Sicherheit für Informationssysteme

Thema: Transaktionsverarbeitung in verteilten Datenbanksystemen

Betreuung: Pavel Vogel
Bearbeitung: Andreas Hirschvogel, Tobias Krummen

Stand: Dienstag, 6. Juni 2002

Inhaltsverzeichnis:

1. Transaktionsbegriff ... 4
  1.1 Eigenschaften ... 4
  1.2 Operationen ... 5
  1.3 Zustandsübergänge einer Transaktion ... 5
2. Fehler und Fehlerbehandlung ... 7
  2.1 Fehlerarten ... 7
    2.1.1 Lokale Fehler ... 7
    2.1.2 Fehler mit Arbeitsspeicherverlust ... 7
    2.1.3 Fehler mit Festspeicherverlust ... 8
  2.2 Gegenmassnahmen durch Protokollierung ... 8
    2.2.1 Aufbau einer Log-Datei ... 8
    2.2.2 Recovery nach Arbeitsspeicherverlust ... 9
    2.2.3 Recovery nach Festspeicherverlust ... 10
3. Einführung in verteilte Transaktionssysteme ... 11
  3.1 Gründe für eine Verteilung ... 11
  3.2 Verteilungsgesichtspunkte ... 11
  3.3 Schwierigkeiten ... 11
  3.4 Anforderungen an ein Transaktionssystem ... 12
4. Verteilte Transaktionsverarbeitung ... 12
  4.1 Grundlegende Forderungen ... 12
    4.1.1 Atomarität, Wiederholbarkeit & Dauerhaftigkeit ... 12
    4.1.2 Integritätssicherung („Constistency“): ... 13
  4.2 Commit – Protokolle ... 13
    4.2.1 Zentralisiertes 2 – Phasen Commit ... 14
    4.2.2 Lineares 2 – Phasen Commit ... 15
    4.2.3 Hierarchisches 2 – Phasen Commit ... 16
    4.2.4 Presumed - Abort – Protocol ... 17
    4.2.5 Fehlerbehandlung in 2 Phasen Commit Protokollen ... 17
    4.2.6 1 Phasen – Commit ... 17
    4.2.7 3 Phasen – Commit ... 18
    4.2.2 Leistungsbewertung der Protokolle ... 19
  4.3 Synchronisation ... 20
    4.3.1 Synchronisation durch Sperrverfahren ... 20
    4.3.2 Synchronisation durch Zeitmarken ... 20
    4.3.3 Synchronisation durch optimistische Protokolle ... 21
  4.4 Deadlocks ... 21
    4.4.1 Deadlock – Verhütung ... 22
    4.4.2 Deadlock – Vermeidung ... 22
    4.4.3 Timeout ... 22
    4.4.4 Deadlock – Erkennung ... 22
5. Transaktionssysteme ... 23
  5.1 Überblick (Middleware, Realisierungen) ... 23
  5.2 TP - Monitore ... 23
    5.2.1 TP-Monitor-Architektur ... 24
    5.2.2 TP-Monitor-Funktionen ... 25
    5.2.3 System-Management und Recovery ... 29
  5.3 Verteilte Transaktionssysteme ... 30
    5.3.1 Programmierte Verteilung ... 30

5.3.2 Verteilung von Datenbank – Operationen 30
5.3.3 Transaction Routing 30
5.4 Föderative Datenbanksysteme(FDBS) 31
5.4.1 Transaktionsverwaltung in FDBS 31
6. Anwendungsbeispiel 32
7. Abschließende Betrachtung 33
6. Abbildungsverzeichnis 34
7. Literaturverzeichnis 34

1. Transaktionsbegriff

Ein Benutzer verwendet eine Datenbank, indem er eine Folge von Zugriffsbefehlen (Lese- und Schreib/Update-Befehle) auf dieser Datenbank ausführt.

Eine derartige Befehlsfolge einzelner, logisch zusammengehöriger Aktionen wird Transaktion genannt und führt die Datenbank von einem konsistenten Zustand in einen anderen konsistenten Zustand über.

1.1 Eigenschaften

Eine Transaktion muss die sogenannte ACID Eigenschaft erfüllen:

- Atomizität (**a**tomicity)
- Konsistenz (**c**onsistency)
- Isolierte Zurücksetzbarkeit/Serialisierbarkeit (**i**solation)
- Dauerhaftigkeit (**d**urability)

Atomarität besagt, dass alle zu einer Transaktion gehörenden Aktionen nur zusammen ausgeführt werden dürfen. Eine Ausführung nur eines Teils, der zu einer Transaktion gehörenden Aktionen, ist nicht zulässig. Kann eine Transaktion nicht als Ganzes ausgeführt werden, müssen die Änderungen die bereits ausgeführt wurden wieder rückgängig gemacht werden. Man nennt dies "Zurücksetzen einer Transaktion".

Konsistenz bedeutet sicherzustellen, dass die Datenbank nach Ausführung einer Transaktion weiterhin konsistent ist. Eine Transaktion soll eine Datenbank von einem konsistenten Zustand in einen anderen konsistenten Zustand überführen.

Die **Isolierte Zurücksetzbarkeit** gewährleistet, dass andere Transaktionen nicht betroffen sind, wenn eine Transaktion zurückgesetzt werden muss. Diese Forderung führt auch zu dem Begriff der Serialisierbarkeit.

Unter **Serialisierbarkeit** versteht man, dass zu einer Menge von parallel auszuführenden Transaktionen eine serielle Ausführungsreihenfolge für die gleichen Transaktionen möglich sein muss. Werden also Transaktionen parallel ausgeführt, so wird der von ihnen erzeugte Datenbankzustand nur als korrekt angesehen, wenn es mindestens eine serielle Ausführung dieser Transaktionen gibt, die den gleichen Zustand erzeugt.

Mit **Dauerhaftigkeit** ist gemeint, dass die Änderungen einer erfolgreich abgeschlossenen Transaktion dauerhaft in der Datenbank gespeichert werden und bleiben, selbst wenn z.B. das System abstürzt.

Zu der ACID-Eigenschaft werden von Transaktionen noch zwei weitere grundlegende Eigenschaften gefordert:

1. **Recovery**: Fehler, die unweigerlich auftreten, sollen behoben werden können, und es soll für die automatische Datensicherung gesorgt werden, um die Auswirkung solcher Fehler minimal zu halten.

2. **Synchronisation**: Damit die Datenbank konsistent bleibt, müssen die vielen Transaktionen auf gemeinsamen Datenbeständen im Mehrbenutzerbetrieb aufeinander abgestimmt werden.

1.2 Operationen

Die grundlegenden Operationen, die auf Datenbankebene angewendet werden, sind vereinfacht dargestellt nur Schreib- und Lese-Zugriffe. Um nun die Transaktionsverarbeitung zu steuern sind zusätzliche Operationen auf der Ebene der Transaktionen notwendig:

- **begin of transaction** (BOT): Mit diesem Befehl wird der Anfang einer Folge von Befehlen auf Datenbankebene gekennzeichnet.

