und Weise auch wirklich funktioniert, muß sichergestellt sein, daß tatsächlich alle Datenobjekte in einem Schreibquorum aktualisiert werden. Deswegen kontrolliert der schreibende Knoten nach erfolgter Operation, ob alle Datenobjekte aktualisiert wurden.

Mit dem gewichteten Abstimmen kann man den Ausfall von Knoten und Netzwerkpartitionen behandeln, ohne zwischen diesen beiden Fällen unterscheiden zu müssen. Falls ein Knoten die für eine Anfrage nötige Anzahl von Stimmen nicht erhält, wird er die Operation einfach nicht durchführen, egal ob die Ursache dafür der Ausfall einzelner Knoten oder eine Netzwerkpartitionierung ist.

Wenn ein Netzwerk in mehrere Gruppen zerfällt sind generell folgende Szenarien denkbar:

1. Eine Gruppe hat ein Lese- und ein Schreibquorum (d.h. $v_{l\&s} \geq \max\{v_l, v_s\}$), während alle anderen Gruppen keins von beiden haben. In dieser Gruppe sind alle Anfragen durchführbar, in den anderen gar keine.

2. Einige Gruppen haben ein Lesequorum, aber keine Gruppe hat ein Schreibquorum. In diesem Fall können Leseanfragen in einigen Gruppen durchgeführt werden, aber die Kopien des Datenobjekts bleiben unverändert.

3. Keine Gruppe hat ein Lesequorum (und damit auch kein Schreibquorum). In keiner Gruppe können Anfragen bearbeitet werden. Dies kann passieren, wenn die Gruppen klein werden.

Es ist einsichtig, daß mit dem gewichteten Abstimmen Situationen verhindert werden, in denen Inkonsistenzen auftreten können. Allerdings kann es aufgrund der statischen Stimmverteilung schnell zu Konstellationen kommen, in denen gar keine Operationen an den Datenobjekten mehr möglich sind. Weiter unten werden daher Verfahren vorgestellt, die die Stimmvergabe dynamisch an den Zustand des Netzwerks anpassen.

In homogenen Systemen wird gewöhnlich davon ausgegangen, daß jedem Datenobjekt eine Stimme zukommt. In diesem Fall gibt es für die Bestimmung der Quoren v_l und v_s zwei extreme Varianten, die hier kurz erwähnt werden sollen:

1. *Einer lesen, alle schreiben*:

   Dabei sind $v_l = 1$ und $v_s = v$ (damit $v_l + v_s > v$ gilt). Es kann jeder einzelne Knoten gelesen werden, aber an einer Schreiboperation müssen alle Knoten teilnehmen. Selbst wenn nur ein einzige Kopie eines Datenobjekts unerreichbar wird, kann das Datenobjekt nicht mehr aktualisiert werden.

2. *Mehrheitliches Abstimmen*:

   Hierbei gilt $v_l = v_s = \lceil \frac{v}{2} \rceil$. Für jede Lese- bzw. Schreiboperation muß die absolute Mehrheit der Stimmen erlangt werden. Im Falle einer Netzwerkpartition sind dann in der Partition, die die absolute Mehrheit der Knoten enthält (so es diese gibt) alle Operationen erlaubt, in den anderen keine.

Hierarchisches Abstimmen Es ist offensichtlich, daß der obige Ansatz einen hohen Kommunikationsaufwand erfordert. Kurz erwähnt werden soll, daß es verschiedene Ansätze dazu gibt, diesen Aufwand durch Hierarchisierung des Abstimmprozesses zu reduzieren [Jal94].

6.2.5 Abstimmung mit dynamischer Adaption

Nachfolgend sollen zwei Verfahren beschrieben werden, die die Abstimmung dynamisch an die Situation im Netz anpassen. Dabei soll das Mehrheitliche Abstimmen als Grundlage dienen.

Dynamische Abstimmung Die Technik der Dynamischen Abstimmung unterscheidet sich insofern von der oben erwähnten Mehrheitlichen Abstimmung, als sie die Mehrheit der Stimmen zum *Zeitpunkt der Anfrage erreichbaren Datenobjekte* erfordert.

Jedem Datenobjekt D_n auf dem Knoten n wird eine Versionsnummer $v_{D;n}$ zugeordnet, die die Anzahl der erfolgreichen Aktualisierungen dieses Datenobjektes angibt. Die höchste Versionsnummer bezeichnet die aktuellen Datenobjekte. Eine Gruppe von Knoten heißt *Mehrheitspartition*, wenn sie die Mehrheit der aktuellen Datenobjekte enthält. Außerdem ist jedem D_n ein Zähler $a_{D;n}$ beigefügt, der angibt, wie viele Datenobjekte an der letzten Aktualisierung teilgenommen haben. Zu Beginn ist $a_{D;n}$ gleich der Anzahl der Kopien des Datenobjekts.

Ein Knoten kann ein Datenobjekt aktualisieren, wenn es zu einer Mehrheitspartition gehört. Daher muß der Knoten zunächst ermitteln, ob das Datenobjekt einer Mehrheitspartition zugeordnet werden kann. Der Knoten fordert hierzu von allen Datenobjekten, die erreichbar sind, $v_{D;n}$ und $a_{D;n}$ an. Dann wird das Maximum v_{max} aus den $v_{D;n}$ und das Maximum a_{max} aus den $a_{D;n}$ ermittelt. Wenn die Anzahl der Datenobjekte deren $v_{D;n}$ gleich v_{max} ist kleiner ist als $a_{max}/2$, dann stellt die Menge der Datenobjekte keine Mehrheitspartition dar und die Anfrage wird zurückgewiesen. Ansonsten bilden die Datenobjekte eine Mehrheitspartition und alle Datenobjekte werden aktualisiert.

Sobald ein Knoten feststellt, daß das von ihm verwaltete Datenobjekt nicht aktuelle Daten enthält, muß der Knoten das Datenobjekt aktualisieren. Dazu muß der Knoten allerdings erst sicherstellen, daß er zu einer Mehrheitspartition gehört. Ansonsten wird das Datenobjekt nicht aktualisiert.

Dynamische Stimmen-Neuverteilung Während die Technik der dynamischen Abstim-

mung ihre Sicht stets auf die Mehrheitspartition reduziert, arbeitet die dynamische Stimmenneuverteilung mit einer Anpassung der Gewichtung der einzelnen Datenobjekte.

Dabei sind wieder verschiedene Verfahren zur Verteilung der Stimmen an die verbliebenen Datenobjekte denkbar. Hier sollen nur zwei Erwähnung finden.

Beim ersten Verfahren erhält ein einzelnes Datenobjekt die Stimmrechte des ausgefallenen Datenobjekts und übernimmt damit vollständig dessen Funktion. $v(D)$ sei die Anzahl der Stimmen eines Datenobjekts D. Nun soll das Datenobjekt B die Funktion eines ausgefallenen Datenobjekts A übernehmen. Das Neue $v(B)$ muß nun so gewählt werden, daß B die Stimmrechte von A und B abdeckt. Wenn durch $v(B) := v(B)+2v(A)$ definiert wird steigt die Anzahl der Stimmrechte insgesamt um $2v(A)$ und die absoulte Mehrheit um $v(A)$. Es kann gezeigt werden, daß jede absolute Mehrheit mit den Stimmen von B gebildet werden kann. Als Nebeneffekt wird das Datenobjekt B "mächtiger", da es jetzt häufiger zur Bildung von Mehrheiten benötigt wird.

Beim zweiten Verfahren werden die Stimmen des ausgefallenen Datenobjekts auf eine Gruppe von Datenobjekten verteilt. Falls sich in der Mehrheitspartition n Datenobjekte befinden kann ein Datenobjekt D dadurch ersetzt werden, daß den n Datenobjekten $\lceil \frac{2v(D)}{n} \rceil$ Stimmen zusätzlich zugeordnet werden.

Falls nun ein Datenobjekt wieder Anschluß an die Mehrheitspartition findet, so müssen seine Stimmrechte ebenfalls an die neue Situation angepasst werden, da es sonst eventuell keine Bedeutung bei Abstimmung mehr hat. Eine Möglichkeit besteht darin, jeden einzelnen Knoten Protokoll darüber führen zu lassen, wieviele Stimmen er durch den Ausfall eines bestimmten Datenobjektes zusätzlich erhalten hat. Wenn dieser Knoten wieder Anschluß findet werden dann diese Stimmrechte aufgegeben.

Eine andere Möglichkeit wäre die Anzahl der Stimmen eines zurückkehrenden Datenobjektes zu erhöhen. Diese Methode wird die Gesamtzahl der Stimmen so lange erhöhen, bis alle Datenobjekte wieder erreichbar sind. Dann werden die Stimmrechte wieder auf den Ausgangszustand zurückgesetzt.

6.3 Replikation von Prozessen

Das Ziel der Replikation von Prozessen liegt darin, daß eine verteilte Berechnung auch dann zu einem Ergebnis führt, wenn einer oder mehrere der beteiligten Prozesse fehlschlagen.

6.3.1 Grundsätzliche Überlegungen am Beispiel von Remote Procedure Calls

Ein Remote Procedure Call (RPC) ist eine Erweiterung des Prozeduraufrufkonzeptes für verteilte Systeme. Ein Prozeß, der auf einem Knoten abläuft kann eine Prozedur auf einem anderen Knoten aufrufen. Wenn eine Prozedur A eine Prozedur B aufruft wird A der *direkte Aufrufer* von B genannt. Wenn zwei, oder im Falle geschachtelter RPCs mehrere, Prozesse an der Berechnung beteiligt sind, birgt dies die Möglichkeit, daß ein Teil der Berechnung aufgrund des Ausfalls eines Knotens fehlschlägt. Wenn eine per RPC aufgerufene Prozedur nicht ordnungsgemäß terminiert, kann dies dazu führen, daß der aufrufende Prozeß "unendlich" lange auf ein Ergebnis wartet. Entsprechend kann es passieren, daß eine Prozedur keinen Aufrufer mehr hat, an den sie ihr Ergebnis zurückmelden kann.

In den unten stehenden Verfahren wird eine Prozedur immer als deterministisch betrachtet, d.h. von einem gegebenen Startzustand aus wird bei gleicher Eingabe immer dieselbe Folge von Befehlen ausgeführt.

Primary Site Verfahren Dieser Ansatz verwendet Prozesspaare, wobei der eine Prozeß, wie in Abschnitt 6.2.2, der *primäre* und der andere der *Backup* Prozeß ist. Wenn der primäre Knoten einen RPC erhält wird eine Kopie an den Backup Knoten gesandt. Wenn der primäre Knoten bei der Ausführung des Aufrufs versagt wird der Backup Knoten aktiv.

Beim Einsatz dieses Verfahrens kann es passieren, daß eine Prozedur zweimal ausgeführt wird, bevor es zu einem Ergebnis kommt - einmal vom primären Knoten, dann vom Backup. Dies kann in einigen Fällen inakzeptabel sein. Als Erweiterung dieses Schemas wäre es deswegen vorstellbar den Ausführungszustand der Prozedur immer wieder an den Backup Knoten zu übertragen, so daß dieser mit dem zuletzt übermittelten Zustand fortsetzen kann.

Replizierter Aufruf Eine andere Methode, einen RPC gegen Fehler abzusichern, liegt darin, die aufgerufene Prozedur auf viele Knoten zu verteilen. Solange mindestens eine dieser Prozeduren bis zum Ende kommt, wird der Aufrufer ein Ergebnis erhalten. Hier soll ein Ansatz Namens *Circus* beschrieben werden.

In einem Circus werden Prozeduren auf viele Knoten repliziert. Die Menge der Replikate einer Prozedur wird als *Troupe* bezeichnet. Immer wenn eine Prozedur aufgerufen wird, führen alle Mitglieder der Troupe diese aus.

Die Troupe, die eine RP aufruft, wird *client troupe* genannt und eine Troupe, die aufgerufen wird, wird *server troupe* genannt.

Wenn eine client troupe einen RPC ausführt ruft jedes der c Mitglieder der troupe alle Mitglieder der server troupe auf, d.h. jedes der s Mitglieder der server troupe erhält c Aufrufe. Die server troupe soll allerdings nicht $c * s$ Aufrufe, sondern nur s Aufrufe ausführen. Deswegen werden alle RPCs mit Sequenznummern versehen. Da die Prozeduren deterministisch sind, werden gleiche Aufrufe, die gleichen Sequenznummern tragen. Mit dieser Informationen können die Mitglieder der server troupe überzählige Aufrufe ausfiltern und, bis auf die Adresse, an die das Ergebnis geliefert werden soll, verwerfen.

In der Implementation eines circus wartet ein client troupe Mitglied normalerweise auf die Ergebnisse aller Mitglieder der server troupe. Wenn das Versagen eines server troupe Mitglieds offenbar wird, wird dessen Ergebnis verworfen. Falls es sich nur Ausfälle von Prozessen handelt können, alle Ergebnisse bis auf eins verworfen werden. Ansonsten bietet diese Methode die Möglichkeit das "richtige" Ergebnis durch Mehrheitsentscheidung auszuwählen.

Dieses Verfahren ist in Bezug auf die Anzahl der durch das Netzwerk übertragenen Nachrichten recht kostenintensiv. Die Anzahl der Nachrichten ist $O(c * s)$. Wenn das Netzwerk multicasting, also die gleichzeitige Übertragung von Nachrichten an mehrere Adressaten, unterstützt, werden nur $O(c + s)$ Nachrichten gebraucht.

Ein kombinierter Ansatz Fehlermaskierung wird hierbei dadurch realisiert, daß Kopien einer Prozedur auf verschiedenen Knoten zur Verfügung stehen. Jede Kopie wird als *Inkarnation* bezeichnet. Wie beim Primary Site Ansatz sind die Inkarnationen logisch linear angeordnet, wobei für die $i$te Inkarnation, die $i+1$te den Backup bildet. Aufrufe werden an die primäre Inkarnation geleitet, die die erste der Liste ist, die nicht ausgefallen ist. Der Aufruf wird dann jeweils an die nächste funktionsfähige Inkarnation übertragen, so daß schließlich alle Kopien der Prozedur den Aufruf durchführen. Falls die primäre Inkarnation ausfällt übernimmt deren Backup diese Rolle und liefert das Ergebnis.

Falls eine Prozedur einen weiteren RPC durchführt, wird der Aufruf nur von der primären Inkarnation getätigt. Die Backups führen diesen Aufruf nicht aus, sie warten nur auf das Ergebnis. Die primäre Inkarnation reicht dieses Ergebnis, analog zu den Aufrufen, an die Backups weiter, bis alle das Ergebnis haben. Falls der direkte Aufrufer den Erhalt des Ergebnis nicht bestätigt, wird es direkt an dessen Backup weitergereicht.

In dem vorgestellten Verfahren gibt es vier Sorten von Nachrichten:

1. *call*: RPC an entfernten Knoten.
2. *result*: Ergebniswerte eines RPC.
3. *done*: Nachricht, die an den Backup gesandt wird, um mitzuteilen, daß der Aufruf beendet ist.
4. *ack*: Bestätigung für den Erhalt einer der obigen drei Nachrichten.

Falls auf die Versendung einer Nachricht nicht in einer angemessenen Zeit mit einer *ack* Nachricht geantwortet wird, wird der Status des angerufenen überprüft. Falls dieser ausgefallen ist wird die Nachricht erneut übertragen, diesmal an den Backup. Diese einfache Regel stellt sicher, daß jede aktive Inkarnation eines clusters die Nachricht erhält.

Ablauf ohne Fehler Abbildung 16 zeigt ein Beispiel, in dem ein cluster A, bestehend aus zwei Inkarnationen, einen cluster B mit 3 Inkarnationen aufrufen soll. Die primäre Inkarnation A_1 des Aufruferclusters sendet eine call-Nachricht an die primäre Inkarnation B_1 des anderen clusters. Zuvor wird eine Kopie der call-Nachricht an A_2 weitergegeben. B_1 startet die Prozedur und leitet die Nachricht an B_2 weiter. B_2 reagiert ebenso. B_3 hat keinen Backup und antwortet B_2 mit einer ack-Nachricht, worauf B_2 eine ack-Nachricht an B_1 liefert und B_1 an A_1. Damit hat der cluster A die Bestätigung für den Prozeduraufruf. Am Ende der Prozedur sendet B_1 eine result-Nachricht an A_1. A_1 gibt diese weiter an A_2. A_2 antwortet mit ack. A_1 sendet ack an B_1. In gleicher Weise propagiert jetzt B_1 eine done-Nachricht an B_2 und B_3.

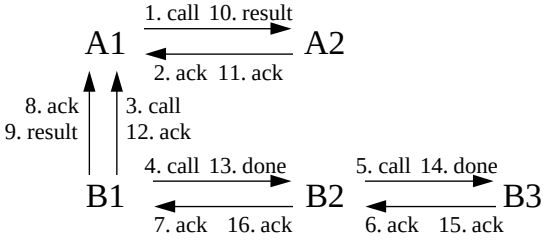


Abbildung 16: Ablauf ohne Fehler

Ablauf mit Fehlern Nachdem ein RPC an den primären Knoten des aufgerufenen clusters gesandt wurde, wird eine ack-Nachricht abgewartet. Falls diese ausbleibt, wird der RPC an den Backup des primären Knotens gesandt. Hat der Backup die call-Nachricht bereits erhalten wird die doppelte Nachricht einfach verworfen. Falls der primäre Knoten ausfällt, nachdem der Aufrufer die ack-Nachricht bereits erhalten hat, hat dies für diesen keine Bedeutung mehr, da der Backup die Funktion des primären Knotens automatisch übernimmt.

Falls der primäre Knoten des aufgerufenen clusters ausfällt, nachdem er die result-Nachricht abgeschickt hat, aber noch bevor er die done-Nachricht an seinen Backup geschickt hat, so wird der Aufrufer mehrere Kopien des Ergebnisses erhalten. Der Aufrufer beantwortet Duplikate einfach mit einer ack-Nachricht.

Falls eine Inkarnation auf eine call- oder done-Nachricht nicht mit einer ack-Nachricht antwortet, dann wird die Nachricht an den Backup der Inkarnation gesandt.

Falls die primäre Inkarnation des Aufruferclusters ausfällt, bevor sie das Ergebnis empfangen und an die Backups weitergeleitet hat, wird deren Backup die call-Nachricht erneut absenden. Der aufgerufene cluster wird diese call-Nachricht verwerfen und sofort mit einer result-Nachricht anworten.

6.3.2 Ausfallsicherheit beim asynchronen Nachrichtenaustausch

Eine Möglichkeit ein System von Prozessen, das mit asynchronem Nachrichtenaustausch kommuniziert, gegen Ausfälle abzusichern, besteht darin, das ganze System in einen früheren, konsistenten Zustand zurückzusetzen. Dieser Ansatz erfordert das Zurücksetzen vieler, eventuell sogar aller noch verbliebenen Prozesse. Eine andere Methode, die nur die fehlgeschlagenen Prozesse zurücksetzt, bedient sich eines Nachrichtenlogbuchs. Hier wird der betroffene Prozeß auf einem anderen Knoten von einem zuvor gesicherten Zustand aus neu gestartet. Nach dem Zurücksetzen ist der Zustand des Systems wahrscheinlich noch nicht konsistent, da eventuell Nachrichten an den fehlgeschlagenen Prozeß verloren gegangen sind, oder dieser Prozeß bereits Nachrichten an andere Prozesse versandt hat, von denen der neugestartete Prozeß nichts weiß. Um sicherzustellen, daß das System einen konsistenten Zustand erreicht müssen diese Nachrichten wiederhergestellt und erneut übertragen werden. Dafür wird das Nachrichtenlogbuch geführt, in dem

während des normalen Ablaufs Nachrichten gespeichert werden, die dann zum Zeitpunkt der wiederherstellung verfügbar sind.

