Rainer Lutz:
Transaktionsverwaltung auf der Basis von Transaktionsbeziehungsnetzen
(Diplomarbeit)

Universität Stuttgart, Institut für Parallele und Verteilte Höchstleistungsrechner IPVR,
Breitwiesenstraße 20–22, 7000 Stuttgart 80

Das Dokument ist über den ftp–Server:

Rechner:	ifi.informatik.uni-stuttgart.de
Internet:	129.69.211.1
Benutzer:	anonymous
Passwort:	beliebig (nicht leer)
Pfad:	/pub/APRICOTS
Format:	PostScript

erhältlich (in Vorbereitung).

Fragen können per e-mail an
Rainer.Lutz@informatik.uni-stuttgart.de
gerichtet werden.

Die vorliegende Arbeit wurde mit LaTeX, BibTeX und makeindex erzeugt.

Vorwort

Wie üblich ist der Autor zu Beginn gehalten, den Leser darüber zu informieren, was den Inhalt seines Werkes ausmacht und wie dieser am besten aus dem Dokument heraus zu erarbeiten ist. Ich möchte mit dem letzteren beginnen, weil es mir ein Anliegen ist, kurz die Umstände zu beleuchten, die zu diesen Seiten geführt haben.

Die Zielrichtung einer Diplomarbeit ist zweigeteilt. So soll sie zum einen die Befähigung des Diplomanden am Ende seines Studiums belegen, eine eigenständige Arbeit erstellen zu können. Er hat sich, aufgrund thematischer Vorgaben, spezifisches Wissen anzueignen, um dieses mithilfe seines Erfahrungshorizontes in globalen Kontext zu setzen, Bezüge zu schaffen oder gar Neuerungen daraus abzuleiten. Auf der anderen Seite dienen Diplomarbeiten konkret auch dazu, Forschungsarbeit zu leisten — sprich den wissenschaftlichen Mitarbeitern in ihrer Arbeit behilflich zu sein. Oftmals sind zwar die Ziele dieser Projekte klar definiert, jedoch in den meisten Fällen dürfte nicht offenbar sein, wie diese zu erreichen sind oder ob sie überhaupt realisierbar sind.

Vor diesem Hintergrund wird deutlich, daß es durchaus von Bedeutung ist, nicht nur Lösungen für gestellte Aufgaben präsentieren zu können, sondern auch Probleme existierender Ansätze, Alternativen zu oder Bewertungen von vorhandenen Theorien benennen, oder im einfachsten Fall erst einmal Aufgaben definieren zu können.

Die Aufgabenstellung der nun vorliegenden Arbeit war, die Idee der Transaktionsverwaltung auf Beziehungen in ein Gesamtkonzept zur Modellierung und Verarbeitung langlebiger Aktivitäten einzubringen. Wie bei jeder Integration sind die Ansatzpunkte für Untersuchungen, betreffen sie Potentiale oder Reibungsverluste, zahlreich und vielschichtig. So lautete die Frage genau genommen nicht *Wie* diese Transaktionsverwaltung auszusehen hat, sondern *ob* sie die Anforderungen des Systems erfüllen kann, ob ihr Einsatz überhaupt möglich ist und erst in zweiter Linie, wie dies dann aussehen könnte. Sie werden also die Verarbeitung von Transaktionen auf der Basis von Beziehungen nur in Ansätzen beschrieben finden; genau so weit, wie sich eindeutige Aussagen zum derzeitigen Stand finden lassen. Auch werden einige Anmerkungen in Bezug auf die Realisierung oder die Verteilte Verarbeitung enthalten sein. Jedoch erfüllt ein erheblicher Teil der Arbeit den Zweck, Fragen zu stellen und Ansatzpunkte für neue Überlegungen zu liefern.

Gemäß der Devise, daß jede Aussage, die angezweifelt wird, zumindest ihre Diskussionswürdigkeit bewiesen haben dürfte, entlasse ich Sie nun — ich hoffe, die Thematik ist für Sie ebenso neu, wie sie es für mich war — in die Diskussion zum zuvor benannten Thema.

Es sei lediglich noch angefügt, daß die Arbeit an diesem Komplex nicht abgeschlossen ist, nicht abgeschlossen sein kann, was Ihre Kritik, Ihr Lob und Ihre Ergänzungen nur noch wertvoller macht. Vielen Dank.

Inhaltsverzeichnis

Abbildungsverzeichnis

Legende

Abkürzungen

ACID .... Atomarity, Consistency, Isolation, Durability
AP .... Applikation
APRICOTS ...... A Prototype Implementation of the ConTract-System
ConTract .... Controlled activities, controlled transaction compound
CS .... Communication-System
CM .... ConTract-Manager
DBMS .... Database Management System
GTRID .... Global Transaction Identifier
RM .... Resource Manager
RMCall .... Aufruf eines Resource Managers
SS .... Step Computation Server
TBN .... Transaktionsbeziehungsnetz
TBNM .... Transaktionsbeziehungsnetz-Manager
TM .... Transaction Manager
TAV .... Transaktionsverwaltung

Schriftarten im Text

Sans Serif .... Zustände von Transaktionen und Akronyme
Italic .... Neuer Begriff
Fett .... Hervorhebungsmerkmal

Motivation

In dieser Arbeit soll, ausgehend von den Überlegungen zum ConTract–Modell, untersucht werden, inwieweit eine Transaktionsverwaltung auf Beziehungen zwischen Transaktionen möglich ist und wie diese aussehen könnte. Als Basis dazu dienen zum einen die Arbeiten im APRICOTS–Projekt sowie [CRta], aber auch [RG91], in denen ein Realisierungsansatz des ConTract–Modells bzw. die grundlegenden Überlegungen zu Beziehungen zwischen Transaktionen geschildert werden.

Im Laufe der Arbeit ist ein Übergangsprotokoll (Commit-Protokoll) zu entwickeln und Aspekte der verteilten Verarbeitung zu untersuchen. Darüberhinaus ist der bisherige Entwurf der Verwaltung von Transaktionsbeziehungsnetzen [Sch91], der ConTract–Manager [Wah91] und das Kommunikationssystem [Bur91] auf Stimmigkeit mit der Transaktionsverwaltung hin zu überprüfen. Besonders im Falle der Verwaltung der Netze sind deren Implementierungsansätze kritisch bezüglich Realisierbarkeit und Angemessenheit zu beleuchten, um in weiteren Phasen der Entwicklung sowohl Entwurfsfehler beheben wie auch neue Aspekte eingliedern zu können.

Kapitel 1 stellt eine kurze Einführung in die Thematik der (verteilten) Transaktionsverarbeitung dar. Kapitel 2 schafft dann die theoretische Grundlage, indem Beziehungen zwischen Transaktionen definiert und erläutert werden. Diese Überlegungen werden wiederum in Kapitel 4 auf sogenannte Transaktionsbeziehungsnetze erweitert. In Kapitel 5 wird ein Protokoll zur Verarbeitung von Transaktionen auf der Basis von Beziehungen eingeführt. Die speziellen Anforderungen im Rahmen der ConTract–Verarbeitung ist Gegenstand des 6. Kapitels, Kapitel 7 hingegen versucht den Rahmen zu erweitern und zeigt generelle Schwierigkeiten und Möglichkeiten des Modells auf. Kapitel 8 schlußendlich ist der Verteilung gewidmet.

Dabei sei besonders auf das Kapitel 3 verwiesen, das besonders auf die Thematik der Transaktionsverarbeitung und die Unterschiede des vorliegenden Modells zur klassischen Vorgehensweise aufzeigt.

Am Ende der Arbeit befindet sich eine kurze Zusammenfassung, die auch auf die Ansätze zur Weiterarbeit eingeht.

Kapitel 1

Grundlagen

1.1 Einführung in die verteilte Transaktionsverarbeitung

An dieser Stelle soll eine kurze Einführung in die Architektur der verteilten Transaktionsverarbeitung gegeben werden. *Architektur* umfaßt hierbei die Komponenten und ihre Schnittstellen. Das verteilte Transaktions-Management integriert all diese Komponenten zu einem verläßlich arbeitenden, verteilten System.

1.1.1 Transaktionsverarbeitung

Transaktionsverarbeitung beschreibt die Ausführung von Zugriffen auf gemeinsamen Daten, oftmals Datenbanken. Eine *Transaktion* ist die Ausführung einer Menge von Operationen (auf gemeinsamen) Ressourcen, mit den folgenden Eigenschaften:

Atomicity Die Transaktion wird nur ganz oder gar nicht ausgeführt. Dementsprechend sind auch bei erfolgreicher Beendigung alle und im Fehlerfall keine der Effekte wirksam. Auch werden keine Zwischenergebnisse (vor Beendigung) nach außen hin sichtbar.

Consistency Eine Transaktion führt von einem konsistenten in einen (nicht unbedingt neuen) konsistenten Zustand, einen Einbenutzerbetrieb und störungsfreien Ablauf vorausgesetzt.

Isolation Dabei wird garantiert, daß eine Transaktion isoliert von allen anderen Transaktionen abläuft. Unerwünschte Wechselwirkungen werden vermieden.

Durability Nach erfolgreichem Beenden der Transaktion ist garantiert, daß die Effekte und Änderungen unter keinen Umständen wieder verloren gehen können.

Diese Eigenschaften werden als das ACID–Paradigma bezeichnet.

Eine Transaktion wird durch die 'Klammer' *begin-of-transaction* und *end-of-transaction* (beziehungsweise durch *rollback* im Fehlerfall) gebildet.

Transaktionen können einander nicht überlappen oder einschachteln. Sie stehen in keiner Beziehung oder Informationsaustausch zueinander. Wir nennen sie *flache Transaktionen*. Transaktionen mit Beziehungen oder Schachtelung nennen wir *komplexe Transaktionen*,

1.1.2 Architektur einer verteilten Transaktionsverarbeitung

Systeme verteilter Transaktionsverarbeitung sind gegliedert in zahlreiche Komponenten mit unterschiedlichen Aufgaben.

So kennen wir zunächst das Anwendungsprogramm oder *Applikation*. Dieses führt Interaktionen mit dem Benutzer durch, nimmt Dienste der Transaktionsverwaltung wahr und wandelt Informationen in beiden Richtungen. Wir unterscheiden *Clients* von *Servern*. Ein Client generiert Aufträge, setzt diese ab, wartet auf Ergebnis und präsentiert dies dem Benutzer. Der Server demarkiert eine Transaktion (durch BOT und EOT), ruft verschiedene *Resource Manager* auf, um auf Daten zuzugreifen, und beantwortet den Auftrag des Client entsprechend. Ein Server kann auch weitere Aufträge absetzen.

Ein *Resource Manager* implementiert die Transaktionssemantik. Er koordiniert den gemeinsamen Zugriff verschiedener Transaktionen auf 'seine' Ressourcen und garantiert dabei die oben eingeführten ACID–Eigenschaften. Ein Resource Manager meldet seine Teilnahme an einer Transaktion bei der Transaktionsverwaltung an. Mit dem Begriff *Resource Manager* implizieren wir im folgenden *Recoverable Resource Manger*. Das heißt, daß er nicht nur die Effekte abgebrochener Transaktionen rückgängig machen kann, sondern auch nach Systemfehler einen Zustand erreicht, in dem alle vormals aktiven Transaktionen abgebrochen und alle bis dato erfolgreich beendeten Transaktionen dauerhaft sind. Darüber hinaus gehen wir davon aus, daß die Resource Manger das zum verteilten Commit notwendige *2-Phase-Commit-Protokoll* unterstützen. Dieses setzt voraus, daß ein Resource Manager auf Anfrage der Transaktionverwaltung einen Zustand (*Prepared*) erreicht, von dem aus das Beenden oder Abbrechen der Transaktion garantiert werden kann. Dieser Zustand ist persistent zu halten.

Die *Transaktionsverwaltung* realisiert die Transaktionsabstraktion. Sie ist verantwortlich für die Atomarität einer Transaktion, indem sie für die beteiligten Resource Manager entscheidet, ob eine Transaktion erfolgreich beendet werden kann oder abgebrochen werden muß. Sie stellt Operationen für die Demarkation von Transaktionen und die Teilnahme eines Resource Managers an einer Transaktion zur Verfügung und wickelt das Commit-Protokoll ab.

Um die Kommunikation der Komponenten transparant zu halten, enthalten solch beschriebene Systeme meist einen *Communication Manager*. Er adressiert physisch die benannten Komponenten und sorgt für die sichere Kommunikation. Die einzelnen Komponenten müssen also weder wissen, auf genau welchem Rechner sich der Kommunikationspartner befindet, noch dafür sorgen, daß Nachrichten auch dort ankommen. Desweiteren sorgt der Communication Manager für die Konvertierung von Daten in verschiedene Darstellungen und propagiert dedizierte Informationen an die verschiedenen Komponenten — seien es Systeminformationen oder Informationen über Transaktionen. Oftmals nehmen Communication Manager den Dienst der Transaktionsverwaltung zur Sicherung der Nachrichten in Anspruch.

1.2 Übersicht

Solch eine Architektur ist im wesentlichen Bestandteil auch unseres Szenarios. Flache Transaktionen haben jedoch die hinlänglich bekannten und in [Gra81] beschriebenen Nachteile und Einschränkungen. Deshalb wurde im Rahmen des APRICOTS–Projekts Augenmerk auf *Definition und Kontrolle langlebiger und verteilter Verarbeitung*, komplexe Verarbeitung und Transaktionsmodelle gelegt. Hieraus entstanden *ConTracts*. Dies stellt kein neues, erweitertes Transaktionskonzept dar, sondern ist vielmehr ein Programmiermodell, das die Eigenschaften der Persistenz, der Konsistenz, Recovery, Synchronisation und der Kooperation ganzer Abläufe (statt einzelner Transaktionen) umfaßt. Hier werden Methoden beschrieben, die Applikationen erlauben, deskriptiv den Kontrollfluß und die Beziehungen zwischen Transaktionen zu beschreiben[1].

In Folge werden gänzlich neue Anforderungen an die Transaktionsverwaltung gestellt. Es muß zwischen Applikation und Transaktionsverwaltung eine Absprache bezüglich der Transaktionen stattfinden. Das „klassische Modell“ muß seine Entsprechung in einer neuen Darstellung finden. Die Beziehungen, nicht schlicht nur die einzelnen teilnehmenden Transaktionen, sind Einflußgrößen bei der Abarbeitung und Koordination von Transaktionen, robust gegenüber Störgrößen (Transaktionsfehler, Systemfehler).

Große Aktivitäten können aus kleineren modelliert werden. Komplexe Transaktionsmodelle müssen verwaltet und kombiniert werden. Die Abhängigkeit im Netz verteilter Transaktionen müssen berücksichtigt werden, Delegation von Koordination vorgesehen werden. Neue Arten atomarer Einheiten (*Steps*) sind in das System zu integrieren, was seinen Niederschlag in Komplexität und Koordination der beteiligten Komponenten findet.

Die vorliegende Arbeit versucht nun, einen Ansatz zur Transaktionsverarbeitung in diesem neuen Szenario zu liefern. Sie stützt sich dabei wesentlich auf bereits existierende Arbeiten. So wurden in frühzeitig die neue Transaktionskonzepte und der Hintergrund der ConTracts beschrieben [WR90][WR91]. Im Rahmen weiterer Untersuchungen wurde u.a. dann der Begriff der Transaktionsbeziehungen und der Transaktionsbeziehungsnetze in die Thematik aufgenommen. Arten und Auswirkungen von Beziehungen zwischen Transaktionen sind dann [RW90] und [Wä90] untersucht worden. Grundlage sind die in [CRR91] und [CRta] eingeführten Beziehungstypen.

Resultat war die Definition einer operationalen Semantik sowie der Entwurf und Realisierung von Basismechanismen zur Verwaltung von Transaktionsbeziehungen und –netzen [Sch91].

Dies bildete die Grundlage für die Transaktionsverarbeitung auf der Basis von Transaktionsbeziehungsnetzen. Dabei lag der Schwerpunkt im wesentlichen darauf, das völlig neue Gebiet, die Möglichkeiten und Einschränkungen verstehen und beschreiben zu lernen.

Eine prototypische Implementierung fand dann im Rahmen der APRICOTS–Projektes statt. Dies erst half, Zusammenhänge und Hintergründe zu verstehen.

An dieser Stelle sei besonderer Dank an Herrn Holger Schütz gerichtet. Die im Zusammenhang mit seiner Arbeit geführten Diskussionen konnten — dank unendlicher Beherrschtheit — immer zu einem faktischen Ergebnis und Verbesserung unserer Theorien geführt werden.

[1] vgl. CICS, UTM [MW88].

Auf seine Arbeit [Sch91] sei auch als Grundlage für das Verständnis der im nachfolgenden aufgeführten Erkenntnisse verwiesen.

Da jede der beiden Arbeiten als eigenständig betrachtet werden muß, ließ es sich naheliegenderweise nicht ganz vermeiden, Erläuterungen und Erkenntnisse doppelt aufzuführen. Für jeweils vorweggenommene Ergebnisse oder fehlende Beweise auf der einen oder anderen Seite möchten wir uns vorab entschuldigen. Wir hoffen aber, der geneigte Leser vermag durch Kombination der Ausarbeitungen einen entsprechend verständlichen und detaillierten Einblick zu gewinnen.

Kapitel 2

Das Transaktionsmodell

2.1 Das grundlegende Modell

2.1.1 Die Idee

Betrachten wir die Entwicklung heutiger Software-Systeme, so erkennen wir eine starke Zunahme an Größe, Komplexität, aber auch die steigenden Anforderungen bezüglich Sicherheit und Performanz. Zudem laufen solche System zunehmend parallel und verteilt ab. Während Parallelität vornehmlich der Steigerung der Performanz dient, stellt die Verteilung die logische Konsequenz aus der Architektur moderner Systeme dar: Modellierungsschwerpunkt ist nicht mehr die einheitliche Daten– und Rechnerzentrale, sondern die Variation von Hardwarekomponenten und deren Plazierung vor Ort, aufgeteilte Kapazität, Dezentralisierung von Daten, Integration in Netze, und so weiter. Als Folge erhalten wir physisch und logisch heterogene Systeme. Applikationen im herkömmlichen Sinne den Veränderungen anzupassen und in der Realisierung darauf Rücksicht zu nehmen, scheint ein möglicher, wenn auch teurer Ansatz. Hinzu kommt, daß moderne Systeme dynamisch in ihrer Gestalt sind, Flexibilität also eine Grundvoraussetzung ist. Wir erkennen, daß moderne Betriebssysteme in verstärktem Maße der Abstraktion zuträglich sein müssen, deklarative Ansätze zu bieten haben und selbstverständlich in ihrer Funktionalität den neuen Gegebenheiten rechnung tragen müssen [Reu90].

Wir greifen uns aus der Fülle der Aufgaben das Gebiet der Transaktionsverarbeitung heraus. Applikationen verwenden Transaktionen, um ihre Arbeit in Einheiten zu unterteilen. Diese Einheiten definieren dann Vorgänge, die sich zur Wahrung der Konsistenz und Integrität von anderen (normalen) Vorgängen der Applikation unterscheiden. Die Applikation kennt (normalerweise) nur sich und ihre Welt. Diese ist jedoch nur virtuell realisiert, in der realen Welt sind Interdependenzen die Regel, Wechselwirkungen, die in geeigneter Weise durch eine zentrale Stelle zu koordinieren sind. Nur so kann die Abstraktion für die Applikation in vernünftigem Maße aufrecht erhalten werden. Hierfür sind Transaktionen inzwischen selbstverständliche Mittel [CP84].

Die Applikation *demarkiert* Beginn und Ende einer Transaktion und führt dazwischen Arbeit aus, über deren Effekte sie erst zum Ende der Transaktion entscheiden muß. Die Arbeit besteht in der Regel aus dem Lesen, Verändern und Schreiben von Objekten; Daten die allgemein zur Verfügung stehen und somit auch von anderen genutzt werden können. Darüber hinaus kann sie sich auf die Eigenschaften der Transaktion verlassen, ohne diese jemals kontrollieren zu müssen.

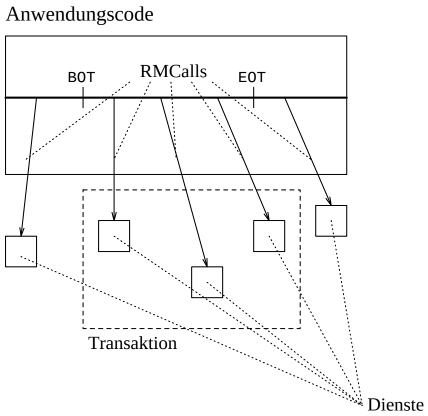


Abbildung 2.1: Dienstaufrufe unter einer Transaktion

Welche Aktionen durch die Applikation während einer Transaktion angestoßen werden, ist abhängig von den jeweiligen Gegebenheiten. So haben natürlich die Daten selbst erheblichen Einfluß auf die Bearbeitung an sich. Allein die Zahl der Datensätze, die verarbeitet werden sollen, oder deren Inhalte steuern den zugrundeliegenden Algorithmus. Auch der Status der jeweils ausgeführten Operation und das diese Operationen implementierende System (z.B. das DBMS) greifen regulierend ein.

Für die Abfolge der verschiedenen Transaktionen trägt die Applikation die Verantwortung. Jedoch gibt es hier ebenfalls zu kategorisierende Einflüsse. Denn oftmals ist die Wahl der nächsten Transaktion vom Ergebnis der vorhergehenden abhängig. Dabei umfaßt der Begriff *Ergebnis* zum einen den Erfolg oder Mißerfolg der Vorgängertransaktion, aber auch die durch diese verarbeiteten Daten. Diese werden gewissenmaßen „global weitergereicht“ oder lokal über Mittel der Applikation (z.B. Variablen) transferiert. Selten aber finden wir eine explizite, sichere Weitergabe, denn diese ist in heutigen Systemen speziell durch die Applikation zu implementieren, gängige Programmiersprachen stellen keinerlei Unterstützung diesbezüglich zur Verfügung.

Hier genau erkennen wir Ansatzpunkte für unsere Arbeit. Zum einen soll durch die Applikation beschrieben werden können, welche Verbindungen zwischen ihren Verarbeitungsschritten existieren. Nur so ist es möglich, längerfristig Abstraktion von den Daten zu wahren. Wir unterscheiden dabei verschiedene Verknüpfungselemente. Zunächst die schon erwähnten Datenkorrelationen verschiedener Transaktionen. Es hat sich gezeigt, daß es nicht

möglich sein kann, völlige Datenunabhängigkeit zwischen Transaktionen einer Applikation zu schaffen. Der Weg, einfach (Elefanten–)Transaktionen zu verwenden, ist hinlänglich als nicht sinnvoll deklariert worden. Einhergehend mit einer neuen (natürlichen) Strukturierung von Verarbeitungseinheiten sind denn auch immer neuer Transaktionsmodelle für viele (Spezial–)Fälle entwickelt worden, jedes mit Vorteilen, manche mit erheblichen Nachteilen; generell führen Spezialanwendungen nicht gerade zu große Flexibilität von Systemen. Neue Strukturierung ist notwendig, doch die jeweils resultierenden *Beziehungen* zwischen den letztendlichen Verarbeitungselementen wollen wir in Ausprägung und Auftreten auf ein gemeinsames, allgemeines Basismodell zurückführen.

In einem integrierenden Ansatz soll versucht werden, auf Basis von Transaktionsbeziehungen sowohl deklarative Mittel für die Applikation zur Verfügung zu stellen, wie auch die dynamisch während der Abarbeitung und Abfolge der jeweiligen Transaktionen sich ergebenden weiteren Beziehungen mit einzubeziehen. Wir bezeichnen im folgenden erstere als *strukturelle Beziehungen*, letztere als *dynamische Beziehungen* zwischen Transaktionen.

Halten wir fest: Es soll der Fluß zwischen Transaktionen definiert und kontrolliert gehandhabt werden. Diese Aspekte werden im ConTract-Modell ausführlich behandelt [WR91] [WR90]. Aus dieser Arbeit gehen auch gewisse Modellierungsgrundvoraussetzungen hervor, unter anderem die Basisideen für die Transaktionsverarbeitung auf Transaktionsbeziehungsnetzen, dem Gegenstand der vorliegenden Abhandlung. Unser Ziel sollten wir uns immer vor Augen halten: Transaktionen und ihre Beziehungen explizit definieren und dynamisch verarbeiten zu können. Damit einhergehend schreiben wir uns die Persistenz der Beziehungen, Konsistenz und Rücksetzbarkeit auch im Bereich der Transaktionsverwaltung auf das Banner.

2.1.2 Das klassische Transaktionsmodell

Wir wollen nun das Konzept der Transaktionen etwas näher betrachten, um zu erkennen, was ihre Mächtigkeit ausmacht und wo ihre Grenzen liegen.

Wie schon angeführt, unterliegen Transaktionen dem ACID–Paradigma. Dies erlaubt es einer Applikation in recht einfacher Weise, ihre Arbeit in sichere Verarbeitungseinheiten zu partitionieren. Für jede dieser Einheiten ist sichergestellt, daß entweder alle oder keine ihrer Ergebnisse (Atomarität) sichtbar werden, die Ergebnisse in einem konsistenten Zustand hinterlassen werden (Konsistenz), die Einheit sich nicht um Einflüsse anderer, gleichzeitig stattfindender Arbeit kümmern muß (Isolation) und letztlich alle sichtbar gewordenen Ergebnisse dies auch „für alle Zeit“[1] bleiben (Dauerhaftigkeit). Dies alles erreicht die Applikation, indem sie eine Transaktion demarkiert und in ihrem inneren die Dienste der sogenannten *Resource Manager* wahrnimmt. Ein recht einfaches Prinzip, wie gesagt.

Betrachten wir das Gesagte eingehender, so lassen sich recht schnell die Grenzen dieser Verfahrensweise aufzeigen.

[1] mindestens aber bis zur nächsten Änderung...

Lange vs. Kurze Transaktionen

Betrachten wir zunächst die Atomarität[2]. Sie war unter anderem dadurch gekennzeichnet, daß entweder alle oder keine Ergebnisse außerhalb der Transaktion sichtbar werden, sprich die Aufgabe ganz oder gar nicht erfüllt worden ist. Wohl gemerkt, wir sprechen von **genau einer** Transaktion.

Beispiel 2.1: *Trip planning*

> Betrachten wir den Vorgang einer Reisebuchung. Die Reise gehe (in Anlehnung an die Referenz) von Bolinas (Kalifornien) nach Ripa (Italien). Als weitere Gegebenheit — soviel Konstruktion muß gestattet sein — existiert kein Direktflug, so daß verschiedene Abschnitte der Reise sukzessive geplant und gebucht werden müssen. Der Zugriff auf verschiedenste Datenbanken der Fluggesellschaften, Bahnunternehmen, Mietwagen-Firmen usw. ist notwendig.
>
> Wie aber sollen wir nun diese Aufgabe in Transaktionen unterteilen? Nennen wir alles eine Transaktion, passiert folgendes: Nehmen wir an, es sind schon die ersten Teile der Reise geplant (und gebucht!). Nur bei der Zwischenstation in der bis dato 'erreichten' Stadt ist kein Hotelbett mehr frei. Offensichtlich muß die Arbeit des letzten Schrittes zurückgenommen werden, eine andere Stadt angesteuert werden. Aber die Zurücknahme (oder deutlicher das Rollback) setzt die Ergebnisse der gesamten Transaktion, also die gesamte, bisher erledigte Arbeit (auch die Taxibuchung vom Heimatort zum nächsten Flughafen) zurück, so daß von neuem begonnen werden muß.
>
> Gut, werden Sie sagen, unterlegen wir nur jeweils dem letzten Abschnitt der Reiseplanung eine Transaktion. Nur, was machen wir, wenn der Kunde sich mitten in der Planung entschließt, zur Konkurrenz zu gehen? Sie können lediglich die letzte Buchung zurücknehmen, alle vorherigen sind ja abgeschlossen, und somit nach dem Prinzip der Dauerhaftigkeit nicht mehr zurückzunehmen.

Wir können die Gegenüberstellung langer und kurzer Transaktionen noch ein wenig fortführen. Um die Isolation zu gewährleisten, sperren herkömmliche Datenbanken die von einer Transaktion verwendeten Objekte. Andere Transaktionen können diese dann nicht oder nur eingeschränkt gleichzeitig benutzen, müssen also gegebenenfalls warten oder werden abgebrochen. Bei kurzen Transaktionen ist dies weniger ein Problem, da die Konflikthäufigkeit nur sehr gering und die Wartezeit zudem kurz ist. Bei langen Transaktionen ist jedoch eine wesentlich höhere Konflikthäufigkeit gegeben, eine hohe Rücksetzrate und/oder lange Wartezeiten die Regel.