- **commit**: Dieser Befehl kennzeichnet den erfolgreichen Abschluss der Transaktion. Die geänderten Daten werden in die Datenbank zurückgeschrieben und somit dauerhaft gespeichert.

- **abort**: Durch diesen Befehl wird der Abbruch der Transaktion gekennzeichnet. Das Datenbanksystem muss sicherstellen, dass die bis hierher ausgeführten Befehle der Transaktion auf Datenbankebene auf den Zustand vor dem Beginn der Transaktion zurückgesetzt werden, um die Konsistenz zu wahren.

Diese drei Befehle reichen bei klassischen Transaktionssystemen aus, doch bei Transaktionen, die über einen längeren Zeitraum andauern, ist es sinnvoll, zwischenzeitlich den Stand der Transaktion zu sichern. Auf diese Sicherungspunkte kann dann zurückgegriffen bzw. die laufende Transaktion zurückgesetzt werden, falls ein Fehler auftritt. Diese Erweiterung wird durch folgende zwei Befehle realisiert:

- **define savepoint**: Dieser Befehl definiert einen Sicherungspunkt, auf den sich die (noch aktive) Transaktion zurücksetzen lässt. Das Datenbankmanagementsystem hat nun dafür zu sorgen, daß alle ab diesem Zeitpunkt erfolgenden Operationen "`gemerkt"' werden und nicht festgeschrieben werden, da die Transaktion immer noch durch einen **abort** abgebrochen werden kann.

- **backup transaction**: Dieser Befehl setzt die Transaktion auf den Stand der zuletzt angelegten Sicherung zurück. Es hängt von der Funktionalität des Systems ab, ob der Stand auch auf zuvor angelegte Sicherungspunkte zurückgesetzt werden kann. Was allerdings mehr Ressourcen benötigt.

1.3 Zustandsübergänge einer Transaktion

Eine Transaktion befindet sich in verschiedenen Stadien, bis sie schliesslich durch ein **commit** festgeschrieben wird. Diese Verschiedenen Zustände sind in Abb.1 dargestellt und sollen hier nun erläutert werden:

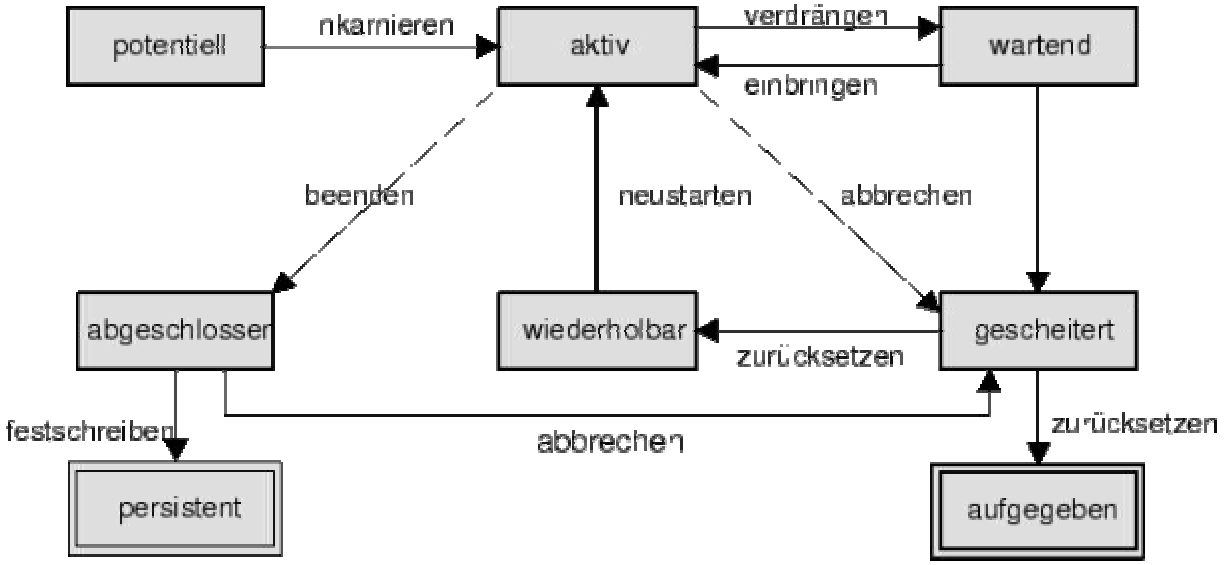

Abb1: Zustandsübergänge eine Transaktion

- **potentiell**: Die Transaktion ist zur Ausführung vorbereitet und wartet darauf zu *inkarnieren*, d.h. in den Zustand *aktiv* überzugehen.

- **aktiv**: Aktive Transaktionen werden gerade abgearbeitet und konkurrieren um die zur Verfügung stehenden Systemressourcen (Hauptspeicher, Rechenzeit)

- **wartend**: Sollte das System überlastet sein, kann die Transaktionsverwaltung einige aktive Transaktionen in den Zustand *wartend* verdrängen. Wenn das System wieder weniger ausgelastet ist, können die Transaktionen nacheinander wieder eingebracht, d.h. aktiviert werden.

- **abgeschlossen**: Durch den *commit*-Befehl wird eine aktive Transaktion beendet. Die Wirkung *abgeschlossener* Transaktionen kann aber nicht immer gleich festgeschrieben werden, da vorher gegebenenfalls noch Konsistenzprüfungen durchgeführt und entstandene Konflikte behoben werden müssen.

- **persistent**: Die Wirkungen von abgeschlossenen Transaktionen werden nach Überprüfung der Konsistenz durch *festschreiben* dauerhaft in der Datenbank gespeichert. Damit ist die Transaktion *persistent*, dies ist einer der zwei möglichen Endzustände einer Transaktion.

- **gescheitert**: Es ist eine Vielzahl von Ereignissen denkbar, die eine Transaktion daran hindern erfolgreich abzuschliessen. Ein Abbruch durch den Benutzer, Systemfehler oder bei abgeschlossenen Transaktionen festgestellte Konsistenzverletzungen können zum Scheitern der Transaktion führen.

- **wiederholbar**: Sollte eine Transaktion scheitern, so kann sie gegebenenfalls *wiederholt* werden. Dazu muss jedoch der Zustand der Datenbasis auf den Beginn der Transaktion zurückgesetzt werden, und diese dann durch einen Neustart erneut zur Wirkung kommen lassen.

- **aufgegeben**: Eine Transaktion wird jedoch aufgegeben, wenn sie auch durch mehrmaliges wiederholen nicht ausgeführt werden kann. Hier wird nun die Transaktion komplett zurückgesetzt und nimmt den Endzustand *aufgegeben* ein.