6.4 Völliger Systemausfall und Bestimmung des letzten ausgefallenen Systems

Ein *Totalausfall* liegt dann vor, wenn alle Prozesse, die gemeinsam an einer Aufgabe arbeiten ausfallen, oder, im Falle der Datenreplikation, alle Datenobjekte nicht mehr zugänglich sind.

Per Definition kann ein Totalausfall von keinem der oben diskutierten Verfahren verdeckt werden. Es kann aber mindestens versucht werden den Zustand vor dem Totalausfall wieder herzustellen, wenn die Systeme neu gestartet werden.

Dazu muß festgestellt werden können, welches Objekt als letztes "am Leben" war, da dieses die aktuellste Sicht des Gesamtsystems besitzt.

In den nachfolgenden Betrachtungen wird davon ausgegangen, daß die Anzahl der zusammengehörigen Objekte statisch ist und daß die Zugehörigkeit eines Objektes zu einer Objektgruppe allen anderen Objekten dieser Gruppe ebenfalls bekannt ist. Es soll angenommen werden, daß der Ausfall eines Objektes von den anderen erkannt werden kann. Diese Erkennung erfolgt üblicherweise zeitlich versetzt, da häufig timeout Verfahren benutzt werden. Diese Ausfalldetektion kann durch eine *failed(i)*-Nachricht modelliert werden, die den Ausfall des Objekts i mitteilt und erst mit einiger Verzögerung den Adressaten erreicht. Da die Zeit bis zur Erkennung und die Zeit des Ausfalls eines Objekts nicht deterministisch sind besteht das generelle Problem die Menge von Objekte zu bestimmen, die zuletzt ausfielen.

Vorbemerkungen In einem verteilten System ist es grundsätzlich problematisch anzugeben, ob ein Ereignis vor einem anderen stattfand. Eine Teilordnung, in Form einer "geschahvor" Relation, kann auf der Menge der Ereignisse in einem verteilten System definiert werden.

Diese Relation, dargestellt durch $\rightarrow$, ist wie folgt definiert:

1. Wenn a und b Ereignisse im selben Prozeß sind und a in diesem Prozeß vor b stattfindet, dann gilt $a \rightarrow b$.
2. Wenn Ereignis a das Versenden einer Nachricht darstellt und b das Empfangen derselben, dann gilt $a \rightarrow b$.
3. $\rightarrow$ ist transitiv, d.h. wenn $a \rightarrow b$ und $b \rightarrow c$ gilt, gilt ebenso $a \rightarrow c$.
4. Zwei Ereignisse konkurrieren wenn weder $a \rightarrow b$ noch $b \rightarrow a$ gilt.
5. Aus $a \rightarrow b$ folgt, daß a das Ereignis b beeinflussen kann. Konkurrierende Ereignisse haben keinen Einfluß aufeinander.

Genauso wie der Ausfall eines Objekts mit der *failed(i)*-Nachricht modelliert wurde, kann die Reihenfolge der Ausfälle mit $\rightarrow$ beschrieben werden: Seien i und j zwei Objekte. Genau dann gilt i *fav* j (*fav* steht für "fiel aus vor"), wenn ein Ereignis a in j existiert, so daß gilt *failed(i)*$\rightarrow a$.

Mit obigen Relationen läßt sich nun die Menge L spezifizieren, die die zuletzt ausgefallenen Objekte enthält: $L = \{j \mid \neg\exists i : (j \ \mathit{fav}\ i)\}$.

Um L nach einem Totalausfall bestimmen zu können muß zu den Objekten in nichtflüchtigem Speicher folgende Information gehalten werden:

1. Die Menge B_i der dem Objekt i bekannten Objekt j.
2. Die Menge $F_i \subseteq B_i$ der Objekte, von denen i eine *failed*-Nachricht erhielt.

B_i und F_i sollen jeweils i enthalten. Die Menge F_i wird *vollständig* genannt, wenn sie alle Objekte enthält, die vor i ausfielen.

Da $i \in L$ nur dann gilt, wenn kein Objekt eine *failed(i)*-Nachricht erhalten hat, folgt: $L = \bigcup B_i \Leftrightarrow \bigcup F_i$.

Nach einem Totalausfall wird nur ein kleiner Teil der Objekte verfügbar sein, und zwar die, die sich wieder "erholt" haben. Sei R die

Verfahren	Partitionierungs-behandlung	Falsch antwortende Prozessoren	Aufwand	Kommunikations-aufwand
Daten				
Versionsvektor	ja	nein	niedrig	niedrig
Primary Site	ja[1]	nein	niedrig	niedrig
State Machine	nein	ja	mittel bis hoch	hoch
Voting	ja	ja	mittel	hoch
Prozesse				
Primary Site	-	nein	niedrig	niedrig
Circus	-	ja	mittel	mittel
Kombiniert	-	nein	hoch	hoch

Tabelle 10: Die Verfahren im Überblick

Menge der Objekte, die wieder erreichbar sind. Während des Totalausfalls ist R leer. Je mehr Objekte wieder erreichbar werden, desto größer wird R.

Um L aus der Teilmenge R bestimmen zu können kann eine vorläufige Menge $L_R = \bigcup B_i \Leftrightarrow \bigcup_{i \in R} F_i$ bestimmt werden, die mögliche Elemente von L enthält. Sicherlich gilt $L_R \subseteq L$.

Bestimmung von L bei vollständigem F_i Falls die F_i vollständig sind, gilt $L_L = L$. D.h. wenn die F_i der Elemente von L vollständig sind, reicht diese Information aus, um L aus den F_i der Elemente von L selbst zu bestimmen. Weiterhin kann gezeigt werden, daß gilt: $L_R \subseteq R \rightarrow L_R = L$. Damit läßt sich ein einfacher Algorithmus zur Bestimmung von L formulieren. Jeder Knoten i, der zu R dazukommt gibt sein F_i per Rundruf bekannt. Daraufhin überprüfen die Empfänger, ob obige Bedingungen erfüllt sind. Wenn ja, dann ist $L = R$ und das System kann seine Arbeit wieder aufnehmen.

6.5 Gegenüberstellung der vorgestellten Verfahren

Die verschiedenen hier vorgestellten Verfahren unterscheiden sich zum Teil recht erheblich im Aufwand, der zu ihrer Implementierung und ihrem Betrieb nötig ist. Tabelle 10 fasst die wesentlichen Eigenschaften noch einmal kurz zusammen. Die Spalte 'Aufwand' bezieht sich dabei auf den Implementierungsaufwand. Die Aussagen der beiden Spalten, die sich auf den Implementierungs- bzw. Kommunikationsaufwand beziehen sind mit Vorsicht zu geniessen, da sich durch geeignete Wahl der zugrundeliegenden Kommunikationsarchitektur bereits viele Voraussetzungen für den jeweiligen Algorithmus erfüllen lassen.

Literatur

[Jal94] P. Jalote: Fault Tolerance in Distributed Systems. Prentice Hall 1994. Kapitel 7 und 8, Seite 257-352.

[Sch90] F. B. Schneider: "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial". *Computing Surveys of the ACM*, 22(4):299-319, Dezember 1990.

[1] Wenn die Partitionierung als solche erkennbar ist.

7 Das fehlertolerante UNIX-System ft(UNIX)

NILS LORENSCHEIT

7.1 Einführung

Das fehlertolerante Multimikrorechnersystem ft(UNIX) ist ein verteiltes UNIX-System, welches Hardwarefehler toleriert. Diese können sporadisch auftreten oder permanenter Natur sein. Hardwarefehler können im Mikrorechner, im Massenspeicher oder im Ein//-Ausgabesystem neutralisiert werden. ft(UNIX) ermöglicht die Einteilung der Prozesse in unterschiedliche Zuverlässigkeitsklassen. Die Effizienz des Datenaustausches zwischen den einzelnen Mikrorechnern und damit die Effizienz des gesamten Systems wird durch ein Nachrichtentransportsystem erhöht, welches für Datenverifikation und Systemsynchronisation zuständig ist. Ferner wird die Fehlertoleranz auf der Verbindung zwischen den Mikrorechnern und der Peripherie durch ein redundantes Bussystem gewährleistet.

7.2 Konzeption und Systemarchitektur

7.2.1 Motivation des Systemdesigning

Mit ft(UNIX) wurde ein fehlertolerantes Multimikrorechnersystem geschaffen, das vor Datenverfälschung und -verlust durch nichtreproduzierbare und permanente Hardwarefehler schützt. Dabei können Fehler im Rechner selbst, auf dem Weg zum Massenspeicher, im Massenspeicher selbst sowie auf dem Weg vom oder zum zeichenorientierten E/A-Gerät auftreten. Um ft(UNIX) optimal einsetzen zu können, sind folgende Vorraussetzungen notwendig:

- Das System sollte transaktionsorientiert arbeiten, d.h. tritt ein Fehler auf, kann das System auf den letzten fehlerfreien Zustand zurüchgesetzt werden.
- Die Anwendung sollte auf konventionellen LAN - Netzen mit UNIX als Betriebssytem verwendet werden können.
- Ein hoher Unabhängigkeitsgrad von der darunterliegenden Hardware sollte für die Software von Nutzen ein.

7.2.2 Prozeßklassen

Der ft(UNIX)-Nutzer benötigt keine Informationen über die Fehlertoleranz oder die Systemstruktur, die Erweiterungen zur Fehlertoleranz sind völlig transparent. Ihm stehen alle konventionellen UNIX-Funktionen zur Verfügung. Seine einzige Aufgabe besteht darin, den Grad der Fehlertoleranz festzulegen, mit dem die Anwendung laufen soll. Auf diese Art und Weise ist es möglich, jede Anwendung nur mit dem Aufwand laufen zu lassen, den sie Wirklich benötigt. Der Anwender kann unter drei verschiedenen Zuverlässigkeitsklassen wählen:

- Klasse-1 Prozesse verhalten sich wie konventionelle Prozesse, d.h. sie produzieren unter Umständen falsche Ergebnisse, wenn es während der Verarbeitung zu Hardwarefehlern kommt.
- Klasse-2 Prozesse laufen doppelt ab, d.h. jeder Prozeß besitzt einen Bruderprozeß, mit dem er sich vergleichen kann. Im Falle eines Hardwarefehlers schlägt der Prozeß fehl. Es kann dann kein gültiges Ergebnis geliefert werden.
- Klasse-3 Prozesse laufen dreifach ab und haben somit die Möglichkeit, einen vielleicht vorhandenen Hardwarefehler zu bemerken, seinen Ursprung zu detektieren und im Falle eines transistenten Fehlers die Datenbestände zu restaurieren bzw. im Falle eines permanenten Hardwarefehlers eine Systemrekonfiguration einzuleiten.

7.2.3 Ziele

Die beiden wichtigsten Ziele sind die Verhinderung des Verlustes von Daten und der Kontrolle des Systemes, wenn ein einzelner Rechner

Abbildung 17: Das Gesamtsystem

fehlerhaft arbeitet. Zusätzlich dazu wird versucht, einen möglichen Fehler zu lokalisieren, ohne Verhaltensänderungen des konventionellen UNIX-Systemes zu verursachen.

7.2.4 Zusätzlich notwendige Hardware

Da das Signalaufkommen für einen Rechnerknoten überhand nehmen würde, wurde ein Nachrichtentransportsystem entwickelt, daß den größten Teil der Kommunikation zwischen Einzelrechner und Netz autonom erledigt. Der Message Transport Controller(MTC), auf dem das Nachrichtensystem(NTS) basiert, rahmt dem Mikrorechner ein und sitzt zwischen ihm und den konventionellen Kontrollern. Die Kommunikation des einzelnen Rechners mit seiner Peripherie kann, wie Bild 17 zeigt, nur durch den MTC hindurch geführt werden. Dabei werden die Daten auf Korrektheit geprüft.

Die Rechner kommunizieren über ein redundantes System von seriellen Verbindungen. Jeder MTC besitzt seine eigene Übertragungsleitung, über die es seine Informationen an alle anderen Rechner schickt und genau soviele Empfangsleitungen, wie Rechner im System enthalten sind. Anhand der Leitung auf der die Informationen übermittelt werden, kann man also feststellen, wo die Quelle sitzt.

Für die Fehlertoleranz bei der Übertragung zwischen zeichenorientierten Ein/Ausgabegeräten und den Rechnerknoten sorgt ein dafür konzi-

Abbildung 18: Die Struktur des Softwarerahmens

pierter Datenbus. Dieser besteht aus zwei getrennten Bussen mit Token, sodaß Inkonsistenzen festgestellt werden können.

7.2.5 ft(UNIX)-Softwarestruktur

Die Fehlertoleranzmechanismen sind Teil der Systemsoftware. Der UNIX-Kernel wird durch eine zusätzliche Softwareschicht eingerahmt, die, für den Anwender transparent, die notwendigen Maßnahmen zur Synchronisation, Sperrung und Datensicherheit gegenüber Hardwarefehlern vornimmt.

Fehlerhafte Prozeßergebnisse, die sich durch unterschiedliche Parameter, unterschiedliche Abfolgen von Systemaufrufen oder einem zu großen Zeitverbrauch bemerkbar machen, werden hier ermittelt. Daher wird der obere Teil des Softwarerahmens durch die Resultatsprotokolle und der Schnittstelle au den Systemaufrufen über-

wacht. Ferner sind im oberen Teil des Softwarerahmens die Zuverlässigkeitsklassen und die dafür notwendigen Prozeduren festgelegt. Im unteren Teil wird die Verbindung zwischen den redundanten Prozessen auf verschiedenen Rechnern festgelegt. Bild 18 zeigt den Aufbau des Softwarerahmens.

Neben den Prozessen selbst wird auch der Datentransfer zwischen Prozeß und E/A-System überwacht. Datentransfers laufen grundsätzlich durch den MTC und aktivieren die Verifikationsmechanismen.

7.3 Das Nachrichtentransportsystem (NTS)

Das NTS ist das Herzstück von ft(UNIX). Es basiert auf einem speziellen Message Transport Controller(MTC), der den konventionellen Kommunikationskontroller, der für gewöhnlich die Kommunikation zwischen Rechner und Peripherie lenkt, ersetzt. Aus der Sicht des einzelnen Rechners sieht das Netz der MTC's wie ein großer globaler Speicher aus, auf den alle Rechner zugreifen können. Jeder MTC besitzt nun einen programmierbaren Filter, der von allen hereinkommenden Daten die für seinen Rechner relevanten Daten herausfiltert und vergleicht bzw. votiert. Unter Votieren versteht man den Entscheidungsvorgang über die Gültigkeiten von Daten bei unterschiedlichen Ergebnissen. Praktisch bedeutet dies in einem Klasse-3 Prozeß , sich für das Ergebnis zweier Rechner zu entscheiden, wenn der dritte Rechner andere Ergebnisse liefert.

Ferner kann der MTC Probleme selbstständig lösen, sodaß er nur in den seltesten Fällen gezwungen ist, seinen Rechner durch einen Interrupt zu unterbrechen. Auf diese Weise verschwindet der Flaschenhals, den die Nachrichtenübertragung zwischen Rechnern für gewöhnlich darstellt. Die für das Nachrichtentransportsystem notwendige Systemsoftware wurde transzendent, d.h. ohne nach außen hin sichtbare Wirkung auf bereits bestehende Strukturen, Eigenschaften und Verhaltensweisen, um den Kernel herumprogrammiert, sodaß fast alle konventionellen Funktionalitäten von UNIX erhalten bleiben. Der neue Softwarerahmen läßt sich, wie bereits erwähnt, in zwei Teile unterteilen, die einen unterschiedlichen Unabhängigkeitsgrad besitzen, ein Teil ist zu den Prozessen um lokaleRechner orientiert, der andere zum Netz.

7.3.1 Der Nachrichtentransport

Die Kommunikation zwischen verschiedenen Rechnern geschieht über ein eigens dafür zur Verfügung stehendes Bussystem(Siehe Bild 19). Jeder Rechner hat seinen eigenen privaten seriellen Anschluß, auf dem nur er Daten sendet und genau soviele mit unterschiedlichen seriellen Leitungen verbundene Receiver, wie andere Rechner sich im Netz befinden. Auf diesen werden nur Daten entgegegenommen. Somit entfällt der Systemaufwand für Anfragen und Zuteilungen von Bussen bzw. die Zuordnung von ankommenden Daten zu einem bestimmten Rechner. Die Folge ist, daß die Leistungsfähigkeit des Nachrichtentransportnetzes nicht leidet, wenn die Größe der einzelnen Datenpakete stark variiert. Auch kleinere Datenmengen oder Signale werden sofort weitergeleitet, was für das Synchronisieren und Sperren von Daten unbedingt notwendig ist.

7.3.2 Datenrepräsentation und Verifikation in einem MTC

Nachrichten werden in aller Regel wie folgt ausgetauscht. Die zu sendende Botschaft wird in einen Buffer geschrieben. Von dort sendet sie der Transmitter. Am anderen Ende der Leitung legt der Receiver die Nachricht wieder in einen Buffer ab, wo der Zielrechner sie entnehmen kann. Der damit verbundene Aufwand ist für ft(UNIX) entschieden zu groß, da alle Rechner alle zum Votieren relevanten Daten an alle im ft(UNIX)-Netz befindlichen Rechner schicken. Hinzu kommt in erschwerenderweise, daß der Empfänger gar nicht alle Nachrichten benötigt.

Der MTC ist ein Kontroller mit einem eigenen Speicher. Dieser ist in genau soviele gleiche Adressräume aufgeteilt. wie es Rechnerkno-

Abbildung 19: Das Nachrichtensystem

ten gibt. Der lokale Rechner, zu dem der MTC gehört, schreibt nur in den Adressraum, der ihm zugeordnet ist(Privat Area). Die anderen Adressräume enthalten jeweils eine Kopie der Privat Area eines anderen Rechners. Wenn ein Rechner in die Privat Area seines MTC's hineinschreibt, werden automatisch alle geänderten Daten an alle andernen Kontroller gesendet, sodaß alle Rechner mit einer geringen Verzögerung alle notwendigen Daten lokal zur Verfügung haben, wenn sie sie brauchen. Wichtig dabei ist, daß der Prozessor fremde Daten mit größerer Geschwindigkeit und ohne zusätzliche Aktion lesen kann, andererseits aber nicht in seiner Arbeit gestört wird, wenn die Datenbestände im MTC-Speicher aktualisiert werden. Es entfallen die Wartezeiten bei der Datenübermittlung.

Bei einer Leseoperation auf ein Objekt im MTC-Speicher, zum Beispiel ein Dateisystemblock, können gleichzeitig begleitende Funktionen, wie eine Verifikation versteckt ablaufen. Jedes Datenobjekt besitzt ein Feld von Registerpaaren, von denen eines auf ein nächstgelegenes gleiches, d.h. redundantes, Datenobjekt in einem Adressbereich, der eiem anderen Rechner angeordnet ist, zeigt. Das andere Register zeigt auf eine Speicherstelle in der Event Unit Map. Für jedes Objekt ist eine Reihe in dieser Map reserviert, so wie für jeden Adressbereich eine Spalte zur Verfügung steht. Wenn nun bei einem Lesezugriff auf ein Objekt einen der redundanten Kopien in ihrem Inhalt von den andernen

Kopien abweicht, so kann dies in der Event und Error Queue festgehalten werden. Nach einem Interrupt leitet der Prozessor dann die Fehlerbearbeitung ein.