Beispiel 2.2: *Bulk Update*

> Ein einfacher zu erläuterndes Problem ist dies: Stellen Sie sich vor, eine Bank möchte am Ende des Monat bei 1.000.000 Konten den Saldo aktualisieren, indem entsprechend Beträge belastet oder gutgeschrieben werden. Warum nicht alles unter einer Transaktion ablaufen lassen? Schließlich muß die Bilanz stimmig sein.
>
> Nehmen wir an, 934 767 Konten seien abgearbeitet worden — was angesichts der Zahl wohl ziemlich lange dauern dürfte — dann tritt ein Systemfehler auf, woraufhin der Rechner „abstürzt“. Natürlich muß die Transaktion beim Wiederanlauf rückgängig gemacht werden, was auch nicht gerade schnell gehen dürfte. Außerdem ist die bis dato gemacht Arbeit hinfällig. Also warum nicht jedes Konto unter einer eigenen Transaktion bearbeiten? Nun, dann würde keinerlei Information nach dem Systemfehler darüber bekannt sein, wieviel Konten bearbeitet wurden und welche; ein Fortfahren könnte bedeuten, daß einem Konto mehrfach gutgeschrieben oder belastet wird, was nicht gerade im Sinne von Bank und Kunde sein dürfte.

[2] Die im folgenden angeführten Beispiele gehen auf [RG91] zurück.

Strukturierung von Transaktionen

Erinnern wir uns nochmals: Ein Applikation demarkiert eine Transaktion, das heißt, sie definiert Beginn und Ende der Verarbeitungseinheit. Dazwischen nimmt sie Dienste von Resource Managern wahr. Was aber, wenn in einem komplexen, modularen System, auch ein Resource Manager Transaktionen zur Strukurierung und Kontrolle seiner Arbeit verwenden möchte. In dem bisher beschriebenen Modell müssen die weiteren Transaktionen, genauer deren Resource Manager-Aufrufe, der ursprünglichen Transaktion zugeordnet werden. Denn zum einen kennen wir bisher keine Mittel, hierarchisch Transaktionen anzuordnen; das Modell macht hierüber keine Aussagen. Zum anderen müssen die „Untertransaktionen“ auf Objekte zugreifen dürfen, die auch „obere“ schon verwendet haben, so daß die gesamte Transaktion nicht *verklemmt*. Das Modell macht aber nur Aussagen über die Isolation **einer** Transaktion, sprich: soll auf das gleiche Objekt zugegriffen werden, muß dies innerhalb einer Transaktion oder durch verschiedene Transaktionen hintereinander geschehen.

Anforderungen an die Transaktionsverarbeitung

Wir haben eine offensichtliche Diskrepanz zwischen langen und kurzen Transaktionen. Lange Transaktionen auf dem herkömmlichen Modell sind nicht adäquat, da sie im Fehlerfall unnötig viel Rücksetzung implizieren und während ihrer Aktivierungszeit die Parallelität durch gehaltene Sperren stark vermindern. Kurze Transaktionen haben letzteres Problem nicht, nur kann hier nicht beliebig weit (zum Beispiel bis zum Anfang der gesamten Abarbeitung) zurückgesetzt werden. Auch kann keine Information von Transaktion zu Transaktion weitergereicht werden und bestehen in derzeitigen DBMS keine ausreichenden Möglichkeiten, transaktionsübergreifend zu synchronisieren. Da herkömmliche Programmiersprachen jedoch weder die Information sichern können noch den Zustand eines Programms persistent zu machen gestatten, stellt die Aufgabe, zusammengehörige Transaktionen in Bezug setzen zu können, ein erhebliches Problem dar.

Wir erkennen folgende Anforderungen an die Transaktionsverarbeitung:

1. Transaktionen sind nicht mehr nur kurze Vorgänge, sondern werden auch zur Beschreibung komplexer, langer Aktivitäten verwenden. Sie werden in komplexen Systemen modular aufgebaut.
2. Somit muß die Modellierung und Kontrolle komplexer Strukturen von Transaktionen ebenso möglich sein, wie die kontrollierte Freigabe von Zwischenergebnissen.

3. Die Anwendung muß darüber entscheiden können, wieviel der erfolgten Arbeit und Ergebnisse im Fehlerfall zurückgesetzt werden sollen.

4. Der Zustand der Verarbeitung muß persistent sein, daß heißt, es muß zu jedem Zeitpunkt die Arbeit unterbrochen und an dieser (oder in angemessener Entfernung weiter vorne liegenden) Stelle wiederaufgesetzt werden können. Dies gilt für Fehlerfälle wie auch für durch den Anwender initiierte Unterbrechungen.

2.1.3 Ansätze erweiterter Modelle

Aus den zuvor beschriebenen Überlegungen resultierten zahlreiche Ansätze, die versuchten, die der subjektiven Einschätzung nach markantesten Einschränkungen zu beheben. Teilweise wurde dies mit relativ geringem Aufwand erreicht, oftmals mußten jedoch erhebliche Anstrengungen unternommen werden, das gesteckte Ziel zu erreichen; niemals hingegen konnten alle beschriebenen Anforderungen erfüllt werden. Auch deshalb gibt es bis heute nicht *den* Ansatz, das Allheilmittel, sondern nur Lösungsansätze für spezielle Anwendungsszenarien.

Im folgenden wollen wir einen kurzen Überblick über eine Auswahl von Modellen geben, um daraus abzuleiten, welche Mittel zur Behebung von Mißständen existieren und welche Annahmen und Forderungen allen Modellen gemein ist. Sie werden dabei jeweils mit den zuvor beschriebenen Anforderungen 1-4 verglichen (die Modelle sind in [Gra81] [Mos81] [RG91] [Wal84] [GMS87] beschrieben).

Flache Transaktionen mit Savepoints: Die hier zugrundeliegende Idee ist die, innerhalb einer Transaktion zu einem früheren Zustand zurücksetzen zu können. Dazu werden an ausgezeichneten Punkten durch die Applikation sogenannte *savepoints* initiiert, bei der der jeweils aktuelle Zustand der Transaktion gesichert wird. Im weiteren Verlauf kann die Anwendung, z.B. im Rahmen einer Fehlerbehandlung, die Transaktion auf solch einen Zustand zurücksetzen. Dies gilt jedoch nicht im Fall von Systemfehler und Abbruch der Transaktion. Hierbei muß der Atomarität wegen die gesamte Transaktion zurückgesetzt werden.

Deshalb werden in einer Erweiterung sogenannte *persistente savepoints* eingeführt. Diese ermöglichen beim Wiederanlauf (nach Rücksetzen des letzten Intervalls) das Aufsetzen und somit die Reduktion der verlorengegangenen Arbeit.

Probleme dieser Ansätze liegen jedoch darin, daß die Applikation bezüglich der Kontrolle voll verantwortlich ist, das heißt, sie muß dafür sorgen, daß beim Wiederanlauf entsprechend aufgesetzt wird. Auch birgt das Rücksetzen innerhalb einer Transaktion gewisse Gefahren, da bekannte Programmiersprachen keine Transaktionssemantik kennen. Die Folge ist, daß die Transaktion sich zwar wieder in einem früheren Zustand, die Anwendung jedoch weder im Code noch in den Werten ihrer Variablen einfach zurücksetzen kann; ein Koordinationsproblem im Kontroll– und Datenfluß entsteht. Desweiteren können noch nicht komplexe Strukturen behandelt werden oder Zwischenergebnisse frei– und Sperren frühzeitig aufgehoben werden.

Chained Transactions: Der grundlegende Gedanke ist der gleiche wie bei Savepoints, nur daß hier die jeweils in einem Intervall erledigte Arbeit beendet wird — also in den Zustand Commit übergeht. Offensichtlich jedoch müssen gewisse Informationen in die Nachfolgetransaktion übernommen werden, wie z.B. Sperren oder SQL-Cursor-Informationen. Das Commit der alten und der BOT der neuen Transaktion werden zu einer atomaren Funktion zusammengefaßt, die den Kontext der Transaktion weiterhin an die Applikation bindet.

Intention dieses Ansatzes ist, mehr Parallelität durch Freigabe nicht mehr benötigter Sperren und Reduktion des bei einem Systemfehler auftretenden Verlustes an Arbeit zu erzielen. Es wird nämlich lediglich die aktuelle Transaktion abgebrochen bzw. zurückgesetzt, alle Ergebnisse vorheriger Transaktionen bleiben der Dauerhaftigkeit wegen weiterhin sichtbar. Der lange Vorgang wird in kleinere Einheiten unterteilt, wobei jedoch nicht mehr die Möglichkeit besteht, die gesamte Arbeit abzubrechen. Gerade hierin und im Umstand, keine komplexen Strukturen modellieren zu können, liegen die Schwächen dieses Ansatzes; wie sollte wohl das Beispiel „Trip planning“ hiermit gestaltet werden?

Nested Transactions: Auch *Nested Transactions* verfolgen die Idee der Strukturierung, allerdings nicht innerhalb von Sequenzen, sondern vielmehr hierarchisch. Dabei kann eine Transaktion *Subtransaktionen* besitzen, Sperren an diese vererben und nach Beendigung zurückerhalten. Darüberhinaus legt das Commit-Protokoll eine Ordnung für die Beendigung fest, so daß Kinder vor ihren Vätern beendet werden und der Abbruch einer Transaktion den Abbruch jedes ihrer Kinder zur Folge hat. Die Blätter dieses Baumes sind flache Transaktionen.

Betrachten wir die Struktur größerer Anwendungsprogramme, so stellen wir fest, daß sie in Einheiten, z.B. Module genannt, unterteilt sind. Solche Module erreichen Effekte über ihre Aufrufschnittstelle. Ansonsten haben sie, sofern sie nicht globale Variablen verwenden, keine Seiteneffekte. Bei Datenbankapplikationen ist dies anders, denn Datenbanken sind letztlich auch große, globale Variablen, so daß Seiteneffekte nicht vermieden werden können, es sei denn, man unterlegt den Datenmanipulationen innerhalb der Module eine Transaktion. Offensichtlich sind Ausführfolgen in Programmen hierarchisch; und genau deshalb bietet sich das Prinzip der Nested Transactions für die Unterstützung in der Modularisierung von Programmen ergänzend an.

Somit ist die Anforderung, komplexe Applikationen modellieren zu können, erfüllt. Betrachtet man jedoch die Rücksetzbarkeit, holen die Programmiersprachenprobleme den Programmierer schnell wieder ein, denn: selbst wenn das Transaktionsmodell erlaubt, auf einen beliebigen Zeitpunkt innerhalb der Baumabarbeitung zurückzusetzen, bieten herkömmliche Programmiersprachen diese Unterstützung nicht. Es kann nicht an eine beliebige Stelle in irgendeinem Modul zurückgesetzt werden, geschweige denn der Laufzeitstack in geeigneter Weise restauriert werden. Dieser Diskrepanz wegen scheint nur sinnvoll, die gesamte Transaktion als atomar zu betrachten und gegebenenfalls zurückzusetzen. Da Sperren aber nur intern weitergereicht werden können, alle Sperren erst bei Bearbeitung der obersten Transaktion freigegeben werden, erhalten wir dieselbe Problematik, wie bei langen Transaktionen[3].

[3] Dies versuchen *Multilevel Transactions* zu beheben, indem sie Subtransaktionen das vorzeitige Commit

Mini Batch: Ein erhebliches Problem stellt offenbar dar, die Position in der Aufführung einer Applikation und den Zustand von Transaktionen in Korrelation zu setzen. Die Informationen über Position der Applikation sind inherent in Laufzeitstack, Aktivierungsblöcken und so weiter verborgen und somit nicht zugänglich. Deshalb wird im *Mini Batch*–Ansatz ein einfaches, sequentielles Programm unterlegt, daß den jeweiligen Verarbeitungszustand explizit global verwaltet. So können z.B. im Beispiel „Bulk update" jeweils 100 Kontenänderungen zu einer Transaktion zusammengefaßt werden und nach deren Beendigung in der Datenbank vermerkt werden, wie weit das Programm in seiner Arbeit gelangt ist. Im Falle eines Systemfehlers und nachfolgendem Aufsetzen kann anhand dieser Information in der Verarbeitung fortgefahren werden; lediglich die Arbeit der letzten Transaktion muß erneut durchgeführt werden.

Diese Strukturierung gelingt offensichtlich jedoch nur bei jeweils gleichartigen Vorgängen — komplexe, geschachtelte Strukturen bleiben weiterhin ein Problem. Wenn sie allerdings möglich ist, erreicht man einen hohen Grad an Parallelität, nicht jedoch ohne Einschränkung. Nehmen wir an, neben der Abänderung aller Konten läuft noch ein Bilanzierungsprogramm, daß die Werte bestimmter Konten liest und addiert. So kann es sein, daß dieses dabei auf Konten alten und neuen Zustandes zugreift, die Bilanz also zwei verschiedene Zeitpunkte umfaßt und schlichtweg unausgeglichen oder falsch ist[4].

Zusammenfassung: Natürlich existieren etliche weitere Ansätze; wir wollen es hiermit allerdings bewenden lassen und die Ergebnisse zusammentragen. Folgende Beobachtungen können gemacht werden:

1. Die jeweilige Arbeit soll ein kleinere Einheiten unterteilt werden, wobei jedoch noch die Atomarität der Gesamtmenge gewährleistet werden soll.

2. Viele Optimierungen beruhen darauf, Zwischenergebnisse bekanntzugeben und nicht mehr benötigte Sperren aufzugeben. Die jeweilige Einheit wird beendet (Commit) und kann nur durch Kompensation rückgängig gemacht werden. Aus der Abgeschlossenheit der jeweiligen (kleinen) Verarbeitungseinheiten müssen wir folgern, daß ein Mechanismus zur Weitergabe bestimmter Informationen (Sperren, Cursor, ...) zur nachfolgenden Einheit bestehen muß; die Informationen werden allgemein *Kontext* genannt.

3. Wir erkennen zahlreiche Zusammenhänge zwischen den Verarbeitungseinheiten. So führt das Rücksetzen einer beendeten Transaktion zur Ausführung der zugehörigen Kompensationstransaktion, im hierarchischen Modell das Scheitern des Vaters zum Abbruch aller Kinder. Der Beweis, daß sich obige Transaktionsmodelle durch einige elementare Beziehungsmuster zwischen den beteiligten Transaktionen beschreiben lassen, wird in [CRta] [CRR91] [RG91] erbracht.

und die Freigabe von Sperren erlauben. Zur Wahrung der Atomarität werden Kompensationstransaktionen verwendet. Jedoch ist auch deren Steuerung aus der Applikation heraus diffizil, da normalerweise keinerlei Informationen des Laufzeitsystems über den aktuellen Zustand eines Programms bekannt sind.

[4]Eine Lösung wäre, die im Laufe eines Minibatches angefaßten Konten vorab alle zu sperren und nach einem Intervall die entsprechenden freizugeben. Diese Semantik ist jedoch äußerst anwendungsspezifisch und entspricht nicht der Regel.

4. Diese Zusammenhänge sind von Modell zu Modell verschieden und allesamt fixiert in der jeweiligen Definition und realisiert durch das zugehörigen Commit-Protokoll.

5. Ein großes Problem stellt dar, Struktur und Ausführung des Programms mit dem jeweiligen Zustand der Transaktionen in Deckung zu bringen. Herkömmliche Programmiersprachen bieten hierzu keinerlei Unterstützung an.

6. Das betrifft insbesondere die Autonomie in der Rücksetzung. Sie sollte auf jede beliebige Stelle möglich sein und in Übereinstimmung mit dem jeweiligen Programm erfolgen. Das setzt jedoch eine Laufzeitumgebung voraus, die in der Lage ist, den jeweiligen Zustand einer Applikation persistent zu machen und die mit der jeweiligen Verwaltung der Transaktionen zu kooperieren vermag.

Was wir in einem erweiterten Modell benötigen, sind unter anderem:

- Beziehungen zwischen Verarbeitungseinheiten, um die Zusammenhänge, Sequenzen und Hierarchieeigenschaften darzustellen,

- Kontext für Transaktionen, um über Transaktionsgrenzen hinweg die Verarbeitung in Zusammenhang setzen zu können, und Kompensationsmechanismen zur Wahrung der semantischen Atomarität,

- ein Laufzeitsystem, das persistent Zustände verwalten kann,

- anwenderbeschreibbare Synchronisation, um in hohem Maße Parallelität zu gestatten.

Ein System, das versucht, obige Aspekte zu beleuchten, ist im ConTract-Modell beschrieben.

Inhalt der vorliegenden Arbeit sind dabei die Beziehungen zwischen Transaktionen, deren persistente, konsistente Verwaltung und Interpretation. Dabei wird versucht, in einem deklarativen Ansatz die Modellierung von Einheiten und deren Zusammenhänge durch die Anwendung beschreiben zu lassen. Gleichzeitig sollen auch solche Abhängigkeiten erfaßt werden, die transparent für die Applikation sind, wie zum Beispiel resultierende Leseabhängigkeiten oder ähnliches. Die gesamte Beschreibung ist somit in sich dynamisch, daß heißt, auch während der Abarbeitung, zum Beispiel im Rahmen eines Commit-Protokolls, kann sich die Beschreibung ändern. Dies hat konkret Einfluß auf Verwaltung, Konsistenzkontrollen, das eigentliche Commit-Protokoll und anderes. In gleichem Atemzug wird es jedoch möglich sein, beliebige Transaktionsmodelle innerhalb einer Anwendung zu kombinieren.

Auf das Modell der Transaktionsbeziehungen wird nun im folgenden eingegangen werden. Begriffe der Konsistenz oder Atomarität und andere müssen neu definiert, Beziehungsarten erläutert werden. Anschließend wird eine Verarbeitungsmodell in Form eines Übergangsprotokolls auf Transaktionsbeziehungen aufgezeigt. Aspekte der Verteilung und Integration in ein reales Betriebsmodell folgen.

2.1.4 Begriffe und Notation

Transaktionskennung

Zunächst müssen wir, da wir gleichzeitig mit mehreren Transaktionen umgehen müssen, Identifikatoren für diese einführen: GTRID Um nicht gleich Beschränkungen zu unterliegen, deklarieren wir diese als global eindeutig und für alle Zeiten gültig (eternal unique), daß heißt, eine GTRID wird nicht — auch nicht nach Systemabsturz und –wiederanlauf — erneut vergeben. Um im folgenden 2 GTRIDs zu unterscheiden, indizieren wir diese, so daß GTRID_i von GTRID_j verschieden ist, wenn $i \neq j$.

Zustände von Transaktionen

Die Literatur auf diesem Gebiet ist vornehmlich englisch-sprachig, weshalb einige der Begriffe in das Deutsche übernommen wurden. Auch in dieser Arbeit sei dies gestattet; eine deutsche Phrase in diesem Zusammenhang zu verwenden verwirrt nur und erscheint künstlich. Für die Demarkation der Transaktionen verwenden wir die klassischen Begriffe `begin-of-transaction` (BOT), `end_of_transaction` (EOT) und `rollback` (Rollback). Dies sind Termini, wie sie die Applikation gegenüber der Transaktionsverwaltung verwendet.

In der Zusammenarbeit mit den Resource-Managern hingegen sind die Begriffe join_transaction, prepare_to_commit, abort_transaction und commit_transaction gebräuchlich.

Der teilweise verschiedenen Semantik, des erweiterten Umfanges und nicht zuletzt auch der klaren Trennung von Applikation, Resource Manager und Transaktionsverwaltung wegen, kennt die Transaktionsverwaltung, genauer die Transaktionsbeziehungsnetzverwaltung, die Begriffe `installed`, `running`, `finished`, `aborted` und `committed`. `Installed` kennzeichnet, daß eine Transaktion der Transaktionsverwaltung bekannt gemacht worden ist, eventuell schon Beziehungen eingetragen worden sind, jedoch die Transaktion noch nicht begonnen wurde. Das wiederum wird durch `running` beschrieben. `Finished` entspricht im wesentlichen dem Zustand `Prepared`, daß heißt, die Transaktion läuft nicht mehr und erwartet ihr Ende.

2.2 Beziehungen zwischen Transaktionen

Mit Beziehungen definieren wir Einschränkungen zwischen und Aktionen auf Transaktionen, die durch gewisse Gegebenheiten erforderlich sind oder werden und durch Transaktionsübergänge oder Netzmodifikationen wirksam werden.

Obwohl in der Bedeutung nicht gänzlich zu trennen, wollen wir folgende Begriffe unterscheiden: Hinlänglich bekannt sind *Transaktionsmodelle*. Ausgehend vom Modell der „flachen Transaktionen“ erweitern sie dieses und reichern es semantisch an. Sie definieren im allgemeinen bestimmte, nicht variable Klassen von Transaktionen verschiedener Mächtigkeit oder Funktionalität sowie gewisse ausgezeichnete Operationen zwischen diesen, seien es Sperrweitergaben oder auch Abbruchreihenfolgen und andere. Somit ist im allgemeinen ein fixes Gerüst von Beziehungen vorgegeben, das von der Applikation mit realen Transaktionen ausgefüllt werden kann. Als *Transaktionsbeziehungsnetz* (TBN) hingegen bezeichnen wir die explizite Darstellung von Beziehungen zwischen einheitlichen Transaktionen gleicher Mächtigkeit und Funktionalität. Dieses umfaßt, Übergangs– und Commit-Protokolle vorausgesetzt, die Festlegung der Abfolgen von Transaktionsübergängen und Ereignisse bzw. Aktionen. Somit kann ein TBN verschiedene Transaktionsmodelle beinhalten und vereinen, genauso aber auch keines explizit repräsentieren. Es stellt einen Baukasten, gewissermaßen ein Programmiermittel für Transaktionen und ihre Abhängigkeiten und Einflüsse aufeinander dar.

Wir unterscheiden in unserer Untersuchung drei Klassen von Beziehungen: *logische Beziehungen*, *zeitliche Beziehungen* und *Kontrollfluß-Beziehungen*. Hinzu kommen Ereignismeldungen; wir nennen sie *Events*.

2.2.1 Logische Beziehungen

Betrachten wir zum Beispiel die Leseabhängigkeit der Transaktion T_2 von T_1. Diese impliziert, daß, wenn T_1 abbricht, auch T_2 abzubrechen hat[5], da von T_2 verwendete Werte nicht mehr gültig sind. Wir schreiben hierfür

$$\mathrm{T}_1 \rightarrow_a \mathrm{T}_2$$

Diese wenn–dann-Bedingungen nennen wir *logische Beziehungen*. Sie stellen Restriktionen bezüglich der Menge möglicher Transaktionsübergänge dar.

Definition 2.1: *Logische Beziehung*

> ***Eine logische Beziehung ist eine wenn–dann-Bedingungen zwischen T_i und T_j derart, daß wenn T_i in den Zustand Z_i übergeht, T_j den Zustand Z_j erreichen muß.***
>
> *Die allgemeine Schreibweise lautet:*
>
> $$T_i \;{}_{Z_i}\!\rightarrow_{Z_j}\; T_j$$

Anmerkung:

> Der kürzeren Schreibweise wegen wird, wenn Z_i=Z_j ist, Z_i weggelassen.
>
> $\mathrm{T}_1 \;{}_a\!\rightarrow_a \mathrm{T}_2$ wird dann $\mathrm{T}_1 \rightarrow_a \mathrm{T}_2$ geschrieben. Darüber hinaus kann der Pfeil auch in die entgegengesetzte Richtung deuten; die Bedeutung ist dann entsprechend umgekehrt zu lesen.

Die Eigenschaft einer Gruppe von Transaktionen, die entweder das Commit oder das Abort aller beteiligter Transaktionen fordert, erreicht man zum Beispiel durch die logische Beziehung des gegenseitigen Commits, $\mathrm{T}_i \leftrightarrow_c \mathrm{T}_j$.

Definition 2.2: *Gegenseitige logische Beziehung*

> ***Eine gegenseitige logische Beziehung ist eine genau–wenn–dann-Beziehung zwischen 2 Transaktionen T_i und T_j derart, daß T_i genau dann, wenn T_j in den Zustand Z_j übergeht, in den Zustand Z_i übergehen muß.***

[5] Synchronisation durch *locking* vorausgesetzt

Die allgemeine Schreibweise lautet:

$$T_i\ {}_{Z_i}\!\leftrightarrow_{Z_j}\ T_j$$

Sie entspricht der Aggregation (Und-Verknüpfung) der beiden einseitigen Beziehungen

$$T_i\ {}_{Z_i}\!\rightarrow_{Z_j}\ T_j \text{ und } T_i\ {}_{Z_i}\!\leftarrow_{Z_j}\ T_j$$

2.2.2 Zeitliche Beziehungen

Auch *zeitliche Beziehungen* schränken die Abfolge der Transaktionsübergänge der beteiligten Transaktionen ein. Während die logischen Beziehungen jedoch lediglich die Forderung bezüglich eines definierten Abgleichs von Zuständen zweier Transaktionen zu einem beliebigen Zeitpunkt erheben, deklarieren zeitliche Beziehungen eine Reihenfolge der Übergänge.

Zum Beispiel definiert die Beziehung $T_1\ {}_c\!<_b\ T_2$, daß das Commit der Transaktion T_1 **vor** dem Beginn der Transaktion T_2 liegen muß.

Definition 2.3: *Zeitliche Beziehungen*

Eine zeitliche Beziehung legt fest, daß Transaktion T_i den Zustand Z_i erreicht haben muß, bevor Transaktion T_j den Zustand Z_j erreicht.

Die allgemeine Schreibweise lautet:

$$T_i\ {}_{Z_i}\!<_{Z_j}\ T_j$$

2.2.3 Kontrollfluß-Beziehungen

Kontrollfluß-Beziehungen stellen nun, im Gegensatz zu den bisher eingeführten Beziehungen, das „aktive Element“ des Beziehungsnetzes dar. Sie definieren explizit, welcher Zustandsübergang durch einen anderen impliziert wird. Während also logische Beziehungen maximal das Eintreten eines Folgeübergangs (zu irgendeinem späteren Zeitpunkt) fordern können, lösen Kontrollfluß-Beziehungen diese direkt aus.

Definition 2.4: *Kontrollfluß-Beziehungen*

Geht eine Transaktion T_i in den Zustand Z_i über, so wird für die Transaktion T_j der Übergang in den Zustand Z_j ausgelöst.

Die allgemeine Schreibweise lautet:

$$T_i\ {}_{Z_i}\!\mapsto_{Z_j}\ T_j$$

Kontrollfluß-Beziehungen koordinieren genau genommen die Übergänge von Transaktionen, die miteinander einhergehen müssen, um die Semantik des jeweiligen Transaktionsmodells oder –netzes zu erfüllen.

Es bleibt zu bemerken, daß zum Beispiel die Beziehung $T_1\ {}_a\!\mapsto_b\ T_2$ nicht bedeutet, daß beim Abbruch von T_1 automatisch der Code, der der Transaktion T_2 unterliegt, gestartet wird; darüber kann die Transaktionsverwaltung in der Regel nichts wissen. Es bedeutet lediglich den Beginn der Transaktion T_2.

Um sie vom Kontrollfluß eines Programms zu unterscheiden, nennen wir diese Beziehungen im folgenden *Trigger*.

2.2.4 Ereignismeldungen

Da wir später für die Abbildung von Steps auf Transaktionen verstärkt Informationsaustausch zwischen der Komponente, die die Steps verwaltet, und der Transaktionsverwaltung benötigen, führen wir an dieser Stelle noch die *Ereignismeldungen* oder, wie im folgenden genannt, *Events* ein.

Definition 2.5: *Events*

Events werden bezüglich des Erreichens eines bestimmten Zustands einer Transaktion definiert. Erreicht eine Transaktion T_i den Zustand Z_i, so wird eine Nachricht mit entsprechendem Inhalt generiert.

Es ist ganz offensichtlich, daß die verschiedenen, dargstellten Beziehungen nicht orthogonal sind. So kann z.B. der Abbruch einer Transaktion sowohl aus einer logischen Beziehung wie auch aus einem Trigger abgeleitet werden. Wir sind uns dessen bewußt, nur konnten wir noch keinen Beweis führen, welche Kombination von Beziehungsklassen als *minimale Basis* für die Transaktionsverwaltung ausreichend wäre. Da jedoch die vorliegende Transaktionsverwaltung vornehmlich als Basis für die Abwicklung von ConTracts dienen soll, und in diesem Modell der Kontrollfluß explizit in den Kontrollbereich des ConTract-Managers verlagert wurde, fallen Kontrollflußbeziehungen und zeitliche Beziehungen aus der Betrachtung heraus; schließlich entscheidet der CM aufgrund von Status und Ergebnis eines Steps, wann welcher weitere Step gestartet wird. Es findet also fortwährend wechselseitig Einfluß auf die Abfolge von Steps (und Transaktionen) statt, wobei der Transaktionsverwaltung lediglich die Aufgaben, logische Abhängigkeiten und Ereignismeldungen zu betrachten bzw. generieren.

Im Rahmen der vorliegenden Arbeit wurde den logischen Beziehungen eine semantisch recht große Bedeutung zugedacht. Sämtliche weiteren Ausführungen, seien es Betrachtungen von Transaktionsbeziehungsnetzen oder das Commit-Protokoll, basieren auf der ausschließlichen Verwendung dieser Beziehungen. Allein bei der Abwicklung von Steps kommen die Ereignismeldungen ergänzend hinzu. Wir werden zeigen, daß diese Vorgehensweise für ein allgemeines Transaktionsmodell unbefriedigend ist und auf geeignete, andere Verfahren verweisen. Es bleibt zu beachten, daß das APRICOTS–Projekt für diese Arbeit im Vordergrund stand.

Nicht zuletzt sollen die Erkenntnisse uns jedoch helfen, einen umfassenden Ansatz für ein allgemeines Transaktionsmodell herzuleiten. Dazu mögen die aufgezeigten Probleme und Unzulänglichkeiten dienen.