2. Fehler und Fehlerbehandlung

Transaktionen bestehen wie schon des öfteren erwähnt aus einzelnen Operationen. Da eine Transaktion aber immer als ganzes ausgeführt werden soll, gibt es vielerlei Zeitpunkte an denen Fehler auftreten können und sich nicht immer vermeiden lassen. Unvermeidbare Fehlerquellen stellen z.B. Einflüsse von aussen dar, die die Hardwarekomponenten des Systems beschädigen. Es ist aber von grösster Wichtigkeit die Konsistenz der Daten sicherzustellen. An diese Stelle treten die Recovery-Massnahmen des Datenbankmanagementsystems.

2.1 Fehlerarten

Im Grunde genommen kann man auftretende Fehler in drei Klassen unterteilen:

1. lokale Fehler mit noch nicht festgeschriebenen Transaktionen

2. Fehler mit Arbeitsspeicherverlust

3. Fehler mit Festspeicherverlust

2.1.1 Lokale Fehler

Lokale Fehler sind Fehler, die nur die Transaktion betreffen und die Datenkonsistenz der Datenbank selber nicht gefährden. Diese können sein:

- Fehler im Anwendungsprogramm

- expliziter Abbruch (*abort*) der Transaktion durch den Benutzer

- systemgesteuerter Abbruch der Transaktion, um beispielsweise eine Verklemmung zu beheben

Diese lokalen Fehler können durch ein **lokales Undo** behoben werden, wodurch die Wirkungen der gerade aktiven Transaktion, die den Fehler verursacht hat, wieder rückgängig gemacht werden.

2.1.2 Fehler mit Arbeitsspeicherverlust

Damit Datenbanksysteme schnell und effizient arbeiten können, verrichten sie ihre Arbeit natürlich im schnellen Arbeitsspeicher. Dort werden die Transaktionen durchgeführt und später dann auf die Platte geschrieben um dauerhaft gespeichert zu werden. Da der Arbeitsspeicher jedoch nur flüchtig ist, geht der Inhalt mit seinen evtl. schon abgearbeiteten Transaktionen bereits bei einen Stromausfall verloren. Das Transaktionsparadigma verlangt, dass

- (globales) **Undo**: alle durch nicht abgeschlossene Transaktionen schon in die materialisierte Datenbasis eingebrachten Änderungen rückgängig gemacht werden

- (globales) **Redo**: alle noch nicht fest gespeicherten Änderungen durch abgeschlossene Transaktionen nachvollzogen werden

Hierbei wird allerdings davon ausgegangen, dass der Festspeicher auf der die materialisierte Datenbank gespeichert ist keinen Schaden hat. Nur der konsistente Zustand braucht nicht vorhanden sein, da dieser durch das Undo und Redo wieder hergestellt werden soll.

2.1.3 Fehler mit Festspeicherverlust

Diese Art von Fehler betrifft direkt die Hardware und kann durch folgende Situationen verursacht werden:

- "head crash", der die Festplatte zerstört
- Feuer/Erdbeben
- Softwarefehler, z.B. im Plattentreiber

Diese Art von Fehlern stellt die wohl schwerwiegenste Art dar. Denn hier geht der gesamte Datenbestand verloren, falls keine Vorkehrungen dagegen getroffen wurden.

2.2 Gegenmassnahmen durch Protokollierung

Wir haben nun einen kurzen Überblick über die Möglichkeiten gesehen, die dazu führen können, dass sich eine Datenbasis nicht in einen konsistenten Zustand befinden kann. Dem kann man entgegenwirken, indem man alle Operationen die Ausgeführt werden in sogenannten Log-Dateien protokolliert. Somit können Änderungen nachvollzogen und rückgängig gemacht werden.

2.2.1 Aufbau einer Log-Datei

In der Protokoll-Datei werden alle Änderungsoperationen, welche die Transaktion durchführt mitgeschrieben. Jeder Eintrag hat folgenden Aufbau:

[LSN, TA, PageID, Redo, Undo, PrevLSN]

- Die **Redo**-Operation gibt an, wie die Änderungen nachvollzogen werden können.
- Die **Undo**-Operation gibt an, wie die Änderungen rückgängig gemacht werden können.
- **LSN** (Log Sequence Number), hiermit wird der Eintrag in der Log-Datei eindeutig gekennzeichnet. Diese Nummer steigt mit jedem neuen Log-Eintrag kontinuierlich an.
- **TA** beschreibt die Transaktionskennung, welche die Änderungen durchgeführt hat.
- **PageID** beschreibt die Speicherseite des Datenbank-Puffers, in dem die Änderungen vollzogen wurden.
- **PrevLSN** ist ein Zeiger auf den vorhergehenden Log-Eintrag der zugehörigen Transaktion.

Beispiel:

Schritt	T_1	T_2	Log
			[LSN,PageID,TA,Redo,Undo,PrevLSN]
1.	BOT		[#1,T_1,BOT,0]
2.	$r(A,a_1)$		
3.		BOT	[#2,T_2,BOT,0]
4.		$r(C,c_2)$	
5.	$a_1:=a_1-50$		
6.	$w(A,a_1)$		[#3,T_1,P_A,$A-=50$,$A+=50$,#1]
7.		$c_1:=c_1+100$	
8.		$w(C,c_1)$	[#4,T_2,P_C,$C+=100$,$C-=100$,#2]
9.	$r(B,b_1)$		
10.	$b_1:=b_1+50$		
11.	$w(B,b_1)$		[#5,T_1,P_B,$B+=50$,$B-=50$,#3]
12.	commit		[#6,T_1,commit,#5]
13.		$r(A,a_2)$	
14.		$a_2:=a_2-100$	
15.		$w(A,a_2)$	[#7,T_2,P_A,$A-=100$,$A+=100$,#4]
16.		Commit	[#8,T_2,commit,#7]

Abb2: Verzahnte Ausführung zweier Transaktionen und das erstellte Log

Hier betrachten wir die verzahnte Ausführung zweier parallel laufender Transaktionen T_1 und T_2. Zu beachten ist hier, dass die Log-Einträge für die Operationen **BOT** und **commit** nur aus LSN, der Operation und der PrevLSN bestehen. PrevLSN hat bei der Operation BOT natürlich den Wert 0, da sie den Beginn der Transaktion darstellt und vorher kein Eintrag vorhanden sein kann. Die Befehle $r(X,x_i)$ stellt einen Lese-Zugriff des Wertes X in die Variable x_i und der Befehl $w(X,x_i)$ den zugehörigen Schreib-Zugriff der Variable x_i in den Wert X dar.