7.3.3 Der programmierbare Ereignisfilter

Geräte, die im Hintergrund arbeiten, wie das bei dem MTC der Fall ist, müssen in einem multiprocessing System in der Lage sein, dem Prozessor mitzuteilen, daß sie einen bestimmten Zustand erreicht haben. Innerhalb des MTC gibt es eine Vielzahl von Mechanismen, wie zum Beispiel für Datenverwaltung, Repräsentation oder Verifikation, die an einem bestimmten Punkt(Event) den Prozessor in ihren Ablauf miteinbeziehen. Es gibt verschiedene Typen von Events, jedoch nur eine Basisfunktion, die einen Event auslöst. Es ist dies der Schreibzugriff auf vorher definierte Event Activation Word. Wenigstens jedes letzte Datenwort eines Speichersegmentes ist ein Event Activation Word. Es ist jedoch möglich, zusammen mit dem Objektzugriffstyp, jedes Datenwort eines Objektes getrennt als Event Activation Word zu definieren.

Jedes Objekt besitzt ein Event Reference Register. Der Inhalt dieses Registers zeigt auf eine Stelle in dem Unit Map Register. Außerdem wird im Event Reference Register festgelegt, welche Art (Kombination oder Alternativ) und ob der Event mit oder ohne Interrupt stattfinden soll.

Bei einem kombinierten Event befindet sich eine Reihe von gesetzten Bits im Unit Map Register, die alle einem anderen, am Event teilnehmenden, Objekt zugeordnet sind. Wird auf einem der so klassifizierten Objekte auf das Event Activation Word zugegriffen, wird das betreffende Bit in dem Unit Map Register gelöst. Ist dies das letzte gesetzte Bit, wird die benötigte Event Control Adresse in der Event und Error Queue eingetragen und der Prozessor unterbrochen.

Bei einem alternativen Event wird der Event angestoßen, wenn nur eines der betreffenden, gesetzten Bits gelöscht wird. Ansonsten sind beide Typen gleich. Zusätzlich ist das System in der Lage, ausbleibende Ereignisse mit einem timeout-Event zu beantworten.

7.3.4 NTS-Anwendungen

An zwei Beispielen soll nun die Arbeitsweise des NTS gezeigt werden. Beispiel 1 erläutert die Verifikation der Korrektheit eines Prozesses und damit der zugrundeliegenden Hardware. Ein Prozeß möge redundant auf meheren Rechnern laufen. Seine Einzelschritte und -ergebnisse werden mitprotokolliert. Das Protokoll besteht aus einer Liste von miteinander verketteten Segmenten. Der Listenkopf ist mit der Prozeßstruktur identisch, es gibt Verweise auf das erste und das letzte Element. Innerhalb eines Segmentes befinden sich Zeiger auf das erste und das letzte benutzte Wort. Wie bereits angesprochen besitzt nun jeder MTC die Kopie der Privat Area aller anderen MTC's. Also besitzen auch die MTC's, auf deren Rechnern die redundanten Prozesse laufen, je eine Kopie des Protokolles aller anderen dazugehörigen redundanten Prozesse. Wenn nun auf dem letzten Protokoll auf das letzte Wort des Segmentes schreibend zugegriffen wird, mag das einen Event auslösen, da es sich bei dem letzten Wort eines Segmentes wie erwähnt um ein Event Activation Word handelt. Danach wird, für den Prozess ohne weiteren Aufwand, kontrolliert, ob die Protokolle übereinstimmen.

In Beispiel 2 dreht es sich um den schnellen globalen Sperralgorithmus. Dieser ersetzt bei ft(UNIX) die normalerweise bei Einzelprozessor-UNIX Prozessen übliche Sperre. Der schnelle globale Sperralgorithmus wurde notwendig, da es beim üblichen einfachen Sperren von Objekten zu Problemen mit den redundanten Prozessoren gekommen wäre. Der Algorithmus basiert auf der Prozeßidentifikationsnummer (PID) und erlaubt parallelen Zugriff redundanter Prozesse mit gleicher PID. Für jedes Objekt wird im MTC-Speicher festgehalten, in welchem Zustand es sich befindet und von welchem Prozeß es gerade benutzt wird. Es gibt 5 verschiedene Zustände, in denen sich ein Objekt befinden kann, es sind dies "nicht-im-Gebrauch", "Gesperrt", "im-Gebrauch", "Kon-

flikt" und "Wartezustand". Diese Zustände werden im Objektzugriffstyp-Register abgelegt, wo sich auch ein Verweis auf die logische Oder-Funktion befindet, auf die gleich näher eingegangen wird. Ferner gibt es für jedes Objekt die Möglichkeit, die PID des Prozesses abzulegen, der das Objekt gerade in Gebrauch hat und eine Minimumfunktion, die in der Lage ist, im Konfliktfall demjenigen Prozeß das Nutzungsrecht des Objektes zuzusprechen, der die kleinste PID besitzt.

Ein Prozeß möchte nun ein Objekt benutzen. Er testet mit der logischen Oderfunktion, ob dieses schon in Gebrauch ist. Die logische Oderfunktion fragt nun die Zustände des Objektes in der Privat-Area und allen Kopien von entfernter MTC_Speichern ab. Dies ist mit geringem Zeitaufwand möglich, da alle redundanten Kopien des Objektes über das Registerfeld miteinander verzeigert sind(siehe Kapitel 7.3.2). Sollte das Objekt noch nicht in Gebrauch sein, wird es gesperrt, andernfalls wird der Wartezustand eingetragen. Anschließend wird der Sperrwunsch an die Kopien der Privat Area auf den anderen MTC-Kontrollern gesendet. Nun muß der Prozessor die Zeit abwarten, die für die Übermittlung des Sperrwunsches benötigt wird. Erreicht den lokalen MTC dabei der Sperrwunsch eines entfernten Prozesses, wird in den Konfliktzustand gewechselt, ansonsten gelangt das Objekt in den Besitz des Prozesses. Aus dem Konfliktzustand heraus wird nach einer Wartezeit erneut ein Sperrwunsch gesendet. Hatte der entfernte Prozeß zu einem früheren Zeitpunkt das Objekt lokal belegt, so ist er bereits in den Besitzt des Objektes gekommen. Ansonsten wird mit Hilfe der Minimumfunktion und der PID die Objektvergabe entschieden.

7.3.5 Die Hardwarestruktur

Vorraussetzung für eine effiziente Umsetzung der Softwarekonzepte von ft(UNIX) ist die hardwareunterstützte Kommunikation zwischen den Rechnern. Im folgenden wird der Message Transport Controller(MTC) beschrieben, ohne den das Nachrichtentransportsystem nicht möglich gewesen wäre. Der MTC wurde auf einem Vier-slot-Q-Bus Modul entwickelt und gliedert sich in 5 Bereiche.

- Der Objektorientierter Speicher(OOS): Die Struktur des OOS prägt das gesamte Verhalten des NTS. Zeitkritische Algorithmenteile wie die Listensuche werden durch Hardware realisiert. Der Speicher ist durch Gatterlogik erweitert worden, sodaß Zugriffe, die Objektfunktionen erfordern, direkt durch die Hardware erkannt und optimal unterstützt werden. Der Speicher wurde daher in Objektspeicher für Objektdaten und Deskriptorspeicher für funktionelle Bits zum Erkennen des Funktionstyp unterteilt. In beiden Speichern wurden Verwaltungsstrukturen eingeprägt. In den OOS integriert ist die Objekt-Management-Unit(OMU) mit 2 wichtigen Aufgaben. Zum einen bildet sie den Benutzeradreßraum auf den tatsächlich vorhandenen Speicher ab. Gleichzeitig wird die Adresse der Objektbeschreibung des aktuellen Objektes ermittelt.

- Die Execution Unit(EU): Die EU steuert das MTC, sie ist das Rechen- und Steuerwerk des MTC's. Neben den Verwaltungsaufgaben für Ergebnis-, Fehler- und Timeoutwarteschlange berechnet sie auch die Objektfunktionen wie Addition, Adreßberechnung, Bitmaskierung usw. Die EU hat uneingeschränkten Zugriff auf den objektorientierten Speicher und besitzt eine masterähnliche Rolle am internen Bus. Realisiert wurde die EU durch einen Transputer.

- Die Sender - Empfänger-Einheit: Hier werden automatisch Daten über den Transceiver an andere MTC's gesendet, wenn von seiten des Hostrechners ein Schreibzugriff auf den OOS erfolgt. Andererseits laufen hier die seriellen Verbindungen von allen anderen MTC's zusammen. Der Sender ist vom Systembus durch eine 2 KByte tiefen FIFO - Speicher entkoppelt und empfangene Daten werden automatisch im objektorientierter Speicher abgelegt, wobei parallel Objektmechanismen angestoßen werden, die z.B. zu einer Ereignisauslösung am UNIX-Prozessor führen können. Um nicht-relevante Daten so früh wie möglich auszusondern, besitzen die Empfänger Tabellen, mit denen jedes

Abbildung 20: Aufbau des Kontrollers

Segment im virtuellen MTC-Adreßraum ausgeblendet werden kann. Eintreffende Daten eines ausgeblendeten Segments werden ignoriert.

- Das Systembus-Interface: Stellt die Verbindung zwischen MTC und Host her.

- Der interner Bus und die zentrale Steuerung: Der interne Bus stellt die Verbindung zwischen den bisher besprochenen Teilen her. Diese greifen in erster Linie auf den OOS zu. Dabei muß ein Zugriff exklusiv sein. Arbiterfunktion, d.h. zentrale Buszuweisung nach Abfrage und Zustandskontrolle werden dabei von der zentralen Steuerung übernommen. Der Bus wird im Daten-/Adreßmultiplex betrieben.

7.4 Das Character-Device I/O-System

In der UNIX-Welt gibt es 2 verschiedene Arten von Pheripheriegeräten, blockorientierte und zeichenorientierte. Zu den letzteren gehören unter anderem Terminals und Zeilendrucker. Das Ein-/Ausgabesystem der zeichenorienterten Geräte wird durch eine zusätzliche virtuelle Treiberschicht in die Fehlertoleranz miteinbezogen.

7.4.1 Das Netzwerk

Die Verbindung zwischen der zeichenorientierten Peripherie und den Rechner wird durch

einen seriellen Doppelbus hergestellt. Die Kommunikation ist rahmenorientiert. Jeder Rahmen wird durch eine 16 Bit lange Prüfziffer gesichert, die Länge des Informationsfeldes ist jedoch variabel. Der Zugriff auf den Bus wird durch ein Token-Bus Protokoll gesichert. Dieses wird von Station zu Station in einer Art logischem Ring geschickt. Nur die Station, die gerade das Token besitzt, darf auf den Bus zugreifen. Diese Methode der Buszuteilung ist für ft(UNIX) gut geeignet, da eine Station mit einem Hardwarefehler schnell eingegrenzt werden kann. Jeder der beiden redundanten Token-Busse besitzt sein eigenes Token, eine Synchronisation findet nicht statt.

7.4.2 Der Terminaltreiber

Wie beim konventionellen UNIX werden auch bei ft(UNIX) alle zeichenorientierten Ein-/Ausgabeaktivitäten von einem Terminaltreiber kontrolliert. Und wie bei UNIX werden bei ft(UNIX) alle Informationen über ein zeichenorientiertes Ein-/Ausgabegerät in der tty-Struktur zusammengefaßt. In ft(UNIX) wird die tty-Struktur jedoch in die Datenbasis des Nachrichtentransportsystems verlagert. Dadurch ist jeder Rechner in der Lage, die Kontrolle über die Struktur zu erlangen.

Wäre die Struktur im Prozessor angesiedelt, könnte es Probleme mit dem Gerät geben, wenn der Rechner fehlerhaft arbeitet, d.h. andere Rechener könnten auf das Gerät nicht mehr ohne weiteres zugreifen.

Der tty-Treiber ist in 2 Schichten, den virtuellen tty-Treiber und den lokalen tty-Treiber, unterteilt. Aufgabe des virtuellen tty-Treibers ist es, die redundanten Prozesse auf das Ausgabegerät abzustimmen, sodaß nur eine Ausgabe stattfindet. Der lokale tty-Treiber sitzt eine Schicht tiefer und hat die Aufgabe, das Gerät physikalisch zu steuern.

7.5 Das Dateiensystem

7.5.1 Probleme und Aufgabenstellung

Die einzelnen Massenspeicher, die die Hardware-Grundlage für das Dateisystem bilden, sind jeweils einem bestimmten Rechner zugeordnet. Sollte also ein Rechner ausfallen, sind damit auch die auf seinen lokalen Massenspeicher befindlichen Daten nicht mehr zugänglich. Als Lösung für dieses Problem wurden die lokalen Dateisysteme zu einem globalen, fehlertoleranten Dateissystem zusammengefaßt, daß von jedem Subsystem soviele redundante Kopien auf verschiedenen Rechnern besitzt, wie das gewünscht wird. Dabei besitzen alle Rechner die gleiche einheitliche Sicht auf das Dateissystem. Die einem Prozeß bekannten Informationen über das Dateisystem ändern sich also nicht, wenn dieses durch Rekonfiguration verändert wird. Ein Prozeß ist ferner nicht in der Lage, anhand des globalen Verzeichnisses festzustellen, auf welchem Rechnern sich die Orginale einer Datei befinden bzw. ob sie lokal oder entfernt zur Verfügung steht. Zusammen mit den geheimen Redundanzgeraden der Dateien stellen diese Charakteristika die Transparenz des Systems sicher. Die einzelnen Dateisubsysteme werden während der Initialisierungsphase durch 'mount'-Befehle zum globalen Dateisystem zusammengesetzt. Der Redundanzgrad der einzelnen Dateien richtet sich somit nach dem Redundanzgrad des Subsystems, dieser ist jedoch einstellbar. Der optimale Redundanzgrad sollte sich nach den Kriterien Verfügbarkeit, Zugriffsgeschwindigkeit und Verwaltungsoverhead richten und nicht größer werden als drei.

Aufgrund der Redundanz ist es nicht mehr möglich, das einzelne physikalische Geräte angesprochen werden können, vielmehr werden nun virtuelle Geräte angesprochen, hinter denen sich der Redundanzgrad der physikalischen Geräte verbirgt. Wechselbare Massenspeicher können jedoch als einfach-redundante Subsysteme angesprochen werden. Zu beachten ist ferner, daß der Redundanzgrad von Prozessen unabhängig ist vom Redundanzgrad der von ihnen benutzten Dateien. Daher passieren die Datenströme

zwischen Prozeß- und Dateisystem einen Votiervorgang, der verhindert, daß sich Fehler ausbreiten und den jeweils notwendigen Redundanzgrad anpasst. Hierzu werden die Funktionen des Nachrichtentransportsystems benutzt(siehe auch NTS).

7.5.2 Verwaltung der Dateisysteme und der virtuellen Geräte

Die Strukturen zur Fehlertoleranz wurden innerhalb des Gesamtsystems so angeordnet, daß jeder Informationsfluß zwischen zwei verschiedenen Subssystemen durch sie hindurchgehen muß und die Daten dabei koordiniert und überwacht werden. Es kann deshalb davon ausgegangen werden, daß ein Fehler vom zuletzt aktiven Teil (Prozeß oder Treiber) verursacht worden ist. Liegt der Fehler auf der Treiberseite, übernimmt es ein spezieller, hochzuverlässiger Datenverwaltungsprozeß, mithilfe interner Tabellen die Fehleranalyse einzuleiten und anschließend eine Restauration der Daten vorzunehmen oder die Rekonfiguration der virtuellen Geräte zu veranlassen.

7.5.3 Gerätezustände und Fehlerbehandlungen

Die blockorientierte Peripherie, bzw. deren einzelne Gerätetreiber, wird innerhalb der virtuellen Blockpufferschicht in Tabellen verwaltet, die auch festhalten, ob und wie häufig Fehler aufgetreten sind. Zunächst ist ein Gerät im aktiven Zustand. Bei dem ersten Fehler, der bei der Verifikation der Daten entdeckt wird, ändert sich dieser Zustand des Treibers in suspekt. Solange ein Treiber nicht im suspekten Zustand verbleibt, wird er an allen Aktivitäten beteiligt, seine Ergebnisse werden jedoch nur noch mit Vorbehalten ausgawertet, d.h. im Zweifelsfall wird der Fehler einem suspekten Gerät zugeordnet. Treten verstärkt weitere Fehler im Zusammenhang mit der Nutzung des Peripheriegerätes auf, wird sein Zustand von suspekt in defekt umgewandelt und ein hochzuverlässiger Verwaltungsprozeß veranlaßt die Rekonfiguration des Systems.

7.5.4 Mechanismen zur Verwaltung der virtuellen Geräte

Bei einem Schreib- oder Lesezugriff auf ein virtuelles blockorientiertes Gerät müssen die Aufträge innerhalb der virtuellen Blockpufferschicht auf diejenigen Rechner, die die zugehörigen physikalischen Geräte besitzen, verteilt werden, die Rückmeldungen müssen gesammelt werden und ausgewertet an die darüberliegenden Schichten des fehlertoleranten UNIX-Kerns weitergegeben werden. Zu diesem Zweck bedient sich die virtuelle Schicht einer Tabelle, in der zu jedem virtuellen Gerät verzeichnet ist, auf welchen Rechnern sich die zugehörigen physikalischen Geräte befinden. Diese Tabelle enthält auch Zeiger auf die oben erwähnten Tabellen mit den Zustandsinformationen der einzelnen physikalischen Geräte. Selbstverständlich unterliegen diese Tabellen, wie alle Objekte des fehlertoleranten UNIX-Kerns, auf die von mehreren Rechner zugegriffen wird, der Objektverwaltung.

Der Zugriff auf diese Tabellen durch einen Anwenderprozeß (den Verwalterprozeß für das Datersystem) erfolgt mit Hilfe des Systemaufrufs "ioctl" auf ein spezielles "character device", hinter dem sich die virtuelle Blockpufferschicht verbirgt. Es stehen die Betriebsarten "lesen", "schreiben/verändern", und "warten auf Fehler" zur Verfügung, die dem Verwaltungsprozeß die vollständige Information und Kontrolle über die Konfiguration des Dateisystems ermöglichen.

7.6 Zusammenfassung

Die Fehlertoleranz wird in diesem System im wesentlichen durch die Einkapselung der Rechner durch den Nachrichtentransportkontroller erreicht. Die Daten müssen grundsätzlichen diesen Kontroller passieren, wenn sie von einer Komponente des LAN-Netzes zu einer anderen geschickt werden. Es ist daher recht leicht, eine fehlerhaft arbeitende Komponente zu identifizieren, vorausgesetzt, der MTC arbeitet nicht selbs fehlerhaft. Zudem erscheint eine Erweiterung des Systemes recht aufwendig, da mit je-

dem neuen Rechner auch ein neuer interner Bus für den MTC benötigt wird.

Literatur

[1] F. Demmelmeier, P. Fischbacher, G.Koller
Communications in configurable fault-tolerant and distributed UNIX-Systems
Aus: International Symposium on Fault-Tolerant Computing Systems, Vienna/Austria 1.-4. Juli 1986

[2] G. Färber
Basisfunktionen für die UNIX-Implementierung in einem fehlertoleranten Multimikrorechnersystem
Abschlußbericht

8 Verteilte objektorientierte Datenbanksysteme

FRANK FOCK

8.1 Einführung

Lokale Netzwerke (**L**ocal **A**rea **N**etworks) werden heute in den vielfältigsten Umgebungen eingesetzt. Vornehmlich wurden allerdings bis heute Daten intensive Anwendungen zentral auf Großrechnern (Mainframes) ausgeführt. Workstations bieten heute ein besseres Preis-/Leistungsverhältnis als Mainframes, da ihre Anschaffungs- und vor allem Wartungskosten erheblich niedriger sind. Dezentralisierung wird also immer bedeutender: früher gab der Benutzer am Terminal seine Daten ein und der Großrechner verarbeitete diese zentral, heute sollen Daten soweit wie möglich lokal, d.h. von Workstations eines LAN, verarbeitet werden. So soll eine erhebliche Leistungssteigerung des Gesamtsystems erreicht und gleichzeitig die Hardware-Kosten minimiert werden. Trotzdem soll es auch möglich sein, zentral gemeinsame Daten zu halten, auf die von jeder Station zugegriffen werden kann. Diesen und anderen Anforderungen wollen verteilte objektorientierte Datenbanken gerecht werden.