Kapitel 3

Gegenstand der Untersuchung

Wir haben bislang die herkömmliche Transaktionsverarbeitung und die Grundlagen des Transaktionsmodells mit seiner Notation kennengelernt. Auch sollte klar geworden sein, warum wir überhaupt eine Transaktionsverwaltung auf Beziehungen brauchen.

Die Frage, der wir uns nun zuwenden wollen, ist „Was eigentlich ist eine Transaktionsverwaltung auf Beziehungen?“. Gleichermaßen sollen diese Ausführungen als Leitfaden durch die folgenden Kapitel dienen, die die jeweiligen Themenkomplexe ausführlich behandeln.

Eine primäre Aufgabe einer Transaktionsverwaltung ist die Koordination der einzelnen Transaktion und der an ihr partizipierenden Resource Manager zu einem eindeutigen Ergebnis. Hier kennen wir das 2-Phase-Commit mit seiner einfachen Entscheidung: Entweder alle Resource Manager beenden die Transaktion (Commit) oder alle brechen sie ab (Abort). Diese Verfahren ist für verteilte Transaktionen geeignet und im Laufe der Zeit um Fehlertoleranz erweitert und bezüglich der Kommunikation optimiert worden.

Selbstverständlich hat auch das vorliegende Modell verteilte Transaktionen zu berücksichtigen. Im Vordergrund steht jedoch die **Koordination von mehreren Transaktionen, um die Bedeutung des jeweiligen Transaktionsbeziehungsnetzes zu erfüllen**. Die Auswirkungen sind dramatisch. So haben wir nicht mehr den einfachen Zusammenhang von Resource Manager und Transaktion und die damit verbundene, einfache Entscheidung. Wir müssen (beliebige) Abhängigkeiten von Transaktionen im Netz behandeln und dementsprechend dann die Entscheidungen für die einzelnen Transaktionen treffen. Ergo kann die Entscheidung nicht mehr isoliert für eine Transaktion, sondern nur unter Beachtung des jeweiligen Kontextes (anderer Transaktionen) getroffen werden. Ein Transaktionsübergang ist nicht mehr nur eine finale Operation, die eine Entscheidung auslöst; sie kann vielmehr auch abgelehnt werden oder gar andere Transaktionsübergänge zur Folge haben. Umstände, die in der TAV und der Applikation ihren Niederschlag finden müssen.

Beispiel 3.1: *Was ist Transaktionsverwaltung auf TBN?*

> Das Beispiel soll die Thematik ein wenig verständlicher machen (Abbildung 3.1). Wir sehen hier verschiedene Transaktionen in einem Netz. Gehen wir davon aus, diese sind von einer Anwendung so eingetragen worden und nun beginnt die Verarbeitung. Daraus resultieren nun Aufträge an die Transaktionsverwaltung, die zum Beispiel das Ende einer Transaktion betreffen. Generell mag solch einem Auftrag nichts entgegenstehen, wir jedoch müssen das Szenario genauer betrachten. Vielleicht existieren

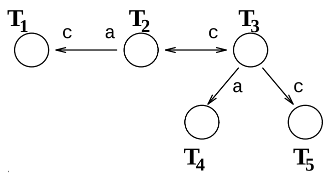


Abbildung 3.1: Verwaltung auf Transaktionsbeziehungen

ja aufgrund der Beziehungen und der Zustände anderer Transaktionen Bedenken. Vielleicht müssen mit dem Übergang weitere Übergange einhergehen.

Allein die Tatsache, daß mit einem Transaktionsübergang andere Übergänge einhergehen können — ja sogar deren Voraussetzung sind — fordert eine neue Definition es Atomaritätsbegriffs, diesmal auf Ebene der Transaktionsbeziehungsnetze.

Beispiel 3.2: *Atomarität bei TBN-Operationen*

So kann zum Beispiel das Commit von T_2 nur dann durchgeführt werden, wenn auch das von T_3 erfolgt. Beide oder keine; Atomarität über Transaktionsgrenzen hinweg.

Ebenso muß die Konsistenz eines Netzes beschrieben werden. Wann gilt ein Netz als gültig? Wann ist es überhaupt erfüllbar? Hinzu kommt, daß, wie zu zeigen sein wird, auch Modifikationen am jeweiligen Netz Transaktionsübergänge auslösen können.

Beispiel 3.3: *Koordination bei Netzmodifikation*

Nehmen wir zum Beispiel an, für T_3 sei ein Commit gefordert worden, das aufgrund der Beziehung zu T_2 verzögert wurde. Wird nun eben jene Beziehung gelöscht, spricht der Beendigung von T_3 nichts mehr entgegen.

Wenn wir wie selbstverständlich von der Koordination von Transaktionen sprechen, darf dies nicht darüber hinwegtäuschen, daß wir Neuland betreten. Bisher war niemals die Rede davon, eine Transaktion zu verzögern. Wohl war dies für die Resource Manager ein dem 2-Phase-Commit entsprechendes, normales Vorgehen. Aber nun sind ja auch die Transaktionen bezüglich ihrer Entscheidung abzustimmen, diese müssen das jeweilige Ergebnis garantieren können; also, wie exemplarisch gezeigt, auch auf dieser Ebene eine 2-Phasigkeit? Es muß für Transaktionen nicht nur jeweils eine Entscheidung getroffen werden, sondern für die Entscheidungsfindung von Transaktionen die Erreichbarkeit eines — aufgrund der Beziehungen — geforderten Zustands anderer Transaktionen überprüft und garantiert werden.

Beispiel 3.4: *2-Phasigkeit und 2-Schichtigkeit*

Wurde für T_3 das Commit gefordert, so wurde diese verzögert, weil keine Aussage darüber existierte, ob T_2 sein Commit würde *erreichen* können. Dieser Prepare–Zustand muß nun solange aufrechterhalten werden, bis auch T_2 sein Commit fordert und garantiert. Die Commit-Zustände sind dann, analog herkömmlicher Transaktionssemantik, auf die einzelnen Transaktionen, sprich die partizipierenden Resource

Manager abzubilden; das verlangt eine 2-schichtige Abwicklung von Netz– und Transaktionskoordination.

Als nächstes sind mehrere Beziehungen, wie sie zwischen 2 Transaktionen definiert sein können, in Bezug zueinander zu setzen. Wir müssen prüfen, ob bestimmte Beziehungen zum jeweiligen Zeitpunkt überhaupt verträglich zueinander sind oder sich etwa widersprechen.

Beispiel 3.5: *Konsistenzkontrolle im Netz*

Würde zum Beispiel die Eintragung ${}_c{\rightarrow}_a$ zwischen T_2 und T_3 vorgenommen, stünde dies im Widerspruch zu der schon bestehenden Beziehung, denn T_3 kann entweder erfolgreich beendet oder abgebrochen werden, aber nicht beides.

Die Auswirkungen von Beziehungen auf irgendwo im Netz plazierte Transaktionen sind zu untersuchen und in einem *Übergangsprotokoll* zu berücksichtigen.

Beispiel 3.6: *Transitivität*

Im Beispiel folgt aus dem Auftrag Commit(T_2) nicht nur die Forderung Commit(T_3) (hierfür ist ja eine Beziehung eingetragen), sondern auch die Forderung Commit(T_5), obwohl zwischen T_2 und T_5 keine direkte Beziehung eingetragen war.

Überhaupt ist das Protokoll für Zustandsübergänge nicht erst bei der Commit-Entscheidung gefragt; vielmehr muß das Netz schon während seiner Entstehung koordiniert werden, schließlich führte ja schon das Einfügen von Beziehungen zu Konsistenzkontrollen und Einflüssen auf Transaktionen.

Ein weiterer, großer Unterschied liegt in der Behandlung von Fehlerfällen. Im herkömmlichen Sinne führt ein Fehler zum Rücksetzen der Transaktion, die Ergebnisse wurden nach außen hin einfach nicht sichtbar. Dies war immer möglich, solange die Transaktion noch nicht beendet war.

Im Falle der Transaktionsbeziehungsnetze ist die Sache nicht ganz so einfach. Wohl können auch hier die Transaktionen zurückgesetzt werden, ja oftmals kann dies gar nicht verhindert werden, wie zum Beispiel der Abbruch von T_1.

Nur, wenn eben jener Abbruch im Widerspruch zu den Beziehungen und den daraus resultierenden Forderungen an diese Transaktion steht? Ist nämlich zuvor schon T_2 abgebrochen worden, **mußte** T_1 ihr Commit erreichen, aber nun?

Es ist zwischen verschiedenen Fehlerklassen zu unterscheiden und zu untersuchen, welche davon verhindert oder behoben werden können. Selbst dann ist noch zu klären, wie das Vorgehen in solchen Fällen auszusehen hat. Bisher wurden jegliche Zusammenhänge von Transaktionen im Applikationscode realisiert, ebenso die Fehlerbehandlung. Das hat seinen Sinn darin, daß die Applikation „die Sicht der Welt kennt". So muß eine Fehlerbehebung in Zusammenarbeit von Applikation und Transaktionsverwaltung geschehen, denn: das Netz repräsentiert eben diese Welt (aus Sicht der Anwendung). Es ist nicht mehr damit getan, daß die TAV ein Scheitern einer Transaktion meldet; vielmehr müssen Informationen über die Ursachen oder Möglichkeiten zur Behebung zur Verfügung gestellt werden. Die Frage ist, ob nicht auch das Netz in traditionellem Sinn zum Beispiel zurückgesetzt werden kann, und wie weit? Schließlich sind die im Netz verzeichneten Transaktionen nicht so einfach rückgängig zu machen, unterliegen sie doch dem ACID-Paradigma und somit der Dauerhaftigkeitsgarantie.

Auch im Falle der Verteilung sind neue Begriffe und Verfahren notwendig. Da auch Transaktionen verschiedener Rechner Beziehungen zueinander eingehen können, müssen, neben verteilten Transaktionen, auch verteilte Transaktionsbeziehungsnetze verwaltet und Übergänge darauf koordiniert werden. Darüberhinaus müssen diese beiden Aufgaben aufeinander bezüglich der Richtigkeit und Performanz abgestimmt werden.

Mehr noch als im Falle der verteilten Transaktionen ist auf effiziente Speicherung der Netze und optimale Protokolle zu deren Koordination zu achten. Schließlich ist die Zugriffshäufigkeit auf die Informationen wesentlich höher, nicht zuletzt der Konsistenzprüfung bei jeder Operation und der Transitivität der Auswirkungen wegen. Letztere stellt uns vor die Aufgabe, möglichst früh unnötige Informationen zu eliminieren, das Netz zu reduzieren.

Größere Schwierigkeiten dürfte der Umstand bereiten, daß zu den Berechnungen für die Übergangsentscheidungen ja eventuell das ganze Netz zur Verfügung stehen muß. Wie ist dies effizient zu erzeugen? Was sind die Kriterien dieser Effizienz? Welchen Einfluß hat zum Beispiel die physische Speicherung? Wirkt die Koordination der Transaktionen der Koordination des Netzes in Bezug auf die Performanz nicht entgegen?

Dies alles sind Fragen, die die Realisierung und Handhabbarkeit der Transaktionsverwaltung auf Beziehungen betreffen. Darüber hinaus existieren noch zahlreiche theoretische oder auch ganz praktische Fragen. Es ist zum Beispiel zu klären, welche Zusammenhänge zwischen Transaktionen überhaupt bestehen. Sind diese rein auf den Kontrollfluß bezogen, oder haben wir auch eine zeitliche Dimension vorliegen? Welche dieser Beziehungsklassen gereicht uns zu einer minimalen Basis, die jegliche Anforderungen erfüllen kann? Gibt es überhaupt diese minimale Basis?

Ein ganz anderer Katalog von Fragen ergibt sich aus der Anforderung der Einbindung in das ConTract–Modell. Hier sollen den Steps Transaktionen unterlegt werden. Es hat sich zu zeigen, ob sich gewisse Modellierungsrichtlinien herausfiltern lassen, die für die dort verwendeten, wiederkehrenden Sprach– und Transaktionskonstrukte als Beziehungsschemata gereichen können. Auch muß wohl, da die Transaktionsverwaltung ja auch Entscheidungen bezüglich Transaktionen trifft (zum Beispiel Abort) und somit Einfluß auf den Kontrollfluß nimmt, eine Interaktion zwischen der Skriptverwaltung und der TAV spezifiziert werden. In die Thematik der TAV muß auch aufgenommen werden, inwieweit spezielle Funktionalität des ConTract–Modells, wie zum Beispiel Suspendierung, Kompensation oder Migration, spezielle Verfahren notwendig machen [KR] [WR91].

Nicht alle bisher aufgezeigten Fragen können im Rahmen diese Arbeit behandelt und beantwortet werden. Zu den offenen Fragen gesellen sich Themen der Synchronisation bei Datenzugriffen oder auch der Recovery. Diese müssen zunächst außen vor bleiben.

Fassen wir die Fragestellungen nochmals kurz zusammen:

- Klärung der Bedeutung von Beziehungen und deren Einfluß auf die Koordination von Transaktionen.
- Erweiterung der Beziehungen in einen Netzbegriff unter Berücksichtigung der transitiven Auswirkungen.
- Entwicklung eines Übergangsprotokolls, das die Abfolge der Transaktionsübergänge entsprechend der jeweiligen Netzbedeutung korrekt abwickelt.

- Analyse der Korrektheitsaspekte dieses Übergangsprotokolls. Betrachtung der möglichen Fehlerfälle und Ansätze zu deren Behebung.
- Kritische Betrachtung des vorliegenden Transaktionsmodells bezüglich Vollständigkeit und Angemessenheit.

Kapitel 4

Das Transaktionsbeziehungsnetz

Aus den bislang geschilderten Komponenten lassen sich nun *Transaktionsbeziehungsnetze* modellieren. Im folgenden wollen wir in den Abschnitten jeweils ein abschließendes Beispiel zur Verdeutlichung verwenden. Dieses bezieht sich dann immer auf das in Abbildung 3.1 gezeigte TBN. Für die graphische Darstellung der Transaktionszustände, werden die in Abbildung 4.1 dargestellten Symbole verwendet.

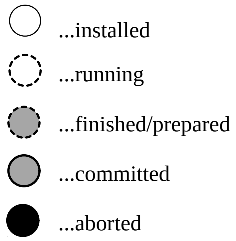


Abbildung 4.1: Zustände der Transaktionen im Beispiel

4.1 Grundlagen

Definition 4.1: *Transaktionsbeziehungsnetz (TBN)*

> *Das Transaktionsbeziehungsnetz umfaßt Transaktionen, deren Zustände und Beziehungen untereinander. Dabei werden (transitiv) zusammenhängende Transaktionen als ein Netz bezeichnet. Transaktionen ohne Verbindung untereinander gehören zu verschiedenen Netzen. Eine einzelne Transaktion stellt ebenfalls ein Netz dar.*

Ein TBN drückt explizit die Zusammenhänge und Abhängigkeiten verschiedener Transaktionen zu– bzw. voneinander zum jeweiligen Zeitpunkt aus. Generell werden die Beziehungen direkt zwischen Transaktionen definiert. Aus verschiedenen Beziehungen zwischen 2 Transaktionen resultiert eine *aggregierte Beziehung*; diese entspricht einer Und-Verknüpfung, welche erfüllbar sein muß.

Nicht jede Konstellation von Transaktionen, Zuständen und Beziehungen stellt eine gültige Ausprägung eines TBN dar. Gültige Formen bezeichnen wir als *konsistente TBN*, ungültige entsprechend als *inkonsistente TBN*[1].

Definition 4.2: *Konsistenz eines TBN*

> ***Ein TBN ist genau dann in einem konsistenten Zustand, wenn keine direkt widersprüchlichen Beziehungen eingetragen sind und auch keine unerfüllten Forderungen bezüglich Transaktionszuständen bestehen.***

Besteht zwischen 2 Netzen TBN_1 und TBN_2 eine Beziehung B, so gelten die Auswirkungen von B nicht nur für die beiden durch B verbundenen Transaktionen, sondern 'pflanzen' sich auf alle Transaktionen in TBN_1 und TBN_2 fort.

4.2 Verwaltung von TBN

Transaktionsbeziehungsnetze werden durch das Generieren von Transaktionen und das Eintragen von Beziehungen erzeugt. Dabei und bei Übergängen von Transaktionen in neue Zustände kommt es dann zu einer Interpretation des jeweiligen Netzes. Als *Operationen auf TBN* definieren wir das Eintragen und Austragen von Beziehungen sowie Übergänge von Transaktionen.

4.2.1 Garantie der Terminierung

Die Verwaltung auf TBN leistet nun die Garantie, Transaktionsübergänge derart zu koordinieren, daß zu jedem Zeitpunkt das TBN sich in einem konsistenten Zustand befindet, im besonderen, wenn alle Transaktionen einen finalen Zustand (Commit oder Abort) erreicht haben; andernfalls wird während der Abarbeitung auf *Inkonsistenz* erkannt und eine entsprechende *Fehlerbehandlung* eingeleitet. Die TBN-Verwaltung implementiert die *Semantik des TBN*. Diese Verwaltung auf TBN wird durch den *Transaction Manager* (TM) realisiert.

Eine Operation überführt ein TBN in ein neues TBN. Beide TBN müssen sich in einem konsistenten Zustand befinden, ansonsten kann die Operation nicht ausgeführt werden. Bezüglich der Überführung eine TBN in einen neuen Zustand wird also die Atomarität und die Konsistenz-Eigenschaft gefordert; hinzu kommen die Dauerhaftigkeit und Isolation.

[1] Inkonsistenzen können nicht gänzlich vermieden werden. Wir werden dies in Abschnitt 7.2 behandeln

4.2.2 Eigenschaften von TBN-Operationen

Definition 4.3: *Atomarität und Konsistenz von TBN-Operationen*

> ***Die Verwaltung von TBN führt eine Operation wie das Ein– und Austragen von Beziehungen oder Transaktionszustandsänderungen genau dann aus, wenn das daraus resultierende TBN einen konsistenten Zustand einnimmt. Ansonsten wird die Operation abgelehnt bzw. anderweitig behandelt.***
>
> ***Kann die Operation nicht abgelehnt werden (z.B. ein Abbruch einer Transaktion) und befindet sich das Netz anschließend in einem inkonsistenten Zustand, wird eine entsprechende Fehlerbehandlung eingeleitet.***

Anmerkung:

> Dies entspricht nicht gänzlich der Atomaritätseigenschaft von Transaktionen, da bei diesen in jedem Fall die aktuelle Bearbeitung abgebrochen und auf den alten Zustand zurückgesetzt werden kann.
>
> Wir definieren jedoch als Ziel der TBN-Verwaltung die erfolgreiche Erfüllung der durch das TBN definierten Semantik. Inkonsistenz ist auf eine Diskrepanz zwischen modellierter und realer Welt zurückzuführen, die einer Ausnahmebehandlung bedarf.

Einmal durchgeführte Operationen sind, entsprechend der Bedeutung bei Transaktionen, *dauerhaft*. Sie können nicht rückgängig gemacht werden.

Anmerkung:

> Ein– bzw. Austragen von Beziehungen stellen inverse Operationen dar. Folgen aus einer Modifikation eines Netzes nicht weitere Operationen, kann durch die entsprechend inverse Operation der ursprüngliche Zustand des TBN wiederhergestellt werden. Aus Sicht der TBN-Verwaltung macht dies jedoch keinen Unterschied; vielmehr werden beide Operationen unabhängig voneinander betrachtet und ausgeführt.
>
> Aus der Eigenschaft, daß TBN monoton restriktiver werden mit dem Eintragen weiterer Beziehungen und Übergängen von Transaktionen, folgt, daß das Austragen von Beziehungen jederzeit möglich ist.
>
> Insbesondere bei Transaktionsübergängen kann keine Rücknahme der Operation vorgenommen werden, da der Zustandsgraph einer Transaktion lediglich das Fortschreiten in einer Richtung kennt. Eine semantische Atomarität kann aus verständlichen Gründen auf dieser Ebene nicht realisiert werden, da die TBN-Verwaltung keinerlei Einblick in den Inhalt der jeweiligen Transaktionen besitzt.

Desweiteren garantiert die TBN-Verwaltung, daß jede Operation genau einmal durchgeführt wird. Dies gilt auch im Fehlerfall.

Gleichzeitig muß gefordert werden, daß Operationen unabhängig von der Ausführung anderer Operationen durchgeführt werden (*Isolation*). Abhängigkeiten zwischen Operationen können weder beschrieben noch verwaltet werden, was aber auch nicht der zugrundeliegenden Intention von TBN entspräche. So soll die Nutzung der Dienste der TAV transparent für die Anwendung und somit unabhängig von anderen Anwendungen oder Ereignissen sein. Erreicht wird dies durch Serialisierung der Operationen. Das heißt, Operationen werden sukzessive behandelt und entsprechend für durchgeführt oder gescheitert erklärt. Wollten wir parallel mehrere Operationen auf einem TBN ausführen, wäre die Einführung einer neuen Abstraktionsebene — mit einer Klammerung *begin-tbn-operation* und *end-tbn-operation* — nötig. Das Generieren von TBN durch Eintragen von Beziehungen bedarf jedoch wohl nicht der Parallelität und Übergänge von Transaktionen parallel und in Abhängigkeit voneinander durchzuführen, dürfte mit Beziehungen einfacher sein.

Hingegen Operationen verschiedener Komponenten in Abhängigkeit zu setzen, widerspräche der Forderung der Transparenz. Ganz davon abgesehen, daß durch TBN jene Beziehungen und Abhängigkeiten beschrieben werden sollten, die wir eben auch durch parallele Aktivitäten erhalten.

4.2.3 Dynamik von TBN

Wir hatten die Reihenfolge von Operationen auf TBN nicht eingeschränkt. Es können also Netzmodifikationen auch dann noch erfolgen, wenn bereits Transaktionsübergänge stattgefunden haben. Dies stellt uns vor ein Interpretationsproblem.

Beispiel 4.1: *Reihenfolgeproblematik*

> Nehmen wir an, für eine Transaktion T_1 wird ein Übergang gefordert, dem aufgrund des derzeitigen Netzes stattgegeben wird. Anschließend wird eine Beziehung eingetragen, die, wenn der Zustandsübergang jetzt erst erfolgen sollte, diesen verhindern würde. Soll nun, nach Eintragen der Beziehung, der Übergang rückgängig gemacht werden? Wohl kaum, denn, so hatten wir es definiert, eine erfolgte Operation kann nicht rückgängig gemacht werden, der Transaktionsübergang sowieso nicht.
>
> Folge: Wollten wir nachträglich in der geschilderten Weise interpretieren, müßten wir das Netz für inkonsistent erklären.

Da einerseits eine Nachinterpretierung aller bisher erfolgter Operationen immens teuer und das Vorgehen auch nicht sonderlich optimistisch wäre, definieren wir den Zustand eines TBN als **für alle Zeiten gültig**. Wir folgern daraus, daß auch beim Eintragen von Beziehungen die Validität des Netzes bezüglich der derzeitigen Transaktionsbeziehungen überprüft werden muß. Im obigen Beispiel müßte also das Eintragen der Beziehung abgelehnt werden, wenn daraus eine Inkonsistenz resultieren sollte.

Wesentlich ist jedoch, daß das TBN dynamisch in Form und Bedeutung ist. Es kann zu jedem Zeitpunkt erweitert werden. Neue Transaktionen und Beziehungen können eingetragen werden, je nach dem, was die entsprechende Situation bzw. die in die Applikationen implementierten Algorithmen erfordern.

Die Reihenfolge im Vorgehen bei der Arbeit mit TBN aus Sicht der Applikation ist in der Regel die Festlegung einer Struktur von Transaktionen, z.B. entsprechend existierender Transaktionsmodelle mittels *struktureller Beziehungen*. Während der Abarbeitung der „*Portion of Work*" kommen dann, z.B. durch Synchronisation oder Parallelität, weitere Beziehungen, sogenannte *dynamische Beziehungen* hinzu. Lösen wir uns von der Vorstellung irgendwelcher festgelegter Modelle, kann die Applikation die jeweiligen Transaktionen in Beziehung zueinander setzen, entsprechend der Strukturierung ihrer Daten und Algorithmen. In einem nächsten Schritt lösen wir uns von der fertigen, festgelegten Applikation und verwenden den Begriff *Entwicklung* oder *Vorgang*, in dessen Verlauf erst aufgrund verschiedener Ergebnisse die Notwendigkeit der jeweiligen Teilarbeiten, Transaktionen und Beziehungen offenbar wird (diesen Gedanken verfolgen wir bei der Beschreibung des ConTract-Modells weiter).

In jedem Fall sollte deutlich sein, daß in diesem deklarativen Modell zunächst die Beziehungen beschrieben werden sollen, um dann während der Ausführung mit dynamischen Beziehungen angereichert zu werden, beide Beziehungsarten vereint in einer Darstellung.

4.3 Realisierung

Wir haben bisher Eigenschaften der TBN beschrieben und Anforderungen an die TBN-Verwaltung hergeleitet. Als nächstes soll untersucht werden, inwieweit und durch welche Methoden sich die gesteckten Ziele erreichen lassen. Dabei wird nicht die gesamte Bandbreite der Möglichkeiten behandelt werden können; es sollen nur rudimentäre Mechanismen, die sich für eine Realisierung als evident herausgestellt haben, aufgeführt werden.

4.3.1 Speicherung der TBN

Die TBN bestehen aus einer Reihe von Informationen über Transaktionen, deren Zustände und Beziehungen. Hinzu kommen Informationen, die zum einen aus Sicht der TBN-Verwaltung, zum anderen aus der Integration in eine reale, verteilte Umgebung resultieren.

Was aber macht denn nun ein spezifisches TBN aus? Vorrangig zu erwähnen bleiben natürlich die Transaktionen an sich. Zu ihrer Unterscheidung dient eine global eindeutige Kennung (GTRID, siehe 8.1), die auch Adreßinformationen für die Koordination verteilter Transaktionen beinhaltet. Als nächstes besitzt jede Transaktion einen Zustand, wie z.B. running oder finished, und eventuell eine für sie existierende Forderung, z.B. Demand Commit. Um die Transaktionssemantik zu implementieren, sind der Transaktion die involvierten Resource Manager und andere Komponenten zugeordnet. Hinzu kommen Informationen, wie z.B. Zugriffsrechte, Zeitpunkt der Generierung, und andere.

Weitere markante Elemente eines TBN sind selbstverständlich die Beziehungen zwischen den Transaktionen, respektive die Events für einzelne Transaktionen.

Darüber hinaus bietet es sich aus Gründen der Verläßlichkeit an, Informationen über die Verfügbarkeit verschiedener im System verteilter Komponenten zu speichern, um zum Beispiel im Rahmen der Commit-Protokolle geeignet auf Fehlerfälle reagieren zu können. Ob diese explizit im Bereich der TBN-Verwaltung zu hinterlegen sind oder auf Anfrage vom Betriebs– oder Kommunikationssystem zur Verfügung gestellt werden sollen, ist an dieser Stelle zweitrangig.

Wie später noch gezeigt werden soll, besteht eine Optimierung bei der Abwicklung der Übergangs– und Commit-Protokolle in der Bündelung von Informationen in spezifische Nachrichten. So verlagern wir die Information, welche Resource Manager an einer Transaktion beteiligt sind, zum jeweiligen TBN-Koordinator. Bei der TBN-Verwaltung wird für eine Transaktion statt der beteiligten Komponenten lediglich vermerkt, welche Knoten des Systems involviert sind.

An dieser Stelle taucht jedoch die Frage auf, welche Informationen aus dem TBN erfragt werden bzw. wie die Informationen zu speichern sind, um effiziente *Zugriffspfade* zu realisieren. Greifen wir zunächst auf die TBN-Operationen zurück, so können wir folgende Zugriffe entdecken: Bei einem Übergangswunsch für eine Transaktion müssen Informationen über die Transaktion und ihre weiteren Daten (Status, beteiligte Komponenten/Knoten) zur Verfügung stehen. Zur Entscheidung über die Durchführbarkeit des Übergangs muß die Umgebung, sprich die Beziehungen der Transaktion untersucht werden. Dazu müssen also alle Beziehungen und Events dieser Transaktion ermittelt werden.

Identifikation für eine Transaktion ist ihre GTRID, weshalb sich eine Speicherung von Transaktionsinformationen und zugehörigen Beziehungen und Events über diesen 'Schlüssel' anbietet. Ähnliches gilt für die Operationen zur Modifikation von TBN.

Desweiteren müssen Zugriffe über Komponentenkennungen auf Transaktionen effizient möglich sein. Meldet das Betriebssystem zum Beispiel den Programmfehler eines Resource Managers, muß ermittelt werden, in welche Transaktionen dieser involviert war. Dabei scheint es nicht sinnvoll, alle Transaktionen anzufassen.