Wenn wir den Log-Eintrag Nummer drei betrachten, sehen wir, dass dieser sich auf die Transaktion T_1 und die Seite P_A bezieht. Soll nun die Aktion wiederholt werden, also das sogenannte **Redo** ausgeführt werden, muss A um den Wert 50 erniedrigt werden. Genau andersherum verhält es sich mit dem zurücksetzen, dem **Undo**, hier soll A um den Wert 50 erhöht werden um den Ursprünglichen Zustand von A wieder zu bekommen. Der zugehörige vorherige Log-Eintrag trägt die Nummer 1.

Diese Art von Speicherung, wie die Operationen in der Log-Datei protokolliert sind, nennt man *logische* Protokollierung. Es sind beide **Redo** und **Undo** Operationen relativ zum Wert aufgeführt. Auf der anderen Seite steht die *physikalische* Protokollierung, in der die Werte für die **Redo** und **Undo** Operationen direkt gespeichert werden. Diese Speicherbilder werden *Images* genannt. In unserem Beispiel hiesse das beim dritten Eintrag, hat A vorher den Wert 1000, so würde dies der **Undo**-Wert sein und 950 der **Redo**-Wert.

2.2.2 Recovery nach Arbeitsspeicherverlust

Tritt nun bei der Arbeit mit der Datenbank ein Fehler mit Arbeitsspeicherverlust auf, so gibt es zwei Arten von Transaktionstypen, die durch die Recovery-Massnahmen bearbeitet werden müssen:

- **Winner**: Dies sind Transaktionen, die erfolgreich abgeschlossen sind und wiederholt werden müssen.

- **Loser**: Transaktionen die während ihrer Ausführung unterbrochen wurden. Diese müssen in ihrer Wirkung rückgängig gemacht werden.

Die Herstellung der Konsistenz der Datenbank geschieht nun in drei Phasen:

1. **Analyse**: In dieser ersten Phase wird die Log-Datei nach den Transaktionen durchforstet, die in dem entsprechenden Zeitraum durchgeführt wurden. Dies geschieht mit den **BOT**-Einträgen im Protokoll. Ist die Transaktion mit einem **commit** abgeschlossen, so zählt sie zu den *Winnern*. Sollte kein zugehöriger Abschluss gefunden werden, wird die Transaktion zu der Menge der *Loser* gezählt.

2. **Redo-Phase**: Hier wird die Protokoll-Datei von vorne durchlaufen, und nochmals ausgeführt. Es soll der Zustand bis zum Zeitpunkt des Fehlers wiederhergestellt werden.

3. **Undo-Phase**: Da nun die Datenbank auf dem neuesten Stand gebracht wurde, müssen jetzt noch jene Transaktionen rückgängig gemacht werden, die sich in der Menge der *Loser* befinden. Dazu wird jetzt die Log-Datei rückwärts durchlaufen und alle *Winner*-Transaktionen ausser Acht gelassen. Hier macht sich die Verwendung der *PrevLSN* bemerkbar, da mit ihrer Hilfe die Datei sehr schnell und effizient durchlaufen werden kann.

2.2.3 Recovery nach Festspeicherverlust

Die Wiederherstellung nach einem Fehler mit Arbeitsspeicherverlust, setzt voraus, dass die Datenbank an sich und die Protokoll-Datei physisch intakt sind. Bei einen Fehler, der jedoch diese Dateien zerstört, muss auf Archiv-Kopien zurückgegriffen werden. Es gibt dann zwei Arten, wie der Recoverymechanismus arbeiten kann.

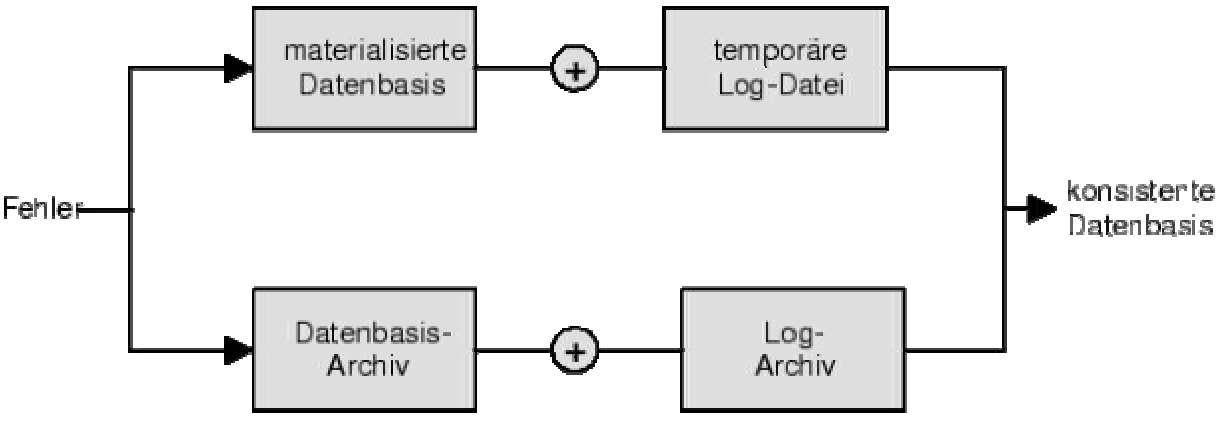

Abb3: Zwei Recoveryarten

- Mit einer redundanten Kopie der materialisierten Datenbasis und einer temporären Log-Datei. Dies stellt die schnellere Art dar, eine konsistente Datenbasis wiederherzustellen. Hierbei sind die Datenbasis und die Log-Datei physisch noch vorhanden.

- Mit einer Kopie aus dem Datenbasis-Archiv und dem zugehörigen Log-Archiv. Dieser langsamere Pfad muss bei zerstörtem Festspeicherinhalt gewählt werden.

3. Einführung in verteilte Transaktionssysteme

3.1 Gründe für eine Verteilung

Am Anfang steht die Frage, wieso man Transaktionssysteme, bzw. die Daten überhaupt verteilt. Es gibt Gründe aus vielen Bereichen, die dafür sprechen:

- Wirtschaftlich gesehen sind primär die reduzierten Hardwarekosten zu nennen. Es ist nämlich oft billiger viele kleine und mittlere Systeme, statt einem Großen zu verwenden. Kleine Systeme bieten heute zu einem relativ geringen Preis bei guter Verfügbarkeit eine sehr große Leistung. Zum anderen können natürlich bereits bestehende Kommunikationssysteme (Netzwerke) zur Kommunikation zwischen den einzelnen Systemen verwendet werden.
- Organisatorisch müssen die Systeme nicht zwingend homogen sein. Viele Systeme erlauben eine heterogene Umgebung, was für eine lokale Autonomie und die Unterstützung lokaler Strukturen sehr gut ist. Damit können mit relativ geringem Aufwand bestehende Datenbanken integriert werden.
- Technisch gesehen erhöht sich die Leistung eines solchen Systems durch die parallele Ausführung vieler Anfragen auf mehreren Systemen. Zum anderen bietet ein verteiltes Transaktionssystem eine hohe Skalierbarkeit, da der Betreiber je nach Auslastung und Finanzkraft neue Rechner zum bestehenden Verband hinzufügen kann. Die Verteilung bietet aber auch eine höhere Verfügbarkeit und Zuverlässigkeit, da es in unterschiedlicher Entfernung andere Systeme die die Funktion eines Ausgefallenen übernehmen kann. Im Idealfall kann dieser Ausfall dem Benutzer komplett verborgen werden.