8.1.1 Was ist ein verteiltes ooDBMS?

Eine *verteilte Datenbasis* ist eine Sammlung von Daten,

- die über mehrere, autonom agierende, über ein Netzwerk verteilte Rechner verteilt sind;
- die logisch zusammenhängen derart, daß Anwendungen bestehen, die auf Daten mehrerer Rechner zugreifen;
- auf die in ihrer Gesamtheit von jedem Rechner des Netzwerks aus zugegriffen werden kann;
- die so verwaltet werden, daß jede Anwendung den Eindruck einer zentralen Datenbasis erhält.

Entsprechend ist ein *verteiltes objektorientiertes Datenbasisverwaltungssystem* (dooDBMS; engl.: distributed object-oriented Data Base Management System) ein Programmsystem, daß auf der Grundlage einer dezentralen Kontrolle eine verteilte objektorientierte Datenbasis verwaltet.

Man kann auch sagen ein verteiltes ooDBMS ist die Kombination von objektorientierter Programmierung (ooP) mit Datenbanktechnologie zur besseren und flexibleren Modellierung von datenintensiven Applikationen.

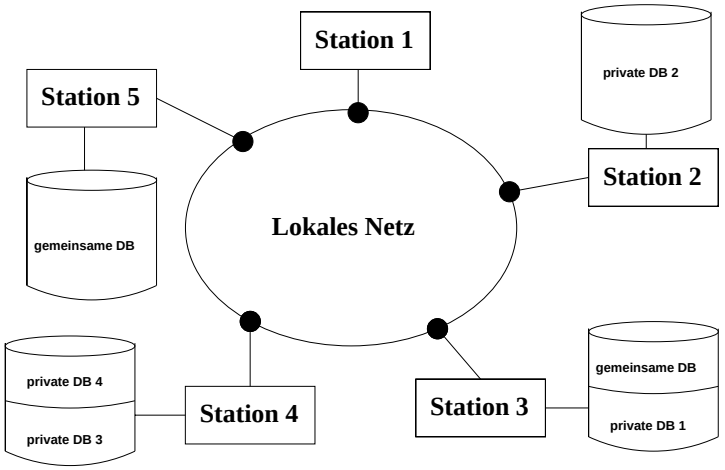


Abbildung 21: Beispiel für eine Architektur eines verteilten ooDBMS

8.1.2 Warum verteilte ooDBMS?

Verteilte Datenbanksysteme bieten generell erhöhte Verfügbarkeit, Zuverlässigkeit und Performance sowie geringere Hardware-Kosten. Der potentielle Performance Gewinn resultiert aus der Möglichkeit, die Bearbeitung von Anfragen über ein geeignetes Netz auf mehrere Prozessoren zu verteilen. Die potentiell erhöhte Zuverlässigkeit entsteht durch die mögliche Daten- und Kontroll Redundanz. Mit einem geeigneten Netzwerk ist es möglich Hard- und Software-Resourcen zu teilen, was die Produktivität erhöht und die Kosten senkt [7].

Ein weiterer, vielleicht sogar der wichtigste, Vorteil ist die Erweiterbarkeit: das System kann leicht an kurz- und langfristige Änderungen angepaßt werden, ohne das dadurch eine signifikante Beeinträchtigung des Systems hervor-

gerufen würde. Kurzfristige Änderungen beinhalten variierende Auslastungen sowie Stations- oder partielle Netzausfälle. Langfristige Änderungen sind solche mit grundlegenden Modifikationen an den Erfordernissen und Inhalten des Systems.

Objektorientierte Datenbanken haben ihre bekannten Vorteile vor allem im Bereich der Datenkonsistenz, der Anwendungswartbarkeit und in der Möglichkeit sehr komplexe Datenstrukturen zu modellieren. Die Kombination dieser Möglichkeiten und der eines verteilten Dantenbanksystems prädestiniert verteilte ooDBMS für den Einsatz in Design-Umgebungen wie z.B. Computer-Aided Design (CAD) und Computer-Aided Software Engineering (CASE). Trotzdem sind Design-Umgebungen nur eines unter vielen Beispielen innerhalb eines weitergefasten Anwendungsgebiets, das man als „datenintensive Programmierung“ bezeichen könnte.

8.2 Design verteilter ooDBMS

Beim Design eines verteilten ooDBMS bieten sich verschiedene Möglichkeiten dieses auf die speziellen Bedürfnisse einer Anwendung anzupassen. Im folgenden interessieren sollen dabei aber nur die Gestaltungsmöglichkeiten die vornehmlich durch die *Verteilung* des ooDBMS entstehen. Die wichtigsten Bereiche, die eine besondere Problematik durch den Aspekt der Verteilung bekommen, sind Replikations-, Zugriffs- und Konsistenzkontrolle sowie Zuverlässigkeit und Transparenz.

8.2.1 Architektur verteilter ooDBMS

Die Verteilung eines DBMS birgt eine Reihe neuer Probleme gegenüber der zentralen Version eines DBMS. Trotzdem ist die Verteilung sehr wichtig, da Anwendungen, die ooDB Technologie benötigen, in der Regel in Netzwerkumgebungen zu Hause sind. Frühe kommerzielle ooDBMS (z.B. GEMSTONE) benutzten Client-/Server-Architekturen. Durch die Entwicklung der Mikroprozessortechnologie konzentriert sicher aber die Leistungsfähigkeit solcher verteilter Systeme immer mehr in den Client-Workstations. Dadurch ist die Verteilung einer ooDB innerhalb eines Netzwerks von Workstations sehr interessant geworden. Tatsächlich existieren schon ooDBMS (z.B. ONTOS und verteiltes ORION [2]), die eine Art von Datenverteilungstransparenz unterstützen.

ORION ist ein Beispiel für ein verteiltes ooDBMS, daß als föderales System konzipiert wurde. Es ist verwaltet gemeinsame und privaten Datenbasen (vgl. Abb. 22). Auf gemeinsame Datenbasen können alle Benutzer zugreifen, während auf private nur ihr Besitzer zugreifen kann. ORION besitzt keine Client-/Server-Architektur, da die gemeinsame Datenbasis auf verschiedene Workstations verteilt sein kann. Bei ORION muß der Anwender nicht wissen auf welcher Workstation sich ein Objekt der gemeinsamen Datenbank befindet, um darauf zugreifen zu können (Abb. 22). Jedoch ist bei Objekten aus einer privaten Datenbasis von vornherein klar, auf welcher Workstation sie liegen. Es ist also nur möglich ein Objekt der gemeinsamen Datenbasis auf eine beliebige Station zu verteilen, während die Objekte der privaten Datenbasen an den Aufenthaltsort dieser gebunden sind. Eine vollständige Verteilungstransparenz (siehe 8.2.5) wäre erst erreicht, wenn der Anwendung auch über den Aufenthaltsort von Objekten der privaten Datenbasen keine Kenntnisse vorlägen.

8.2.2 Zuverlässigkeit

Replikationen. Eine interessante Möglichkeit die Zuverlässigkeit und Leistung eines verteilten ooDBMS zu steigern, ist die Verwendung von Replikationen. Ein Replikationsschema erlaubt es Kopien eines Objekts auf unterschiedlichen Stationen zu halten. Das erlaubt den Ausfall einer Reihe von Stationen ohne Funktionalitätseinbuße. Nur die Kopie der ausgefallenen Station ist dann nicht mehr verfügbar, doch eine Kopie des betroffenen Objekts auf einer anderen Station kann etwaige Anfragen weiterführen. Die wichtigen Probleme, die im Zusammenhang mit Replikationsschemata stehen, sind Konsitenzerhaltung der Kopien, die Synchronisation

der Aktivitäten verschiedener Clients und die Störungen durch partielle Netzwerkausfälle.

Ein einfacher Ansatz ist es, nur unveränderbare Objekte zu replizieren. Diese können ohne Konsitenz- und Synchronisationsprobleme repliziert werden. Leider ist dieses Schema auf den Teil der unveränderbaren Objekte beschränkt, so daß es keine allgemeine Lösung bringt.

Ein alternativer Ansatz ist es, eine Kopie besonders als Primärkopie (*primary copy*) auszuzeichnen. Die anderen Kopien werden als Sekundärkopien (*secondary copies*) bezeichnet, geordnet und auf verschiedene Stationen gehalten. Nichtverändernde Zugriffe (Lesezugriffe) können von jeder Kopie bedient werden. Modifizierende Zugriffe (Schreibzugriffe) können nur von der Primärkopie bearbeitet werden. Die Veränderungen müssen dann von diesem Objekt an die Sekundärkopien weitergegeben werden.

Es gibt zwei Variationen des Primärkopie Schemas: das statische und das dynamische. Im statischen Primärkopie Schema werden modfizierende Anfragen im Fall eines Ausfalls der Primärkopie solange zurückgestellt, bis die Primärkopie wieder verfügbar ist. Im Gegensatz dazu bietet das dynamische Primärkopie Schema auch für modifizierende Anfragen zusätzliche Objektverfügbarkeit. Wenn eine Primärkopie ausfällt (zum Beispiel durch Netzwerkpartitionierung), dann weißt das System einer der Sekundärkopien die Rolle der neuen Primärkopie zu.

Die dynamische Variante birgt allerdings auch zusätzliche Probleme. Wenn eine Sekundärkopie eine Primärkopie ersetzt, so muß sie auch deren Identität und Resourcen (z.B. Schnittstellen, Sperren) übernehmen, damit für die Clients des ausgetauschten Objekts keine Veränderung entsteht. Schwieriger zu behandeln sind dagegen Netzwerkpartitionierungen. Hier kann es passieren, daß die Partitionierung eine Primärkopie von ihren Sekundärkopien trennt, so daß auf beiden Seiten der Teilung jeweils eine Primärkopie mit dazugehörigen Sekundärkopien generiert wird. Wird der Netzwerkfehler behoben, so müssen etwaige Konflikte zwischen den beiden Primärkopien aufgelöst werden.

Der dritte Ansatz für ein Replikationsschema ist das der gleichberechtigten Objekte. In diesem Schema gibt es keine ausgezeichnete Kopie eines Objekts. Schreib- und Leseanfragen können von jeder Kopie bedient werden. Trotzdem ist eine Zusammenarbeit einiger oder aller Kopien nötig, um eine Anfrage zu bearbeiten. Dieses Schema toleriert den Ausfall einer begrenzten Anzahl von Replikate, ohne das Schreibzugriffe zurückgestellt werden müßten. Außerdem ist dieses Schema weniger anfällig gegenüber Netzwerkpartitionierungen.

Rücksetzen. Für die Zuverlässigkeit eines verteilten ooDBMS ist es wichtig, daß es nach Fehlern von Objekten, Workstations oder des Netzwerks wieder in einen stabilen Zustand überführt werden kann. Dies wird im allgemeinen durch wiederherstellen des letzten konsistenten Zustands erreicht, der durch die Festschreibungsprozedur auf Sekundärspeicher geschrieben wurde. Alle nach diesem Zustand bis zum Auftreten des Fehlers ausgeführten Operationen gehen beim *Rücksetzen (roll-back)* verloren. Rücksetzverfahren sind einfach aber effizient und zudem sichern sie fast immer die erfolgreiche Wiederherstellung eines stabilen Systemzustands.

Anders ist dies beim sogenannten *roll-forward* Verfahren: hier wird versucht alle Operationen, die bis zum Zeitpunkt des Fehlers ausgeführt wurden, ebenfalls wiederherzustellen. So sieht es aus, als sei gar kein Fehler aufgetreten. Das Problem dabei ist nur, daß unter Umständen dabei der schon aufgetretene Fehler wieder und wieder erzeugt werden könnte.

Um korrekt zurücksetzen zu können, muß im Sekundärspeicher genug Information vorhanden sein, um ein persistentes Objekt in einen konsistenten Zustand zurückzuführen. Ein dooDBMS kann beim Festschreiben entweder den gesamten Zustand eines Objekts sichern *(checkpoint schemes)* oder nur die relativen Änderungen zu einem vorher gespeicherten Zustand sichern *(log schemes)*.

Beim ausschließlichen Sichern von Objektabzügen, wird der vollständige Zustand eines veränderten Objekts auf Sekundärspeicher gesichert, wenn die zugehörige Transaktion festge-

schrieben wird. Während der Vorabfestschreibungsphase der Festschreibungsprozedur wird die modifizierte Version eines Objekts auf Sekundärspeicher gesichert, ohne die alte Version zu zerstören. In der Festschreibungsphase wird dann einfach die alte durch die neue Version ersetzt. Wird zu irgendeinem Zeitpunkt die Transaktion (siehe 8.2.3) abgebrochen, so wird die modifizierte Version einfach gelöscht.

Ein Vorteil dieses Verfahrens ist die gute Ausnutzung des Sekundärspeichers, da jeweils nur eine Kopie eines Objekts gehalten wird.

8.2.3 Konsistenz

Transaktionen. Eine wichtige Aufgabe eines verteilten ooDBMS ist die Transaktionsverwaltung. Sie stellt (mehr oder weniger) [2] sicher, daß die Daten nach Ausführung einer Aktion wieder konsistent sind. Transaktionen sollten vier Transaktionsparadigmen genügen:

- *Atomizität.* Transaktionen werden entweder erfolgreich abgeschlossen oder haben keinen Effekt.
- *Konsistenz.* Eine Transaktion bewirkt einen konsistenten Datenbasiszustand, sofern sie auf einen konsistenten Datenbasiszustand aufsetzte.
- *Isolation.* Nebenläufige Transaktionen laufen jede für sich so ab, als ob sie für sich allein abliefen.
- *Dauerhaftigkeit.* Die Wirkung einer erfolgreich abgeschlossenen Transaktion geht nicht verloren. Einzige Ausnahme ist ein katastrophaler Systemausfall.

Eine einzelne Transaktion kann mehrere Folgetransaktionen hervorrufen, die ganz unterschiedliche Objekte betreffen können. Eine Transaktion ist nur dann erfolgreich abgeschlossen, wenn alle zugehörigen Transaktionen erfolgreich abgeschlossen wurden. Eine erfolgreich abgeschlossene Transaktion wird *festgeschrieben*, während eine Transaktion, die nicht erfolgreich abgeschlossen werden konnte, *abgebrochen* wird. Wird eine Transaktion festgeschrieben, werden ihre Änderungen auf Sekundärspeichermedien geschrieben. Wenn eine Transaktion abgebrochen wurde, werden alle Änderungen an Objekten rückgängig gemacht (Abb 23).

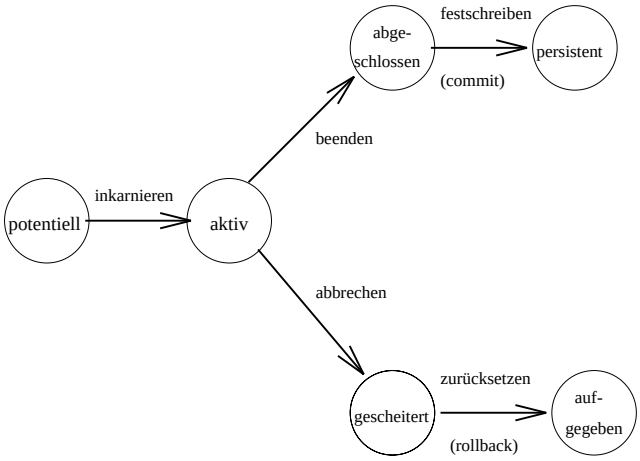


Abbildung 22: Zustandsübergangsdiagramm einer Transaktion

Die Festschreibungsprozedur (commit procedure) sichert die Atomizität. Die bekannteste Prozedur ist das Zwei-Phasen-Festschreibung-Protokoll [1], [8]. Die erste Phase dient dabei der Sicherung der Wiederholbarkeit einer Transaktion, falls diese fehlschlägt. An deren Ende ist die Transaktion unverwunbdbar geworden und kann nun in der zweiten Phase abgeschlossen werden. Dieser Grundgedanke läßt sich auch auf den verteilten Fall übertragen. Die Erweiterung betrifft die erste Phase, in der die Transaktionsverwaltung darauf bestehen muß, daß sich alle beteiligten Knoten in der Entscheidung einig sind. Dies schlägt sich in der *globalen Festschreibungs-Bedingung* nieder:

1. Wenn auch nur ein Agent für einen Abbruch stimmt (beispielsweise weil er einen Transaktionsfehler oder eine Verklemmung entdeckt hat), muß die verteilte Transaktion abgebrochen werden.
2. Stimmen sämtliche Agenten dem Festschreiben zu, so wird die verteilte Transaktion festgeschrieben.

[2] Aus Gründen der Performance kann es nützlich sein, daß die Konsistenz der Daten nicht immer völlig gesichert wird

Dieses Protokoll ist einfach und effizient, trotzdem kann es bei Netzwerkfehlern zu Blockierungen kommen.

Die Verantwortung für den Aufruf der Festschreibungsprozedur liegt entweder beim DB oder seinen Benutzern. Das einfachste Schema das *Anforderungsschema*. Hier ist der Benutzer verantwortlich dafür, wann eine Festschreibung durchgeführt wird. Der Benutzer muß nicht jede Transaktion festschreiben lassen. Das kann die Leistung eines verteilten ooDBMS erheblich steigern, da der Festschreibungsprozess vor allem bei verteilten ooDBMS sehr kostspielig sein kann. Das Problem dabei ist nur, daß die Konsistenz der Daten gefährdet wird: werden zum Beispiel eine Reihe von persistenten Objekten durch eine Transaktion verändert und diese Änderungen nicht festgeschrieben. In der Zwischenzeit verändert eine zweite Transaktion einen Teil dieser Objekte (aber nicht alle) und wird festgeschrieben. Nach einem Systemausfall würde dann ein inkonsistenter Zustand wiederhergestellt, da die Änderungen der ersten Transaktion nur unvollständig wiederhergestellt wurden.

Um sicherzustellen, daß alle Transaktionsparadigmen eingehalten werden, muß die Transaktionsverwaltung vollständig vom System ausgeführt werden. Im einfachen *Transaktionsschema* wird im Sinne des Zwei-Phasen-Festschreibung-Protokolls nach jeder erfolgreichen Änderung eine Festschreibung durchgeführt. komplette Transaktion abgebrochen. Der Nachteil dieses Schemas ist der Overhead durch die Festschreibungsprozedur und daß eine Transaktion komplett abgebrochen wird, wenn nur eine zugehörige Aktion fehlschlägt.

Eine Erweiterung des Transaktionsschemas ist das *verschachtelte Transaktionsschema*. In diesem Schema können Untertransaktionen fehlschlagen, ohne daß notwendigerweise die gesamte Transaktion fehlschlägt. Untertransaktionen können hier je nach Bedarf entweder wiederholt oder einfach ignoriert werden. Natürlich kann dabei auch die gesamte Transaktion abgebrochen werden. Die Änderungen der Untertransaktionen, die erfolgreich abgeschlossen wurden, hängen dann vom gelingen der gesamten Transaktion ab. Das verschachtelte Transaktionschema erlaubt eine feinere Handhabung von auftretenden Fehlern. So können Fehler behandelt werden, solange andere Untertransaktionen fortgeführt werden können. Ein Nachteil dieses Schemas ist allerdings der erhöhte Aufwand der Objektrepräsentation, da Untertransaktionen als ganzes rückgängig gemacht werden können müssen.