Wie schon beschrieben, sind die Auswirkungen von Beziehungen sowohl aggregiert als auch transitiv zu berücksichtigen. Die Aggregation war die Und-Verknüpfung von Beziehungen zwischen zwei bestimmten Transaktionen; die Transitivität benötigten wir, um die Iteration der Ereignisse durch das Netz abzudecken und koordiniert entscheiden zu können. Das Fortpflanzen von Beziehungswirkungen nennen wir *Propagation* [Sch91]. Da nun aber bei jeder Operation auf das TBN zurückgegriffen werden muß, um die Konsistenzerhaltung zu prüfen, müssen diese Informationen berechnet vorliegen. Der Ansatz, die *transitive Hülle der Beziehungen*, also jede aggregierte und transitiv resultierende Beziehung zwischen allen Transaktionen explizit zu speichern (im allgemeinen als *Masche* bezeichnet), bringt zwei entscheidende Vorteile: Zum einen müssen diese Informationen nicht mehrfach berechnet werden, zum anderen führt die Aggregation auch der transitiven Beziehungen zu einer Fokussierung in bestimmten Knotenpunkten. Bei Berechnungen der transitiven Auswirkungen wird anhand der Beziehungen im Netz „entlang berechnet", also die transitive Hülle iterativ gebildet. Sind hingegen aber alle Informationen zwischen allen Transaktionen schon zu Beginn der Berechnung bekannt, kann die Zahl der Iterationen vermindert und außerdem eine etwaige Inkonsistenz frühzeitig in Knotenpunkten[2] erkannt werden.

Zwei weitere Anforderungen sollen noch beleuchtet werden: die „Übermittlung ganzer Netze", relevant im Zusammenhang mit der Koordination/Delegation im Bereich verteilter TBN, und die „Abbildung in Primär– und Sekundärspeicher", wie wir sie für temporäre Sicherungskopien oder im Zusammenhang mit der Recovery benötigen. In beiden Fällen ist das Netz als ganzes und vor allem in „kompakter Form" manipuliert. Eine iterative Durchwanderung entlang der Beziehungen zur Bildung einer Kopie im Hauptspeicher ist hierbei aus Performanzgründen nicht vertretbar, weshalb besonderes Augenmerk auch auf die Repräsentation von TBN gelegt werden muß.

4.3.2 Betrachtung der TBN-Eigenschaften

Den Operationen auf TBN hatten wir gewisse Eigenschaften zugeschrieben, zu deren Realisierung im folgenden einige Anmerkungen erfolgen sollen.

Wir hatten definiert, daß Operationen auf TBN unter anderem atomar und konsistenzerhaltend sind. Vor Durchführung einer Operation heißt uns dies, die Konsistenz des

[2] Ein solcher Knotenpunkt wäre z.B. eine Transaktion, die zwei TBN verbindet.

potentiell neuen TBN zu prüfen und gegebenenfalls die Operation, wenn möglich, abzulehnen; Anmerkungen zur Fehlerbehandlung folgen. Desweiteren muß die Operation als ganzes erfolgen. Diese im Normalfall triviale Erkenntnis erlangt im vorliegenden Modell jedoch entscheidendes Gewicht. Da wir Auswirkungen auf Transaktionen im Netz explizit von bestimmten Ereignissen abhängig machen, führen eben einfache Operationen zu einer Reihe von weiteren Operationen — insgesamt transitiv, wie schon gezeigt.

Somit ist wohl der erste Ansatz im Design zur Realisierung von TBN-Operationen leicht nachzuvollziehen: Bevor eine Berechnung auf dem TBN stattfindet, wird zunächst eine *lokale Sicherungskopie* angelegt, auf die im Fehlerfall 'zurückgesetzt' werden kann.

Anmerkung:

> In Anlehung an [Sch91] werden die Begriffe *future* und *current* verwendet. 'Future' bezeichnet dabei das potentiell neue TBN, auf dem die zur Durchführung der Operation nötigen Berechnungen erfolgen und in dem die neuen Daten gespeichert werden, 'Current' stellt das vor Durchführung der Operation gesicherte TBN dar.

Die Berechnungen im TBN sind nun der Versuch, ein eindeutiges, konsistentes TBN durch Ausführung der gewünschten Operation zu ermitteln. Dazu wird faktisch davon ausgegangen, die Operation sei erfolgt; anschließend werden die (transitiven) Auswirkungen berechnet. Besteht bezüglich einer Transaktion nun eine eindeutige Restriktion, das heißt, gibt es für sie eine eindeutige Aussage über einen Zustand, den sie erreichen muß, um das TBN konsistent zu erhalten, so ist diese Restriktion eine Voraussetzung (*Demand*) für die Durchführung der Operation. Da beliebig viele Transaktionen in einem TBN verzeichnet sein können, kann es natürlich ebensoviele Voraussetzungen geben, wobei die Durchführung der Operation in diesem Fall von der Erfüllung **aller** Voraussetzungen abhängt. In diesem Sinne, muß die Atomarität der Operation auf die Erfüllung aller Folgeoperationen erweitert werden, um die Frage der Ganzheitlichkeit zu beantworten. Dabei ist Unsicherheit mit Ablehnung zu beantworten; ist für eine Voraussetzung nicht in eindeutiger Weise ihre Erfüllung zu garantieren, muß in pessimistischer Strategie von ihrem Scheitern ausgegangen werden, die Operation kann nicht erfolgen.

Gerade die Problematik der Unsicherheit führt uns zu einer weiteren Entscheidung: Bei Unsicherheit wird ein Scheitern angenommen, die Operation nicht durchgeführt. Nur warum? Warum wird der bis dato berechnete Zustand nicht erhalten, bis weitere Informationen erlauben, eindeutige Aussagen zu treffen? Das hieße in unserem Fall folgendes[3]: Das future TBN müßte „offen gehalten" werden. Da neue Informationen aber nur durch Operationen in ein TBN gelangen können, müßten nun mehrere Operationen im Rahmen einer Neuberechnung eines TBN erfolgen dürfen. In anderen Worten: Mehrere Operationen würden zu einer atomaren Berechnung zusammengefaßt.

Nur stellt uns eben dies vor unlösbare Probleme: Wir können nicht erkennen, welche Operationen zusammen eine atomare Einheit bilden sollen. Wird ein TBN offen gehalten, weil eine Operation in Unsicherheit verblieb, werden alle weiteren Operationen auf dem temporären Netz ausgeführt; wir wollen ja mit einem Erfolg der ursprünglichen Operation rechnen. Außerdem können weitere Operationen nicht auf dem current TBN berechnet werden, da wir ja eben die durch sie neu eingebrachten Informationen zur Lösung der ersten

[3] Nennen wir Operationen mit solch unsicherem Ergebnis *gefährliche Operation*, dir nicht zwangsläufig zur Inkonsistenz des TBN führen müssen, aber können.

(unsicheren) Operation hinzuziehen wollten. Ein weiteres Problem ist die Rücksetzung. Wird nämlich auf ein Scheitern der ersten Operation erkannt, nachdem weitere Operationen in das TBN eingegangen waren, werden auch diese zurückgesetzt. Da wir jedoch bei Applikationen und Resource Managern keine Funktionalität der 'Zurücknahme' kennen, werden Operations-Aufträge synchron abgewickelt und erst bei Erfolg oder Scheitern eine entsprechende Antwort erzeugt. Solange ein TBN offen ist, kann noch keine Aussage getroffen werden und keine Bestätigung erfolgen.

Beispiel 4.2: *Parallele Operationen auf TBN: nicht lösbar?*

An die TBN-Verwaltung ergeht für das in Abbildung 4.2 dargestellte TBN der Auftrag Commit(T_2). Aufgrund der berechneten Voraussetzungen kann diesem nicht

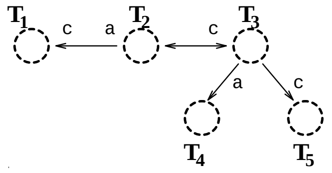


Abbildung 4.2: Mehrere Operationen in einer Einheit? (1)

sofort stattgegeben werden, das Commit von T_3 kann ja nicht garantiert werden. Das TBN bleibt 'offen', der Auftrag wird suspendiert. Der nächste Auftrag betreffe das Eintragen einer Abort-Beziehung zwischen T_4 und T_5, welchem, aufgrund der Topologie des Netzes, stattgegeben wird: die Beziehung wird (im temporären Netz)

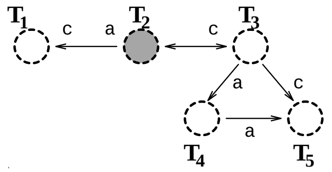


Abbildung 4.3: Mehrere Operationen in einer Einheit? (2)

eingetragen (Abbildung 4.3). Schließlich trifft ein Auftrag ein, z.B. Abort(T_3), der die erste Operation undurchführbar macht, sie also scheitern läßt. Es muß auf den Zustand des TBN vor der ersten Operation zurückgesetzt werden, nur dann ist auch das Eintragen der Beziehung gescheitert. Würde das Eintragen der Beziehung erneut gefordert, ohne daß das TBN offen ist, wäre dies wahrscheinlich erfolgreich; ein nicht ganz nachvollziehbares Verhalten aus Sicht der Applikation.

Das Problem verschärft sich noch: Der zweite Transaktionsübergang kann als Abort

nicht abgelehnt werden. Infolge der zwischenzeitlich eingetragenen Beziehung zwischen T_4 und T_5 wird der Abbruch der Transaktion T_5 berechnet, die, würden wir im ursprünglichen TBN rechnen, nicht betroffen wäre, eben weil jene Beziehung hier nicht existierte.

Zusammenfassend betrachten wir Unsicherheit als Scheitern; Operationen werden im oben beschriebenen Sinne ganzheitlich durchgeführt oder abgelehnt. Mehrere Operationen können **nicht** zu einer atomaren Einheit zusammengefaßt werden.

Anmerkung:

Ein weiterer Ansatz wäre, gefährliche Operationen durchzuführen, die Liste der Voraussetzungen teilweise unerfüllt zu belassen. Wir nehmen damit das Risiko in kauf, daß zu einem späteren Zeitpunkt eine Inkonsistenz dadurch auftritt, daß ein einer Voraussetzung entgegengesetztes Ereignis eintritt. In diesem Fall verlagern wir die Problematik in den Bereich der Fehlerbehandlung von inkonsistenten TBN.

Hiermit erreichen wir, quasi als Nebenprodukt, auch die Isolation der Operationen, da wir sie ja serialisieren müssen. In gleichem Atemzug soll auch die Dauerhaftigkeit von TBN-Operationen erfaßt werden. Ist also eine Operation durchgeführt worden, hat die TBN-Verwaltung zu garantieren, daß dies, lapidar gesagt, auch so bleibt; die Wirkungen einer Operation bleibt bestehen und kann nur durch Wirkungen anderer Operationen beeinflußt werden. Besonders im Fehlerfall, also wenn z.B. ein Systemfehler auftritt, muß nach einem Wiederanlauf der zuletzt gültige, konsistente Zustand des TBN restauriert werden. Die eventuell zu diesem Zeitpunkt berechnete Operation ist, sofern sie noch nicht abgeschlossen war, abzubrechen. Da wir die temporären Berechnungen nicht stabil machen, erfolgt letzteres nahezu automatisch, lediglich im Log muß der Erfolg einer Operation hinterlegt sein und es muß dafür gesorgt werden, daß die „angefangene Operation", respektive deren Auftrag, sich noch in der Warteschlange der eingehenden Nachrichten befindet.

Für das stabil Machen von Informationen müssen wir jedoch zwei Ebenen unterscheiden: Das TBN und seine Operationen an sich sowie die darauf ablaufenden Transaktionsprotokolle (*Übergangsprotokoll* und *2-Phase-Commit*). Generell könnte das TBN entweder physisch gesichert (*Image*) oder aus den Operationen sukzessive rekonstruiert werden. Gehen wir jedoch davon aus, daß im übrigen System die Transaktionen ihren Zustand weiterhin innehaben, würde die Rekonstruktion aus Operationen heraus analog dem Reihenfolgeproblem auf gewisse Schwierigkeiten stoßen. Es kann also nur ein TBN_j aus einem TBN_i durch erneute Ausführung der Operationen (entsprechend *Redo*) rekonstruiert werden, wenn dies unabhängig von den Transaktionszustandsänderungen, die während der ursprünglichen Operationensequenz stattfanden, geschehen kann. Andernfalls muß das komplette Netz gesichert bzw. geladen werden. Da die Abhängigkeit der Operationen von den Transaktionszuständen aber erheblich ist, wäre ein häufiges Sichern der gesamten TBN-Informationen die Folge, der Aufwand wohl nicht gerechtfertigt. Deshalb wird vorgeschlagen, keine „echte Rekonstruktion" durchzuführen, sondern nur die jeweils neue Netz-Struktur aus einer alten anhand eines Struktur-Logs zu erzeugen. Daß heißt, für jede Operation muß ermittelt werden, wie sich die Struktur des Netzes verändert hat, und genau diese Information ist zu protokollieren. Unter der *Strukturänderung* des TBN verstehen wir sowohl des Einfügen und Löschen von Beziehungen, wie auch die Veränderung von anderen TBN-Informationen, wie Transaktionszuständen, Auslösen von Trigger und so weiter.

Beim Wiederanlauf der TBN-Verwaltung und Rekonstruktion des TBN wird die zuletzt erfolgte Sicherung der TBN-Informationen herangezogen und sukzessive um Informationen aus Operations-Log und Struktur-Log ergänzt. Hieß eine Operation die TBN-Verwaltung zum Beispiel, eine Transaktion in den Commit-Zustand zu versetzen, so muß beim *Redo* zum einen der neue Transaktionszustand vermerkt werden[4], zum anderen muß das Netz in seiner Struktur entsprechend angepaßt werden, als ob die Operation durchgeführt worden wäre, inklusive aller Folgeänderungen, wie zum Beispiel Zustandsänderungen anderer Transaktionen.

Es bleibt noch zu vermerken, daß der Eintrag für Strukturänderung vor dem Eintrag für die erfolgreiche Beendigung der Operation im Log steht; beim Redo wird die Strukturänderung nur dann ausgeführt, wenn ein zugehöriger Operationseintrag existiert. Außerdem muß zur Operation die Kennung der Nachricht, die Auftrag für diese Operation war, mitgesichert werden. Wird nämlich nach der Recovery festgestellt, daß die erste Nachricht in der Warteschlange der Aufträge dieselbe Kennung trägt, wie der letzte Eintrag im Log, dann ist der Systemfehler zwischen dem Schreiben des Log-Eintrags und dem Entfernen der Nachricht aus der Warteschlange aufgetreten; in diesem Fall ist die Nachricht zu entfernen und zu ignorieren.

Zu bemerken bleibt schlußendlich, daß in dem beschriebenen Ansatz zum Zeitpunkt der Berechnung lediglich ein gültiges, gesichertes und ein temporäres, neues TBN existiert. Explizite Zwischensicherungen (*Savepoints*) sind nicht vorgesehen. Deshalb müssen gesicherte Images nicht unterschieden (versioniert) werden.

[4]In der Sicherung steht ja noch der Zustand der Transaktion **vor** dem Commit.

Kapitel 5

Ein Übergangsprotokoll auf Beziehungen

Das Übergangsprotokoll ist nunmehr der Part, der, nachdem die Anwendung das Transaktionsszenario definiert hat, das Netz geeignet interpretiert, um die Arbeit im Sinne der Applikation und der Transaktionssemantik koordiniert ablaufen zu lassen.

Im folgenden soll im Ansatz ein Übergangsprotokoll eingeführt werden. Es werden dabei anhand von Beispielen die jeweils notwendigen Kriterien und Schritte aufgezeigt werden. Die Form des Protokolls, die wir am Ende ereichen werden, ist ganz sicher weder vollständig noch endgültig. Dazu sind noch weitere Arbeiten erfolderlich. Wir werden denn vornehmlich auch nur die logischen Beziehungen betrachten und nachweisen, daß diese allein nicht ausreichend sind, und deshalb Events im Rahmen der ConTract–Abwicklung noch hinzunehmen. Die zeitlichen Beziehungen und Trigger bleiben jedoch vorläufig außen vor.

Zu Beginn des Kapitels werden zunächst einige Begriffe festgelegt. Anschließend werden verschiedene Kriterien für das Protokoll eingeführt. Am Ende des Abschnitts werden die Erkenntnisse, Anforderungen und Probleme nochmals zusammengefaßt.

5.1 Grundlagen

Definition 5.1: *Übergangsprotokoll*

> *Wir definieren das „Übergangsprotokoll auf Transaktionsbeziehungen“ als eine Instanz, die die Abfolge von Transaktionsübergängen in solcher Weise koordiniert, daß die Semantik des jeweiligen Netzes erfüllt ist, keine Inkonsistenzen im Netz auftreten und keine widersprüchlichen Forderungen bezüglich des Beendens an eine Transaktion existieren.*

Das Übergangsprotokoll auf Transaktionen berechnet also zu einem *Übergangswunsch*[1] die dafür existierenden (transitiven) Prädikate und versucht diese zu erfüllen. Je nach Konstellation ist der Übergang durchgeführt, befinden sich auch andere Transaktionen in einem

[1]Ein Übergangswunsch ist die Forderung an die Transaktionsverwaltung, eine Transaktion in einen bestimmten Zustand zu überführen. Solch ein Wunsch kann z.B. durch eine Applikation oder einen Resource Manager gestellt werden.

neuen Zustand, werden die definierten Events nach außen hin gemeldet. Oder der Übergang wird abgelehnt, im schlimmsten Fall ist das Netz inkonsistent.

Transaktionssemantiken

Wir müssen im folgenden, um die Übersicht zu wahren, zwischen verschiedenen Sichten der Transaktionssemantik unterscheiden.

Eine Anwendung spricht bezüglich der Transaktion von *begin-of-transaction* und *end-of-transaction* und bezeichnet hiermit gegenüber der Transaktionsverwaltung Beginn und Ende einer Transaktion.

Ein Resource-Manager nimmt an einer Transaktion teil, und kann nach Beendigung seiner Aufgaben die Ergebnisse in einen Endzustand bringen. Der Endzustand Commit bedeutet, daß die Ergebnisse stabil und sichtbar werden, bei Abort werden die Werte zurückgesetzt. Um mehrere Resource-Manager koordiniert zu einem Ende zu führen, stellen diese eine 'Zwischen-Zustand' (prepared) zur Verfügung, von dem sie garantieren, daß sie das Commit und Abort erreichen können und bei einer Recovery auch diesen Prepare-Zustand wieder erreichen.

Die Transaktionsverwaltung dazwischen muß die eine in die andere Sicht überführen, zum Beispiel eine Commit-Entscheidung treffen und durchsetzen, und jeweils eine Schnittstelle zur Verfügung stellen. Die Zustände, die wir innerhalb der Transaktionsverwaltung für eine Transaktion kennen, sind installed, running, finished, committed und aborted. Wir vermeiden somit Namenskonflikte innerhalb von Beschreibungen, die sich auf mehrere der erwähnten Sichten oder Ebenen beziehen. Die Abbildung 5.1 stellt dies dar. Eine Unterscheidung ist

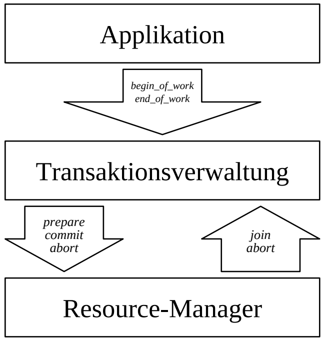


Abbildung 5.1: Verschiedene Sichten von Transaktionssemantik

deshalb zu treffen, da wir im Rahmen der Verwaltung auf Transaktionsbeziehungsnetzen (zunächst) die Transaktionen an sich, nicht die daran partizipierenden Resource Manager koordinieren. In diesem Sinne werden dann zum Beispiel auch ganze Transaktionen verzögert.

Wir koordinieren somit Transaktionen zu einem konsistenten TBN und Resource Manager zu wiederum ACID–Transaktionen.

5.2 Beziehungen im Übergangsprotokoll

Logische Beziehungen stellen Einschränkungen in den möglichen Abfolgen von Transaktionsübergängen dar. Diese Einschränkungen realisiert das Übergangsprotokoll, indem es zu einem Übergangswunsch entscheidet, ob dieser generell möglich ist — z.B. ist ein BOT nach einem Commit wohl kaum für ein und dieselbe Transaktion möglich —, unter welchen Prämissen und welche weitere Restriktionen sich aus diesem Übergang ergeben.

Beispiel 5.1:

Betrachten wir die Beziehung: $T_1 \leftrightarrow_c T_2$. In Worten bedeuted dies:

> *T_1 erreicht genau dann den Zustand* Commit*, wenn auch T_2 den Zustand* Commit *erreicht.*

Gehen wir davon aus, daß beide Transaktionen aktiv (d.h. running) sind. Wird dann für Transaktion T_1 das EOT gefordert, müssen folgende Betrachtungen angestellt werden:

1. Der Übergang stellt keinen Widerspruch zum Zustand der Transaktion dar. Ein Commit für einen Zustand running ist legitim.
2. Der Übergang stellt auch keinen Widerspruch zu irgendeiner Forderung bezüglich dieser Transaktion dar, es existiert nämlich im Moment keine.
3. Der Übergang führt zu keiner Netz-Inkonsistenz.

Wir können also eine *Commit-Betrachtung* durchführen.

Werten wir nun die Beziehung zwischen den Transaktionen aus, so hieße ein Commit von T_1, daß T_2 in jedem Fall auch den Zustand Commit erreichen muß, sonst wäre die Netz-Bedeutung verletzt. Nun ist aber offensichtlich, daß über den Ausgang der Transaktion T_2 zum jetzigen Zeitpunkt keine Aussage getroffen werden kann, sie kann ja noch abbrechen.

In Anlehnung an das 2-Phase-Commit müssen wir somit T_1 in einen Zustand versetzen, aus dem heraus der ursprüngliche noch erreicht werden kann und weiterhin das Netz erfüllbar bleibt, unabhängig vom Ausgang von T_2. Die Transaktionsverwaltung holt sich gewissermaßen die *Votierung der Transaktion* T_1 für das anstehende Commit ein. Eine Zustimmung bedeutet, daß die Transaktion das Commit genauso **garantieren** kann, wie das Abort. Der Unterschied zum 2-Phase-Commit liegt darin, daß wir die Votierung schon einholen, bevor alle beteiligten Komponenten (hier Transaktionen statt Resource Manager) zu einer Vortierung bereit sind (T_2 ist ja noch aktiv). Wir nützen somit die *Garantie-Eigenschaft* des Zustandes prepared über einen längeren Zeitraum und **über Transaktionsgrenzen hinweg**, d.h. , nach der Votierung erfolgt nicht das sofortige Commit der Transaktion.

Wir versetzen also T_1 in den Zustand prepared (oder in unserer Terminologie finished). Irgendwann stellt nun T_2 ihrerseits einen Übergangswunsch. Gehen wir vom Idealfall aus, was EOT bedeuten würde. Es gelten analog die Betrachtungen von oben. Das Commit von T_2 setzt in gleicher Weise das Commit von T_1 voraus. Dieses kann hingegen — T_1 ist immer noch prepared — garantiert werden, auch spricht nichts im Netz gegen diese Übergänge (z.B. durch weitere Beziehungen). Somit kann T_2 in den Zustand Commit versetzt werden, und bei Erfolg auch T_1. Die Beziehung zwischen T_1 und T_2 ist somit *erfüllt*.

Fordert T_2 hingegen das Abort, entsteht für T_1 die Forderung, ebenfalls abzubrechen. Nun, ein Abort aus einem Zustand prepared heraus ist ebenfalls möglich. Und da wir den Abbruch von T_2 wohl kaum verhindern können, ist das Netz auf diese Weise zu erfüllen.

Zustand einer Beziehung

Für eine Beziehung wollen wir unterscheiden, in welchem Zustand sie sich befindet, um den Einfluß auf das jeweilige Protokoll prägnant ausdrücken zu können. Demnach bezeichnen wir eine Beziehung als *eingetragen*, wenn sie im Netz bekannt ist, aber noch kein Ereignis zu einer Berücksichtigung innerhalb des Protokollablaufs führte. Wurde sie hingegen schon einmal ausgewertet, und muß sie auch bei weiteren Berechnungen noch herangezogen werden, so nennen wir sie *aktiv* (in obigem Beispiel also nach dem Commit-Wunsch von T_1). Eine Beziehung gilt als *erfüllt* oder *final*, wenn sie in Zukunft nicht mehr berücksichtigt werden muß, in diesem Fall also wenn beide Transaktionen den gleichen Endzustand (Commit oder Abort) erreicht haben. Entsprechend heißt eine Beziehung *unerfüllbar*, wenn aus ihrer Auswertung heraus Forderungen erwachsen, die nie mehr erfüllt werden können.

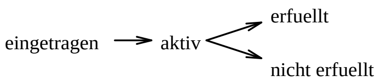


Abbildung 5.2: Zustände einer Beziehung

5.3 Ein erster Ansatz

Die im Beispiel aufgetretenen Kriterien für ein Übergangsprotokoll wollen wir noch einmal zusammenfassen.

1. Kriterium: *Widerspruch zum eigenen Zustand*

Für eine Transaktion kann kein Übergang gefordert werden, der für sie nicht erfüllbar ist. Ein Abbruch einer Transaktion kann immer erfüllt, weil nicht verhindert werden. Abbildung 5.3 veranschaulicht dies.

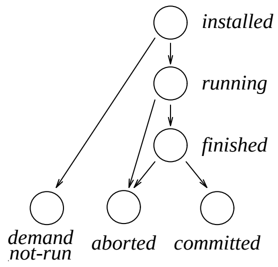


Abbildung 5.3: Der Zustandsautomat für Transaktionen

Definition 5.2: *Demand und Agenda of Demands (AoD)*

Ein Transaktionsübergang einer Transaktion kann, abhängig von den eingetragenen Beziehungen und den Zuständen der beteiligten Transaktionen, Forderungen bezüglich Transaktionsübergängen anderer Transaktionen implizieren. Diese Forderungen nennen wir 'Demands'. Eine Liste von Demands heißt „Agenda of Demands".

Ein Demand kann folgende Ausprägungen besitzen: Demand Run, Demand Commit, Demand Abort *und* Demand not Run[2].

Definition 5.3: *Erreichbarer Transaktionszustand*

Wir nennen einen Transaktionszustand für eine Transaktion 'erreichbar', wenn dieser vom augenblicklichen Zustand der Transaktion und unabhängig von irgendwelchen Beziehungen des Netzes als nächster Zustand möglich ist.

Nachstehende Tabelle definiert die Erreichbarkeit:

Erreichbarer	aktueller Transaktionszustand				
Demand	Installed	Running	Finished	Committed	Aborted
Run	ja	–	–	–	–
Commit	–	–	ja	–	–
Abort	ja	ja	ja	–	–
Not-Run	ja	–	–	–	(ja)

Wir benötigen dies später im Übergangsprotokoll als eine der Abbruchbedingungen.

Definition 5.4: *Nicht erreichter Demand*

Die Einträge in einer Agenda of Demands, deren Forderung bezüglich des Transaktionszustandes noch nicht erfüllt sind, bezeichnen wir als „nicht-erreicht". Dabei werden nur die möglichen neuen Zustände berücksichtigt.

[2]Dies stellt den Quasiabbruch einer Transaktion dar, die noch nicht begonnen wurde [Sch91].

Auch dies spielt im Zusammenhang mit Abbruchbedingungen eine Rolle. Die AoD wird nämlich im Rahmen des Übergangsprotokoll iterativ bearbeitet, denn aus einem jeweiligen Übergang können sich neue Konstellationen ergeben, die weitere Übergänge ermöglichen oder fordern. Wenn also z.B. alle Forderungen innerhalb der AoD nicht mehr *erreichbar* sind, kann die Bearbeitung an dieser Stelle der Commit-Betrachtung abgebrochen werden.

Wir können somit ein weiteres Kriterium einführen:

2. Kriterium: *Atomarität der Erfüllung der Agenda of Demands*

> Die aus einem Übergang resultierenden Demands stellen Prämissen für die Durchführung eben dieses Übergangs dar. Kann die Agenda gänzlich erfüllt werden, d.h., können alle Forderungen erfüllt werden, so ist der ursprüngliche Übergang ungefährdet durchführbar. Andernfalls ist er zu verzögern oder abzulehnen.

Es müssen natürlich noch ein paar Dinge zur Berechnung der Agenda erklärt werden[3]. Wie wir im Beispiel gesehen haben, impliziert ein Commit von T_1 faktisch das Commit von T_2. Genauer gesagt, gehen wir für die weitere Demand-Berechnung explizit davon aus, dem Übergangswunsch sei stattgegeben worden, die Transaktion befinde sich im neuen Zustand. Dies müssen wir an jeder Stelle der Berechnung auch, denn: wird dem Übergang stattgegeben, so muß das „neue Netz" eben die neue Zustandsmenge aller Transaktionen repräsentieren, und dies selbstverständlich konsistent.

Die möglichen nächsten Zustände für T_2 sind dann auf einen Zustand, nämlich das Commit, reduziert.

Nehmen wir an, für T_2 seien weitere Beziehungen zu noch anderen Transaktionen im Netz verzeichnet. Dann müssen wir davon ausgehen, daß die Restriktion bezüglich T_2 (der Demand Commit) auch dort zu Einschränkungen führt. Diese müssen schon zum jetzigen Zeitpunkt berücksichtigt werden. Würden wir dies nicht tun, könnten wir verfrüht einem Übergangswunsch stattgeben, dieser löst in der Folge dann weitere Übergangswünsche (genau die nicht berücksichtigten, transitiven Demands) aus, welche zu Widersprüchen führen könnten. Den ursprünglichen Übergang können wir an dieser Stelle jedoch nicht mehr zurücknehmen! Ergo muß die Agenda of Demands iterativ *transitiv* berechnet werden[4]. Dabei können an bestimmten Stellen mehrere Möglichkeiten für eine Transaktion bestehen. Offensichtlich sollte in solchen Fällen optimistisch vom Commit der Transaktion ausgegangen werden.