3.2 Verteilungsgesichtspunkte

Daten und Funktionalität können nach unterschiedlichen Gesichtspunkten verteilt werden.

- Nach der Funktionalität: Meist werden nicht an allen Standorten eines großen Unternehmens die gleichen Daten bzw. die gleichen Funktionen benötigt.
- Verteilung nach der Last: Dies wird heute meist bei Webanwendungen praktiziert. Häufig nachgefragte Dienste oder Daten werden auf mehrere Rechner verteilt, um eine schnelle Abarbeitung der Anfragen zu ermöglichen.
- Nach Sicherheitskriterien: Die Daten oder Funktionen können an verschiedenen Standorten vorgehalten werden. Dadurch erhöht sich zwar der Aufwand durch die nötigen Replikationsverfahren (Vortrag 5), aber im Falle eines Ausfalls sind die Daten an einem anderen Ort noch redundant vorhanden. So können eventuelle Ausfallzeiten und Datenverlust vermieden oder ganz verhindert werden.

3.3 Schwierigkeiten

Durch die Verteilung ergeben sich einige Schwierigkeiten.

- Wie bei Datensystemen üblich, müssen Redundanzen vermieden werden. Das ist in verteilten Systemen sehr komplex, da sich die Daten an unterschiedlichen Standorten befinden.
- Ein Konsistenzverlust, der durch den Ausfall von Teilen des Rechensystems entstehen kann, muss verhindert oder wieder rückgängig gemacht werden.

- Komplexe Ablaufkontrolle: Dazu mehr im nächsten Kapitel. Allgemein müssen Methoden gefunden werden, um die Abläufe auf unterschiedlichen Systemen zu koordinieren.
- Erhöhte Komplexität bei der Programmerstellung. Das ergibt sich daraus, dass die Verteilung und Ihre Funktionalitäten implementiert werden müssen. Eventuell muss der Einsatz in heterogener Umgebung und auf unterschiedlichen Plattformen bedacht und integriert werden.

3.4 Anforderungen an ein Transaktionssystem

Die Anforderungen an ein verteiltes Datenbanksystem sind weitgehend identisch mit den Gründen die für ein verteiltes Datenbanksystem sprechen (siehe oben). Zusätzlich zum oben genannten muss noch Verteilungstransparenz gefordert werden. Der Benutzer soll sich ja nicht damit beschäftigen müssen, auf welchem Teil der Datenbank er gerade arbeitet. Auch eine einfache Systemadministration muss gewährleistet werden. Zwar muss man hier mit einer Verkomplizierung gegenüber zentralen Systemen rechnen, doch zur Komplexitäts-begrenzung sollte es möglich sein, die Administrationsaufgaben zu automatisieren oder zumindest durch entsprechende Tools zu unterstützen und zu vereinfachen. Für ein geographisch verteiltes System ist jedoch noch mal mit einer Erhöhung des Aufwands zu rechnen, da meist die Administration für jedes System einzeln abzuwickeln ist.

4. Verteilte Transaktionsverarbeitung

4.1 Grundlegende Forderungen

Wie für alle Transaktionen müssen wir auch für die verteilten Transaktionen bestimmte Forderungen stellen, um deren korrekte Funktion gewährleisten zu können. Zum einen müssen die schon bekannten ACID - Bedingungen (**A**tomarität, **C**onsistency, **I**ntegrität, **D**auerhaftigkeit) erfüllt werden. Darüber hinaus muss noch für eine Synchronisation der Anfragen, für das Logging der Transaktionen, für eine eventuelle Recovery der Datenbank nach einem Absturz, und die Vermeidung oder Verhinderung von Deadlocks gesorgt werden.
Diese Anforderungen sollen durch verschiedene Commit – Protokolle und die Transaktions-Verwaltungen sichergestellt werden. Diese Protokolle sollen vom Leistungsgesichtspunkt her natürlich möglichst wenige Nachrichten zur Koordination und möglichst wenige Schreibvorgänge auf die Log – Dateien benötigen, da beide Faktoren ja die potentielle Antwortzeit für eine Transaktion verlängern und zu mehr Overhead führen.

4.1.1 Atomarität, Wiederholbarkeit & Dauerhaftigkeit

Um bei einer verteilten Transaktionverarbeitung die Atomarität der Datenbank – Operationen gewährleisten zu können, muss man sich vor einer globalen Transaktion aller an dieser Transaktion beteiligten Knoten zunächst für eines der möglichen Protokolle entscheiden. Die Transaktion muss entweder an allen Knoten erfolgreich sein (COMMIT), oder aber an allen Knoten abgebrochen werden (ABORT), d.h. es ist ein Commit – Protokoll nötig, das die Korrektheit der Transaktion trotz der Verteiltheit unseres Systems verwirklicht.
Dabei soll es dann aber die Dauerhaftigkeit der Transaktionen auch im Fehlerfall sicherstellen. Das erfordert die Umsetzung eines einheitlichen Logging Konzepts, das in die Commit – Protokolle integriert sein muss. Das Logging nimmt hier deshalb eine so zentrale und wichtige Rolle ein, da für eine effiziente und genaue Recovery nach einem Absturz oder bei auftretenden Fehlern möglichst genaue Zustandsberichte über den letzten Zustand des

Datenbanksystems benötigt werden. Dieser Zustand lässt sich nur über den Inhalt der Log – Datei rekonstruieren. Aus diesem Grund ist ein gutes Logging – Konzept unabdingbar für ein gutes Commit – Protokoll.
Ganz klar ist, dass eine gleiche Anfrage, die zweimal ausgeführt wird, beide Male dasselbe Ergebnis bringen soll. Andernfalls wäre die Arbeit des Datenbanksystems nicht nachvollziehbar.

4.1.2 Integritätssicherung („Constistency“):

Die 2. der vier ACID – Eigenschaften. Eine Änderungstransaktion darf nur zum Commit kommen, falls alle Integritätsbedingungen (Wahrung der Primärschlüsseleindeutigkeit und referentiellen Integrität in rel. Datenbanken) erfüllt sind. Die Überwachung kann weitgehend wie im zentralen Fall gehandhabt werden. Nur bei den Fällen, in denen die Integritätsbedingungen mehrere Relationen betreffen und somit erst verzögert am Ende der Transaktion überprüft werden können, erfolgt die Überprüfung im 2-Phasen – Commit in der 1. Phase. Es gehen also nur solche Knoten in den Prepared – Zustand, die die Integritätseigenschaften erfüllen.