Für die Transaktionsverwaltung objektorientierter DBMS spielen vor allem folgende drei Probleme eine Rolle:

1. Objekte können sehr komplex sein, so daß Sperrmöglichkeiten von unterschiedlicher Granularität wichtig sind;

2. Die Unterstützung von dynamischer Schema-Evolution benötigt effizente Algorithmen für die Aktualisierung der Schema-Daten;

3. Um den unterschiedlichen Anforderungen von Zielanwendungen zu genügen, müssen verschiedene Synchronisationsschemata angeboten werden (z.B. pessimistische und optimistische).

Auf 2. wird in 8.2.5 näher eingegangen und auf 1. und 3. weiter unten in diesem Abschnitt.

Aus Gründen der Performance werden verteilte Transaktionenschemata in kommerziellen Systemen nicht im vollen Umfang eingesetzt. Einige (z.B. Oracle) erlauben dem Benutzer nur eine Datenbank gleichzeitig geöffnet zu haben, so daß keine verteilten Transaktionen auftreten können, während andere (z.B. Sybase) die Grundanforderungen des Zwei-Phasen-Festschreibung-Protokolls erfüllen, aber dem Anwender überlassen die Festschreibungen zu koordinieren. Das verteilte ooDBMS erzwingt nicht die Atomizität der Transaktionen, stellt aber die Werkzeuge zur Verfügung, die der Anwender braucht, um diese selbst zu erzwingen. Es gibt allerdings auch verteilte ooDBMS, die das komplette Zwei-Phasen-Festschreibung-Protokoll benutzen (z.B. Ingres und Non-Stop SQL).

Synchronisation. Eine weitere wichtige Funktion eines verteilten ooDBMS ist es zu verhindern, daß mehrere Transaktionen, die dasselbe Objekt betreffen, sich gegenseitig beeinflussen. Ein Objekt darf solange nicht von einer Transaktion verändert werden, solange eine andere ihre Änderungen noch nicht festgeschrieben hat. Um also die Konsistenz und Isolation der Transaktionen zu sichern (s.o.) ist eine *Synchronisation* der Transaktionen notwendig. Es existieren viele Synchronisationsschemata, jedoch kann man alle grob in zwei Klassen einteilen: pessimistische und optimistische.

Beim *pessimistischen* Synchronisationsschema werden Konflikte von vornherein ausgeschlossen. Eine Transaktion, die auf ein Objekt zugreifen möchte, das auch schon von einer anderen benutzt wird, wird solange unterbrochen, bis das Objekt wieder frei ist (d.h. die auf diesem Objekt aktive Transaktion festgeschrieben wurde). Das Zwei-Phasen-Sperr-Protokoll ist das bekannteste pessimistische Synchronisationsschema, aber auch Zeitmarkenverfahren, Semaphoren und Monitore werden benutzt [1].

Beim *optimistischen* Synchronisationsschema werden Konflikte nicht vermieden, sondern bevor eine Transaktion festgeschrieben wird, wird sie auf *Serialisierbarkeit* getestet, um sicherzustellen das ihre Änderungen an einem Objekt nicht mit den Änderungen einer anderen Transaktion kollidieren, die schon festgeschrieben wurde. Das System untersucht also, ob die Daten, die einer Transaktion zu Grunde lagen, noch aktuell sind, oder in der Zwischenzeit durch eine andere Transaktion verändert wurden. Sind sie aktuell, so kann die Transaktion festgeschrieben werden, ansonsten muß sie abgebrochen werden. Um die Wahrscheinlichkeit eines solchen Abbruchs zu reduzieren, sollte die Granularität der Objekte recht fein sein. Das optimistische Synchronisationsschema erlaubt ein Maximum an Nebenläufigkeit, da Transaktionen nie unterbrochen werden (anders im pessimistischen Synchronisationsschema).

Das größte Problem der optimistischen Variante [9] ist, daß, obwohl eine Transaktion erfolgreich abgeschlossen werden konnte, sie unter Umständen trotzdem rückgängig gemacht werden muß. Trotzdem bietet diese Variante gerade für verteilten ooDBMS Vorteile: zum Beispiel bei einer Aufteilung der Daten eines verteilten ooDBMS in private und öffentliche Datenbanken wie bei ORION (siehe 8.2.1) wird schon von vornherein eine Gruppierung der Objekte (Daten) vorgenommen. In der privaten Datenbank einer Station X werden sich sicher schon im Hinblick auf die Netzbelastung nur solche Objekte befinden, die auch hauptsächlich von dieser Station genutzt werden. In den öffentlichen Datenbanken werden sich nur solche Objekte befinden, die nicht einem Bereich zuzuordnen sind und im Klassenschema eher in der Nähe der Wurzel anzusiedeln sind. Daher werden sich Konflikte im Normalfall in Grenzen halten und ein Maximum an Nebenläufigkeit bleibt gewährleistet. Denn das ist ja gerade der Sinn eines *verteilten* ooDBMS.

Natürlich hat auch die Kombination von pessimistischen und optimistischen Synchronisationsschema ihre Vorteile. So kann man zum Beispiel das pessimistische Schema für Anfrage verwenden, die mehrere Objekte betreffen, und das optimistische nur für solche, die nur ein Objekt betreffen.

Zu erwähnen bleibt noch der erhöhte Aufwand Verklemmungen zu lösen, der durch die Verteilung von objektorientierten Datenbasen entsteht. Verklemmungen werden auch hier dadurch gelöst, daß ein Wartegraph aufgebaut wird und dieser auf Zyklen untersucht wird. Nur reicht es in diesem Fall nicht einen lokalen Wartegraphen eines Knotens (Workstation) zu erstellen, es muß vielmehr ein globaler Wartegraph (GWG) erstellt werden. Trotz einiger Optimierungsmöglichkeiten [1] entsteht ein erheblicher Kommunikationsaufwand, um Zyklen im GWG zu erkennen. Das ist der Grund für die Attraktivität des Zeitmarkenverfahren, obwohl es unnötig viele Transaktionen zurücksetzt. Man kann aber trotzdem mit Sperren arbeiten und das Problem der Zyklensuche umgehen, indem man jeder Anforderung eine Zeitschranke setzt, innerhalb derer sie befriedigt werden muß. Sehr schwierig gestaltet sich hierbei allerdings die Wahl der Länge der Zeitschranke. Ist sie zu groß oder zu klein sinkt der

Durchsatz des Systems, zugleich hängt die optimale Größe auch von der Systemlast ab.

8.2.4 Sicherheit

Die Sicherheit eines verteilten ooDBMS ist ein oft vernachlässigter Aspekt. Jedoch gerade in Multiuser-Systemen, wo die verschiedenen Benutzer unterschiedlichen Sicherheitsstufen unterliegen und so auf unterschiedlichen Objektmengen zugreifen können, wird Sicherheit immer wichtiger.

Ein verbreitetes Sicherheitsschema ist das *Potentialschema* [10], daß den Objektschutz in das Namensschema integriert. Die Kennung eines Objekts besteht dabei aus zwei Feldern: dem Namensfeld und dem Berechtigungsfeld. Das Namensfeld spezifiziert das entsprechende Objekt, das Berechtigungsfeld spezifiziert die Daten/Operationen des Objekts, die angefordert werden können. Ein Berechtigungsfeld bezieht sich auf genau ein Objekt, jedoch kann ein Objekt mehrere Berichtigungsfelder besitzen. So kann der Besitzer eines Objekts für unterschiedliche Clients unterschiedliche Rechte vergeben.

Ein anderer Sicherheitsmechanismus ist das *Kontroll-Prozedur-Schema* [11]. Hier hat jedes Objekt eine spezielle Prozedur durch die alle eingehenden Anforderungen gehen müssen. Sie prüft die Autorisation des Clients und terminiert alle ungültigen Aufrufe. Diese Verfahren ist sehr flexibel und kann fast jedes Sicherheitsschema unterstützen.

8.2.5 Transparenz

Ziel eines verteilten ooDBMS sollte es sein, den Datenzugriff durch den Benutzer von den Einzelheiten der darunterliegenden Implementation zu trennen. Der Benutzer sollte nicht wissen müssen, an welchem Ort des Netzwerks sich ein Objekt befindet oder wo und wieviele Replikationen eines Objekts existieren. Dieses wird durch verschiedene Arten von *Transparenz* erreicht: *Verteilungstransparenz* und *Replikationstransparenz*. Die Datenbank sollte also für den Benutzer eine logische Einheit bilden (auch wenn sie physikalisch verteilt ist), so daß dieser auf die verteilte Datenbank zugreifen kann, als ob sie eine zentralisierte sei. Die Anfragesprache eines transparenten verteilten ooDBMS unterscheidet sich also idealerweise nicht von der eines herkömmlichen DBMS.

Volle Transparenz ist allerdings kein allgemein anerkanntes Ziel. Es wird argumentiert, daß volle Transparenz die Handhabung, die Modularität und vor allem die Performance beeinträchtigt. Trotzdem sollte in Zukunft das verteilte ooDBMS volle Transparenz bereitstellen, denn der Benutzer sollte im Sinne der einfachen Handhabung eines solchen Systems von Kenntnissen über Details der Implementation verschont bleiben.

Man unterscheidet im Allgemeinen vier Arten von Transparenz:

1. *Verteilungstransparenz* (auch als *Netzwerktransparenz* bezeichnet) besagt, daß Anwendungen die Existenz des Netzwerks nicht wahrnehmen. Dies schlägt sich im wesentlichen in zwei Eigenschaften nieder. *Ortstransparenz* stellt eine Anwendung von der Kenntnis des Orts der Speicherung frei und macht sie dadurch ortsunabhängig. *Namenstransparenz* sorgt für netzweit eindeutige Namensgebung aller Objekte, ohne diese Namensgebung auch der Anwendung abzuverlangen.

2. *Replizierungstransparenz* verbirgt eine etwaige Replizierung von Daten vor den Anwendungen, so daß diese nicht durch den Zusatzaufwand mehrfacher Änderungen oder Wahl der günstigsten Kopie belastet werden.

3. *Fragmentierungtransparenz.*
   Werden Fragmente eines Objekts oder Klasse von Objekten auf unterschiedliche Knoten verteilt, so soll die Beschaffung sowie Ort und Art ihrer Verbindung bei Anfragen der Anwendung verborgen bleiben.

4. *Fehlertransparenz* verbirgt Ausfälle von Knoten oder Kommunikationsverbindungen sowie Netzwerkpartintionierungen vor den Anwendungsprogrammen.

Wenn das verteilte ooDBMS eine Anfrage bearbeitet, so muß es zunächst feststellen, welches Objekt betroffen ist und auf welcher Workstation sich dieses befindet, um ihm die Anfrage zusenden zu können. Jedes Objekt muß eine (netzweit) eindeutige Kennung besitzen, die sich über die gesamte Lebensdauer des Objekts nicht ändern und nach einer Benutzung nicht wiederbenutzt werden darf. Der Lokalisierungsmechanismus muß flexibel genug sein, um Objekten zu erlauben von einer Workstation zu anderen bzw. von einer privaten zu einer öffentlichen Datenbank (und umgekehrt) zu wechseln.

Die einfachste Methode ist es, den Aufenthaltsort eines Objekts in die Objektkennung zu codieren. Das System weiß also schon durch die Objektkennung auf welcher Workstation sich das betreffende Objekt befindet. Der Nachteil dieses Ansatzes ist es, daß ein Objekt während seiner gesamten Lebensdauer an einen Ort gebunden ist.

Ein anderer Ansatz ist das der *verteilten Namesgebern*. Bei diesem Schema erstellt das System eine Gruppe von Namensgeberobjekten, die auf einer Zahl aber nicht unbedingt auf allen der Workstations gehalten werden. Diese Objekte kooperieren untereinander, so daß sie zusammen alle Informationen über den Aufenthaltsort eines jeden Objekts besitzen. Es gibt zwei Variationen dieses Schemas. Bei der ersten Variation hat jedes dieser Objekte die gesamte Ortsinformationen, so daß jedes für sich alle Ortsanfragen bearbeiten kann. Bei der zweiten Variante besitzt jedes dieser Objekte nur einen Auschnitt der Gesamtinformationen. Kann ein Objekt eine Anfrage nicht beantworten, so gibt sie diese weiter. Das Hauptproblem dieser Lösung ist die Synchronisation zwischen den Objekten und die Notwendigkeit, die Namensgeberobjekte zu benachrichtigen, sobald ein Objekt seinen Ort wechselt.

Ein weiterer Ansatz ist das Cache/Broadcast-Schema. Jede Workstation hält einen kleinen Cache der Aufenthaltsorte der zuletzt angeforderten Objekte. Steht der Aufenthaltsort eines angeforderten Objekts im Cache, so wird die Anfrage an die betreffende Workstation gesendet. Wenn das Objekt dort nicht mehr existiert, wird eine entsprechende Nachricht zurückgesendet. Ist der Cache veraltet oder enthält er das angeforderte Objekt nicht, so wird eine Nachricht an alle Stationen gesendet (broadcast), um den richtigen Aufenthaltsort des entsprechenden Objekts zu ermitteln. Jede Workstation, die eine solche Nachricht erhält, sucht intern nach dem angegebenen Objekt. Diejenige Workstation, die es gefunden hat, meldet den Aufenthaltsort der anfragenden Workstation zurück und deren Cache wird aktualisiert.

Dieses Schema kann sehr effizient sein, da der Aufenthaltsort eines Objekts im lokalen Cache gefunden werden kann. Trotzdem ist es flexibel, da ein Objekt von einem Ort zum anderen wechseln kann und das, ohne daß andere Workstations oder ein Namensgeberobjekt benachrichtigt werden müssen. Von Nachteil ist nur, daß das Netzwerk durch die ungezielten Anfragen stark belastet wird und daß alle Workstations dabei belastet werden, obwohl nur eine direkt betroffen sein müßte.

Vorwärtsverweisende Zeiger können die meisten der oben genannten Schemen erweitern. Der vorwärtsverweisende Zeiger ist die Referenz zum neuen Aufenthaltsort eines Objekts und befindet sich am vorherigen Aufenthaltsort des Objekts. Beim Suchen eines Objekts muß das System also nur dieser Kette der Vorwärtsverweise folgen, um zum gesuchten Objekt zu gelangen. Von Nachteil ist allerdings der erhöhte Aufwand für die Zeigerverwaltung. Außerdem wird das System mit zunehmender Zahl dieser Verweise störanfälliger, da durch Netzwerk- oder Worksationausfälle einige Zeiger verloren gehen können. Denkbar wäre hier, daß diese Vorwärtsverweise ab einer gewissen Kettenlänge oder zu lastarmen Zeiten aufgelöst werden.

8.2.6 Schema-Management

Die Problematik des Schema-Managements wird am Beispiel des föderalen verteilten ooDBMS ORION (Abb. 22) deutlich. ORION [2] benutzt ein Schema (eine logische Klassenhierachie) für alle (private und gemeinsame) logischen Datenbanken des Systems. Anders als

beim Multi-Schema Ansatz hat jede Datenbank zwar ein eigenes Schema, jedoch sind diese voneinander nicht unabhängig, sondern Teil eines globalen Schemas. Abbildung 24 zeigt den Multi-Schema-Ansatz bei dem jede Datenbasis als unabhängige Datenbasis aufgefaßt wird, die dann auch eine eigene logische Klassenhierachie besitzt. Abbildung 25 zeigt den Ein-Schema-Ansatz bei dem die Datenbasen nicht unabhängig voneinander sind, so daß also ein einziges globales Schema existiert. Das Schema der gemeinsamen Datenbank ist eine Untermenge des globalen Schemas; trotzdem beinhaltet es auf jeden Fall die Wurzel der globalen Hierachie. Das Schema der privaten Datenbasis ist ebenso eine Untermenge des globalen Schemas. Sie kann aus einer oder mehreren Teilhierachien der globalen Klassenhierachie bestehen. Der Teil des globalen Schemas, der zur gemeinsamen Datenbasis gehört, wird in jedem Knoten repliziert. Mit anderen Worten: die Klassenschema der gemeinsamen Datenbasis ist für alle Knoten gleich. Als Erweiterung zum gemeinsamen Schema besitzt jeder Knoten ein Schema für seine private Datenbasis. Wenn nun ein Objekt einer Klasse C einer privaten Datenbasis Objekt der gemeinsamen Datenbasis wird *(check in)*, dann wird das globale Schema automatisch so aktualisiert, daß die Definitionen der Klasse C und alle ihrer Oberklassen (direkte und indirekte) Teil des Klassenschemas der gemeinsamen Datenbasis werden.

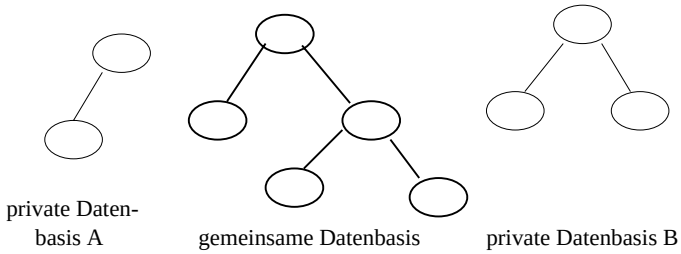


Abbildung 23: Multi-Schema-Ansatz

Ein Nachteil des Ein-Schema Ansatzes ist die Vererbung von Änderungen des Klassenschemas der gemeinsamen Datenbasis an alle privaten Datenbasen. Enfernt ein Benutzer ein Attribut einer Klasse der gemeinsamen Datenbasis, so wird dieses Attribut auch aus allen Unterklassen (auch aus denen von privaten Datenbasen)

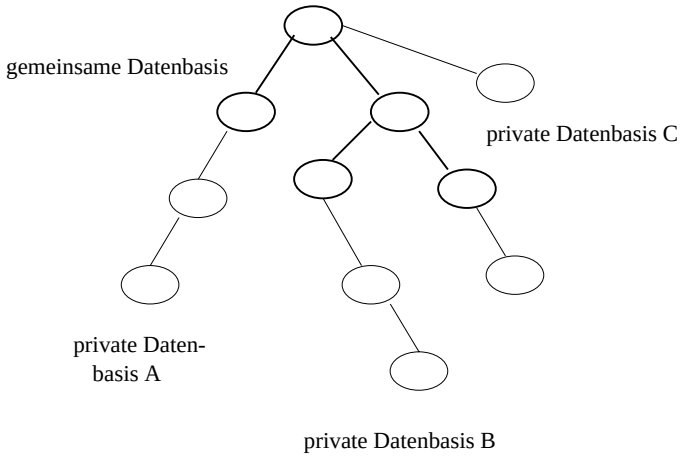


Abbildung 24: Ein-Schema-Ansatz

dieser Klasse entfernt. Eine Lösung dieses Problems würde die Einführung von Versionen des gemeinsamen Klassenschemas einzuführen. Ein Benutzer könnten dann eine neue Version des gemeinsamen Klassenschemas herleiten, die die gewünschten Änderungen wiederspiegelt, anstatt sie direkt an der logisch einmaligen Kopie des gemeinsamen Schemas durchzuführen. ORION unterstützt keine Versionen, so daß Änderungen an dem gemeinsamen Klassenschema nur möglich sind, wenn diese die Schemata der privaten Datenbasen nicht berührt (ausgenommen die private Datenbasis des Benutzers, der Änderungen durchführen möchte).