Im Laufe der Berechnungen können, zum Beispiel bei Zyklen, widersprüchliche Demands für eine Transaktion berechnet werden. An dieser Stelle muß dann ein *Rücksetzmechanismus* dafür sorgen, daß an einer Stelle mit ursprünglich mehreren Möglichkeiten entsprechend neu aufgesetzt wird, nun aber mit anderer Strategie (z.B. Abort). Kann der Widerspruch dennoch nicht verhindert werden, stellt dies aber eine *Inkonsistenz* des Netzes bezüglich des ursprünglichen Übergangswunsches dar, weshalb diesem nicht stattgegeben werden kann.

Kehren wir jedoch zum Übergangsprotokoll zurück. Wir hatten gefordert, daß eine AoD erfüllt sein muß, bevor der Übergangswunsch durchgeführt werden kann. Was heißt dies aber genau?

[3] Für eine genaue Darstellung der Algorithmen wird erneut auf [Sch91] verwiesen.
[4] zum Beispiel nach dem Verfahren der Delta-Iteration.

Betrachten wir nochmals unser Beispiel. Der Commit-Wunsch von T_1 erzeugt die Agenda: Demand Commit T_1[5], Demand Commit T_2. Nun muß für diese AoD ermittelt werden, ob sie erfüllt ist oder nicht. Offensichtlich ist weder der erste noch der zweite Demand erfüllt, beide Transaktionen haben ja noch den Zustand running. Somit erhalten wir für die Anzahl der nicht erfüllten Demands den Wert 2. Was jedoch ist erreichbar? Das Commit von T_1 wird als erreichbar betrachtet, er wurde ja ursprünglich gefordert, was einem impliziten *Vote 'yes'* entspricht. Das Commit von T_2 hingegen ist nicht (sicher) erreichbar, wir haben darüber keine Aussage. Somit ist die Zahl der erreichbaren Zustände, nämlich 1, kleiner als die der nicht erfüllten, die AoD ist entsprechend nicht erfüllt. Wie bereits geschildert, verbleibt als einzige Möglichkeit, T_1 und ihren Commit-Wunsch zu verzögern, einen prepare_to_commit zu initiieren.

Im weiteren Verlauf des Beispiels fordert schließlich T_2 seinen Commit. Auch hier erhalten wir die AoD: Demand Commit T_1, Demand Commit T_2. Im Gegensatz zur ersten Berechnung jedoch haben wir nun für die Zahl der erreichbaren Transaktionszustände die Zahl 2, da den Zustand Commit von T_1 garantiert werden kann. Ergo können beide Transaktionen in das Commit überführt werden können.

Anmerkung:

> Das Commit von T_2 könnte theoretisch noch scheitern; es war bis dato lediglich das Commit gefordert, jedoch dieses noch nicht durch ein prepare-to-commit garantiert. Wäre T_1 bereits zuvor committed worden und würde T_2 noch abbrechen, wäre dies ein Widerspruch. Deshalb müssen wir das Verfahren geringfügig modifizieren.
>
> Zum einen können wir beim Übergangswunsch das Commit aus dem running heraus auch als nicht erreichbar definieren. Wir erhalten somit in der Regel einen Iterationsschritt mehr. Im Beispiel wird beim Commit-Wunsch von T_2 erkannt, daß die AoD nicht erfüllbar ist, weil T_2 nicht aus running heraus committed werden kann. Somit ist zunächst, ganz natürlich, T_2 zu verzögern (prepare_to_commit). In einem erneuten Durchgang ist das Commit von T_2 erreichbar (und garantierbar!), sie ist ja im Zustand prepared. Somit ist die gesamte AoD erfüllbar, beide Transaktionen können beendet werden. Wir müssen jedoch das Verfahren auch insofern modifizieren, daß diese zusätzliche Iteration bei einem Netz mit nur einer Transaktion (ohne Beziehungen) **nicht durchgeführt** wird. Denn, wie wir noch sehen werden, das 2-Phase-Commit, das an einer Transaktion beteiligte Resource Manager koordiniert beendet, wird von einem *Transaktionskoordinator* durchgeführt. Dieser wird vom *TBN-Koordinator* jeweils beauftragt, eine Transaktion (unabhängig von Beziehungen) in den Zustand Prepared, Committed oder Aborted zu versetzen. Würden wir also den Algorithmus nicht weiter modifizieren, erginge an den Transaktionskoordinator der Auftrag zum Prepare und direkt im Anschluß (und bei Erfolg) der Auftrag zum Commit. Da im vorliegenden Fall aber genau diese Reihenfolge bekannt ist, kann sofort das Commit gefordert werden; der Transaktionskoordinator hat dies geeignet zu behandeln, sprich Votierungen der Resource Manager einzuholen und direkt im Anschluß das Commit durchzuführen.
>
> Auf der anderen Seite, könnten wir auch vor der Berechnung der AoD eine Transaktion, die einen Commit-Wunsch stellt, generell in den Zustand prepared versetzen. Bei der anschließenden Berechnung wäre ihr Commit somit erreichbar, ein Abort weiterhin noch möglich. Wir nehmen hiermit aber in kauf, daß wir eine einzelne Transaktion unnötig in diesen Zwischenzustand versetzen und somit überflüssige Kommunikation zwischen TBN-Koordinator und Transaktionskoordinator betreiben.

[5] den Demand für die Transaktion, die den Übergangswunsch stellt, nehmen wir o.B.d.A. mit in die AoD auf.

5.3.1 Gestatten des Übergangswunsches

Ist die gesamte AoD erfüllt, dann kann wie gesehen dem Übergangswunsch stattgegeben werden, das heißt, die Transaktion in den gewünschten Zustand versetzt werden. Durch die Erfüllung der Prämissen, können auch andere Transaktionen in einen anderen Zustand gewechselt sein.

Somit kann das neu berechnete Netz fürderhin Gültigkeit haben. Es bleibt zu vermerken, daß in der Abfolge von Netzen ein jeweils neues Netz monoton restriktiver ist als ein vorhergehendes, vorausgesetzt wir reduzieren es nicht um Beziehungen und Transaktionen. Im *finalen Netz* haben alle Transaktionen einen Endzustand erreicht, die Restriktion ist maximal (kein neuer Übergang möglich).

Was verbleibt aber zu tun, wenn einem Übergangswunsch nicht stattgegeben wird?

5.3.2 Rücksetzen des Netzes

Offensichtlich müssen wir zu dem Zustand des Netzes zurückkehren, der vor Beginn des Übergangs gegeben war. Aus dieser Rückkehr können keine Demands oder ähliches erwachsen, denn das gesicherte Netz kann maximal genauso restiktiv sein, wie das potentiell neue. Wir müssen lediglich garantieren, daß wir keine Transaktionsübergänge durchgeführt haben, die später zu Inkonsistenzen führen. Dies wäre der Fall, wenn wir T_1 in den Zustand Commit versetzt hätten, obwohl T_2 noch läuft. Ein Abort von T_2 würde dann zu einem inkonsistenten Netzes führen.

Da wir jedoch als einzigen Übergang ein Prepare durchführen, und aus diesem sowohl Commit wie auch Abort möglich ist, kann dieses Problem im geschilderten Verfahren nicht auftreten.

5.4 Erste Zusammenfassung

Wir haben bisher gesehen, daß ein Übergangswunsch nicht zum aktuellen Zustand der Transaktion in Widerspruch stehen darf und er auch nur dann sofort oder überhaupt erfüllt werden kann, wenn die Agenda of Demands vollständig erfüllt werden kann und keine Inkonsistenzen im Netz verbleiben. Andernfalls ist der Übergangswunsche zu verzögern oder abzulehnen. In diesem Fall ist das Netz auf den ursprünglichen Zustand zurückzusetzen.

Die Atomarität kann desweiteren nur gewährleistet werden, wenn die AoD transitiv berechnet und vollständig erfüllt wird. Ein einmal vollzogener Übergang ist nicht mehr zurücknehmbar, er ist *dauerhaft*.

Um die Konsistenz zu sichern, bedürfen wir der *2-Phasigkeit*. Das Commit mehrerer involvierter Transaktionen muß bei der Erfüllung einer Agenda koordiniert und in Übereinstimmung geschehen, weshalb in Anlehnung an das 2-Phase-Commit ein Zustand Prepare hinzugenommen werden muß[6]

[6]und zwar auf Transaktionsebene!

5.5 Modifikationen des Netzes

Es ist zu zeigen, daß das Eintragen von Beziehungen ebenfalls gewissen Prämissen unterliegen kann und mithin das Übergangsprotokoll in Anspruch nimmt und auch das Problem „Löschen von Beziehungen“ soll kurz angerissen werden.

Beispiel 5.2: *Eintragen neuer Beziehungen*

> Gegeben sei folgende Situation: Es seien zwei Transaktionen, T_1 und T_2, im Netz eingetragen, zunächst ohne Beziehung zueinander. T_1 sei im Zustand committed, T_2 sei derzeit noch aktiv. Wir gehen davon aus, daß die beendete Transaktion T_1 trotzdem im Netz verbleibt, z.B. weil weitere Beziehungen zu T_1 dies erforderlich machen.
>
> In dieser Situation soll nun die Beziehung $T_1 \leftrightarrow_c T_2$ eingetragen werden. Offensichtlich darf, soll diese Beziehung Gültigkeit besitzen, T_2 nur im Zustand Commit beendet werden, ein Abort wäre eine Inkonsistenz. Da das Commit für T_2 jedoch nicht garantierbar ist, muß die Beziehung abgelehnt werden. Das Eintragen der Beziehung muß nicht, kann aber das Netz in einen inkonsistenten Zustand führen, wenn T_2 nämlich abbricht. Diese Operation hatten wir als *gefährlich* bezeichnet und führen sie deshalb nicht durch.

Anmerkung:

> Eigentlich macht das Eintragen von Beziehungen zu finalen Transaktionen keinen Sinn. Jedoch nehmen finale Transaktionen durch die Propagation durchaus Einfluß auf aktive Transaktionen; das Beispiel ist der Einfachheit halber so gewählt worden.

Wir müssen also erneut auf das ursprüngliche Netz zurücksetzen. In jedem Fall kommt es zu keiner ‘Verzögerung’, denn nur eine einen Übergangswunsch stellende Transaktion kann verzögert (d.h. prepared) werden, nicht aber eine Beziehung.

Beispiel 5.3:

> Nehmen wir wieder obiges Szenario, T_1 sei allerdings abgebrochen worden. Das Eintragen der neuen Beziehung $T_1 \rightarrow_a T_2$ (was zum Beispiel als „*Leseabhängigkeit*“ interpretiert werden kann) würde sofort ein Abort von T_2 fordern, was nach der bisherigen Logik erfüllbar wäre. Wenn nun aber T_2 erst begonnen hatte, **nachdem** T_1 abgebrochen wurde, konnte es auch keine vermeintlich falschen Ergebnisse lesen, ein Abbruch von T_2 also somit hinfällig.

Wir erkennen, daß entweder das Eintragen von Beziehungen zu finalen Transaktionen gesondert definiert werden muß oder die logische Beziehung für diesen Fall nicht hinreichend genau genug definiert ist.

5.5.1 Löschen von Beziehungen

Da mit dem Eintragen von Beziehungen das TBN monoton restriktiver wird, ist beim Löschen das Gegenteil der Fall. Außerdem können durch das Austragen keine (neuen) Demands entstehen, weshalb schlußendlich das Löschen von Beziehungen immer möglich ist.

Jedoch können dabei durchaus Übergänge von Transaktionen ermöglicht werden, die bislang aufgrund eben jener Beziehungen verzögert wurden. Es muß nun entweder für jede Transaktion, die sich in einem verzögerten Zustand befindet, überprüft werden, ob ihr Übergang nun möglich ist; dies kann keine Demands betreffen, da die AoD nach Definition ja leer sein muß.

Ein Problem stellt hingegen die Realisierung der Löschoperation dar. Die Auswirkungen von Beziehungen wurden bei ihrem Eintragen aggregiert und propagiert, sprich die Informationen wurden verdichtet. Kamen zum Beispiel „neutrales Verhalten" und „direkter Einfluß" zur Deckung, so wurde daraus bei der Aggregation der „direkte Einfluß" abgeleitet [Sch91]. Beim Austragen müssen aber eben jene ursprünglichen Informationen rekonstruiert werden.

Der von Schütz vorgeschlagene Ansatz geht so vor, daß er das TBN sukzessive aus den Operationen wieder aufbaut, wobei die zu löschende Beziehung außen vor bleibt. Das hat aber zur Folge, daß zum einen alle ursprünglichen Beziehungen (nicht nur die aggregierten) gespeichert werden müssen. Zum anderen ist dieses Verfahren extrem teuer.

Hier zeigt sich, daß bei den Verfahren der Repräsentation von TBN andere Ansätze untersucht werden müssen.

Kapitel 6

TAV in der ConTract-Abwicklung

6.1 Das ConTract-Betriebsmodell

Der Zweck dieses Modells liegt darin, langlebige Aktivitäten zu modellieren und abzuwickeln. Ein ConTract ist in diesem Zusammenhang die unverwundbare und konsistente Ausführung einer (beliebigen) Sequenz von vordefinierten Aktionen, *Steps* genannt, entsprechend einem explizit spezifizierten Kontrollfluß. Ein ConTract ist mithin nicht anderes, als ein Programm mit einem einer parallelen Umgebung entsprechendem Kontrollfluß, mit persistenten Variablen, parallelem Zugriff auf gemeinsame Objekte unter applikationsorientierter Synchronisation und mit einer präzisen Fehlersemantik [WR91][WR90].

Viele der Aufgaben werden aus dem Verantwortungsbereich der Applikation in das System verlagert, wie zum Beispiel die Ausführung und Kontrolle paralleler Aktivitäten oder auch die Verteilung der Aufgaben im System.

Ein Step stellt die elementare Verarbeitungseinheit dar. Er beschreibt, je nach Modellierung, zum Beispiel einzelne Datenbankanfragen oder ganze Vorgänge, wie das Buchen eines Fluges mit Platzreservierung und so weiter. Steps werden in herkömmlichen Programmiersprachen implementiert. Sie kennen keine interne Parallelität; der Programmierer muß sich nicht um Asynchronität, Kommunikation zwischen Komponenten, Lokalität von Ressourcen, Synchronisation oder Recovery sorgen; er schreibt in gewissem Sinne „normale Applikationen".

Das Skript hingegen beschreibt die Struktur komplexer Aktivitäten mittels Kontrollfluß zwischen Steps. Diese sind somit der algorithmische Part, während Skripts die Abfolge definieren. Hier wird auch die Parallelität festgelegt, deklarativ Objekte und Transaktionen beschrieben, und so weiter. Für die Implementierung steht eine eigene Sprache zur Verfügung, die Codierung erfolgt (weitgehend) unabhängig von der der Steps.

Die atomaren Einheiten werden sukzessive ausgeführt, abhängig von Kontrollfluß und den jeweiligen Daten oder Ergebnissen. Im Falle von Fehlern, wird nach dem Prinzip der *Forward Recovery* verfahren, sprich es wird versucht, das aufgetretene Problem anhand vorab für diese Situation definierte Aktionen zu beheben. Dies ist besonders im Zusammenhang mit realen Aktionen von erheblicher Bedeutung, kann hier doch nicht, wie im klassischen Transaktionsmodell, jede Transaktion abgebrochen und deren Ergebnis rückgängig gemacht werden. Aus diesem Grund und der Persistenz wegen, wird für einen Step seine jeweilige 'Umgebung' explizit in sogenanntem *Kontext* gespeichert. Dieser umfaßt zum Beispiel Programmvariablen, Sessionparameter, Fensterinhalte der Oberfläche, Datenbank-Cursor, Dateideskriptoren, Aufrufparameter, Ergebniswerte und andere. Gleichzeitig dienen diese Informationen auch der expliziten, transaktionsübergreifenden Nutzung von Daten, zum Beispiel durch Weitergabe von Sperren oder ähnlichem. Da ein Step, beziehungsweise die ihm unterlegte Transaktion, sich nach Beendigung im Zustand Commit befindet, kann er/sie nicht rückgängig gemacht werden (Dauerhaftigkeit). Hier wird der Begriff der semantischen Atomarität oder *Kompensation* eingeführt, daß heißt die Ergebnisse eines Steps werden durch einen weiteren (ausgezeichneten) Step rückgängig gemacht, gleichbedeutend einer Invertierung.

Die Synchronisation der (globalen) Daten wird semantisch vollzogen. Ein Step kann nur die lokale Konsistenz der Daten garantieren, weswegen Prädikate oder in diesem Fall Invarianten vor bzw. nach Ausführung eines Steps überprüft werden müssen.

Gleichzeitig müssen die Steps (Transaktionen) explizit und dynamisch in Beziehung zueinander gesetzt werden können, denn Abhängigkeiten existieren während der Lebensdauer des Skripts und sind nicht auf die Dauer einer Transaktion begrenzt. So muß die Transaktionsverwaltung, die der Abwicklung von Steps in ConTracts zugrunde liegt, zunächst einmal Mittel zur Verfügung stellen, den jeweiligen Steps Transaktionen zu unterlegen. Jeder Step wird, sofern das Skript nicht Gegenteiliges definiert, unter Transaktionssemantik ausgeführt. Auch hat der Skriptprogrammierer die Möglichkeit, verschiedene Steps zu atomaren Einheiten — sprich Transaktionen — zusammenzufassen, woraus sich automatisch eine Hierarchie von Transaktionen ergibt. Hinzu kommt, daß im vorliegenden System verschiedene herkömmliche Systeme, wie zum Beispiel Datenbanksysteme, integriert und genutzt werden können sollen. So ist durchaus angedacht, daß Steps zur Verwaltung ihrer Daten SQL-Datenbanken verwenden. Die verschiedenen Systeme und deren Transaktionsmechanismen und –verwaltung in geeigneter Weise zu integrieren, stellt eine weitere Aufgabe an die Transaktionsverwaltung dar. Da, wie wir noch sehen werden, zwischen Transaktions– und Stepverwaltung explizit unterschieden wird, müssen Mechanismen vorgesehen werden, um Kontrollfluß und Abhängigkeiten zu koordinieren. So muß die Transaktionsverwaltung zum Beispiel Ereignisse, wie den Abbruch oder die Beendigung einer Transaktion bekanntgeben, da dies Teil der den Kontrollfluß beeinflussenden Informationen ist.

6.2 Die systembildenden Komponenten

Gehen wir von den beiden Einheiten Step und Skript aus. So können wir zur Ausführung folgende Komponenten deklarieren:

ConTract Manager (CM) Der Kontrollfluß, den ein Skript für die Steps definiert, wird durch eine eigene Komponente, den *ConTract Manager* verwaltet. Er entscheidet u.a. anhand von Status und Ergebnis, welche Steps zur Ausführung kommen bzw. welche Fehlerbehandlung notwendig ist. Er stellt die Schnittstelle zur Anwendung zur Verfügung.

Communication System (CS) Ein Skript ist von seiner Umgebung, zum Beispiel der Systemtopologie (Rechner, Netzwerk) unabhängig. Entscheidend ist, daß die Server zur Step-Ausführung innerhalb des Systems bekannt und verfügbar sind. Die Verteilung, Adressierung, Aktivierung, Überwachung oder auch die Nachrichtenübermittlung hingegen sind Aufgaben die in APRICOTS im *Communication System* zusammengefaßt sind.

Transaction Manager (TM) Neben der herkömmlichen Transaktionssemantik hat der *Transaction Manager* die Beziehungen und Abhängigkeiten zwischen Transaktionen zu verwalten und zu koordinieren. Hierbei findet sowohl eine logische Integration — Koordination von Transaktionen und ihren Beziehungen — wie auch eine physische Integration — Einbindung verschiedener Systeme und Abbildung der Transaktionen in fremde Verwaltungen — statt.
In Zusammenarbeit mit der Systemverwaltung wird dies um Aspekte verteilter Umgebungen erweitert.

Step Computation Server (SS) Die eigentlichen Steps kommen in speziellen Komponenten zum Ablauf, sogenannten *Step Computation Servern*. Diese implementieren neben dem eigentlichen Stepcode die Verarbeitung des Kontextes und die Kommunikation.

Resource Manager (RM) Der Stepcode umfaßt Aufrufe Dienste mittels sogenannter `RMCalls`. Diese Dienstleistungen umfassen unter anderem Zugriffe auf Ressourcen, wie zum Beispiel auf Datenbanken oder Dateisysteme, oder führen Berechnungen durch, und so weiter.

Erst das Zusammenspiel dieser Komponenten ermöglicht es, die als Ziel gesetzte Funktionalität zu erreichen. Wir wollen nun schrittweise die dazu notwendigen Abläufe und besonders die Einbeziehung des TMs beschreiben.

6.3 Starten eines Steps

Ein Step ist eine atomare Verarbeitungseinheit aus Sicht eines Skriptes, weswegen ihm auch eine Transaktion unterlegt wird. Er besitzt somit die ACID–Eigenschaften. Ein Step wird vom *Step Computation Server* ausgeführt. Wie dargestellt, entscheidet der CM über die Aktivierung eines Steps. Wo, auf welchem Knoten und Rechner, der Step letztendlich zur Ausführung kommt, liegt im Verantwortungsbereich des CS. Ist der Auftrag erst einmal bei einem SS angekommen, prüft dieser zunächst die Eingangsinvarianten, holt den Kontext für diesen Step aus der Datenbank und startet dann den eigentlichen Step-Code. Nach dessen Beendigung wird der Ausgabekontext zurückgeschrieben [Bur91] [Wah91] [WR91].

Der SS terminiert und der CM erhält die Nachricht von der Beendigung des Steps. Für den CM stellt dieser Vorgang einen einzelnen Aufruf dar. Innerhalb des Stepcodes werden die verschiedenen `RMCalls` abgesetzt, sprich die zu einer Transaktion zusammengefaßten Aktionen ausgelöst. Deshalb sprechen wir dem SS auch die Demarkation der Transaktion zu, die dem einzelnen Step unterliegt. Dazu wird vom CM die GTRID für diesen Step

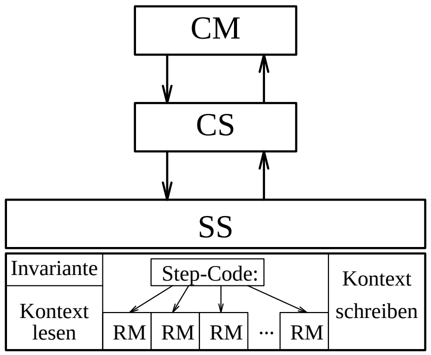


Abbildung 6.1: Schematische Darstellung einer Step-Ausführung

übergeben; der SS meldet dem TM den Beginn der Transaktion und analog auch das Ende. Er entscheidet somit über Erfolg oder Mißerfolg der Transaktion. Aus Sicht des CM hingegen ist ein Step immer erfolgreich. Auch wenn er durch seine Rückgabewerte eventuell ein Scheitern der Transaktion bekannt gibt, so hat er doch eindeutig terminiert und seine Arbeit dem ACID–Paradigma entsprechend erfüllt. Der „neu erzeugte Zustand“ ist eben die Meldung des Scheiterns. Wohlgemerkt: Die Demarkation wie auch das Prüfen der Invarianten oder die Bearbeitung des Kontextes wird vom Code des SS übernommen, geschieht mithin automatisch und transparent für den Programmierer des eigentlichen Stepcodes.

Ein Problem besteht jedoch darin, die Atomarität von Step und Transaktion in Deckung zu bringen. Ergeht ein EOT vom SS an den TM und wird diesem stattgegeben, so muß in gleichem Atemzug eine entsprechende Erfolgsmeldung („*Step_Committed*“) an den CM erfolgen. Tritt zwischen diesen beiden Aktionen ein Fehler auf, kann es fälschlicherweise zu einer Wiederholung der Transaktion kommen. Deshalb müssen beide Aufrufe atomar abgehandelt werden.

Andererseits wäre denkbar, das Absetzen der Meldung „*Step_Committed*“ in die Transaktion aufzunehmen. Dazu richten wir einen Resource Manager ein, dessen Aufgabe es ist, bei seinem Commit eine Nachricht an den CM mit Bekanntgabe des erfolgreichen Beendens des Steps zu schicken. Wir hätten zum einen die Garantie, daß die Nachricht abgesetzt wird und automatisch auch eine Synchronisierung von Step– und Transaktionsende, da das Abschicken eben genau beim EOT des Steps und somit beim Commit des RMs erfolgt.

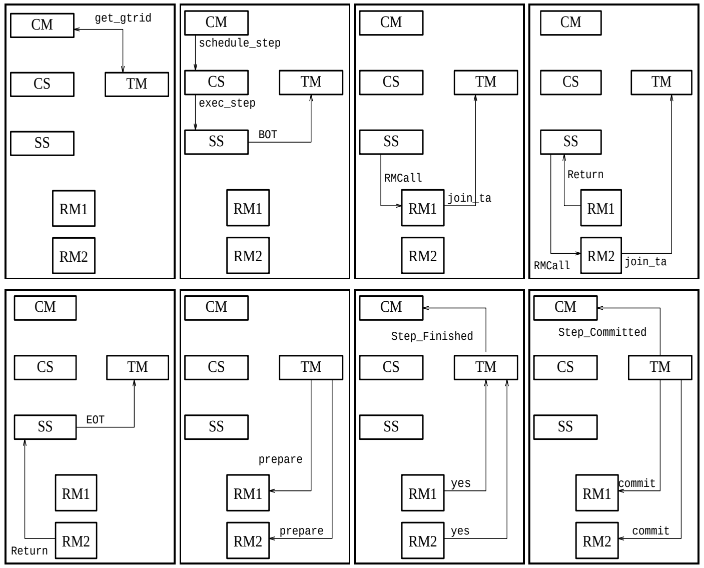
Wie wir noch sehen werden, benötigt der CM allerdings schon vor dem Commit des Steps Information über dessen Beendigung, zum Beispiel wenn dieser aufgrund von Beziehungen verzögert wurde. Es genügt also nicht, wenn der SS vor Beendigung der Transaktion eine Nachricht an den CM schickt, die das Ende des Steps bekanntgibt, denn die Ergebnisse können ja noch nicht garantiert werden. Garantierbar sind sie erst, wenn alle RMs mit ‘Yes’ votiert haben und sich im Zustand prepared befinden. Nun könnte verlangt

werden, der SS müsse eben sein eigenes 2-Phase-Commit implementieren, um nach der vorbereitenden Phase eine Nachricht an den CM erzeugen zu können. Natürlich implizierte dies deutlichen Mehraufwand in Verwaltung und Koordination; schließlich müssen Konflikte mit dem TM in der Abarbeitung der Transaktionen vermieden werden. Wir geben deshalb eine andere Lösung vor, die eine Erweiterung des TMs vorsieht. Dieser sendet nach Erhalt der Votierungen bei positiver Commit-Entscheidung eine entsprechende Nachricht an den CM, der insofern vom (garantiert) möglichen Ende des Steps erfährt („*Step_Finished*“)[1]. Im Rahmen der Commit-Meldungen, muß die Transaktionsverwaltung garantieren, daß jeder involvierte Resource Manager von der Entscheidung erfährt. Fügen wir dieser Menge noch die Nachricht an den CM bei, die das reale Ende des Steps zum Inhalt hat („*Step_Committed*“), erreichen wir mit geringem Aufwand die gewünschten Effekte: In der 2. Phase des COmmit-Protokolls erfährt der CM von der (garantierten) Beendigung der Transaktion bzw. des Steps. Ebenso ist ein Abgleich von Step– und Transaktionsende erreicht.

Anmerkung:

> Nehmen wir die Events zum Übergangsprotokoll hinzu, können wir dieselbe Funktionalität erreichen. Dazu nehmen wir dann folgende Eintragungen vor: „Beim Prepare der betreffenden Transaktion melde ein „*Step_Finished*“, beim Commit entsprechend „*Step_Committed*“.“
>
> Da die Event-Meldungen ebenfalls den zuvor beschriebenen Voraussetzungen begnügen müssen, ersparen wir uns die Erweiterung des TM für die Abwicklung von Steps.

Wir erhalten somit folgenden (skizzierten) Ablauf:



[1] Analog wird im Falle des Scheiterns die Meldung „*Step_Aborted*“ erzeugt.

Zunächst erfragt der CM beim TM eine GTRID. Anschließend übergibt er dem CS den Auftrag, den Step unter dieser Kennung auszuführen. Der CS beauftragt, nach Durchführung von Prüfungen und Auslastungsberechnungen, einen bestimmten SS mit der Ausführung. Dieser meldet dem TM den Beginn der Transaktion und startet dann den Stepcode (Invarianten und Kontext werden außer acht gelassen). Innerhalb des Stepcodes erfolgen diverse `RMCalls`, die zur Partizipierung verschiedener Resource Manager an der Transaktion führen. Nach Abschluß des Codes gibt der SS dem TM das Ende der Transaktion bekannt, welcher nun im Rahmen des 2-Phase-Commit die Votierungen der beteiligten Resource Manager erfragt und bei Erfolg die Nachricht „*Step_Finished*" an den CM absetzt. Im Zusammenspiel mit der Versendung der Commit-Nachrichten wird schlußendlich noch die Nachricht „*Step_Committed*" an den CM übermittelt. Der Prozeß kehrt zum SS zurück; dieser terminiert.