4.2 Commit – Protokolle

Im Folgenden werden einige Commit – Protokolle, die in verteilten Transaktionen zum Einsatz kommen können, vorgestellt.
Generell ist an jedem Knoten/ Rechner in einem verteilten Transaktionssystem ein Transaktionsmanager nötig, der den Ablauf der Transaktion überwacht. Am Heimatknoten, also dem Knoten an dem eine Transaktion startet, kommt diesem auch die Aufgabe des *Commit – Koordinators* zu. Dieser muss die lokalen Commit – Ergebnisse der beteiligten Knoten ermitteln und zu einem globalen Commit – Ergebnis zusammenführen. Alle Systeme, auf denen eine Sub – Transaktion ausgeführt wurde, werden als *Agenten* bezeichnet. Zur Kommunikation zwischen dem Koordinator und den Agenten können zentralisierte (Koordinator kommuniziert mit jedem Agenten direkt), lineare (Sequentialisierung des Commit – Ablaufs) oder hierarchische Ablaufstrukturen (allgemeinste Form) zum Einsatz kommen.

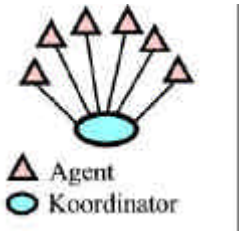

Abb4: Modell Koordinator – Agent

An ein geeignetes Commit – Protokoll werden einige Anforderungen gestellt. Das Protokoll muß korrekt sein, einen geringen Aufwand sowie eine geringe Antwortzeitverlängerung haben und robust gegenüber Rechnerausfällen und Kommunikationsfehlern sein. Zuletzt soll noch die Knotenautonomie erfüllt sein, indem jeder Rechner noch möglichst lange Zeit haben soll eine globale Transaktion von sich aus abzubrechen, damit sich die Datenbank nicht in einem inkonsistenten Zustand befindet, falls der COMMIT – Fall auf einem lokalen Knoten nicht erreicht werden kann.
Da wir aber da von ausgehen, dass unser Commit – Protokoll keine (oder sehr wenige) Fehler macht, ist vor allem der Nicht – Fehler – Fall zu optimieren.

In einem Commit – Protokoll unterscheidet man nach der Anzahl der Phasen, die es zum Ablauf einer Transaktion benötigt. Im Folgenden soll das zentrale 2 Phasen – Commit relativ ausführlich beleuchtet werden, um die grundlegenden Commit – Konzepte in einer verteilten Umgebung zu erklären. Es folgen die übrigen 2 Phasen – Commit – Protokolle und nach einem kurzen Einschub zur Fehlerbehandlung in 2 Phasen Protokollen, werden noch das 1 – Phasen und das 3 – Phasen Commit vorgestellt.

4.2.1 Zentralisiertes 2 – Phasen Commit

Das Basisprotokoll ist das zentralisierte 2 - Phasen Commit. Dieses Protokoll soll im Folgenden ausführlich vorgestellt werden. Danach soll die Effizienz genau bewertet werden.

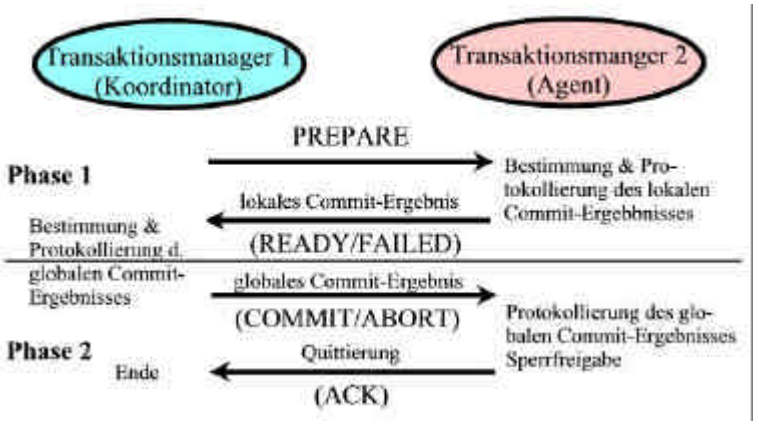

Abb5: Zentralisiertes 2 – Phasen Commit

1. Zu Beginn der 1. Phase sendet der Koordinator am Transaktionsende eine PREPARE Nachricht an alle Agenten, um das jeweilige lokale Commit - Ergebnis zu erfahren.
2. Nachdem die Sub-Transaktion erfolgreich durchgeführt wurde, sichert der Agent die Wiederholbarkeit zu. Dazu schreibt er noch ungesicherte Daten und einen PREPARE - Satz in die Log Datei. Eine READY - Nachricht wird an Koordinator gesandt. Scheitert die Sub-Transaktion, wird ein Abort-Satz in die Log-Datei geschrieben und ein FAILED zum Koordinator gesandt. Daraufhin setzt der Agent die Sub-Transaktion zurück und hebt die Sperren auf, falls Sperren zur Synchronisation verwendet werden. Da die globale Transaktion scheitert, wird die Sub-Transaktion hier beendet.
3. Wenn alle Antwortnachrichten der Agenten beim Koordinator eingetroffen sind, ist Phase 1 beendet. Haben alle mit READY geantwortet, schreibt der Koordinator einen Commit - Satz in die lokale log-Datei. Damit gilt die globale Transaktion als beendet. Gleichzeitig sendet er eine COMMIT Nachricht an die Agenten. Wenn mindestens ein Agent mit FAILED geantwortet hat, schreibt der Koordinator einen Abort-Satz und sendet eine ABORT – Nachricht an die Knoten, die vorher nicht mir FAILED geantwortet haben.
4. Der Agent schreibt nach dem Eintreffen des COMMIT ebenfalls einen Commit - Satz in die Log-Datei und gibt anschließend die Sperren frei. Bei einer ABORT - Nachricht, wird ein Abort-Satz geschrieben und die Sub-Transaktion zurückgesetzt. Sperren werden ebenfalls freigegeben. Zu Bestätigung sendet er noch eine Quittung(ACK - Nachricht) an den Koordinator. Damit ist die globale Transaktion beendet. Wird durch einen Ende-Satz in der log-Datei des Koordinators vermerkt.

Fällt in diesem Verfahren der Koordinator aus, kann es zu langen Blockierungen kommen.
Zur Leistungsbewertung kann man hier hauptsächlich über die Anzahl der Schreibvorgänge und die Anzahl der Nachrichten die gesendet werden müssen messen.
Die Zustandsübergänge von Agent und Koordinator können folgendermaßen dargestellt werden.