Der Multi-Schema-Ansatz ist in sofern einfacher zu handhaben, als jeder Benutzer sein eigenes Schema besitzt und dort alle Änderungen am Schema vornehmen kann. Kompliziert wird es allerdings, wenn man ein Objekt von einer privaten in die gemeinsame (oder umgekehrt) wechseln soll, denn dann müssen die entsprechenden Teile der Schemata der privaten und gemeinsamen Datenbasen miteinander synchronisiert werden. Wenn ein Instanz einer Klasse C von der gemeinsamen in eine private Datenbasis PDB wechselt (check out), so muß die Klassendefinition von C erst Teil des Schemas von PDB werden. Umgekehrt muß eine Klassendefinition C aus PDB zuerst Teil des Schemas der gemeinsamen Datenbasis werden, damit eine Instanz von C von PDB zur gemeinsamen Datenbasis wechseln kann. Das ist ein schwieriges Datenbank-Administrationsproblem für den Benutzer, da dieser entscheiden muß, wo die Po-

sition der Klasse C in der privaten Datenbasis zur Checkout-Zeit ist und wo in der gemeinsamen Datenbasis zur Checkin-Zeit. Außerdem kann C geerbte Attribute und Methoden ihrer Oberklassen enthalten, so daß nicht nur C sondern auch ihre Oberklassen Teil des Schemas der gemeinsamen bzw. privaten Datenbasis werden müssen. Bedingt durch diese Probleme wird der Multi-Schema-Ansatz in der Praxis selten angewandt.

8.3 Ungelöste Probleme

Verteilte ooDBMS haben heute noch nicht den Stand erreicht, den man sich von dieser Technologie wünschen würde. So wurde schon oben erwahnt, daß auf volle Netzwerktransparenz aus Gründen der Systemleistung bisher weitgehend verzichtet wurde. Im Zusammenhang mit der Netzwerktransparenz steht auch die *verteilte Anfragenbearbeitung* (engl.: *distributed query processing*). Immer wichtiger werden Realzeitanforderungen für DBMS, so auch für verteilte ooDBMS. Für Prozessteuerung in Betrieben wird es gerade bei verteilten DBMS nicht ausreichen das System einfach „schnell genug“ zu machen. Aber hier liegt gerade eine Chance der verteilten ooDBMS: in der unternehmensweiten Integration von Daten und Anwendungen von der Entwicklung (CAD) bis zur Fertigung (CAM).

8.3.1 Verteilte Anfragenbearbeitung

Die wesentliche Vorraussetzung für die Einführung verteilter ooDBMS war die Annahme, daß die überwiegende Zahl der Anfragen lokal beantwortet werden kann. Die physische Verteilung der Datenbasis wird durch Fragmentierung des globalen Klassenschemas (siehe 8.2.6), Replizierung und Ortszuweisung auf hohe Verarbeitungslokalität hin ausgerichtet. Andererseits wird die Verknüpfung von lokalen Datenbanken zu verteilten Datenbanken damit begründet, daß es eine genügende Anzahl von Anwendungen gibt, die auf die globalen Datenbasis Bezug nehmen und deshalb zu ihrer Bearbeitung mehrere Knoten in Anspruch nehmen müssen.

Obwohl es einige Vorschläge für Algebren objektorientierter Datenbanken gibt [3], so bleibt die Frage der Anfrageoptimierung bisher ungelöst. Warum ist aber eine Optimierung objektorientierter Anfragen so schwierig? Bei relationalen Datenmodellen wird eine die Anfragen ausdrückende Algebra konstruiert, die dann optimiert wird. Das geht im relationalen Datenmodell so einfach, weil Anfragen mit Hilfe von wohldefinierten Operatoren und sehr einfachen Strukturen (z.B. normalisierte Relationen) ausgeführt werden. Im objektorientierten Modell dagegen können Anfragen Operatoren von gerade neu definierten abstrakten Datentypen enthalten. Jeder neue Typ, der Operatoren einführt, erstellt eine neue Algebra, deren Eigenschaften der Anfragenoptimierung unbekannt sind. Wenn aber die algebraischen Eigenschaften dieser neuen Operatoren nicht bekannt sind, ist es schwierig die Anfrageausdrücke in equivalente, effektivere Formen zu transformieren.

Die Objektkapselung birgt noch ein anderes Problem. So nützt es nichts transformierte Versionen von Anfragen produzieren zu können, solange man nicht die relativen Kosten der Ausführung dieser Anfragen kennt. Die Ausführungskosten hängen typischerweise von der Struktur der Speicherung der Objekte und deren Aggregate ab. Ein Beispiel: wenn eine Menge S als Binärbaum eines Attributs A implementiert ist, so wird das Suchen eines Attributs A in S relativ günstig sein. Das Wisssen um solche Implementierungseinzelheiten widerspricht aber gerade dem Grundsatz der Objektkapselung, aber selbst beim übergehen dieses Grundsatz bleibt die Aufstellung einer Algebra schwierig.

Weiter kompliziert wird die Optimierung der Anfragen durch die Verteilung der Objekte über das Netzwerk. Hier spielen noch Faktoren wie Auslastung der betroffenen Workstations und der Durchsatz des Netzwerks eine Rolle. Problematisch dabei ist nur, daß diese Faktoren zeitabhängig sind und dadurch in keinem Fall im voraus bestimmt werden können. Untersuchungen haben sogar gezeigt, daß im Zusammenhang mit Netzauslastungen auch keine statistischen

Vorhersagen möglich sind und daß daher eine zufällige Verteilung der Last die beste Auslastung bzw. Performance des Netzes erreicht.

8.3.2 Realzeit-Transaktionssysteme

Realzeit-Transaktionssysteme werden für eine Reihe von Anwendungen immer wichtiger. Ein Beipiel für ein Realzeittransaktionsystem ist ein CAM System, bei dem das System den Zustand von Maschinen überwacht, Pressen in der Produktion steuert und statistische Informationen über den Produktionsverlauf sammelt. Damit direkt verbunden ist die CAD Abteilung des Unternehmens, um einen nahtlosen Übergang von Design und Produktion zu gewährleisten. Einige Transaktionen müssen in diesem System also innerhalb einer festen Frist (Zeitschranke, engl.: deadline) abgewickelt sein. Zum Beipiel muß die Information über die Form eines Objekts aktualisiert werden, bevor ein Team von Roboter dieses bearbeiten kann. Die Aktualisierungstransaktion wird aber nur dann als erfolgreich betrachtet, wenn die Daten konsistent für alle Roboter und innerhalb eines festgelegten Zeitintervalls geändert wurden, damit die Roboter mit einer konsistenten Sicht der Situation beginnen können zu arbeiten.

Realzeit-Transaktionen sind komplex, da sie Protokolle benötigen, die neben der Datenkonsistenz auch Zeitvorgaben berücksichtigen müssen. Die Algorithmen und Protokolle müssen Prozessplanung, Synchronisation, Konfliktauflösung, Transaktionsreaktivierung, Verklemmungen, Puffermanagement und Festplatten I/O-Planung integrieren. Jeder dieser Algorithmen oder Protokolle sollte selbst direkt auf Realzeitanforderungen zugeschnitten sein. Dabei wird klar, daß es eben nicht reicht allein das dooDBMS Realzeitanforderungen zu unterwerfen, sondern daß hier eine Integration von Betriebssystem und DBMS notwendig ist (8.4).

Um die Serialisierbarkeit von Transaktionen zu sichern, kann man entweder eine Realzeitversion des Zwei-Phasen-Sperr-Protokolls oder des optimistischen Synchronisationsverfahrens einsetzen. Das optimistische Synchronisationsverfahren hat sich jedoch effektiver als das Zwei-Phasen-Sperr-Protokoll herausgestellt, wenn es zusammen mit prioritätsabhängige Prozessteuerung in Realzeit-Datenbanksystemen integriert wurde.

Zusätzlich zu Zeitschranken gibt es in vielen Realzeit-Transaktionssystemen Transaktionen unterschiedlicher Prioritätsstufen. Diese kann im Zusammenhang mit der verbleibenden Ausführungszeit stehen. Prioritätsstufen und Zeitschranken sind zwei charakteristische Eigenschaften von Realzeit-Transaktionen, aber sie korrelieren nicht notwendigerweise. Eine Transaktion mit früher Zeitschranke hat nicht unbedingt eine hohe Priorität und umgekehrt. Trotzdem kann man grundsätzlich sagen: je größer die Priorität einer Transaktion ist, desto größer ist deren Wert für das System. Der Wert einer Transaktion ist aber auch zeitabhängig, denn eine Transaktion, die innerhalb ihres gegebenen Zeitintervalls nicht abgeschlossen werden konnte, ist sicher dem System weniger Wert, als wenn sie erfolgreich abgeschlossen worden wäre.

Entscheidend, ob eine Transaktion in Echtzeit ausgeführt werden kann, sind aber nicht nur die oben aufgeführten Anforderungen an die Transaktionsverwaltung, sondern auch die Echtzeitfähigkeit der Betriebssysteme und vor allem des Netzwerks auf die das verteilte ooDBMS aufbaut. Mit Realzeitbetriebssystemen wurden schon einige Erfahrungen gesammelt (Prozessteuerung), doch besteht im Bereich der Netzwerke bis zur Größe von MANs (Metropolitan Area Network) noch großer Forschungsbedarf. Mit einfachen Datagrammdiensten auf die dann erst verbindungsorientierte Dienste aufbauen, lassen sich keine Realzeitanforderungen erfüllen und rein verbindungsorientierte Dienste sind zu unwirtschaftlich. Ein Kompromiß der in die Richtung einer Realzeitfähigkeit von Netzwerken geht, ist das ATM-Protokoll (Asynchronous Transfer Mode). Dieses erlaubt es einer Verbindung Mindest-Transferraten zuzusichern, ist aber trotzdem in darunterliegenden Schichten auf verbindungslose Dienste aufgebaut.

8.4 Integration von dooDBMS und Betriebssystemen

Bestehende Betriebssysteme (OS von engl.: Operating Systems) besitzen nicht die Funktionalität, um verteilte ooDBMS zufriedenstellend zu unterstützen. So werden von verteilten ooDBMS Funktionen benötigt wie

- Steuerung verteilter Transaktionen (inklusive Synchronisation und Rücksetzen)
- effizientes Management verteilter, persistenter Daten
- Objektidentitätsverwaltung
- Realzeit-Prozessteuerung

die bestehende Betriebssysteme nicht oder nur unzureichend zur Verfügung stellen (Abb. 26 und 27). Zudem benötigen verteilte ooDBMS auch Änderungen in traditionellen Bereichen der Betriebssysteme (zum Beipiel Taskverwaltung und Puffermanagement).

Betriebssysteme die diese Lücke schliessen möchten, sind verteilte Betriebssysteme (DOS, von engl. distributed operating system) und verteilte objektorientierte Betriebssysteme. Ein verteiltes objektorientiertes Betriebssystem unterstützt Objekt-Abstraktion auf Betriebssystemebene, die es zum Beispiel erlaubt Objekte von verschiedenen Benutzern gemeinsam benutzen zu lassen. Durch die Objektorientierung von Betriebssystemen und Programmiersprachen wird immer weniger zwischen beiden unterschieden. Diese Entwicklung ist Teil eines Prozesses den Nicol [12] *vollständiges System-Design* nennt. Diese Entwicklung dient dazu, auf niederer Ebene effizentere Unterstützung, der auf höherer Ebene entwickelten, Abstraktionen zur Verfügung zu stellen.

Trotzdem besteht das Integrationsproblem nicht zwischen Betriebsystem und DBMS, sondern auch die Netzwerk-Protokolle müssen berücksichtigt werden. Das macht das Problem noch komplexer, denn verteilte Betriebssysteme benutzten in der Regel ihre eigenen Kommunikationsdienste, obwohl diese an den allgemeinen Standards vorbei gehen. Dabei wird es aber

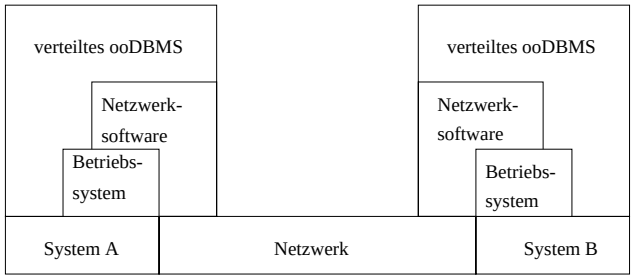


Abbildung 25: Struktur einer Umgebung eines verteilten ooDBMS aufbauend auf herkömmliche Betriebssysteme

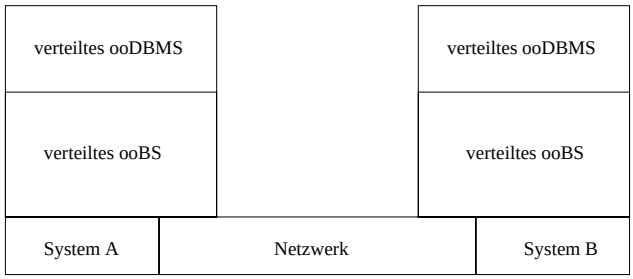


Abbildung 26: Struktur einer integrierten Umgebung

dann schwierig diese Betriebssysteme auf unterschiedliche Netzwerke zu portieren. Die Architekturanforderungen an ein verteiltes objektorientiertes Betriebssystem sollten also flexibel genug sein, um einerseits Funktionen verteilter ooDBMS und verteilter objektorientierter Betriebssysteme zu unterstützen und andererseits Kommunikationsprotokoll-Standards wie ISO/OSI oder IEEE 802 [4] gerecht zu werden. Die Implementation von DBMS Funktionen in das Betriebssystem muß und soll dabei gar nicht umfangreich ausfallen. Es genügen die oben genannten Funktionen, die eben im Betriebssystem effizienter umgesetzt werden können. Auf Benutzerebene können dann ungehindert andere DBMS Funktionen effizient implementiert werden. Dazu am besten geeignet ist das Client/Server-Modell. Beispiele für solche Betriebssysteme sind Amoeba [5] und Mach [6]. Die Objektorientierung von Betriebssystem und DBMS macht die Integration natürlich sehr viel leichter, weil damit schon gleiche Grundstrukturen geschaffen sind. So braucht das ooDBMS sich zum Beispiel nicht um die Objektidentität zu kümmern, da dieses Management vom objektorientierten Betriebssystem schon angeboten wird.

Die Namensgebung ist ein fundamentaler Mechanismus der einem Betriebssystem zur Verfügung steht, um transparenten Zugriff zu Systemresourcen zu gewährleisten. Ob der transparente Zugriff auf verteilte Objekte schon auf Betriebssystemebene geschehen soll, ist eine sehr umstrittene Frage. Dafür sprechen der einfache Benutzerzugriff auf verteilte Objekte und die Möglichkeit Objekte beliebig innerhalb des Systems zu verschieben, was Lastausgleich durch das System ermöglicht. Dagegen spricht der Aufwand, der für den transparenten Zugriff geleistet werden muß, den gar nicht alle Applikationen benötigen und damit unnötige Kosten zu tragen haben. Davon abgesehen ist die Transparenz für verteilte ooDBMS ein sehr wichtiger Punkt. Viele der existierenden verteilten Betriebssysteme haben versucht ihr eigenes Namesschema durchzusetzen, allerdings ohne wirklich Erfolg damit zu haben. Hier gibt es noch viel Raum für Untersuchungen, die den Zusammenhang zwischen Namensschemata, verteilten Namensverzeichnissen und Betriebssystem-Namensgebern ausleuchten.

Für DBMS ist die Verwaltung von *persistenten Daten* sehr bedeutsam. Bei herkömmlichen Betriebssystemen werden persistente Daten vom Dateisystem verwaltet. Bei einer erfolgreichen Kooperation zwischen DBMS und Betriebssystem könnte man sich sogar Vorstellen, daß das DBMS die Funktion des Dateisystems übernimmt. Auch hier sind noch viele Fragen in Sachen Kooperation zwischen DBMS und Betriebssystem unbeantwortet. Die Suche nach Antworten, die Verwaltung persistente Daten betreffend, geht hauptsächlich in drei Richtungen:

- Probleme von ooDBMS mit traditionellen Dateiverwaltungssystemen,
- Persistenz-Unterstützung durch Datenbank-Programmiersprachen und in welcher Art solche Sprachen Betriebssystem-Funktionalität unterstützen können und
- Probleme, die speziell im Zusammenhang mit verteilten Dateisystemem entstehen.

Generell sind verteilte Dateisysteme zur Implementierung eines verteilten ooDBMS ungeeignet, da sie nicht die Möglichkeit bieten, Objekte kleinerer Granularität als Seiten (Dateien oder Blöcke im Dateisystem) zu sperren. Seiten können im ooDBMS mehrere Objekte beeinhalten, so daß dadurch Objekte nicht mehr einzeln gesperrt werden können, was die Nebenläufigkeit stark beeinträchtigt und damit zu Performance-Verlust führt. Zusätzlich sind verteilte Dateisysteme auf die Übertragung von fixen Blockgrößen oder ganzen Dateien zwischen Knoten begrenzt. Die Anfrageoptimierung setzt sich gerade aber das Ziel, die Datenübertragung zwischen Knoten so gering wie möglich zu halten und dabei werden sicher nicht immer ganze Blöcke oder Dateien zu übertragen sein. Es ist also klar, daß zwischen Anfrageprozessor und verteilten Dateisystem eine Kommunikation stattfinden muß und daß neue Muster für die Implementierung des Zugriffs auf entfernte Dateien benötigt werden.

Die Transaktionsverwaltung für ein Realzeitanforderungen unterliegendem verteilten ooDBMS muß in das darunterliegende Betriebssystem integriert sein. Ansonsten gehen die Meinung über die Integration der Transaktionsverwaltung in Betriebssysteme stark auseinander.

8.5 Zusammenfassung

Verteilte objektorientierte Datenbankmanagementsysteme versprechen transparente Verwaltung verteilter und replizierter Daten, gesteigerte Systemzuverlässigkeit durch verteilte Transaktionen, erhöhte Performance durch Ausnutzung von Parallelität zwischen und innerhalb von Anfragen sowie einfachere und kostengünstigere Systemerweiterungen. Heutige Systeme sind allerdings noch ein gutes Stück entfernt, dieses Versprechen einzulösen. Im Weg stehen dabei nicht nur die kommerzielle Umsetzung von Forschungsergebnissen, sondern auch die Lösung einer Reihe von Problemen. Folgende Gebiete, deren Probleme noch nicht vollständig gelöst sind, wurden in dieser Arbeit besprochen:

1. Verteilte Anfragenbearbeitung, die objektorientierte Anfragen optimieren kann, so daß Nebenläufigkeiten besser ausgenutzt oder Teilergebnisse für andere Anfragen mitbenutzt werden können. Dazu nötig ist ein entsprechendes Kostenmodell für objektorientierte Anfragen.

2. Erweiterte Transaktionsmodelle, die die Bearbeitungslokalität bei verteilten ooDBMS besser unterstützen (Kooperation anstatt Konkurrenz).

3. Analyse von Replikation und ihres Einflusses auf die Architektur verteilter ooDBMS und die Entwicklung von Replikationskontrollprotokolle, die die Systemverfügbarkeit erhöhen.

4. Implementationstrategien für Betriebssysteme und verteilte ooDBMS mit dem Ziel bessere Schnittstellen zwischen den Systemen und bessere Kooperation der Systeme zu erreichen.

Literatur

[1] Lockemann, Krüger, Krumm: *Telekommunikation und Datenhaltung.* Carl Hanser Verlag, München, Wien, 1993.

[2] W. Kim, N. Ballou, J. F. Garza, D. Woelk: *A Distributed Object-Oriented Database System Supporting Shared and Private Databases.* ACM Transaction and Information Systems, Vol. 9, No. 1, pp. 31-51, Jan. 1991.

[3] S. B. Zdonik and D. Maier: *Fundamentals of Object-Oriented Databases,* Morgan Kaufmann Publishers, pp. 18–20, 1990.

[4] Andrew S. Tannenbaum: *Computer-Netzwerke,* Wolfram's Verlag, Attenkirchen, 2. Ed, 1992.