6.4 Transaktionen um Steps

6.4.1 Das Beispiel

Beispiel 6.1: *Transaktionen um Steps*

Für die folgenden Ausführungen legen wir das Szenario aus Abbildung 6.2 zugrunde. Es ist Auszug eines Skriptes.

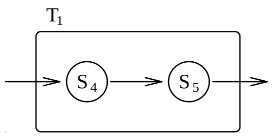


Abbildung 6.2: Eine Sequenz von Steps unter einer Transaktion

6.4.2 Die Ansätze

1. In einem ersten Ansatz lassen wir S_4 und S_5 nicht unter einer eigenen Transaktion ablaufen, obwohl wir das für Steps ja so definiert hatten. Der Ablauf hat folgende Gestalt:

   - Der CM definiert um S_4 und S_5 eine Transaktion T_1 und meldet dem TM einen Beginn von GTRID_{T_1}. Er muß sich lokal vermerken, daß für diese Steps nicht eine separate GTRID angefordert, sondern jeweils die GTRID für T_1 (GTRID_{T_1}) beim Starten übergeben wird.
   - Die SS melden dem TM nach ihrem Start die Teilnahme an der (übergebenen) Transaktion, in diesem Fall GTRID_{T_1} mittels join_transaction.

     Nach Beendigung des Codes erhält der CM die Nachricht „*Step_Finished*".

   - Nach Beendigung beider Steps fordert der CM vom TM ein EOT für T_1. Der TM wickelt ein 2-Phase-Commit-Protokoll mit den beteiligten SS ab, bringt diese in einen Endzustand (Commit oder Abort) und meldet das Ergebnis an den CM.

   Halten wir folgendes fest[2]:

   (a) S_4 und S_5 werden durch die 2-Phase-Commit-Semantik verzögert, so daß die Atomarität von T_1 garantiert ist.

   (b) Wenn einer der beiden Steps abbricht, dann muß auch T_1 (und der andere Step) abbrechen laut 2-Phase-Commit. Dieser Abbruch wird erst beim durch den CM initiierten EOT erkannt, es sei denn, der CM behandelt explizit eine Meldung „*Step_Aborted*".

   (c) Daraus folgt auch, daß für einen gescheiterten Step keine Alternative gestartet werden kann, respektive auf einen früheren Zeitpunkt innerhalb der Transaktion T_1 aufgesetzt werden kann (*savepoint*). T_1 ist gewissermaßen eine flache Transaktion.

   (d) Der SS hat statt dem Beginn der Transaktion seine Teilnahme zu melden. Ebenso darf er die Transaktion nicht am Ende des Steps autonom beenden.

2. Um ein wenig flexibler zu sein, versuchen wir in einem zweiten Ansatz, auch die beiden Steps unter jeweils einer eigenen Transaktion ablaufen zu lassen, und Beziehungen für das koordinierte Commit zu verwenden.

   Um das Commit der Steps zu verzögern und entsprechend zu einem übereinstimmenden Resultat zu bringen, können wir verschiedene Beziehungen verwenden:

   (a) $S_4 \leftrightarrow_c S_5$
   Die gegenseitige Commit-Beziehung verlangt, daß wenn eine der beiden Step-Transaktionen Commit erreicht, auch die andere dies erreichen muß. Resultat ist eine Verzögerung analog dem 2-Phase-Commit. Wenn allerdings eine Transaktion abbricht, dann muß auch die andere abbrechen. Dies entspricht etwa einer flachen Transaktion für S_4 und S_5.

   (b) $T_1 \rightarrow_a S_4$ und $T_1 \rightarrow_a S_5$
   Hier nun werden S_4 bzw. S_5 bei ihrem Commit verzögert, da daß Ende von T_1 ja noch ungewiß ist. Erst wenn T_1 (erfolgreich) beendet wird, können die Steps dies entsprechend.
   Allerdings funktioniert dies nur, wenn das EOT für die Steps **vor** dem EOT für T_1 gefordert wird. Außerdem sind diese Beziehungen unerläßlich im Gegensatz zur Variante (a), denn bei einem Abbruch von T_1 **müssen** auch die Step-Transaktionen abgebrochen werden. Das erreicht die erste Variante alleine noch nicht.

   (c) $T_1 \leftrightarrow_c S_4$ und $T_1 \leftrightarrow_c S_5$
   Eine Verzögerung wird analog erreicht. Auch führt ein Abbruch von T_1 zum Abbruch der Step-Transaktionen. Darüber hinaus impliziert ein Abbruch einer Step-Transaktion den sofortigen Abbruch von T_1, was wir eigentlich verhindern wollten.

[2] S_i steht aus Gründen der Vereinfachung für die Transaktion T_i, die dem Step i unterliegt.

Aus den geschilderten Gründen verfolgen wir im weiteren Variante (b). Das ergibt folgenden Ablauf:

(a) Erneut deklariert der CM eine Transaktion T_1 um die beiden Steps. Für diese fordert er aber dennoch Transaktionskennungen an ($GTRID_{S_4}$ und $GTRID_{S_5}$), um anschließend die Abort-Abhängigkeiten von T_1 nach S_4 bzw. S_5 einzutragen und den Beginn von T_1 bekanntzugeben.

(b) Nun kann S_4 gestartet werden. Der betreffende SS meldet den Beginn der Transaktion $GTRID_{S_4}$. Nach dem Durchlaufen des Stepcodes initiiert er ein $EOT(GTRID_{S_4})$, was zu einer Verzögerung der Resource Manager und der Transaktion führt. Gleichzeitig erhält der CM die Nachricht vom Ende des Steps, zunächst allerdings nur das oben bereits beschriebene „*Step_Finished*“. Jedoch genügt diese Information, um den Step S_5 zu starten.

(c) Dessen Ausführung erfolgt analog. Auch er wird an seinem Ende verzögert.

(d) Die durch diese Verzögerung ausgelöste Nachricht bewegt den CM, nun das Ende der Transaktion T_1 zu fordern. Daraufhin werden S_4 und S_5 in den Zustand Committed versetzt, jeweils die Nachricht „*Step_Committed*“ generiert und schließlich das Ende von T_1 bestätigt.

Was haben wir zusammenfassend erreicht?

- Die Ausführung beider Steps unterliegt insofern einer Transaktion, als ihre Ergebnisse bis zum Ende der umfassenden Transaktion verzögert werden.
- Ein Abbruch der umfassenden Transaktion impliziert auch den Abbruch der beiden Steps.
- Der CM erfährt vom (garantierten, vorläufigen) Ende des ersten Steps und kann diese Information zu Kontrollflußbestimmungen verwenden. Überhaupt ist der Kontrollfluß gänzlich in den CM verlagert, denn ein Scheitern eines der beiden Steps führt aus Sicht der TBN-Verwaltung **nicht automatisch** zum Abbruch des anderen Steps. Vielmehr kann der CM über Maßnahmen zur Behebung, etwa einen alternativen Step, entscheiden.

  Soll hingegen dieser Entscheidungsspielraum nicht explizit zur Verfügung stehen, so sollte Variante (c) (gegenseitige Commit-Beziehung zur umfassenden Transaktion) statt der Abort-Beziehung verwendet werden. Hierbei führt dann der Abbruch eines Steps zum Abbruch des anderen.

Zusammenfassend läßt sich kein generelles Vorgehen festlegen; aber nicht zuletzt darin liegt die Stärke des Modells. Die Anwendung kann explizit definieren, an welchen Stellen automatisch über Erfolg und Mißerfolg von Transaktionen entschieden werden soll.

Kapitel 7

Probleme und Potentiale der TAV auf TBN

7.1 Exemplarische Modellierung von TBN

In diesem Abschnitt soll nun gezeigt werden, wie mit den bisher eingeführten Mechanismen Transaktionsverarbeitung betrieben werden kann. Es soll deutlich gemacht werden, daß jeweils verschiedene Möglichkeiten der Modellierung bestehen, je nach Interpretation und Autonomiegrad der Anwendung. Letztendlich soll noch aufgezeigt werden, daß das bis jetzt geschilderte Modell nicht differenziert genug sein kann, um exakt unsere Anforderungen zu erfüllen.

Doch beginnen wir mit der Modellierung eines Beispiels: Wir wollen uns mit dem Thema der Schachtelung von Transaktionen befassen, wie sie häufig vorkommt und auch natürlich für die Struktur heutiger Programme und Programmiersprachen ist.

Beispiel 7.1: *Geschachtelte Transaktionen*

> Ein Programm beinhalte folgende Sequenz: Eine Funktion A ruft die Funktionen B, C und D sukzessive auf. Dafür umschreibt sie diesen eine Transaktion T_A. Funktion C wiederum ruft, ebenfalls unter einer Transaktion (T_C), die Funktionen E und F auf, entsprechend ruft D die Funktionen G und H unter der Transaktion T_D. Die Funktionen E bis H generieren weitere Transaktionen T_E bis T_H (Abbildung 7.1).

Nun soll zwischen den Transaktionen folgendes gelten:

1. Bricht eine Transaktion ab, so sollen auch deren Subtransaktionen abbrechen. In diesem Fall heißt dies: Bricht T_A ab, so auch T_C und T_D.
2. Die Transaktionen dürfen erst dann ihre Sperren und Ergebnisse frei– bzw. bekanntgeben, wenn die gesamte Transaktion T_A erfolgreich beendet ist.
3. Der Abbruch einer Subtransaktion soll zum Abbruch der Subtransaktionen mit demselben Vater führen. Daß heißt, entweder die gesamte Arbeit unter einer Transaktion wird erfüllt oder keine der Auswirkungen wird sichtbar. Der jeweilige Vater kann über Abbruch der Transaktion entscheiden. Im Beispiel wäre dies bei Abbruch von T_E der Fall: Auch T_F müßte abgebrochen werden; T_C kann über ihren Abbruch entscheiden.

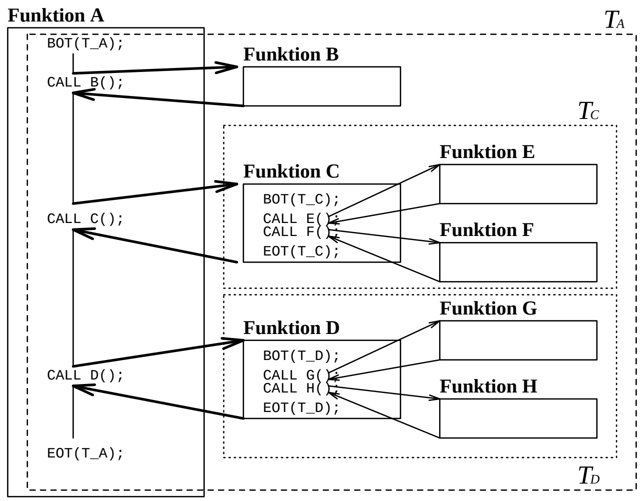


Abbildung 7.1: Schachtelung von Transaktionen

4. Die Subtransaktionen müssen vor ihrem 'Vater' beendet sein.

Betrachten wir obige Abbildung, so erkennen wir, daß Punkt 4 schon erfüllt ist, da das Commit von T_A erst dann gefordert wird, wenn die Funktionsaufrufe zurückgekehrt sind und somit die Subtransaktionen ihre Beendigung erfahren haben. Verbleiben noch die Punkte 1 bis 3.

Um die *all–or–nothing*-Eigenschaft der Subtransaktionen eines Vaters (Punkt 3) zu erfüllen, tragen wir für diese die gegenseitige Commit-Beziehung ($\leftrightarrow_c$) ein, im Beispiel $T_C \leftrightarrow_c T_D$. Laut Definition führt dann der Abbruch einer Subtransaktion zum Abbruch aller anderen Subtransaktionen. Für Punkt 1 führen wir die Abbruchbeziehung $\rightarrow_a$ zwischen Vater und Subtransaktion ein (z.B. $T_A \rightarrow_a T_C$). Ein Abbruch des Vaters fordert dann den Abbruch der Subtransaktion. Gleichzeitig wird, der Definition des Commit-Protokolls folgend, erreicht, daß beim Commit-Wunsch bezüglich einer Subtransaktion diese solange verzögert wird, bis der Vater ebenfalls das Commit erreicht (Punkt 2).

Das vollständige TBN zeigt Abbildung 7.2.

Dieses TBN scheint somit den obigen Anforderungen zu genügen. Was jedoch, wenn die Funktionsaufrufe asynchron und parallel abgesetzt werden? Dann würden die Subtransaktionen nicht automatisch durch die Abfolge von Befehlen des Codes bezüglich ihres Commits synchronisiert, es könnte nicht garantiert werden, daß das Commit des Vaters nach dem der Kinder erfolgt. Ergo müßte im Code eine explizite Synchronisation vorgesehen wer-

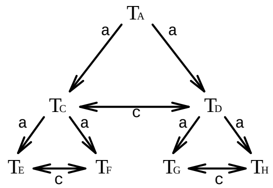


Abbildung 7.2: Modellierung von geschachtelten Transaktionen (1)

den; etwa kann in einer Schleife gewartet werden, bis die Ergebnisse aller Funktionsaufrufe eingetroffen sind.

Nun sollen aber genau solche, wiederkehrenden Aufgaben, die die Kontrolle von Transaktionen betreffen, in die Transaktionsverwaltung eingeschrieben werden. Zusätzlich zwischen Vater und Subtransaktion die Beziehung $<_c$ einzutragen, bringt keine Verbesserung, denn aufgrund der Abbruchbedingung wird die Subtransaktion solange verzögert, bis der Vater das Commit fordert. Durch die neue Beziehung wird aber gerade das Commit des Vaters solange verzögert, bis die Kinder den Zustand Commit eingenommen haben; ein Zyklus.

In einem weiteren Modellierungsansatz definieren wir zwischen Vater– und Subtransaktionen die gegenseitige Commit-Beziehung (z.B. $T_A \leftrightarrow_c T_C$). Punkt 2 ist erfüllt, da erst alle Transaktionen beendet werden können, wenn auch alle das Commit gefordert haben, also auch der oberste Vater. Punkt 1 und 2 sind insofern erfüllt, da ein Abbruch einer Transaktion zum Abbruch aller anderen Transaktionen führt. Punkt 4 ist in abgeschwächter Form erfüllt, da alle Transaktionen gleichzeitig ihren finalen Zustand erreichen. Zwar kann nun der Wunsch nach dem Commit zu einem beliebigen Zeitpunkt gestellt werden, die Reihenfolge also außer acht gelassen werden, jedoch gilt die *all–or–nothing*-Eigenschaft nun für alle Transaktionen des Baumes. Keine Vatertransaktion kann über ihr Ende autonom entscheiden, wenn eine ihrer Subtransaktionen (oder alle) abbrechen; Zurücksetzen entfällt.

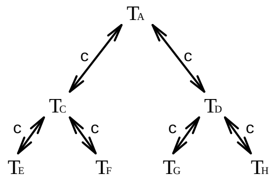


Abbildung 7.3: Modellierung von geschachtelten Transaktionen (2)

Wir sehen folgende Diskrepanz: Aufgrund unseres bisherigen Modells kann die Anwendung zwischen großer Autonomie bezüglich Kontrollfluß/Commit-Entscheidung und Flexibilität entscheiden; einen Mittelweg gibt es für sie nicht. Dazu sind die Auswirkungen von logischen Beziehungen zu mächtig definiert. Ihnen muß gezwungenermaßen die feine Differenzierung abhanden gehen.

Deshalb gehen Reuter und Gray in [RG91] einen anderen Weg: Sie verwenden Trigger, um bei Eintritt eines bestimmten Zustandes einer Transaktion direkt die resultierenden Zustandsänderungen bei anderen Transaktionen festzulegen. Desweiteren kann über *Wait*-Beziehungen die Verzögerung bestimmter Zustände in Abhängigkeit vom Zustand anderer Transaktionen erreicht werden. Hierfür wurden Regeln definiert, die es darüber hinaus erlauben, auch Modifikationen der Strukturen ereignisgesteuert vorzunehmen. So können zum Beispiel beim Beginn einer neuen Transaktion ihre Beziehungen (automatisch) generiert und bei Beendigung entsprechend entfernt werden. Das macht deshalb Sinn, weil in diesem Modell zum einen recht hoher Definitionsaufwand betrieben werden muß — es sind deutlich mehr Einflüsse zu definieren als bei logischen Beziehungen, die diese inherent umfassen. Zum anderen muß die Applikation recht genau über die Einflüsse Bescheid wissen, so daß die Selbstmodifikation als Erleichterung für generelle Fälle zu sehen ist. Leider sind hierfür noch keine Ausführungen bezüglich der Verwaltung und eines Übergangsprotokolls bekannt. Es bleibt interessant, ob sich einer der beiden Ansätze überhaupt als vollständig und praktikabel erweisen kann, und wenn ja, mit welchen Erweiterungen — oder ob schlußendlich eine Kombination erst adäquat verwendbar wird.

7.2 Inkonsistente TBN und Fehlerbehandlung

Wir sprechen nun einen enorm wichtigen Problembereich an. Kaum da wir die Mächtigkeit der TBN erahnen, müssen wir uns mit der Frage von Fehlern befassen. An dieser Stelle werden Vollständigkeitsbeweise und Abschätzungen über die Erreichbarkeit bestimmter Situationen erforderlich, um das Gebild 'TBN' vollständig erfassen und die Verwaltung darauf gestalten zu können. Ungeklärt dürften demnach zahlreiche Ursprünge und Bedeutungen von Fehlern sein, was die Erstellung von Ansätzen zu deren Behebung nicht gerade erleichtert. Wir wissen darum, daß wir äußerst unsicheres Territorium betreten; aber gerade das macht den Reiz der Arbeit aus und soll Vorarbeit für Gedankenspiele und Lösungsansätze leisten.

Da wir mit komplexen Gebilden zu tun haben und diese autonom Informationen für die Applikation verwalten, läßt es sich nicht vermeiden, daß Unstimmigkeiten auftreten, die aus Fehlern der beteiligten Komponenten oder Gegebenheiten des Systems resultieren. Die TBN-Verwaltung leistet ja gerade die Garantie, das definierte Transaktionsgebilde zu einem koordinierten Abschluß im Sinne oben definierter Konsistenz zu bringen. Gewissermaßen realisiert die TBN-Verwaltung die durch die Applikation modellierte Sicht der realen Welt. Konflikte, wie wir sie im folgenden behandeln wollen, stellen nun Diskrepanzen zwischen realer und modellierter Welt dar. Um so klarer wird, daß diese Fehler durchweg **nur in Kooperation** zwischen Applikation und TBN-Verwaltung erfolgen kann.

Zunächst sollen verschiedene Ausprägungen von Inkonsistenz unterschieden werden. Zu Beginn ist die *erkannte Inkonsistenz* zu nennen. Sie bezeichnet die bei einer Berechnung eines neuen TBN erfolgte Beobachtung, daß jenes neue TBN nicht konsistent wäre. Sollen zum Beispiel widersprüchliche Beziehungen zwischen zwei Transaktionen eingetragen werden oder ergibt die transitive Berechnung, daß das potentiell neue TBN niemals erfüllbar

sein wird, so hat die TBN-Verwaltung eine Inkonsistenz erkannt. Der zuvor schon betrachtete Faktor 'Unsicherheit' über den zukünftigen Zustand von Transaktionen hingegen geht in den Begriff *potentielle Inkonsistenz* ein. Die Ursache der Unsicherheit lag darin, daß für eine Transaktion ein geforderter Zustand nicht garantiert werden konnte. Wir hatten TBN-Operationen, die solche unsicheren Berechnungen zur Folge haben, als gefährliche Operationen gekennzeichnet. Schließlich kennen wir noch die *faktische Inkonsistenz*, die, genau genommen, Folge einer potentiellen Inkonsistenz sind. Sie entstehen, da gewisse TBN-Operationen nicht abgelehnt werden können.

Beispiel 7.2: *Faktische Inkonsistenz*

Nehmen wir das in Abbildung 7.4 gezeigte TBN. Hier mußten nach dem Abort von

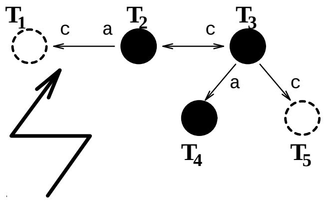


Abbildung 7.4: Gefahr der faktischen Inkonsistenz

T_2 auch die Transaktionen T_3 und T_4 aufgrund der eingetragenen Beziehungen abgebrochen werden. Gleichzeitig muß gefordert werden, daß T_1 den Zustand Commit erreicht. Das kann natürlich nicht garantiert werden, wenn T_1 noch aktiv ist; auf potentielle Inkonsistenz müßte erkannt werden. Andererseits kann der Abbruch von T_2 natürlich nicht verhindert werden kann. Wenn aber auch T_1 abbricht, was ebenfalls nicht zu verhindern ist, dann genau widerspricht die modellierte Welt der Realität. Der Konflikt kann durch keine Mechanismen der TBN-Verwaltung behoben werden.

Was kann nun aber in solchen Fällen getan werden? Es muß zunächst die Frage gestellt werden, wie und wann Inkonsistenzen erkannt werden können. Wie schon erwähnt, findet im Rahmen des Übergangprotokolls bei jeder TBN-Operation zunächst eine direkte Konsistenzprüfung statt. Diese umfaßt den Abgleich verschiedener Beziehungen zwischen 2 Transaktionen bzw. den Test, ob eine Transaktion den geforderten Zustand laut Zustandsübergangsdiagramm überhaupt, und wenn ja ob sicher, erreichen kann. Anschließend wird, so keine Inkonsistenz vorliegt und ausgehend von der faktischen Erfüllung der Operation, transitiv überprüft, ob für das potentiell resultierende TBN noch Pfade zu seiner Lösung existieren, wobei Unsicherheit — so die restriktive Fassung des Übergangsprotokolls — als Scheitern interpretiert wurde. Existieren keine Pfade oder nur unsichere, so ist das ein weiterer Ansatz zur Fehlerbehandlung.

Auf welche Weise kann nun aber Fehlerbehandlung betrieben werden? Nun, erkannte und potentielle Inkonsistenz muß in Ablehnung der gewünschten Operation resultieren. Das TBN behält in diesem Fall Struktur und Zustand; soweit aus Sicht der TBN-Verwaltung. Nicht ganz so leicht fällt die Interpretation der Ablehnung aus Sicht der Anwendung. Denn durch im Laufe der Zeit von anderer Stelle eingebrachte Information kann es durchaus möglich sein, daß zu einem späteren Zeitpunkt genau die soeben noch abgelehnte Operation gestattet wird. Es gibt allerdings keine Aussagen darüber, ob die betreffende Zeitspanne endlich ist, oder gar, wie groß sie genau ist. Die einzige Information, die der Anwendung bereitgestellt werden kann, ist die, ob die Ablehnung von erkannter oder potentieller Inkonsistenz herrührt. Im ersteren Fall sind Fehler in der Struktur des TBN vorhanden, die nur durch weitergehende Modifikation behoben werden können oder einen Abbruch der Applikation implizieren. Allein die potentielle Inkonsistenz nährt die Hoffnung, weitere Operationen verdichten die Informationen, so daß die ursprüngliche Operation möglich wird. Dabei ist allerdings nicht gesagt, daß es überhaupt noch Operationen gibt, die nicht aufgrund von Unsicherheit abgelehnt werden müssen, so daß die Applikation ohnehin ewig warten würde[1].

Halten wir fest: Ablehnung einer TBN-Operation muß durch die Applikation interpretiert und behandelt werden, denn nur sie 'kennt' die reale Welt und ist somit überhaupt in der Lage, die Diskrepanz zur modellierten Welt zu mindern.

Schwieriger wird dieses Unterfangen im Falle von faktischer Inkonsistenz. Betrachten wir unser obiges Beispiel, so wird der TBN-Verwaltung in diesem Fall nichts anderes übrig bleiben, als den Abbruch durchzuführen und auf faktische Inkonsistenz zu erkennen. Nur wem gegenüber wird 'erkannt'? Der Übergangswunsch kann durch einen involvierten Resource Manager oder auch das System initiiert werden. Wie also erfährt die Applikation davon? Welche der an einem Netz partizipierenden Applikationen werden in Kenntnis gesetzt? Da Inkonsistenz ein nach Definition „mächtiger Zustand" ist, muß mit einer extremen Fortpflanzung einer Inkonsistenz, vielleicht sogar im gesamten Netz, gerechnet werden. Macht es da überhaupt noch Sinn, die Arbeit fortzusetzen? Oder wäre es nicht sinnvoller, auf Initiative der TBN-Verwaltung alle Transaktionen und die betroffenen Applikationen abzubrechen? Wie aber kann dies geschehen und wie sieht eine Zusammenarbeit mit dem Betriebssystem aus?

Weil diese Überlegung recht pessimistisch anmuten, definieren wir in APRICOTS, daß die TBN-Verwaltung bei Auftreten einer faktischen Inkonsistenz eine Meldung an den CM generiert, deren Inhalt genau dieser Umstand ist. Ab dato wird die Behandlung der Anwendung überlassen. Ob diese die gesamte Arbeit abbricht oder versucht, mittels Netzmodifikationen, die Inkonsistenz zu beheben, wird offen gelassen. Sowieso ist im derzeitigen Stadium unserer Untersuchungen ungeklärt, ob solche Korrekturen überhaupt möglich sein werden. Das Problem besteht im wesentlichen darin, nicht zuviel Information (über die Inkonsistenz) im Netz zu vermerken, da wir eine große Fortpflanzung erwarten, die jegliche Modifikation unmöglich machen würde. Wie diese Informationsgestaltung auszusehen hat, ist jedoch auch noch nicht abzusehen, denn schließlich dürfen auch nicht zu wenige Informationen vorliegen. Das nächste Problem liegt darin, zu entscheiden, wann die Korrekturen abgeschlossen sind. Denn im allgemeinen werden dazu mehrere Operationen notwendig werden. Wann aber, wenn, wie bisher, keine Klammerung von Operationen möglich ist, kann die 'normale' TBN-Verwaltung wieder aufgenommen werden? Schließlich muß geklärt werden, ob nur der von der Inkonsistenz betroffenen Teil des TBN auf diese Weise isoliert behandelt

[1] Eine Operation, die zu jedem Zeitpunkt akzeptiert werden muß, ist der Abbruch einer Transaktion. Dies macht aber nur dann konstruktiv Sinn, wenn der Abbruch einer Modifikation des inkonsistenten TBN und/oder Alternativtransaktion vorausgeht.

werden kann und ob Operationen an anderer Stelle gestattet werden können, oder ob dies gar zu dann gänzlich unlösbaren Inkonsistenzen führt, da zu ihrer Lösung die Kooperation verschiedender Applikationen nötig wäre.

7.3 Reduktion von TBN

Auf eine etwas einfachere Frage soll am Schluß noch eingegangen werden: die Reduktion von Transaktionsbeziehungsnetzen. Im klassische Transaktionsmodell flacher Transaktionen können sämtliche Informationen über eine Transaktion, nachdem diese einen finalen Zustand erreicht hat, verworfen werden. Dies gilt sowohl für die Transaktionsverwaltung wie auch für die beteiligten Resource Manager oder zum Beispiel auch das Kommunikationssystem. In dem vorliegenden Modell gilt diese einfache Regel jedoch nicht so ohne weiteres. Denn, auch wenn eine Transaktion beendet ist, können noch Beziehungen zu ihr eingetragen sein, die Einfluß auf andere Transaktionen nehmen — sprich die Beziehungen können noch aktiv sein. Aber selbst wenn dies nicht der Fall ist, können wir die Informationen nicht einfach eliminieren. Denn wir hatten erlaubt, neue Beziehungen zwischen Transaktionen zu jedem Zeitpunkt und unabhängig vom Zustand der betroffenen Transaktionen einzutragen, die Konsistenz natürlich vorausgesetzt. Daraus folgt, daß auch zu einer sich in finalem Zustand befindlichen Transaktion Beziehungen definiert werden können.

Natürlich darf an dieser Stelle die Frage nach dem Sinn gestellt werden, denn dynamische Beziehungen, zum Beispiel durch Synchronisation, können nicht mehr auftreten, die finale Transaktion greift schlichterdings nicht mehr auf Daten zu. Und strukturelle Beziehungen durch die Applikation machen zu diesem Zeitpunkt keinen Sinn mehr, denn die aus der Beziehung für eine neue Transaktion resultierende Forderung bezüglich ihres Zustandes ist a priori bestimmbar und eindeutig, da ja das Ergebnis der finalen Transaktion schon eindeutig und unveränderlich ist. Daß heißt, für die neue Transaktion kann nur ein Demand Begin resultieren, was dem sowieso zu erwartendem Starten der Transaktion gleichkommt, ein Demand Abort sein (was ein Starten unnötig macht) oder ein Demand Commit die Folge sein; letzteres wäre nicht zu garantieren, ganz davon abgesehen, daß auch in unserem Modell das Commit einer Transaktion das erwünschte Ziel ist.