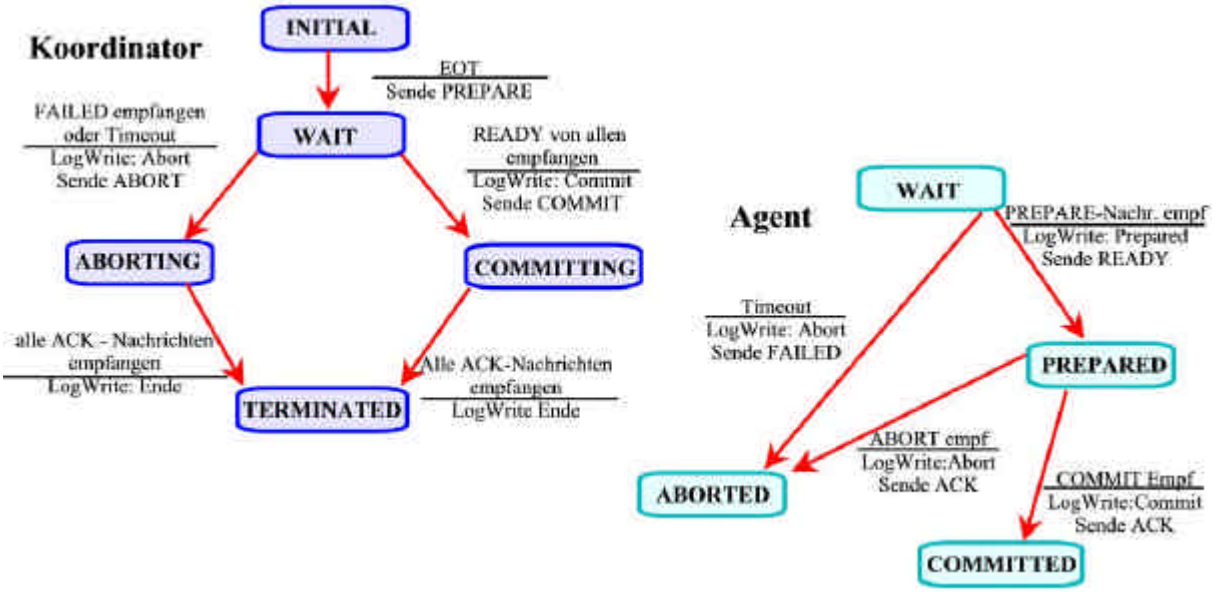

Abb6: Zustandsübergänge von Koordinator und Agent

Im vorliegenden Protokoll können wir für die Kommunikation zwischen Koordinator und Agent pro Agent 4 Nachrichten zählen. Damit bekommen wir für N Knoten 4 * (N – 1) Nachrichten. Pro Agent und Koordinator haben wir zwei Schreibvorgänge in die Log – Datei, die bis auf den „Ende“ Log – Eintrag synchron ausgeführt werden können. → 2 * N – 1 synchrone Log – Vorgänge.

4.2.2 Lineares 2 – Phasen Commit

Hier ist die Commit – Behandlung sequentialisiert.

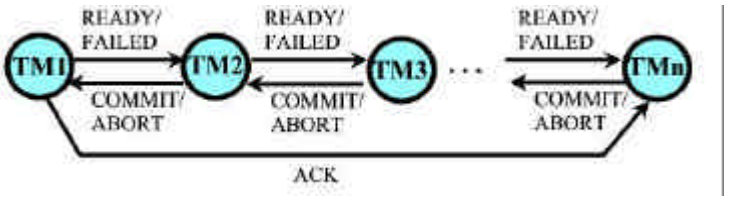

Abb7: Lineares 2 – Phasen Commit

Hier unterscheidet sich die Kommunikationssequenz in den Commit – Phasen. In der 1.Phase läuft die Kommunikation vom Koordinator zum n. Agenten. In der 2.Phase wird der Weg rückwärts gegangen. Dabei bringt sich ein Agent nach Eintreffen der READY – Nachricht selbständig in den „Prepared“ – Zustand. Anschließend gibt er das READY Signal an den nächsten Agenten weiter, falls ein Fehler aufgetreten ist, gibt er das FAILED – Signal weiter. Wenn der letzte Agent das READY – Signal empfangen hat, schreibt dieser das globale Commit der gesamten Transaktion in die Log – Datei und meldet das Ergebnis in umgekehrter Reihenfolge an alle Knoten, die daraufhin ihr lokales Commit protokollieren und die Sperren freigeben. Am Ende der Transaktion schickt der 1. Knoten noch eine Quittung („ACK – Nachricht“) an den letzten Agenten.
Hat einer der Knoten in der 1. Phase ein „FAILED“ gemeldet, so ist für alle nachfolgenden Knoten das Commit – Ergebnis ebenfalls FAILED. Der letzte Agent empfängt diese Nachricht und erklärt die ganze Transaktion für gescheitert.

Daraus kann man sehen, dass hier eigentlich der letzte Agent die Rolle des Koordinators einnimmt, weil er letztendlich darüber entscheidet, ob die Transaktion Erfolg hat oder nicht.
Wie man leicht sehen kann fallen hier die PREPARE – Nachrichten an alle Knoten weg. Damit hat man ein halbiertes Kommunikationsaufkommen. Des Weiteren fallen N-2 ACK – Nachrichten weg, weil die Quittung ja nur vom ersten zum letzten Knoten gemeldet wird.
Insgesamt sind hier also 2 * N -1 Nachrichten zur Kommunikation nötig.
Andrerseits erhält man für eine große Anzahl von beteiligten Knoten durch die Sequentialisierung des Vorgangs sehr hohe Antwortzeiten. Aus der gesammelten Erfahrung sieht man aber, dass meist 2 Knoten an einer Transaktion beteiligt sind. Damit benötigt man drei Nachrichten zur Kommunikation, was effizient erledigt werden kann.

4.2.3 Hierarchisches 2 – Phasen Commit

Dies ist die Verallgemeinerung des Basisprotokolls. Dabei kommuniziert jeder Knoten nur mit seinem direkten Vorgänger und seinem direkten Nachfolger in der Hierarchie. Die Schachtelungstiefe ist hier egal.