[5] A. S. Tanenbaum: *Experiences with the Amoeba Distributed Operating System,* ,Comm. ACM, Vol. 33, No. 12, pp. 46–63, Dec. 1990.

[6] R. Rashid: *Mach: A Foundation of Open Systems – A Position Paper,* Proc. Second Workshop Workstation Operating Systems, pp. 28–34, 1989.

[7] A. J. Stankovic: *Distributed Computing Systems: An Overview,* Casavant T. L., Singhal M.: Readings in Distributed Computing Systems, IEEE Computer Society Press, 1994.

[8] P. Bernstein, N. Goodman: *Concurrency Control in Distributed Database Systems,* ACM Computing Surveys, Vol. 13, Jun. 1981.

[9] S. J. Mullender, A. S. Tanenbaum: *A Distributed File Service Based on Optimistic Concurrency Control,* ACM 10th Symposium on Software Principles, 1985.

[10] A. S. Tanenbaum, S. J. Mullender, R. van Renesse: *Using Sparse Capabilities in a Distributed Operating System,* IEEE Proceedings of the 6th International Conference on Distributed Computing Systems, pp. 558–563, May 1986.

[11] J. S. Banino, J. C. Fabre: *Distributed couple actors: A CHORUS proposal for reliability,* IEEE 3rd International Conference on Distributed Computing Systems, pp. 128–134, Oct. 1982.

[12] J. R. Nicol, G. S. Blair, R. W. Scheifler: *Operating System Design: Towards a Holistic Approach?,* ACM Operating Systems Rev. 21, Vol. 1, pp. 11–19, Jan. 1987.

9 Common Object Request Broker

MARTIN GERCZUK

9.1 Einführung

Die *Object Management Group* (OMG) hat sich zum Ziel gesetzt Standards für die Nutzung von Objekten in verteilten Systemen zu definieren. Ihr gehören inzwischen namhafte Firmen wie unter anderem Digital Equipment Corporation, Hewlett Packard, NCR und SunSoft.

In einem ersten Schritt definierte die OMG eine Architektur, deren Basis der *Object Request Broker* bildet. Dieser soll im folgenden näher beschrieben werden.

9.2 Die Object Management Architecture

Die *Object Management Architecture* (OMA) definiert die folgenden Elemente und ihre Abhängigkeiten:

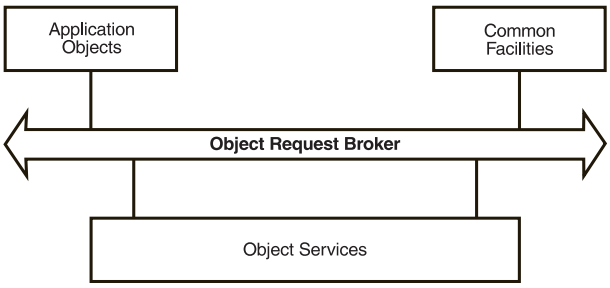


Abbildung 27: Object Management Architecture

Der *Object Request Broker* (ORB) spielt dabei eine zentrale Rolle und bildet sozusagen die Infrastruktur für die Architektur. Die erste konkrete Spezifikation aus der abstrakten OMA betraf folgerichtig den *Object Request Broker*, die in Kürze in der Version 2 vorliegen wird.

Common Facilities und *Object Services* sind Implementierungen von häufig benötigten Objekten. Diese werden von der OMG parallel nach und nach festgelegt.

Standards werden von der OMG in den folgenden Schritten verabschiedet:

- Der Standard wird von einem *Technical Comitee* (TC) als *Request For Proposals* (RFP) vorgestellt. Die Mitglieder der OMG haben nun für einen bestimmten Zeitraum die Möglichkeit, zu dem Vorschlag Stellung zu nehmen oder Korrekturen anzuregen.
- *Task Forces* (TF) bewerten die eingegangenen Vorschläge und arbeiten die endgültige Spezifikation aus, die dann endgültig von einem Board of Directors verabschiedet wird.

9.3 Die Common Object Request Broker Architecture

9.3.1 Begriffe

Ein Objekt (*object*) ist definiert als eine Einheit, die Dienste (*services*) anbietet, die von Dienstnehmern (*clients*) in Anspruch genommen werden können. Der Dienstgeber wird hier Objektimplementierung (*object implementation*) genannt.

Auf ein Objekt wird durch eine Objektreferenz (*object reference*) Bezug genommen. Es ist festgelegt, daß für einen Dienstnehmer eine Objektreferenz systemweit eindeutig ist.

Die Dienstnehmer senden Anfragen (*requests*) an die Objekte, um die Dienste in Anspruch zu nehmen. Eine Anfrage kann Parameter enthalten, um zusätzliche Daten an das Objekt zu übergeben. Diese können vom Dienst als Eingabe, Ausgabe oder auch Ein- und Ausgabeparameter festgelegt sein. Der Dienst kann ein Resultat an den Dienstnehmer zurückliefern. Falls während der Ausführung des Dienstes unvorhergesehene Ereignisse eine korrekte Ausführung unmöglich machen, kann auch eine Ausnahme (*exception*) mit zusätzlichen Informationen, z.B. über die Ursache der Ausnahme, zurückgeliefert werden. In diesem Fall sind die Ausgabeparameter undefiniert.

Die Werte (*values*) der Parameter können die folgenden Typen besitzen:

- Einfache Typen wie Ganzzahlen, Fließkommazahlen, Aufzählungen, Boolsche Werte, Zeichen,

Zeichenfolgen. Der Datentyp any kann für jeden beliebigen Typ stehen. Ein 8-bit Datentyp existiert für die direkte Übermittlung von Binärdaten ohne Interpretation des Inhalts.

- Zusammengesetzte Typen wie Struct, Sequence, Union und Array.
- Objektreferenz

Die von einem Objekt angebotenen Dienste werden in einer Schnittstelle (*interface*) beschrieben. Hier werden die Operationen (*operations*) und die Attribute (*attributes*) eines Objekts mit genauer Angabe der Typen von Parametern, Rückgabewert bzw. Attribut festgelegt.

9.3.2 Übersicht

Der Object Request Broker (ORB) übernimmt nun die Aufgabe die Anfragen vom Dienstnehmer an die Implementierung weiterzureichen.

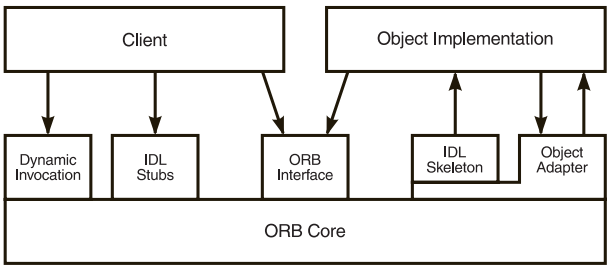


Abbildung 28: Object Request Broker

Er bietet dazu die folgenden Schnittstellen für den Dienstnehmer und die Implementierung an:

- *IDL Stubs* sind Funktionsrümpfe, die es dem Dienstnehmer ermöglichen, die Anfragen wie normale Funktionsaufrufe durchzuführen. Sie werden üblicherweise von Werkzeugen direkt aus der Schnittstellenbeschreibung generiert.
- *Dynamic invocation* bietet die Möglichkeit die Schnittstellenbeschreibung während der Laufzeit zu ermitteln, und die Anfragen entsprechend zu generieren. Dadurch wird es möglich, daß Dienstnehmer auch Dienste von Implementierungen nutzen, die zum Zeitpunkt der Erstellung des Dienstnehmers noch nicht zur Verfügung standen.
- Das *ORB Interface* bietet Funktionen, die sowohl für den Dienstnehmer, als auch die Implementierung nützlich sein können.
- Der *IDL Skeleton* ist ein - üblicherweise durch Werkzeuge aus der Schnittstellenbeschreibung - generierter Rumpf für die Implementierung.
- Die *Object Adapter* stellen für die Implementierung Hilfsfunktionen zur Verfügung.

Die Implementierung eines ORB ist von der Spezifikation völlig offengelassen. Es ist denkbar, daß ein ORB einen eigenen Prozess darstellt, möglicherweise sogar auf einer anderen Maschine wie der Dienstnehmer abläuft, und die Schnittstellen über IPC Mechanismen realisiert werden. Es ist jedoch auch denkbar, daß der ORB nur aus einer hinzubindbaren Bibliothek besteht, und die Implementierung sogar in der selben Instanz wie der Dienstnehmer abläuft. Aus der Sicht des Dienstnehmers und der Implementierung besteht zwischen diesen beiden Alternativen kein Unterschied.

9.3.3 Interface Definition Language

Die *Interface Definition Language* (IDL) dient zur Spezifikation der Objektschnittstellen. Es ist grob eine normale Programmiersprache mit den folgenden wichtigen Merkmalen:

- Es sind nur Deklarationen und keine Definitionen möglich. Die Implementierung eines Objekts bzw. der Aufruf erfolgt also immer in einer anderen Programmiersprache. Genaue Vorschriften für die Abbildung der Sprachelemente von IDL in die verwendete Programmiersprache (*Language Mapping*) sind Bestandteil der CORBA Spezifikation. Derzeit ist die Abbildung auf C komplett spezifiziert. Vorschriften für die Abbildung auf C++, OO COBOL, Smalltalk und Objective C werden derzeit ausgearbeitet. Die Standardisierung der Abbildung durch die OMG bewirkt hohe Portabilität von sowohl Dienstnehmer- als auch Dienstgeber-Implementierungen.

- Da IDL vom System und der verwendeten Programmiersprache unabhängig sein muß, jedoch über eine in IDL definierte Schnittstelle auch ein Datenaustausch zwischen unterschiedlichen Systemen stattfinden kann, sind die Datentypen exakter festgelegt. So ist, im Gegensatz z.B. zur Festlegung in der Programmiersprache C, genau die Bitzahl der Ganz- und Fließkommazahlen spezifiziert. Die Darstellung der Datentypen auf den Systemen kann dabei durchaus unterschiedlich sein. Der ORB übernimmt in diesem Fall die entsprechende Umwandlung z.B. der Byteordnung zwischen Intel und Motorola Prozessoren.

IDL ist bei der Syntax sehr stark an C++ angelehnt.

Datentypen in IDL Es stehen folgende Basis-Datentypen zur Verfügung:

`short`	Ganzzahl im Bereich $\Leftrightarrow 2^{15}..2^{15} \Leftrightarrow 1$
`long`	Ganzzahl im Bereich $\Leftrightarrow 2^{31}..2^{31} \Leftrightarrow 1$
`unsigned short`	Ganzzahl im Bereich $0..2^{32} \Leftrightarrow 1$
`unsigned long`	Ganzzahl im Bereich $0..2^{16} \Leftrightarrow 1$
`float`	32-bit IEEE Fließkommazahl
`double`	64-bit IEEE Fließkommazahl
`char`	8-bit Zeichen nach ISO Latin-1.
`string`	Zeichenkette entweder mit beliebiger Länge, oder durch Angabe von `string<len>` mit festgelegter Maximallänge.
`boolean`	boolscher Wert `TRUE` oder `FALSE`
`octet`	8-bit Binärdaten die bei der Übertragung nicht interpretiert werden. Bei Verwendung von char kann dagegen eine Umwandlung in den jeweiligen system- oder sprachabhängigen Zeichensatz erfolgen.
`any`	Steht für jeden beliebigen Datentyp.
`enum`	Aufzählungstyp.

Für die Bildung zusammengestzter Datentypen stehen die folgenden Möglichkeiten zur Verfügung:

struct

entspricht dem `struct` in C++.

```
struct st {
    short i;
    long l;
};
```

union

ist gegenüber der `union` in C++ erweitert um ein implizites Feld, das angibt welche Variante im speziellen Fall vorliegt.

```
union un switch (short) {
    case 0: short i;
    case 1: string s;
    default: long l;
};
```

sequence

ist ein eindimensionales Array mit wahlweise vorgegebener Maximalgröße. Die tatsächliche Größe kann zur Laufzeit ermittelt werden.

```
// max. 10 short Werte
typedef sequence <short,10> seq1;
// bel. viele Werte:
typedef sequence <short> seq2;
```

array

entspricht den Arrays mit festen Größen in C/C++

```
typedef short arr5[5];
```

Typ- und Konstantendeklarationen Typ- und Konstantendeklarationen entsprechen der Syntax von C++.

Deklaration einer Schnittstelle Eine Schnittstelle wird ähnlich einer C++ Klasse deklariert.

```
interface Sample {

    short Op1(

        in float f,

        out float g,

        inout float h);

    oneway void Op2(in short i);

    attribute float Att1;

    readonly attribute long Att2;

};
```

In obigem Beispiel ist `Op1` eine Operation mit Eingabeparameter `f`, Ausgabeparameter `g` und Ein-/Ausgabeparameter `h`. Der Rückgabewert ist ein `short`. `oneway` gibt an, daß keine Synchronisation des Aufrufers mit der Abarbeitung der Operation erfolgt. In diesem Fall ist also insbesondere nicht gewährleistet, daß die Operation komplett oder fehlerfrei ausgeführt wurde.

Datenelemente werden immer durch das `attribute` Schlüsselwort gekennzeichnet. Durch Angabe von `readonly` kann ein schreibgeschütztes Element erzeugt werden.

Der Name eines Parameters muß im Gegensatz zu C++ immer angegeben werden.

Innerhalb einer Schnittstellendeklaration können auch Typen-, Konstanten und Ausnahmedeklarationen auftreten.

Schnittstellen können mit der gleichen Syntax und Semantik wie in C++ vererbt werden. Dabei ist auch Mehrfachvererbung zulässig.

```
interface Derived : Base1, Base2 {

...

};
```

Es existieren keine mit `public` oder `protected` geschützten Elemente in einer Schnittstellendeklaration.

Ausnahmen Ausnahmen (*exceptions*) sind eigene Datentypen, die üblicherweise Elemente beinhalten, die Aufschluß über die Ursache der Ausnahme enthalten.

```
exception E {

    short Reason;

};
```

Ausnahmen können nur während der Ausführung einer Operation auftreten. Es muß in der Schnittstellenbeschreibung angegeben werden, bei welcher Operation welche Ausnahme auftreten kann.

```
interface If2 {

    short Op(in short i) raises (E);

};
```

Zusätzlich zu den angegebenen Ausnahmen sind Standard Ausnahmen definiert, die bei jeder Operation auftreten können, ohne daß sie explizit angegeben werden müssen bzw. dürfen. Diese umfassen unter anderem Speichermangel, Fehler bei Parameterprüfung, Kommunikationsfehler usw.

Kontext Einer Operation kann wahlweise ein Kontext (*context*) übergeben werden. Dieser besteht aus einer Liste von Zeichenketten. Diesen können vor dem Aufruf Werte zugeordnet werden, die dann von der Implementierung ausgewertet werden können. Ein Kontext in IDL entspricht also in etwa dem environment unter UNIX.

```
interface If3 {

    short Op(in short i)

        context ("Path","User");

};
```

Modul Alle Deklarationen können in ein Modul aufgenommen werden. Dies dient vor allem der Vermeidung von mehrdeutigen Bezeichnern. Die Auflösung von Bezeichnern eines Moduls erfolgt ähnlich wie in C++ mit dem : : Operator. Module können verschachtelt werden.

Verwendung der IDL für die Spezifikation von CORBA Schnittstellen IDL wird in der CORBA Spezifikation auch verwendet, um Schnittstellen zum ORB selbst zu beschreiben. Dies bewirkt, daß durch die Standardisierung der Sprachabbildung die Zugriffe auf einen ORB bei Verwendung der gleichen Programmiersprache portabel zwischen unterschiedlichen Systemen sind.

9.3.4 Sicht des Dienstnehmers auf einen ORB

Ein Dienstnehmer möchte mit Hilfe des ORB Operationen auf Objekten ausführen. Ihm stehen dazu zwei Möglichkeiten zur Verfügung: die Verwendung von *Client Stubs* oder die Verwendung des *Dynamic Invocation Interface*. Beide Arten ergeben das gleiche Ergebnis. Ein Dienstnehmer kann auch abwechselnd die beiden Möglichkeiten für das selbe Objekt nutzen.

Client Stubs Die einfachste Art der Implementierung eines Operationsaufrufs stellt die Benutzung von *Client Stubs* dar. Dies sind aus IDL Deklarationen automatisch generierte Funktionen, die in der für die Programmiersprache üblichen Art aufgerufen werden können. Wie oben bereits angedeutet, ist es dabei völlig unerheblich um welche Programmiersprache es sich handelt, die Umsetzung wird nach den jeweiligen Regeln der *Language Mapping* für diese Programmiersprache vorgenommen.

Anhand eines Beispiels soll verdeutlicht werden, wie die Benutzung von *Client Stubs* in der Programmiersprache C geschieht. Gegeben sei folgende Schnittstellenbeschreibung in IDL:

```
interface If { /* IDL */

    short Op(in short Par);

    attribute short Att;

};
```

Dies generiert die folgenden C Deklarationen, üblicherweise in einer Header-Datei mit gleichem Namen wie die IDL Datei:

```
typedef CORBA_Object If; /* C */

extern CORBA_short If_Op(
```

```
    If o,
    CORBA_Environment *ev,
    CORBA_short Par
);
extern CORBA_short If__get_Att(
    If o,
    CORBA_Environment *ev,
);
extern void If__set_Att(
    If o,
    CORBA_Environment *ev,
    CORBA_short val
);
```

Der Parameter `o` ist die Referenz auf das Objekt, für das die Operation ausgeführt werden soll. `ev` ist ein Zeiger auf eine Datenstruktur, in der unter anderem festgehalten wird, ob bei der Ausführung der Operation eine Ausnahme aufgetreten ist. Der Zugriff auf das Attribut `Att` geschieht über get/set-Funktionen. Bei Deklaration von `Att` als `readonly` würde entsprechend die set-Funktion wegfallen.

Dynamic Invocation Interface Flexibler, aber auch aufwendiger als die Verwendung von *Client Stubs* ist die Verwendung des *Dynamic Invocation Interface*. Hiermit sind für den Aufruf einer Operation grob die folgenden Schritte notwendig:

- Erstellen eines `Request` Objekts mit Hilfe des Namens der Operation (siehe dazu auch ORB Schnittstelle). Der Name ist der selbe wie in der IDL Deklaration der Operation. Die Schnittstelle des `Request` Objekts ist in (Pseudo-) IDL spezifiziert.
- Hinzufügen der Parameter zum `Request` durch Angabe von Paaren von Parametername und Wert. Dies kann wahlweise auch bereits bei der Erstellung des `Request` Objekts geschehen.
- Aufruf der `invoke` Operation auf das `Request` Objekt. Wahlweise kann auch eine asynchrone Ausführung durch Verwendung des Operationspaars `send` und `get_response` gewählt werden.
- Auswertung des Resultats der Operation
- Löschen des `Request` Objekts und Freigeben des damit verbundenen Speicherplatzes durch Aufruf der `delete` Operation.

Das *Dynamic Invocation Interface* wird auf die gleiche Art und Weise angesprochen, wie beliebige andere Objekte, die in einem ORB verfügbar sind. Verständlicherweise ist es nicht möglich, das *Dynamic Invocation Interface* über sich selbst aufzurufen. Es wird ein Mechanismus verwendet, der den *Client Stubs* entspricht. Wie oder auch wo das *Dynamic Invocation Interface* implementiert ist nicht spezifiziert. Es kann also auch durchaus eine Bibliothek sein, die zum Dienstnehmer dazugebunden wird.

Der Vorteil gegenüber der Verwendung von *Client Stubs* ist die Möglichkeit, auch Schnittstellen nutzen zu können, die zum Zeitpunkt der Übersetzung des Dienstnehmerprogramms noch nicht existierten. Die Implementierung der *Client Stubs* kann, muß jedoch nicht durch Nutzung des *Dynamic Invocation Interface* erfolgen.