Verbleiben noch die Transaktionen, die bereits in ein Netz involviert sind. Gehen wir davon aus, daß es zumindest noch eine aktive Transaktion gibt. So können zu ihr noch Beziehungen eingetragen werden, welche — und das ist entscheidend — auch durch die Beziehungen der von uns betrachteten Transaktion T beeinflußt werden, Generell gesagt, können die Informationen einer solchen Transaktion T aus dem Netz entfernt werden, die sich

1. in einem finalen Zustand befinden und die
2. keine aktiven Beziehungen mehr unterhält.

Ein offensichtlich hinreichendes Kriterium ist, daß alle Transaktionen, zu denen direkt Beziehungen von T aus eingetragen sind, ebenfalls in einem finalen Zustand sind, denn dann können die Beziehungen nichts mehr bewirken, sprich aktiv sein. In diesem Kontext können natürlich auch finale Transaktionen ohne Beziehungen und generell finale Beziehungen entfernt werden.

Beispiel 7.3: *Reduktion von TBN*

Betrachten wir nochmals unser Beispiel. Wie wir in Abbildung 7.5 sehen, ist T_4 ab-

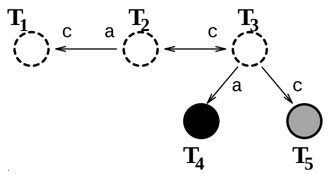


Abbildung 7.5: Reduktion im Beispiel

gebrochen und T_5 erfolgreich beendet. Beide können aus dem TBN entfernt werden, da ihre jeweiligen Beziehungen zu T_3 final sind. Zu dieser Überprüfung bedarf es der ursprünglichen, nicht aggregierten Beziehungen.

Anmerkung:

Die Konsistenzprüfung für Transaktionsübergänge muß noch um eine Randbedingung erweitert werden. Wird für eine finale Transaktion der entgegengesetzte Finalzustand gefordert, zum Beispiel ein Abort für eine Transaktion im Zustand Committed, so wird dies bisher innerhalb der direkten Konsistenzprüfung erkannt. Die Erweiterung sieht vor, daß Forderungen für nicht mehr verzeichnete (weil eliminierte) Transaktionen grundsätzlich als Fehler behandelt werden, so daß inkonsistente und doppelte Forderung auf einen Fehlerfall reduziert werden.

Das Gesagte gilt selbstverständlich nur für die TBN-Verwaltung. Alle anderen Komponenten, wie Resource Manager oder ähnliche, können im allgemeinen nach dem Beenden der jeweiligen Transaktion die Informationen vergessen, denn sie müssen weder Einflüsse auf andere Transaktionen ableiten, noch kann diese Transaktion jemals wieder aktiv werden.

Eine Reduktion des TBN erhöht die Performanz der Abarbeitung, da bei der transitiven Berechnungen weniger Transaktionen angefaßt und Operationen durchgeführt werden müssen. Dennoch, auch die Reduktion ist teuer. So muß, geht eine Transaktion in einen finalen Zustand, überprüft werden, ob sämtliche ihrer Beziehungen inaktiv sind. Gegebenenfalls wird die Transaktion entfernt. Nun war die Transaktion eventuell jedoch auch aktiver Nachbar anderer ebenfalls finaler Transaktionen, so daß auch für diese ihre Eliminierbarkeit überprüft werden muß.

Ein gänzlich anderer Ansatz wäre, das TBN erst mit der Beendigung der Applikation zu verwerfen. Das würde jedoch voraussetzen, daß die TBN-Verwaltung von der Beendigung in Kenntnis gesetzt würde.

7.4 Ausblick

Schließlich sei noch auf die Idee eingegangen, längerwährende Prozesse durch TBN zu begleiten. Bei diesen Vorgängen, wie sie zum Beispiel im Bereich des CAD oder generell beim Engineering vorkommen, kann erst im Laufe der Entwicklung definiert werden, was die nächsten Aufgaben sein werden bzw. wie diese miteinander verbunden sind. Die Verwaltungsstrukturen müssen somit zur Laufzeit erweitert bzw. reduziert werden können, und es muß möglich sein, die Bearbeitung an ausgezeichneten Punkten zu unterbrechen, um erst zu einem späteren Zeitpunkt fortzufahren.

Die deklarative Beschreibung und Dynamik sind aber genau Intention der Transaktionsverwaltung auf TBN. Während der Abarbeitung können neu definierte Aufgaben (Transaktionen) hinzugenommen und semantisch eingepaßt werden, nicht mehr benötigte oder gescheiterte werden eliminiert. Die Strukturierung ergibt sich während der Bearbeitung. Lediglich die Persistenz und die Unterbrechbarkeit muß noch gewährleistet sein. Es bleibt zu untersuchen, an welchen Punkten der Bearbeitung eine Unterbrechung erfolgen kann, wie die Ressourcen in einem solchen Fall zu verwalten sind, und so weiter.

Selbstverständlich ist hiermit noch kein vollständiges System für die Bewältigung solcher Aufgaben geschaffen — aber als Basis für ein offenes Skript im Rahmen der ConTract–Bearbeitung wäre dies durchaus denkbar.

Kapitel 8

Aspekte der Verteilung

8.1 Global eindeutige Kennung

Um Transaktionen auch über Knotengrenzen hinweg identifizieren und unterscheiden zu können, müssen wir die Kennungen dieser Transaktionen (GTRID) global eindeutig halten. Hinzu kommt, daß auch nach einem Fehler und Wiederanlauf eine eindeutige Zuordnung möglich sein muß, was die Wiederverwendung eines Wertes einer Transaktionskennung ausschließt.

8.1.1 Generierung

Daraus folgt, daß entweder die im Netz verteilten Transaktionsverwaltungen sich bezüglich der GTRIDs koordinieren, eine zentrale Stelle im Netz GTRIDs vergibt oder — und diese Lösung verwenden wir — in die GTRID die Identifikation des Rechnerknotens mit aufgenommen wird, da wir von dieser voraussetzen, daß die schon global eindeutig ist. Es müssen dann zusätzlich nur noch die GTRIDs eines Knotens durch Werte unterscheidbar gemacht werden; ein einfacher Zähler genügt hierfür. Die Annahmen, die wir treffen, lauten wiefolgt:

1. Das zugrundeliegende Kommunikations– und Betriebssystem, das das Adressierungsschema im Netz verwaltet, stellt der Transaktionsverwaltung die jeweilige Rechnerkennung zur Verfügung.

2. Diese Rechnerkennung muß zur Adressierung eines Rechnerknotens aus Sicht der Transaktion genügen. Wie wir sehen werden, müssen sich die verschiedenen TMs koordinieren, also miteinander direkt kommunizieren. Die einzige Information, die ein TM jeweils besitzt, ist die Rechnerkennung in obigem Sinne, sie muß bei Aufruf der Kommunikationsdienste als Empfängeradresse genügen.

3. Wir gehen davon aus, daß solch eine Rechnerkennung global eindeutig ist, und auch für alle Zeiten identisch für den jeweiligen Rechner bleibt. Eine erneute Vergabe ein und derselben Ausprägung muß ausgeschlossen sein.

4. Übernimmt ein (z.B. neuer) Knoten die gesamte Arbeit eines anderen unter anderer Kennung, ist dies für die TMs transparent zu halten. Das heißt, die Kommunikationsmechanismen setzen Nachrichten an die 'alte' Adresse in die 'neue' um.

8.1.2 Eindeutigkeit

Ein Anspruch an die GTRID ist, daß ein und derselbe Wert nur einmal vergeben wird. Selbst wenn die Rechnerkennung diesem Anspruch genügen sollte, muß der zusätzliche Zähler zur Unterscheidung der Transaktionen eines Knotens auch diesem Fakt rechnung tragen. Dies impliziert das stabil Machen des Zählers (oder der GTRID) durch Hinterlegung auf stabilem Speicher (z.B. im Log), so daß bei einem Wiederanlauf an der entsprechenden Stelle 'weitergezählt' werden kann.

8.1.3 Lokalisierungsinformation

Ein Problem ist die explizite Verlagerung von Arbeit — sei es ein TBN oder eine Transaktion — auf einen anderen Rechnerknoten, wie es in diesem Modell aus Gründen der Fehlertoleranz oder Performanz vorgesehen ist. In diesem Fall repräsentiert die GTRID nicht mehr den korrekten Wert: Die Kennung des Rechnerknotens kann ja nicht verändert werden, da sonst der gesamte Wert der GTRID verändert wäre. Dies ist aber nicht möglich, da die beteiligten Komponenten (Resource-Manager, Applikation) solch eine Änderung nicht nachvollziehen können. Somit interpretieren wir die oben eingeführte Kennung des Rechnerknotens in der GTRID als den Knoten, auf dem die Transaktion generiert wurde; wir nennen diese Information den *Heimatknoten*. Eine Umsetzung der Empfängeradresse (im Fall einer Verlagerung) muß gesondert vorgenommen werden. Eine Möglichkeit wäre eine außerhalb der GTRID verwaltete Information; besser erscheint es, den aktuellen Rechnerknoten einer Transaktion zusätzlich in der GTRID zu vermerken und entsprechend zu aktualisieren, nicht aber zur Identifikation hinzuzuziehen. Diese Kennung nennen wir *Koordinatorknoten*.

8.1.4 Modellierung

Offensichtlich ist der Gebrauch der GTRID aus Sicht der Applikationen und Resource Manager rein referentiell; der Inhalt muß von ihnen niemals interpretiert werden. Deshalb definieren wir die GTRID als abstrakten Datentypen [X/O90]. Das heißt, ein Programm, das solch einen ADT verwendet, kann nicht direkt auf dessen Werte zugreifen, sondern kennt nur den Verweis auf die entsprechende Struktur.

Für den Umgang mit ADTs werden Funktionen von der ADT-Verwaltung zur Verfügung gestellt, wie z.B. der Vergleich zweier ADTs oder die Generierung eines Hash-Wertes zur effizienten Speicherung. In einer nicht ganz so strikten Variante existieren auch Funktionen, um auf die Werte der ADTs zuzugreifen.

8.1.5 Inhalte

Fassen wir zusammen: Folgende Inhalte schreiben wir der GTRID ein:

- Die global eindeutige Rechnerkennung des Knotens, auf dem die Transaktion generiert wurde (*Heimatknoten*).
- Ein für einen Rechnerknoten eindeutigen Zähler zur Unterscheidung der Transaktionen eines Rechnerknotens (*TA-Zähler*).
- Die global eindeutige Rechnerkennung des Knotens, der aktuell speichernder und koordinierender Knoten für eine Transaktion ist. Diese Information dient nicht der Identifikation (*Koordinatorknoten*).

8.2 Weitere Annahmen bezüglich der Kommunikation

Um die verteilte Verarbeitung isoliert prägnant beschreiben zu können, wollen wir einige grundlegende Annahmen über Basis und Umgebung treffen. Diese Einschränkungen erlauben es uns natürlich, ein weniger mächtiges System darzustellen. Dies soll aber bewußt geschehen, um die Kernpunkte der Verteilung und Verarbeitung beleuchten zu können.

Zunächst verlangen wir die Kommunikation als *sicher*. Darunter wird verstanden, daß ein an die Kommunikationsdienste erteilter Auftrag zur Zustellung einer Nachricht garantiert beim Empfänger eintreffen wird. Ist dieser beim Kommunikationssystem registriert, aber im Moment nicht ansprechbar, so wird die Nachricht gepuffert. Diese Pufferung ist gegenüber Systemfehlern abgesichert. Desweiteren umfaßt der Begriff der Sicherheit, daß die Komponenten im Netz unabhängig von Fehlern ihre Empfängeradresse behalten, zumindest aus Sicht der Sender.

Wir gehen auch davon aus, daß Nachrichten **in genau der Reihenfolge** beim Empfänger eintreffen, in der sie gesendet wurden.

Da Nachrichten der Garantie der Zustellung unterliegen, kann das Fehlen einer Rückantwort nur in zweifacher Hinsicht interpretiert werden:

1. die Arbeit wurde wegen Überlastung (noch) nicht bearbeitet oder
2. der Empfänger ist ausgefallen und es muß eventuell damit gerechnet werden, daß er zur Bearbeitung auch nicht mehr im Stande sein wird.

Wird die Möglichkeit geschaffen, den Status eines Auftrags zu erfragen, kann das Scheitern eines Auftrags eindeutig erkannt werden.

Anmerkung:

> In diesem Zusammenhang sei darauf hingewiesen, daß es sich als Vorteil erweißt, wenn im Rahmen eines Protokolls, das zwischen zwei Komponenten abgewickelt wird, eine gewisse Toleranz bezüglich Mehrfachverschickung von Nachrichten herrscht. Dies kann nämlich dann der Fall sein, wenn eine Komponente nach einem Wiederanlauf nicht exakt definieren kann, welche der Nachrichten schon gesendet wurden und zur Sicherheit latente Aufträge erneut versendet. Da dies aber nicht in direktem Zusammenhang mit dem Kommunikationssystem steht, wollen wir nicht darauf eingehen.

Kurz wollen wir noch auf den Adressierungsmechanismus eingehen, der im folgenden Grundlage bilden soll. Die Adressierung eines Kommunikationspartners geschieht logisch. Als Adressenkomponenten sind global eindeutige Kennungen bekannt, zumindest aber Rechnerknoten, Komponententyp und Komponentennummer falls mehrere Komponenten eines Typs auf dem betreffenden Rechnerknoten anzufinden sind. Selbst diese Informationen sind aber logisch zu gestalten, das heißt die Umsetzung aus einem solchen 'Namen' in eine reale Adresse übernimmt das Kommunikationssystem. Eine logische Adresse einer Komponente ist ewig global eindeutig, die reale kann sich ändern (zum Beispiel bei Migration oder Rekonfiguration).

Anmerkung:

> Es existieren sicherlich weitere Voraussetzungen oder Randbedingungen. Eine genaue Diskussion übersteigt aber den Rahmen dieser Arbeit. Für eine ausführliche Beschreibung des Kommunikationssystems in APRICOTS sei auf [Bur91] verwiesen.

Im Rahmen der Kommunikation gehen wir von folgender Nomenklatur aus:

1. Als einen *Aufruf* bezeichnen wir die Ausführung eines Code-Stücks entsprechend einem Funktionsaufruf einer Programmiersprache.
2. Als *lokale Nachricht* wird eine Information bezeichnet, die mittels der Kommunikationsdienste verschickt wird, wobei der Adressat lokal zum Absender befindlich ist.
3. Hingegen wird eine *entfernte Nachricht* oder *Rechner-zu-Rechner-Nachricht* durch das Kommunikationssystem über das Netz vermittelt.

Lediglich für die letzten beiden Kommunikationsformen wird durch das Kommunikationssystem das abstrakte Adressierungsschema und die Sicherheit garantiert. Hingegen wird bezüglich des Aufwandes folgende Annahme getroffen: Eine Rechner-zu-Rechner-Nachricht ist wesentlich teurer als eine lokale Nachricht, diese wiederum erheblich teurer als ein Aufruf.

8.3 Verteilte Verarbeitung

Kommen wir zum eigentlichen Aspekt der *Verteilung*. Zunächst muß geklärt werden, worin der Sinn einer Verteilung von Arbeit in einem heterogenen System liegt, denn offensichtlich bedeutet dies ein gewisses Maß an Mehraufwand in Speicherung und Verwaltung zusätzlicher Informationen und Koordination asynchroner Aktivitäten.

Wie zu Beginn der Arbeit schon angedeutet, ist die Verteilung von Arbeit eine logische Konsequenz aus der Architektur heutiger und zukünftiger Systeme. Es gibt nicht mehr „die Zentrale“ von Daten und Performanz. Verschiedene Aufgaben werden verschiedenen, oft mehreren Rechner eines Verbundes übertragen. Nicht mehr ein Rechner kann alles, sondern für verschiedene Aufgaben existieren zahlreiche Möglichkeiten, sie erledigen zu lassen. Die eigentliche Arbeit wird zudem parallel von mehreren Systemen des Verbundes gleichzeitig bearbeitet. Die Daten des Gesamtsystems sind partitioniert und auf die Komponenten des Systems (beliebig) verteilt; es muß Informationsaustausch stattfinden und der Zugriff koordiniert werden. Wie wir sehen, gibt es verschiedenste Gründe für die Verteilung, wie zum Beispiel:

- Funktionale Verteilung:
  Bestimmte Aufgaben können nur von bestimmten Komponenten des Verbundes übernommen werden.
- Lastverteilung:
  Aufgaben werden auf verschiedenen Komponenten parallel zueinander bearbeitet. Diese können, müssen aber nicht in Bezug zueinander stehen.
- Datenverteilung:
  Aus verschiedenen Gründen (Erfassung, Kosten, Performanz, Sicherheit, ...) werden Daten nicht mehr ausschließlich zentral gehalten, sondern sind (permanent) bei den Komponenten hinterlegt.

Im vorliegenden Modell muß nun zwischen zwei Ausprägungen der Verteilung differenziert werden: *Verteilte Transaktionen* und *Verteilte Transaktionsbeziehungsnetze*. Verteilte Transaktionen beschreiben den Umstand, daß an **einer** Transaktion Resource Manager verschiedener Rechner partizipieren, die Koordination zur Beendigung also über Rechnergrenzen hinweg erfolgen muß. Verteilte Transaktionsbeziehungsnetze hingegen erfassen, daß **verschiedene** Transaktionen auf verschiedenen Rechnern ablaufen und in Beziehung zueinander stehen. Wohlgemerkt: Jede dieser Transaktionen an sich kann wiederum verteilt sein. Die folgenden Abschnitte werden die Unterschiede und Verfahren beschreiben.

8.3.1 Verteilte Transaktionen

Nehmen wir wieder das Bild der flachen Transaktion zur Veranschaulichung: Eine Applikation demarkiert eine Transaktion und ruft innerhalb der Transaktion verschiedene Resource Manager auf. Bisher sind wir davon ausgegangen, daß sich diese auf demselben Rechnersystem befinden. Dies muß aber nicht der Fall sein. Genauso gut können ein oder mehrere Resource Manager sich auf einem anderen Rechner befinden.

Im Idealfall ist dies der Applikation verborgen, daß heißt, in ihrer virtuell einheitlichen Welt gibt es nur einen Rechner (nämlich ihren) und somit auch keine Verteilung. Aber zum Beispiel eine über Rechnergrenzen hinweg operierende Lastverteilung könnte entscheiden, einen Resource Manager-Aufruf an einen anderen Rechner weiterzuleiten. Da die Applikation jedoch nichts von davon weiß, kann sie die Koordination natürlich weder bewerkstelligen noch unterstützen; die Transaktionsverwaltung muß um diese Funktionalität erweitert werden.

Charakterisierung:

Wollen wir verteilte Transaktionen in ihren Merkmalen beschreiben, so können wir festhalten:

1. Die Interaktionen zwischen einem Resource Manager und der Transaktionsverwaltung beschränken sich, abgesehen von Anmelde– und Abmeldeprotokoll, auf wenige Mitteilungen, wie join_transaction, prepare_to_commit oder Commit/Abort.

2. Jeder Resource Manager kann über seine Votierung isoliert entscheiden.

3. Die Commit-Entscheidung für eine Transaktion ist trivial: Wenn alle an der Transaktion partizipierten Resource Manager ihr Einverständnis gegeben haben, kann die Transaktion in den Zustand Commit überführt werden. Die Zustimmungen sind unabhängig von einander und können asynchron, parallel eingeholt werden (für die Beschreibung des 2-Phase-Commits in sei auf [EMS91] [ML83] [RP] [Ske] [X/O90] verwiesen).

Realisierung:

Wir wollen verteilte Transaktionen wie folgt behandeln: Jeder Resource Manager, der (erstmalig) an einer Transaktion teilnehmen will, meldet dies durch ein join_transaction an. Dabei ist entscheidend, daß der Aufruf an die **lokale Transaktionsverwaltung** erfolgt.

Es ist klar, daß in einem großen, heterogenen System Information zielgerichtet übermittelt werden muß. Das heißt, daß zum Beispiel globale Anfragen zur Feststellung, wer alles an der Transaktion teilgenommen hat, zu teuer sind. Effizienter erscheint, Informationen bei ihrer Entstehung an eine koordinierende Zentrale zu übermitteln, in diesem Fall einem Koordinator für die Transaktion.

Um dies aber transparent gegenüber den partizipierenden Komponenten zu halten, definieren wir als Kommunikationspartner für einen Resource Manager oder eine Applikation die jeweils lokale Transaktionsverwaltung. Diese übernimmt dann die Aufgabe, die erhaltenen Informationen oder Aufträge an die jeweils zuständige Stelle weiterzuleiten. Dies hat in Zusammenarbeit mit dem Kommunikationssystem zu erfolgen, denn nur diese kennt die jeweilige Topologie des Systems und sein Adressierungsschema. Direkt die Transaktionsverwaltung anzusprechen, statt über das Kommunikationssystem die zuständige Transaktionsverwaltung zu ermitteln, hat seinen Sinn darin, daß wenn Transaktionen nicht oder nur zu einem geringen Teil verteilt ablaufen, häufig die lokale Transaktionsverwaltung zugleich Koordinator der jeweiligen Transaktion sein wird. Eine effizient gestaltete Methode zum Aufruf der Verwaltungsfunktionen[1] weist erhebliche Performanzvorteile gegenüber der 'indirekten' Methode auf. Wir nehmen damit in kauf, daß ein Zusammenwirken von Transaktionsveraltung und Kommunikationssystem vereinbart werden muß, um die richtige Zustellung zu realisieren.

Diesem Umstand genau wird mit dem Inhalt der GTRID rechnung getragen. Hieran kann die Transaktionsverwaltung erkennen, welcher TM welches Rechnerknotens für die betreffende Transaktion zuständig ist. Die erweiterte Funktionalität umfaßt die Möglichkeit für einen TM, einen Auftrag an den TM eines anderen Rechnerknotens weiterzuleiten; wir nennen dies *forwarding*. Dabei wird durch das Kommunikationssystem eine Nachricht generiert, die dem Auftrag an den lokalen TM entspricht, als Empfänger jedoch der zuständige entfernte TM eingesetzt ist. Dieser erhält den Auftrag, ohne wissen zu müssen, von welchem Knoten dieser Auftrag letztlich kam. Im weiteren ist nur der Absender, sprich die Kennung des den Auftrag generierenden Resource Manager von Bedeutung, denn diese ist Adresse jeder weiteren Kommunikation aus Sicht des zuständigen TMs.

[1]zum Beispiel mittels sogn. *callbacks*, siehe [RG91].

Da der Aufwand für eine lokale Nachricht sehr viel kleiner sein dürfte als der einer fernen Nachricht, können wir die Indirektion über den lokalen TM als vernachlässigbar betrachten.

Anmerkung:

> Eine Optimierung des Verfahrens stellt dar, wenn wir nicht die einzelnen Resource Manager eines Knotens betrachten, sondern den Knoten als ganzes. Das heißt, statt die Teilnahme jedes Resource Managers eines Knotens an den zuständigen, entfernten TM zu melden, könnte der lokale TM nur die **Teilnahme des Knotens** beim ersten join_transaction an den zuständigen TM melden und diese sowie jede weitere Teilnahme lokal vermerken. Beim Votieraufruf, der nun an die TMs der partizipierenden Knoten (statt an die einzelnen Resource Manager) ergeht, ist dann anhand dieser lokalen Teilnahmevermerke der Aufruf in einzelne lokale Votierungsaufrufe[2] an die Resource Manager des Knotens umzusetzen. Der Kommunikationsaufwand wird erheblich vermindert.
>
> In kauf genommen wird hierdurch jedoch das Aufheben einer Abstraktionsebene. Konnte im vorigen Ansatz der koordinierende TM einen Resource Manager direkt ansprechen, unabhängig wo er sich und ob er sich an der ursprünglichen Stelle befand, so besteht nun eine feste Bindung zwischen Rechnernknoten und dem einzelnen Resource Manager. Folge ist eine verminderte Mächtigkeit in der Fehlertoleranz des Systems. Wir können das Problem jedoch entschärfen, wenn für die lokalen Votierungsaufrufe das Adressierungsschema, sprich das Kommunikationssystem in Anspruch genommen wird. Diese Art der Kommunikation ist zwar gezwungenermaßen teurer im Vergleich zu lokalen Aufrufen (callbacks), aber eine Optimierung wird dennoch erreicht: Statt n Rechner-zu-Rechner-Nachrichten haben wir nun eine Rechner-zu-Rechner-Nachricht (der Votierungsaufruf an den Knoten) und n lokale Nachrichten (sofern die Votierungen noch lokal ergehen, d.h. alle Resource Manager sich noch am ursprünglichen Ort befinden); und da letztere erheblich billiger sein dürften, ist die Leistungssteigerung deutlich.

Halten wir fest:

- Ein Resource Manager wendet sich mit sämtlichen Nachrichten für die Transaktionsverwaltung an den lokalen TM. Dieser leitet gegebenenfalls die Information an den zuständigen, fernen TM weiter. Dazu verwendet er die Information über den *Koordinatorknoten* der GTRID.

- In einer optimierten Version melden die TMs des System lediglich die Teilnahme ihres Knotens an einer Transaktion und verwalten die Liste der an dieser Transaktion partizipierenden, lokalen Resource Manager selbstständig. Votierungsaufrufe gehen dann gebündelt an die TMs auf den jeweiligen Knoten, die diese dann in lokale Votierungen umsetzen, um schließlich eine global für ihren Knoten geltende Votierung an den koordinierenden TM zurückzugeben.

- Zur koordinierten Abwicklung des Commits einer verteilten Transaktion wird ansonsten ein offenes, nichtblockierendes 2-Phase-Commit verwendet.

[2]Die Votierungsaufrufe werden an dieser und weiteren Stellen exemplarisch verwendet. Analog gelten die Ausführungen auch für andere Interaktionen zwischen Transaktionsverwaltung und Resource Manager.

Wir haben bisher folgende Schichtung:

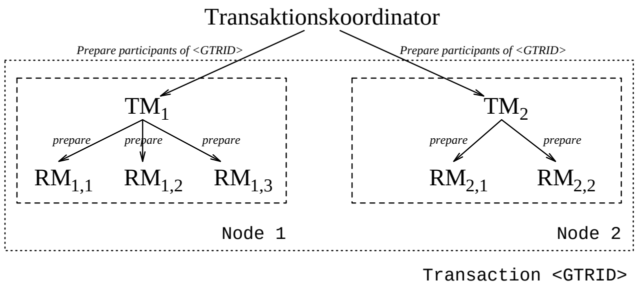


Abbildung 8.1: Koordination verteilter Transaktionen

8.3.2 Verteiltes Transaktionsbeziehungsnetz

Ein *verteiltes Netz von Transaktionsbeziehungen* setzt Transaktionen verschiedener Knoten in Beziehung zueinander. Dies kann zum Beispiel dann der Fall sein, wenn Steps von verschiedenen, im Rechnernetz verteilten SS ausgeführt werden, und die zugehörigen Transaktionen in Beziehung zueinander stehen.

Charakterisierung:

Wie gesehen, haben Operationen, wie das Eintragen von Beziehungen oder Übergangswünsche, weitere Aktionen zur Folge, sämtliche Berechnungen werden transitiv durchgeführt. Das heißt, die Transaktionen und ihre Beziehungen werden im Laufe der Berechnungen in großer Zahl und Häufigkeit angefaßt. Die Prüfung auf Konsistenz und Folgeaktionen gehört in diesem Modell zur wohl teuersten Operation bei der Koordination von Transaktionen und ihren Beziehungen.

So kann zum Beispiel auch die Commit-Entscheidung für eine Transaktion, für die Beziehungen existieren, im seltesten Fall isoliert getroffen werden. Immer werden die Zustände und Entscheidungen bezüglich weiterer Transaktionen Einfluß besitzen. Auch wird durch das Auslösen und die Struktur des Netzes eine bestimmte Auswertungsreihenfolge und –menge vorgegeben.

8.3.3 Grundsätze für die Arbeit mit verteilten Netzen

Zunächst einmal muß nicht nur für das Commit Koordination betrieben werden, sondern für sämtliche Operationen, wie Netzmodifikationsoperationen und Übergänge von Transaktionen. Offensichtlich wird der Transitivität wegen in jedem der Fälle eine *globale Sicht des TBN* benötigt.

Da schon vor Ende der Transaktion Koordinationsarbeit geleistet werden muß, kann die Entscheidung, wer Koordinator ist, nicht bis zum Schluß aufgeschoben werden. Im Gegenteil: Wir müssen den Koordinator a priori festlegen und für Alternativen im Fehlerfall sorgen.

Beziehungen und Transaktionen können sowohl lokal als auch von anderen Rechnerknoten aus in das TBN gelangen. So kann ein Resource Manager zum Beispiel neue Beziehungen eintragen. Desweiteren können auch Netze, die unter Koordination verschiedener TMs stehen, durch Eintragen von Beziehungen zu einem einzigen Netz zusammenwachsen.

Definition 8.1: *Globale Sicht eines TBN*

> ***Unter der globalen Sicht eines TBN verstehen wir die Gesamtheit aller Beziehungen und Transaktionen eines TBN, unabhängig davon, wo die jeweiligen Transaktionen oder Beziehungen ursächlich gespeichert und verwaltet werden.***

Wir müssen die globale Sicht noch weiter differenzieren: so unterscheiden wir zwischen der *virtuellen globalen Sicht* und der *materialisierten globalen Sicht*. Eine materialisierte globale Sicht für einen TM bedeutet nicht anderes, als daß das gesamte Netz mit allen Transaktionen und Beziehungen im entsprechenden Zeitpunkt in einem Knoten zur Verfügung steht. 'Verfügung' beschreibt hierbei die Tatsache, daß ab Beginn der Berechnung bis zu deren Ende keine Modifikation des Netzes auftreten kann[3].

Die virtuelle globale Sicht setzt die Speicherung bei einem TM nicht voraus. Um die transitiven Auswirkungen zu ermitteln, kann bei ihr funktional Information übermittelt werden, indem zum Beispiel Teilergebnisse bei fremden TMs in Auftrag gegeben und auch gebündelt Ergebnisse vermittelt werden. Auch könnten partitionierte Netze verschickt werden, um Teile der globalen Sicht zu materialisieren. Wir erreichen also durch erweiterte Funktionalität eine quasi globale Sicht. Auch hier gilt, daß für eine Berechnung das Netz 'eingefroren' sein muß.

8.4 Koordinatorfunktion

Zunächst soll die Frage geklärt werden, wo das TBN zur Speicherung kommt. Hier haben wir zwei Zielrichtungen, die entgegengesetzt wirken: Zum einen müssen die einzelnen Transaktionen und ihre daran partizipierenden Resource Manager koordiniert werden. Auf der anderen Seite müssen aber auch die Transaktionen untereinander mit ihren Beziehungen koordiniert werden. Das erstere bezeichnen wir als *Transaktionskoordination* , letzteres als *TBN-Koordination*. Diese zwei Koordinationsaufgaben sind nun aber keineswegs orthogonal. Die Information über einzelne Transaktionen und deren Koordination ist Grundvoraussetzung für die TBN-Koordination; sie ist deren Untermenge. Das ist in den weiteren Aufführung von Bedeutung.

Wie aus den Charakterisierungen hervorging, erscheint die Koordination von Transaktionen gegenüber der Koordination im TBN als die wesentlich billigere Komponente, da zum einen deutlich weniger Zugriffe auf die Informationen stattfinden, zum anderen auch die Häufigkeit der Koordination beträchtlich geringer ist, als bei den transitiven Berechnungen im TBN.

[3] Wir hatten eine einzelne Berechnung ja als isoliert definiert.

Mit diesen Grundlagen wollen wir die Möglichkeiten der Speicherung von TBN diskutieren:

1. Partitioniertes TBN:
   Bei einem physisch verteilten Netz werden die jeweiligen Transaktionen des Netzes vom TM des Knotens koordiniert, auf dem die Demarkierung erfolgte.

   Beziehungen zwischen Transaktionen müssen dann über Rechnergrenzen hinweg verwaltet und ausgewertet werden; die globale Sicht ist virtuell.

2. Teilweise redundantes TBN:
   Da häufig Informationen über Transaktionen benötigt werden, die von anderen Knoten koordiniert werden, werden gewisse Transaktionsinformationen dupliziert und redundant auf den verschiedenen Knoten gespeichert, so daß jeweils die benötigte materialisierte globale Sicht vorliegt.

   Schwierigkeiten bestehen darin, Änderungen an Teilen, von denen Kopien verteilt wurden, bekannt zu geben und dazu vorher eine Sperre zu errichten. Die Latenz zur Aktualisierung dürfte recht groß sein. Außerdem existiert ein Entscheidungsproblem: Welche Information wird dupliziert? Da in der Folge der transitiven Berechnung alle Transaktionen angefaßt werden können, müssen Statistiken über die Häufigkeit und über Zeitpunkte der möglichen Wiederverwendung geführt werden — ein erheblicher Aufwand.

3. Vollständige replizierte TBN:
   Deshalb wird bei diesem Ansatz das gesamte Netz auf allen involvierten Knoten physisch gespeichert. Nur müssen dann entweder alle Knoten die jeweilige Operation durchführen oder das gesamte neue Netz übernehmen; ein aufwendiges Unterfangen.

4. Partitionierte TBN mit Küste:
   Untersuchungen haben gezeigt, daß es möglich ist, Informationen über Transaktionen zu aggregieren und auf bestimmte Transaktionen zu fokussieren. Dabei werden auf den Knoten außer den lokalen auch noch genau die Transaktionen gespeichert, zu denen eine direkte Verbindung von mindestens einer der lokalen Transaktionen besteht. Bei der transitiven Berechnung wird dann bis zu genau dieser Küste berechnet, die Informationen in der Küste aggregiert und dann zur weiteren Berechnung an den fernen Knoten übermittelt. Dieses Verfahren versucht, die Vorteile der virtuellen und der materialisierten globalen Sicht zu vereinen.

   Allerdings haben die Untersuchungen auch gezeigt, daß oftmals Zyklen der Berechnung, zum Beispiel im Verlauf der Delta-Iteration, eben diese Küste, und damit Rechnergrenzen, überschreiten, was ein hohes Kommunikationsaufkommen zur Folge hat. Algorithmen, die hier eine entsprechend sinnvolle Aufteilung der Transaktionen zwischen den Rechnerknoten bestimmen, sind nicht untersucht worden; allerdings dürften diese ebenfalls teuer sein.

5. Zentrales TBN:
   Bei allen bisherigen Möglichkeiten konnten zwar Operationen primär lokal ausgeführt werden, da jedoch viele der dazu benötigten Informationen im Netz verteilt gespeichert sind, war **Kommunikation während der Berechnung** die logische Konsequenz. Im Gegensatz dazu wird hier nun das Netz zentral auf einem Knoten gespeichert und von dessen TM koordiniert. Das hat zwar zur Folgen, daß alle TBN-Operationen ferner Komponenten als Aufträge übermittelt und beantwortet werden müssen, jedoch keine Kommunikation zur Berechnung mehr notwendig ist. Lediglich die Koordination der Transaktionen muß noch über zusätzliche Nachrichten bewerkstelligt werden; hier hatten wir den optimierenden Ansatz durch Zusammenfassung zur Knoteneinheiten aber schon erwähnt (siehe 8.3.1, S. 81).

   Es ist klar, daß dieser Koordinator-Knoten eine besonders hohe Verläßlichkeit besitzen muß. Jedoch ist diese nur von diesem einen Knoten abhängig, nicht wie bei einem verteilten TBN von allen von der transitiven Berechnung betroffenen.

   Dieser Ansatz wird im folgenden verfolgt. Zum einen war dadurch die Implementierung der TBN-Verwaltung mitsamt ihren Berechnungen wesentlich einfacher und unabhängig von Kommunikation und Protokollen. Außerdem verspricht das Verfahren der zentral materialisierten Sicht weniger Kommunikationsaufwand als die anderen.

Für dieses zentrale TBN definieren wir den speichernden TM auch als Koordinator. Das Ergebnis einer Transaktion des Netzes wird ebenfalls von diesem TBN-Koordinator ermittelt, jedoch auf Basis der von fernen TMs gelieferten Informationen bezüglich der dortigen Resource Manager. Machen wir ein Beispiel zur Verdeutlichung:

Beispiel 8.1: *Koordination auf TBN im verteilten Fall*

Gehen wir von folgender Situation aus: Das Rechnernetz umfasse die Knoten R_1 und R_2. Auf Knoten R_1 sei das TBN verzeichnet, der TM des Knotens R_1 ist TBN-Koordinator. Das TBN beinhalte 2 Transaktionen, T_1 und T_2, wobei T_1 lediglich lokale Dienste, T_2 sowohl Resource Manager von R_1 wie auch von R_2 zu koordinieren hat. Zwischen den Transaktionen herrsche eine gegenseitige Commit-Beziehung ($T_1 \leftrightarrow_c T_2$). Beide Transaktionen seien aktiv.

Für T_1 werde dann der Übergangswunsch zum Commit gefordert. Anhand des Übergangsprotokolls auf TBN wird erkannt, daß dazu das Commit von T_2 notwendig ist, weshalb T_1 in den Zustand Prepared zu versetzen ist. Es wird also vom TBN-Koordinator an den Transaktionskoordinator der Auftrag „*Prepare(T_1)*“ gegeben. Daraufhin werden Votierungen von den an T_1 partizipierten und in diesem Fall ausschließlich lokalen Resource Managern erfragt. Stimmen alle dem Commit zu — sie sind somit im Zustand Prepared — wird dies als Erfolg an den TBN-Koordinator übermittelt. Die finale Commit–Meldung an die Resource Manager bleibt vorerst aus! Damit ist die Bearbeitung des Übergangswunsches vorerst abgeschlossen.

Stellt nun T_2 ihrerseits den Commit-Wunsch, wird analog das Prepare von T_2 ausgelöst und dem Transaktionkoordinator übermittelt. Dieser erkennt, daß der Knoten R_2 involviert war und übermittelt dem dortigen TM die Aufforderung zum Prepare der betreffenden Transaktion. Der TM auf R_2 erkennt nun seinerseits anhand der von ihm gespeicherten Informationen, welche Resource Manager an der zu beendenden Transaktion beteiligt waren und versetzt diese in den Zustand Prepared. Der Erfolg wird an den TM auf R_1 zurückgemeldet. Dieser hat inzwischen die lokalen Resource Manager votieren lassen, und wenn diese zustimmen und der Erfolg des Prepare von R_2 eingetroffen ist, gilt die Transaktion als im Zustand Prepared. Dies wird dem

TBN-Koordinator bekanntgegeben, woraufhin anhand des Protokolls auf Commit für T_1 und T_2 entschieden wird (Vorgehen analog).

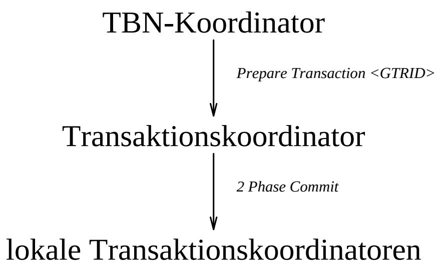


Abbildung 8.2: TBN-Koordination auf Grundlage der Transaktionskoordination

Entscheidend bei diesem Verfahren ist, daß sämtliche Berechnungen auf dem Netz lokal stattfinden können und nur die Transaktionskoordination an fremde Knoten übergeben wird. Setzen wir dort das Vorhandensein von Partizipationsinformationen voraus, kann die Übergabe der Transaktionskoordination, wie geschildert, gebündelt erfolgen. Ein Resource Manager von R_2 nimmt dann an der Transaktion T_2 teil, indem er 'seinem' TM die Teilnahme meldet. Dieser leitet den Teilnahmewunsch an den TBN-Koordinator der Transaktion — erkenntlich aus der GTRID — weiter. Wird der Teilnahme stattgegeben, vermerkt der lokale TM die Partizipation des Resource Manager an der Transaktion und bestätigt diesem die Teilnahme. Analog wird verfahren, wenn dieser Resource Manager wegen irgendwelcher Systemfehler verloren geht, also abbricht. Dies wird dem lokalen TM gemeldet, der allen Transaktionen, an denen dieser Resource Manager Anteil hatte, eine Abort-Meldung zukommen läßt. Gleichzeitig kann er diese Transaktionen lokal abbrechen, daß heißt, den an seinem Knoten involvierten Resource Managern eine Abort-Meldung übermitteln[4].

Wir müssen jedoch noch Aussagen treffen, wer Koordinator für eine Transaktion sein soll. Nun, zunächst definieren wir den TM als Koordinator für eine Transaktion, der den Auftrag zum BOT von der Applikation erhält und die GTRID generiert. Das ist im allgemeinen der zur Applikation lokale TM. In anderen Worten: Koordinator-Knoten und Heimat-Knoten innerhalb der GTRID sind zu Beginn identisch. Da wir jedoch ein physisch zentrales TBN fordern, muß beim Verschmelzen von Netzen ein neuer Koordinator für einen Teil der Transaktionen bzw. der Teilnetze bestimmt werden. Ein Wechsel der Koordinationsverantwortung impliziert den Transfer des TBN von einem Rechnerknoten zu einem anderen. Um die Menge der ausgetauschten Informationen möglichst gering zu halten, definieren wir, daß das kleinere TBN zum großeren TBN hin wechselt. Andere Kriterien wären die Auslastung, Erwartungen über weiteres Wachstum des Netzes oder auch Aufteilung der Ressourcen auf die Rechner und damit verbunden das Kommunikationsaufkommen. Eine

[4]Das obligatorische Abort durch den zuständigen TBN-Koordinator kann dann ignoriert werden.

weitere Behandlung solcher Gesichtspunkte würde aber den Rahmen dieser Arbeit sprengen.

8.5 Delegation von Koordination

Da wir schon zum Zeitpunkt der Entstehung einen Koordinator für ein TBN festlegen, müssen wir damit rechnen, daß die Koordinationsverantwortung im Laufe des Bestehens des Netzes wechselt. So wollen wir, wenn zwei Netze zu einem verschmelzen, sofort die globale Sicht materialisieren, sprich ein TBN wechselt den Rechnerknoten. Auch bei Ausfall eines TM kann die Arbeit und Verantwortung samt dem TBN auf einen anderen TM übergehen. Da im ersten Fall die Verantwortung für einen Teil, im zweiten für die gesamte Arbeit an einen anderen TM übergeht, nennen wir dies *Teildelegation* bzw. *Volldelegation*.

8.5.1 Volldelegation

Nehmen wir an, ein TM eines Rechnerknotens R_1 fällt aus. Das könnte vom dortigen System erkannt werden und anhand der persistent gemachten Informationen ein TM eines Rechnerknoten R_2 mit der Übernahme der Arbeit beauftragt werden. Die Übernahme von Verantwortung muß — so unser Modell — generell möglich sein, wie wir beim Zusammenschluß verschiedener, verteilter Netze noch sehen werden. Bei entsprechender Auslegung des *Recovery-Treibers* kann der ferne TM die Wiederherstellung der Informationen insofern bewerkstelligen, daß er in geeigneter Weise die Arbeit seines verlorengegangenen Kollegen übernehmen kann.

Sämtliche dieser Volldelegationen werden vom Betriebssystem[5] erkannt und initiiert, also niemals von einem TM in Eigenregie.

8.5.2 Teildelegation

Hingegen wird die *Teildelegation* von einem TM in Absprache mit einem anderen TM ausgelöst. Eine Teildelegation ist zum Beispiel beim Zusammenschluß von TBN notwendig, da wir explizit zu jedem Zeitpunkt die materialisierte globale Sicht fordern und es dann auch nur Sinn macht, genau einen Koordinator dafür zu benennen.

Ist zwischen den beiden TMs die Entscheidung, wer von wem nun ein TBN übernehmen soll, getroffen, müssen das Netz und die Transaktionsinformationen übertragen werden. In den zugehörigen GTRIDs muß der neue Koordinator-Knoten eingeschrieben werden. Nun taucht natürlich das Problem der Empfänger-Adresse auf. Sämtliche in die Transaktionen involvierte Knoten und Resource Manager haben in 'ihren' GTRIDs noch die 'alte' Koordinator-Adresse verzeichnet. Hier kommen nun zwei Entscheidungen zum Tragen, die wir bis dato vereinbart hatten:

1. Ein Resource Manager wendet sich immer an seinen lokalen TM. Wir müssen also lediglich allen TMs der Knoten, die in irgendeiner Weise in das zu übertragenden

[5]Hier sei das APRICOTS–Betriebssystem gemeint.

TBN involviert sind, eine Änderungsmitteilung schicken. Die so angesprochenen vermerken dies lokal und senden fortan an die 'neue' Adresse. Der Vermerk ist natürlich persistent zu machen.

2. Da wir die Koordinator-Kennung innerhalb der GTRID explizit **nicht** zur Unterscheidung von GTRIDs hinzuziehen, stimmen GTRIDs mit alter und neuer Koordinator-Adresse (die Resource Manager kennen immer noch nur die alte Adresse) auch weiterhin überein.

Der Latenz in einem großen System wegen, können natürlich noch Nachrichten beim 'alten' TM eintreffen. Dieser hat sie dem 'neuen' TM nachzusenden. Die Information über das Nachsenden kann dann vergessen werden, wenn alle involvierten Knoten die Änderungsmeldung quittiert haben. Um kennzuzeichnen, daß fortan nur noch die neue Adresse verwendet wird und keine alten Nachrichten mehr unterwegs sind, schicken die betreffenden TMs eine Nachricht zur Quittierung an die alte Adresse, alle weiteren gehen an die neue Adresse. Ist von allen TMs eine solche Quittierung beim bisherigen Koordinator eingetroffen, weiß dieser, daß keine Nachrichten bezüglich dieses TBNs mehr an ihn unterwegs sein können — wird hatten ja vorausgesetzt, daß das Kommunikationssystem die Reihenfolge der Nachrichten strikt einhält, er kann das TBN 'vergessen'.

Anmerkung:

> Bisher haben wir den Ansatz, ein TBN zu einem benannten Objekt zu machen und mit einer globalen TBN-Identifikation zu versehen, nicht verfolgt. In diesem Fall müßten dann den beteiligten TMs nicht die Änderungen aller Transaktionen des Netzen mitgeteilt werden, sondern nur eine Änderung bezüglich des TBN. Denn: Wir delegieren ausschließlich vollständige TBN, so daß nicht zwei Transaktionen eines TBN nach der Delegation zwei verschiedene TBN-Koordinatoren haben können.
>
> Allerdings bereitet dieses Verfahren dann Mehraufwand bei der Teilung von Netzen und einer daraus eventuell erfolgenden Delegation, denn zur Unterscheidung der Koordination der beiden resultierenden TBN müssen dann auch unterschiedliche TBN-Kennungen existieren. Bekanntmachung und Umspeicherung sind auch hier die Folge.

8.6 Fazit

Der hohen Kosten der Berechnungen auf dem TBN und der ebenfalls großen Häufigkeit dieser Berechnungen wegen scheint nur der Ansatz, das TBN zentral zu speichern und zu koordinieren, realistisch. In kauf genommen wird dabei, daß für die im Netz eingetragenen Transaktionen, so sie denn seblst verteilt sind, in größerem Umfang Kommunikation betrieben werden muß und daß die zentrale Koordination zu einem „Flaschenhals" werden kann.

Sämtlichen anderen Ansätze, die Replikation oder Partitionierung vorsehen, erscheinen aus Gründen der Abstimmung oder wegen hochkomplexer Entscheidungsalgorithmen als zu teuer.

Kapitel 9

Zusammenfassung und Bewertung

Die Kernfrage war, ob eine Verwaltung von Transaktionen auf der Basis von Beziehungen möglich ist und wie eine solche aussehen könnte. Es hat sich dabei herausgestellt, daß das Thema inhaltlich sehr weit gefächert ist und wohl als ziemlich komplex bezeichnet werden darf, was angesichts der beabsichtigten Mächtigkeit und Allgemeingültigkeit auch nicht weiter verwundert.

Prinzipiell ist eine solche Transaktionsverwaltung realisierbar. Es konnte gezeigt werden, daß eine allgemeine Verwaltung der Beziehungen und der Netzstrukturen ebenso möglich ist, wie ein allgemeines Übergangsprotokoll, wenngleich zunächst einmal auf logischen Beziehungen. Dazu bedurfte es nur der Erweiterung und Verallgemeinerung vorhandenen Wissens, wie zum Beispiel im Fall der 2-Phasigkeit. Jedoch konnten keine generellen Regeln für die Modellierung von Transaktionsstrukturen gegeben werden. Ebenso ist die Fehlerbehandlung in weitesten Teilen weiterhin Aufgabe der Applikation. Es ist nicht zu erwarten, daß es im Rahmen der Anwendungsprogrammierung oder gar der Transaktionsverwaltung die „allgemeingültige Fehlerbehandlung" geben wird, die anhand der Strukturen geeignete Maßnahmen trifft und durchführt. Vielmehr wird der Programmierer jeden einzelnen Fehlerfall berücksichtigen und behandeln müssen.

Hingegen konnte gezeigt werden, daß Ansätze zur Abwicklung verteilter Anwendungen durchaus vorhanden sind. Ob sich die Annahmen bezüglich Speicherung und Koordination letztendlich als richtig herausstellen, muß offen bleiben; genauere Abschätzungen und vergleichende Tests können hier vielleicht Klarheit schaffen.

Was verbleibt zu tun?

Es hat sich herausgestellt, daß die eigentliche TBN-Verwaltung durch ihre transitiven Berechnungen die mit Abstand teuerste Operation in der Transaktionsverwaltung ist. Darauf und auf neu hinzugekommene Funktionalität, wie das Löschen von Beziehungen oder die Übermittlung eines gesamten Netzes, muß verstärkt innerhalb verbesserter Algorithmen eingegangen werden. Überhaupt ist fraglich, ob sich die getroffenen Grundentscheidungen bezüglich Speicherung der Beziehungen, Aggregation und Propagation bei der weiteren Konkretisierung des Modells werden halten lassen, oder ob andere Methoden, wie z.B. das Trigger-Konzept [RG91], bessere Lösungen bieten. Ziel ist die Entwicklung einer minimalen, möglichst orthogonalen Basis von Beziehungen; darauf aufbauend müssen dann Verwaltung, Konsistenzkontrolle und auch das Übergangsprotokoll insofern erweitert werden, daß nun verschiedene Beziehungsklassen integriert werden. Dazu muß selbstverständlich die exakte Semantik jeder einzelnen Beziehung festgelegt werden und die gemachten Aussagen daraufhin überprüft werden, ob sie tatsächlich für alle Beziehungen einer Klasse oder sogar für alle Klassen gelten.

In diesem Zusammenhang sollten Gedanken an die Transparenz und Anwendbarkeit hin zu einer Architektur in ergonomischem Sinne führen, bei der der Anwender genügend Übersicht behalten kann und dennoch nicht jedes Detail kennen muß. Klare Trennung von Kompetenzen zwischen Anwendung und Transaktionsverwaltung müssen gezogen werden; hier sei beispielhaft der Kontrollfluß genannt.

Ein weiteres Gebiet für Überlegungen ist ein Konzept, TBN-Operationen zu Einheiten zusammenfassen und in zumindest geringem Maße vom gegenseitigen Erfolg abhängig machen zu können. Dies wird ganz besonders dann interessant, wenn, mittels einer weiteren Abstraktionsebene, der Anwendung vorgefertigte Transaktionskonstrukte (z.B. Sagas, Nested Transactions) zur Verfügung gestellt werden sollen; hier sollte das Eintragen aller notwendigen Beziehungen erfolgreich sein oder keine Änderung vonstatten gehen.

Im Rahmen der weiteren Entwicklung dürfte es interessant sein, Beispielanwendungen zu realisieren, um die Anwendbarkeit direkt zu überprüfen; erste Erfolge bei der Integration in APRICOTS und die ConTract–Abwicklung sind erkennbar, konnte diese doch ohne Änderung des bis dato gezeichneten Modells vorgenommen werden. Ein anderer Gesichtspunkt ist die Integration existiertender Datenbank-Produkte und deren Einbindung in das Transaktionskonzept. In diesem speziellen Falle sind geeignete Schnittstellen zu finden bzw. Anforderungen zu spezifizieren, wie ein Datenbanksystem in Zunkunft aussehen muß, um geeignet kooperieren zu können.

Zu einer Realisierung eines gesamten, ausgereiften Systems dürfte es vorerst jedoch nicht kommen. Dazu greifen die Konzepte zu sehr in die bestehenden Grundannahmen ein und bedingen die Entwicklung eines kompletten Systems, das dazu noch allgemein genug sein muß, um Korrekturen in Modell und Spezifikation zuzulassen.

Ein persönliche Anmerkung sei zum Schluß gestattet: An die Stelle einer anfänglichen Euphorie ist, ob der Breite des Themengebietes, inzwischen eine heilende Ernüchterung getreten. Zu klar sind die Restriktionen, die das Modell in seiner derzeitigen Fassung auferlegt. Heilend ist dieser Schwund an Enthusiasmus insofern, da die bisher gewonnenen Erkenntnisse auch deutlich eine Richtung und, so die persönliche Einschätzung, einen gehbaren Weg aufzeigen. Kann zudem noch deutlicher das Potential und die Anwendbarkeit in der Praxis vermittelt werden, steht einer wahrhaft interessanten Entwicklung nichts mehr im Wege.

Anhang

Index

Literaturverzeichnis

[Bur91] Holger Burkert. Systemarchitektur und Betriebsmodell für das verteilte Transaktionssystem APRICOTS, 1991. Diplomarbeit, Universität Stuttgart, Institut für Parallele und Verteilte Höchstleistungsrechner, Breitwiesenstr. 20–22, 7000 Stuttgart 80.

[CP84] Stefano Ceri, Giuseppe Pelagatti. *Distributed Databases — Principles and Systems.* McGraw-Hill computer science series, 1984.

[CRta] Panos K. Chrysanthis, Krithi Ramamritham. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. Technischer Bericht, Department of Computer and Information Science, University of Massachusetts, Submitted to 1990 ACM SIGMOD International Conference on Management of Data.

[CRR91] Panos K. Chrysanthis, S. Raghuram, Krithi Ramamritham. Extracting Concurrency from Objects: A Methodology. Technischer Bericht, Department of Computer and Information Science, University of Massachusetts, To appear in SIGMOD 1991.

[ELLR90] A. K. Elmagarmid, Y. Leu, W. Litwin, M. Rusinkiewicz. A Multidatabase Transaction Model for InterBase. In *Proceedings of the 16th VLDB Conference*, Brisbane, Australia, 1990.

[EMS91] J. L. Eppinger, L. B. Mummert, A. Z. Spector. *Camelot and Avalon: A Distributed Transaction Facility.* Morgan Kaufmann Publishers, 1991.

[GMS87] Hector Garcia-Molina, Kenneth Salem. Sagas. In *ACM Very Large Databases*, 1987.

[Gra81] Jim Gray. The Transaction Concept: Virtues and Limitations. In *Proceedings on the 7th VLDB Conference*, S. 144–154, September 1981.

[KR] Johannes Klein, Andreas Reuter. Migrating Transactions. Technischer Bericht, Universität Stuttgart, Institut für Parallele und Verteilte Höchstleistungsrechner, Breitwiesenstr. 20–22, 7000 Stuttgart 80.

[ML83] C. Mohan, B. Lindsay. Efficient Commit Protocols for The Tree of Processes Model of Distributed Transactions. In *Proceedings on the Symposium on Priciples of Distributed Computing*, S. 76–87, 1983.

[Mos81] Moss. *Nested Transactions: An Approach to reliable distributed computing*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1981. PhD thesis 260.

[MW88] Klaus Meyer-Wegener. *Transaktionssysteme*. Teubner Verlag, Stuttgart, 1988.

[Reu90] Andreas Reuter. Performance and Reliability Issues in Future DBMSs. In *Proc. Int. Symposium „DB systems in the 90s"*, Berlin, 1990.

[RG91] Andreas Reuter, Jim Gray. *Implementation of Transaction Processing Operating Systems*. Morgan Kaufmann Publishers, To appear in 1991.

[RP] K. Rothermel, S. Pappe. Open Commit Protocols for the Tree of Processes Model. Technischer Bericht 43.8909, European Networking Center, IBM.

[RW90] Andreas Reuter, Helmut Wächter. Neue Transaktionskonzepte. Technischer Bericht, Universität Stuttgart, Institut für Parallele und Verteilte Höchstleistungsrechner, Breitwiesenstr. 20–22, 7000 Stuttgart 80, Mai 1990.

[Sch91] Holger Schütz. Darstellung und Verwaltung von Beziehungen zwischen Transaktionen, 1991. Studienarbeit, Universität Stuttgart, Institut für Parallele und Verteilte Höchstleistungsrechner, Breitwiesenstr. 20–22, 7000 Stuttgart 80.

[Ske] Dale Skeen. Nonblocking Commit Protocols. In *Proceedings SIGMOD Conference*, S. 132–141.

[Wä90] Helmut Wächter. Ein allgemeines Transaktionsmodell auf der Basis von Transaktions-Beziehungen. Technischer Bericht, Universität Stuttgart, Institut für Parallele und Verteilte Höchstleistungsrechner, Breitwiesenstr. 20–22, 7000 Stuttgart 80, Februar 1990.

[Wah91] Peter Wahl. A ConTract-Manager, 1991. Studienarbeit, Universität Stuttgart, Institut für Parallele und Verteilte Höchstleistungsrechner, Breitwiesenstr. 20–22, 7000 Stuttgart 80.

[Wal84] Bernd Walter. Nested Transactions With Multiple Commit Points: An Approach to the Structering of Advanced Database Applications. In *Proceedings on the 10th International Conference on VLDB, Singapur*, August 1084.

[WBHN] Jonothan Walpole, Gordon S. Blair, David Hutchinson, John R. Nicol. Transaction Mechanism for Distributed Programming Environments. Technischer Bericht, Department of Computing, University of Lancaster, Bailrigg, U.K.

[WR90] Helmut Wächter, Andreas Reuter. Grundkonzepte und Realisierungsstrategien des ConTract-Modells. *Informatik, Forschung und Entwicklung*, 5(4):202–212, 1990.

[WR91] Helmut Wächter, Andreas Reuter. The ConTract Model, 1991. Fakultätsbericht 8/91.

[X/O90] X/Open. *Distributed Transaction Processing: The XA Specification*, x/open preliminary specification edition, April 1990.