Interface Repository Um Informationen über unbekannte Schnittstellen zu erhalten, kann der Dienstnehmer die *Interface Repository* (übersetzt: Ablage für Schnittstellen) verwenden. In der *Interface Repository* sind alle Informationen über die in einem ORB verfügbaren Schnittstellen persistent abgelegt. Es können die Schnittstellen, deren Operationen und Attribute, die Parameter und Rückgabewerte der Operationen und deren Typen, die definierten

Ausnahmen und alle definierten Typen und Konstanten zur Laufzeit abgefragt werden. Dies kann z.B. in CASE Systemen dazu verwendet werden, um Listen der verfügbaren Schnittstellen anzuzeigen, oder im ORB selbst, um zur Laufzeit Parameterüberprüfungen vorzunehmen. Aus der *Interface Repositiory* kann der komplette IDL Quelltext (natürlich ohne Kommentare, aber mit allen Parameternamen!) zurückgewonnen werden.

Die Operationen und Datentypen zum Zugriff auf die *Interface Repository* sind in der CORBA Spezifikation in IDL angegeben. Die Informationen über die Schnittstellen sind also aus Dienstnehmersicht selbst in Objekten abgelegt, auf die mit den üblichen Mitteln zugegriffen werden kann, allerdings ist nicht vorgeschrieben, daß die *Interface Repository* Schnittstellen unbedingt als Objekte implementiert werden müssen. Ebenso ist im CORBA Standard nur festgehalten, wie Informationen über Schnittstellen ermittelt werden können, nicht jedoch wie zusätzliche Schnittstellenbeschreibungen in die *Interface Repository* eingefügt werden. Es ist auch möglich, daß ein ORB mehrere *Interface Repositories* anbietet, wenn z.B. eine Reihe von Objekten vom Verhalten sehr stark von anderen Objekten abweicht (z.B. OODB) oder ein Teil der Objekte über einen zweiten ORB angesprochen werden muß.

Um einen möglichst universellen Zugriff auf die einzelnen Schnittstellenbeschreibungen zu ermöglichen, werden für die Objekte der *Interface Repository* die folgenden Basisobjekte verwendet:

```
module CORBA {

typedef string Identifier;

typedef string RepositoryId;

typedef Identifier InterfaceName

interface Container;

interface Contained;

typedef sequence<RepositoryId>

    RepositoryIdSeq;

typedef sequence<Identifier>

    IdentifierSeq;

typedef sequence<string> StringSeq;

typedef sequence<Container>

    ContainerSeq;

typedef sequence<Contained>

    ContainedSeq;


interface Contained {

    struct Description {

        Identifier name;

        any value;

    };

    attribute Identifier name;

    attribute RepositoryId id;

    attribute RepositoryId defined_in;

    ContainerSeq within();

    Description describe();

};


interface Container {

    struct Description {
```

```
        Contained contained_object;

        Identifier name;

        any value;

    };

    typedef sequence<Description>

        DescriptionSeq;


    ContainedSeq contents(

        in InterfaceName limit_type,

        in boolean exclude_inherited);

    ContainedSeq lookup_name(

        in Identifier search_name,

        in long levels_to_search,

        in InterfaceName limit_type,

        in boolean exclude_inherited);

    DescriptionSeq describe_contents(

        in InterfaceName limit_type,

        in boolean exclude_inherited,

        in long max_returned_objs);

};

} /* end module CORBA */
```

Durch diese beiden Objekte sind folgende Möglichkeiten gegeben:

- Bestimmen des `Container`, der ein `Contained` Objekt beinhaltet. Durch diesen Zugriff kann man z.B. ausgehend von der Beschreibung einer Operation einer Schnittstelle zurück auf die Schnittstelle selbst, und weiter auf das die Schnittstelle enthaltende Modul navigieren.
- Durchsuchen eines `Container` nach allen Einträgen durch `contents` oder gezielt nach einem bestimmten Eintrag durch `lookup_name`, womit sogar rekursiv gesucht werden kann. Bei beiden Operationen kann wahlweise nur nach bestimmten Objekttypen (z.B. alle Konstantendeklarationen) gesucht werden.

Mit diesen beiden Basisklassen wird nun die syntaktische Hierarchie von IDL nachgebildet. Die Wurzel dieser Hierarchie bildet das `Repository` Objekt, das die weiteren Objekte als `Container` enthält. Die weiteren Objekte (`ModuleDef`, `ConstantDef`, `InterfaceDef`, usw.) sind jeweils von `Contained` abgeleitet, falls sie wiederum weitere Elemente enthalten (z.B. `ModuleDef`), zusätzlich auch von `Container`.

Für den Zugriff auf die *Interface Repository* existieren 3 Möglichkeiten:

- Die Schnittstelle eines Objekts (`InterfaceDef`) kann direkt aus der Objektreferenz ermittelt werden (siehe auch ORB Basisfunktionen).
- Ausgehend vom `Repository` Objekt
- Anhand einer bereits bekannten Id (`RepositoryId`) kann direkt über die `lookup_id` Operation der `Repository` auf das entsprechende Objekt zugegriffen werden.

9.3.5 Sicht der Implementierung auf einen ORB

Aufgabe der Implementierung ist es, die in der IDL Spezifikation vorgegebenen Operationen zu implementieren. Die Implementierung kommuniziert mit dem ORB hauptsächlich über einen *Object Adapter*, der allgemeine Funktionen für Objektimplementierung bereitstellt, wie z.B. den Start und das Beenden der benötigten Instanzen, die den Code der Implementierung enthalten oder auch Methoden für die Objektpersistenz.

Für verschiedene Ansprüche können unterschiedliche *Object Adapter* verwendet werden. Eine Art von *Object Adapter* ist in der CORBA Spezifikation als Standard vorgeschlagen, der *Basic Object Adapter* (BOA). Dieser sollte, muß aber nicht für jeden ORB verfügbar sein.

Weitere Object Adapter können weitergehende Funktionalität aufweisen, wie z.B. die Verwendung einer Objektorientierten Datenbank für die Unterstützung der Objektpersistenz von umfangreichen Objekten.

Basic Object Adapter Die Schnittstelle des BOA ist in (Pseudo-) IDL spezifiziert, so daß Implementierungen, die diesen Adapter verwenden einigermaßen portabel sein sollten. Er stellt die folgenden Operationen zur Verfügung:

- Start und Beenden der Implementierung oder auch individueller Objekte

  Die Funktionen zur Aktivierung der Implementierung oder der Objekte sind sehr stark vom verwendeten System abhängig. Diese Informationen sind in der *Implementation Repository* abgelegt. Es ist dabei in der CORBA Spezifikation nur vorgegeben, daß für jede Schnittstelle ein `ImplementationDef` Objekt existieren muß, dessen Aufbau nicht näher festgelegt ist.

  Es ist möglich mehrere Objektimplementierungen in einer Instanz (Prozess) unterzubringen, es kann jedoch auch der Fall vorliegen, daß für jede Methode eine eigene Instanz gestartet wird. Die einzelnen vom BOA unterstützten Vorgehensweisen sind:

  shared server

  In diesem Fall werden mehrere Objekte in einer Instanz implementiert. Wenn das erste dieser Objekte benötigt wird, startet der BOA diese Instanz. Diese Instanz bleibt aktiv, bis sie sich beim Beenden beim BOA abmeldet.

  unshared server

  Hier wird für jedes Objekt eine neue Instanz der Implementierung gestartet. Der BOA verwaltet, welche Instanz für welches Objekt zuständig ist. Daher muß beim Abmelden einer Instanz dem BOA auch mitgeteilt werden, welches Objekt davon betroffen ist.

  server-per-method

  Bei dieser Art startet der BOA für jeden Aufruf einer Operation eine neue Instanz. Es kann sogar vorkommen, daß für das selbe Objekt die selbe Operation auf verschiedenen Instanzen ausgeführt wird. Die Instanzen müssen sich beim Beenden nicht beim BOA abmelden, da dieser keine Informationen über bereits aktivierte Instanzen verwalten muß.

  persistent server

  Diese Form der Implementierung muß außerhalb des BOA aktiviert werden (z.B. bei Systemstart) und sich beim BOA anmelden. Falls die Instanz nicht aktiviert ist und eine Operation für diese Implementierung eintrifft, wird ein Fehler zurückgeliefert.

- Erstellung und Interpretation von Objektreferenzen.

  Der BOA bietet die Möglichkeit, in der Implementierung intern verwendete Objektreferenzen auf die vom ORB verwendeten Objektreferenzen abzubilden. Für die implementierungsinternen Referenzen stehen 1024 Byte als Speicherplatz zur Verfügung. In einfachen Fällen kann dieser Bereich sogar genutzt werden, um die gesamten Objektdaten aufzunehmen.

  Durch die `create` Operation kann nun die Implementierung aus ihrer internen Referenz und durch die Angabe von Schnittstellen- und Implementierungsdefinition eine ORB Objektreferenz erzeugen. `get_id` liefert umgekehrt zu einer ORB Objektreferenz die internen Daten zurück. `dispose` dient zum Löschen einer ORB Objektreferenz.

- Authentisierung des Aufrufers

  Es ist möglich, für den Aufruf einzelner Operationen Zugriffsrechte zu vergeben. Wie dies im einzelnen geschehen kann ist stark vom verwendeten System abhängig, und wurde daher in der CORBA Spezifikation nicht exakt festgelegt. Es ist nur definiert, daß eine Möglichkeit existiert den Benutzer einer Operation zu ermitteln. Die Verwaltung und Überprüfung der Zugriffsrechte der Benutzer zu den einzelnen Objekten oder

Operationen muß innerhalb der Implementierung erfolgen.

- Ausführung der Operationen durch Aufruf der Interface Skeletons (übersetzt: Schnittstellengerippe)

  Das *Interface Skeleton* wird aus einer IDL Spezifikation erstellt. Die Funktionsrümpfe werden durch die selbe Abbildung (*Language Mapping*) erstellt, wie die *Client Stubs*. Der normale Ablauf einer solchen Funktion besteht (in C) in der Ausführung der Operation und einem *return*. Falls während der Abarbeitung der Operation eine Ausnahme auftritt, kann diese durch Verwendung der *set_exception* Operation des BOA an den Aufrufer weitergegeben werden.

  Es ist jedoch nicht festgelegt, wie die Anbindung der Implementierung an das *Interface Skeleton* erfolgt. Es besteht die Möglichkeit, daß der Linker die Anbindung vornimmt, es ist jedoch auch möglich, daß die Implementierung die Funktionsadressen erst beim Start dem System bekannt machen.

9.3.6 ORB Schnittstelle

In der ORB Schnittstelle sind die Funktionen zusammengefasst, die unabhängig vom verwendeten *Object Adapter* sind. Diese Funktionen umfassen:

- Abbildung von Objektreferenzen auf Zeichenketten

  Um Dienstnehmern das Speichern oder die Übertragung von Objektreferenzen zu ermöglichen, kann eine Objektreferenz in eine Zeichenkette, und umgekehrt eine Zeichenkette wiederum in eine Objektreferenz umgewandelt werden.

- Hilfsfunktionen für die Erstellung von Listen für das *Dynamic Invocation Interface*

- Allgemeine Basisoperationen, die auf allen Objekten gültig sind

  Bei jedem Objekt ist es möglich durch `get_implementation`
  die zugehörige `ImplementationDef`, und durch `get_interface` die zugehörige `InterfaceDef` zu ermitteln.

  Mit der Operation `duplicate` ist es möglich, eine Objektreferenz zu kopieren. Es wird dann eine neue Objektreferenz erstellt, die das selbe Objekt bestimmt. Mit `release` kann eine Referenz wieder freigegeben werden.

  Mit der Operation `is_nil` kann getestet werden, ob eine Objektreferenz auf ein Objekt verweist.

  Durch `create_request` kann ein `Request` Objekt erstellt werden, durch das Operationen über das *Dynamic Invocation Interface* ausgeführt werden können.

9.4 Object Services und Common Facilities

Basierend auf dem durch CORBA festgelegten Objektmodel definiert die OMG nach und nach Schnittstellen, die von Anwendungsprogrammen verwendet werden können. Nach OMG Definition teilen sich diese Standards folgendermaßen auf:

Object Services definieren nützliche Schnittstellen, die als Basis für komplexere Schnittstellendefinitionen verwendet werden können.

Common Facilities sind eine Ebene höher angesiedelt und definieren komplexere, anwendungsbezogene Standards, die wiederum teilweise auf *Object Services* basieren.

9.4.1 Object Services

Object Services bieten einen Grundstock an Schnittstellen, die in vielen Bereichen nützlich sein können. Beim Entwurf der Schnittstellen wurde bzw. wird besonderen Wert darauf gelegt, daß die verschiedenen Klassen möglichst unabhängig voneinander spezifiziert wurden, um sie in Art von Baukastensystemen zusammensetzen zu können. Daher wurde keine komplette Klassenhierarchie errichtet, sondern versucht eine möglichst unabhängige Ansammlung von Klassen zu präsentieren.

Die folgenden Schnittstellen sind bereits vollständig spezifiziert:

- *Object Event Notification Service* bildet die Erzeuger/Verbraucher Semantik ab. Es existieren Schittstellen für Erzeuger, Verbraucher und für den Kommunikationskanal.

- *Object Lifecycle Service* dient der Erzeugung, dem Löschen, Kopieren und Verschieben von Objekten.

- *Object Name Service* bietet die Zuordnung von Namen zu Objekten.

- *Persistent Object Service* ermöglicht eine persistente Speicherung des Objektzustands, die über die im ORB vorhandene Persistenz der Objektreferenz selbst hinausgeht.

- *Object Concurrency Control Service* regelt den gemeinsamen Zugriff auf ein Objekt.

- *Object Externalization Service* definiert den Austausch von Objektzuständen über Dateien oder Kommunikationskanäle. Teil der Spezifikation ist auch ein Austauschformat, das alle Implementierungen unterstützen.

- *Object Relationships Service* erlaubt die Definition von Gegenstand/Beziehungs- Graphen auf CORBA Objekten.

- *Object Transaction Service* stellt grundlegende Funktionalität für eine Transaktionsverwaltung zur Verfügung.

Zum Zeitpunkt der Erstellung dieser Ausarbeitung (Anfang 1995) waren folgende Schnittstellen noch im Vorschlagsstadium (RFP):

- *Object Security Services* decken den Zugriffsschutz und die Berechnung von Kosten für erbrachte Dienste ab.

- *Object Time Service* bietet einen Mechanismus für den Abgleich der Uhrzeit über mehrere Maschinen.

- *Object Licensing Service* kann für die Überwachung der Einhaltung von Lizenzbedingungen einer Software verwendet werden.

- *Object Properties Service* dient der dynamischen Erstellung von Objektattributen als Erweiterung zu den statisch in IDL definierten Attributen.

- *Object Query Service* ist eine mit SQL vergleichbare Schnittstelle zur Manipulation von Relationen.

Weitere *Object Services* wurden bereits für die nächsten RFPs vorgeschlagen.

9.4.2 Common Facilities

Die *Common Facilities* werden von OMG zunächst in zwei Gruppen eingeordnet:

Horizontal Common Facilities sind unabhängig von einer gewissen Art von Anwendung (z.B. Benutzeroberfläche), während *Vertical Common Facilities* bestimmten Anwendungsgebieten (z.B. CAD Systeme, Finanzprogramme, etc.) zuzuordnen sind. Letztere sind die größere Gruppe, aber auch die speziellere Art von *Common Facilities*. Es ist jedoch nicht ausgeschlossen, daß Mitglieder dieser Gruppe zu einem späteren Zeitpunkt der ersteren Gruppe zugeordnet werden, wenn sich herausstellt, daß auch andere Anwendungen aus ihnen Nutzen ziehen können.

Die derzeit in Arbeit befindlichen *Horizontal Common Facilities* fallen grob in 4 Gruppen:

- *User Interface Common Facilities* definieren Schnittstellen für Objekte, die etwas mit der Benutzeroberfläche zu tun haben. Dazu gehören u.a. die Anzeige von Objekten auf Bildschirm oder Drucker, die Verwaltung von zusammengesetzten Dokumenten, die Unterstützung von Hilfesystemen und Methoden für die Automatisierung von Vorgängen (Makroprogrammierung).

- *Information Management Common Facilities* bieten Funktionen zur Verwaltung von allgemeinen Unternehmensdaten, wie Datenbanken, aber auch Text- oder grafische Dokumente. Dieser Standard beinhaltet auch die benötigten Dateiformate für den Austausch von Dokumenten und anderen Informationen.

- *Systems Management Common Facilities* stellen standardisierte Hilfsmittel für die Verwaltung von Computersystemen zur Verfügung. Dies umfaßt unter anderm auch die Betreuung von Netzwerken.
- *Task Management Common Facilities* werden für den Datenaustausch von Unternehmensdaten verwendet. Sie umfassen einfachen Nachrichtenaustausch (z.B. E-Mail) bis hin zur Definition von komplexen Abläufen des Informationsaustauschs.

9.5 Zusammenfassung

Durch die große Akzeptanz von CORBA innerhalb der Industrie dürfte einer weiten Verbreitung des Standards nichts im Wege stehen. Die einzige wichtige Firma, die bis jetzt noch nicht der OMG angeschlossen hat, ist Microsoft, da sie mit OLE einen von der Funktionalität ähnlichen eigenen Standard durchsetzen wollen.

Ein großer Vorteil von CORBA ist die einfache Implementierung von Dienstnehmern durch Verwendung der *Client Stubs*. Durch diesen Mechanismus ist es kaum aufwendiger eine Operation auf einem Objekt auszuführen, das möglicherweise an einem unbekannten Ort in einem Netzwerk implementiert ist, als eine normale Prozedur aufzurufen.

Durch die Festlegung der Schnittstellen in IDL und die strenge Definition der *Language Mappings* können Dienste in jeder Programmiersprache (idealerweise) unabhängig vom darunterliegenden System verwendet werden. Dies und die zusätzlichen Anstrengungen der Standardisierung der *Object Services* und der *Common Facilities* dürfte einen großen Schritt auf das Ziel der wiederverwendbaren Software darstellen, das man sich ursprünglich ja von der Objektorientierung versprochen hat.

Schwachstellen sehe ich vor allem in den wenigen nicht exakt festgelegten Punkten in der Spezifikation von CORBA. So ist es anscheinend nicht üblich, daß zwei ORBs unterschiedlicher Hersteller miteinander kommunizieren können, ohne daß einer der beiden den anderen "kennt". So steht zu befürchten, daß sich doch wieder Herstellerstandards entwickeln, die die Portierung von Software von einem System auf ein anderes unnötig erschweren. Dies dürfte vor allem Objektimplementierungen betreffen, die mit der Funktionalität des *Basic Object Adapter* nicht auskommen. Es bleibt in diesem Punkt abzuwarten, welche Erweiterungen mit dem gerade entstehenden CORBA 2.0 Standard vorgenommen werden.

Literatur

[1] *The Common Object Request Broker: Architecture and Specification*, OMG Document Number 93.xx.yy, 1993

[2] *Object Services Architecture*, OMG, 1994

[3] *Common Object Services Specification, Volume I*, OMG Document Number 94-1-1, 1994

[4] *Common Facilities Architecture*, OMG Document Number 94.11.9, 1994

[5] Thorsten Beyer, *Objektbörse; CORBA - OMG Standard für verteilte Objekte*, iX 2/1993, S. 24-33

[6] Tom Mowbray, Ron Zahavi, *Distributed Computing with Object Management*, Connexions - The Interoperability Report, Vol. 7, No. 12, December 1993, S. 18-24