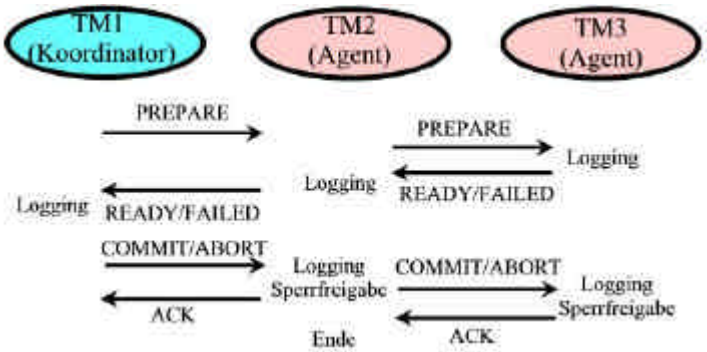

Abb8: Hierarchisches 2 – Phasen Commit

Durch seine Kommunikationsabfolge kann dieses Protokoll sehr gut in Umgebungen eingesetzt werden, in denen nicht jeder Knoten jeden anderen kennt. Jeder Zwischenknoten nimmt dabei die Rolle eines Zwischenkoordinators ein. Gegenüber seinem Vorgänger spielt er die Rolle eines ganz normalen Agenten, für seine Nachfolger ist er der Koordinator. Zunächst wird die PREPARE Nachricht vom Koordinator zum letzten Knoten versandt. Das Logging erfolgt auf dem Rückweg. In der 2.Phase wird das COMMIT bzw. ABORT – Signal, beginnend beim Koordinator, vom Vorgänger zum Nachfolger versandt. Im Nachfolgerknoten erfolgt sofort die Protokollierung und die Sperrfreigabe, sowie die Weitergabe des Signals an seine Nachfolger. Anschließend versendet er seinem Vorgänger das ACK – Signal und ist fertig. Dabei schreibt jeder Zwischenkoordinator ebenfalls einen Ende – Satz in das Protokoll.
Wie man unschwer sehen kann ist das Protokoll sehr allgemein und auch sehr flexibel gehalten worden. Das geht ganz klar auf Kosten der Parallelität und damit der Leistungsfähigkeit: es ist eine höhere Bearbeitungsdauer nötig. Zusätzlich ist für jeden Zwischenknoten noch ein zusätzlicher Logging – Vorgang durch das Schreiben des Ende-Satzes nötig.
Trotz seiner mangelnden Schnelligkeit, vor allem bei großer Schachtelungstiefe, wird das Protokoll sehr häufig verwendet, vor allem in der Form des „Presumed-Abort-Protocol“, das in der Schnelligkeit optimiert wurde. So ist es die Basis für Transaktionsverarbeitung nach ISO/OSI und nach X/Open, welche als Standards relativ etabliert sind.

4.2.4 Presumed - Abort – Protocol

Bei diesem Protokoll handelt es sich um eine Optimierung des hierarchischen 2 – Phasen – Commit Protokolls. Zum Einen wird hier der ABORT – Fall optimiert, indem keine ACK – Nachrichten bei einem Abort versendet werden und keine synchrones Logging stattfindet.
Default - mäßig wird hier ABORT angenommen, falls keine anderen Angaben im Log gefunden werden.
Zum Anderen erfolgt eine Optimierung für lesende Teiltransaktionen. Hier erfolgt kein Logging und die Sperrfreigabe bereits in der 1. Phase. Auch wird die Kommunikation für die zweite Phase einfach umgangen.

4.2.5 Fehlerbehandlung in 2 Phasen Commit Protokollen

Es gibt hier Timeout – Bedingungen sowohl für den Koordinator als auch für die Agenten.
Tritt z.B. ein Timeout auf, wenn sich ein Agent gerade im PREPARED Zustand befindet, so ist dies darauf zurückzuführen, dass entweder die Kommunikation nicht richtig geklappt hat, oder der Koordinator ausgefallen ist. Der Agent versuch auf alle Fälle zunächst beim Koordinator, und wenn dies fehlschlägt, bei manchen Protokollen, bei einem anderen Agenten festzustellen, ob ein globales COMMIT oder ein ABORT erfolgt ist. Tritt bei einem Agenten oder einem Koordinator im WAIT Zustand ein Timeout auf, dann löst der Agent ein unilateral ABORT aus und setzt die Teiltransaktion zurück, der Koordinator setzt die Transaktion komplett zurück und verschickt global ABORT - Nachrichten.
Fällt der Koordinatorknoten aus, kommt die Reaktion auf den momentanen Log-Zustand an.
Ist der Zustand TERMINATED, d.h. abgeschlossen, wird je nach Transaktionsausgang eine UNDO oder REDO – Recovery durchgeführt, so dass keine „offenen“ Teiltransaktionen möglich sind.
Ist der Zustand ABORTING, d.h. im Abbruch, so wird eine UNDO – Recovery eingeleitet und ein ABORT an alle Rechner verschickt, die noch kein ACK an den Koordinator gesandt haben.
Ist der Zustand COMMITTING, also vor dem erfolgreichen Abschluss, erfolgt eine REDO – Recovery und ein COMMIT an alle Rechner, die noch keine Quittung geschickt haben.
In allen anderen Fällen wird einfach eine UNDO – Recovery eingeleitet.
Wenn ein Agent ausfällt, dann wird im Falle des COMMIT - Zustandes eine REDO - Recovery eingeleitet, wenn der ABORT – Zustand bestand oder noch kein 2 – Phasen - Commit Satz in der Log Datei steht, erfolgt ebenfalls ein UNDO. War der Zustand PREPARED, so erfolgt die Anfrage an den Koordinator nach dem Ausgang der Transaktion (Dieser hält die Information noch vor, da noch kein ACK erfolgte).

4.2.6 1 Phasen – Commit

Leider entfällt bei den 2 – Phasenprotokollen vor allem bei kurzen verteilten Transaktionen ein Großteil der Bearbeitungszeit auf die Commit – Behandlung.

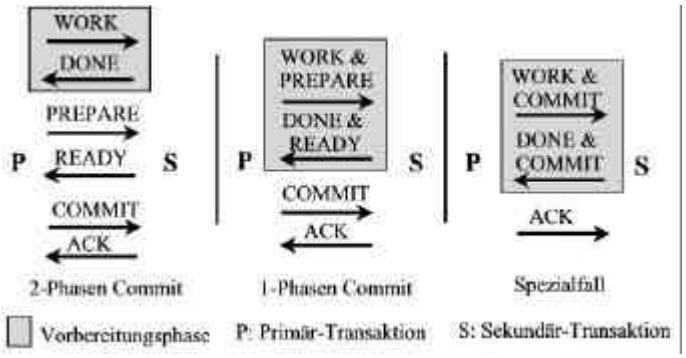

Abb9: Vergleich 1- Phasen und 2 – Phasen Commit

Hier ist es rentabel die PREPARE – Operation mit dem WORK – Aufruf zu kombinieren, der normalerweise dem eigentlichen Commit – Protokoll vorangeht. Nach dessen Durchführung geht der Knoten automatisch in den Prepared – Zustand. Damit konnte die eigentlich 1. Phase eingespart werden. Die eigentliche Commit – Bearbeitung umfasst nur noch eine Phase zur Mitteilung des Ergebnisses.
Von der Leistung ist es ebenfalls besser als die 2 – Phasen – Protokolle: Es können nämlich pro beteiligten Agent 2 Nachrichten eingespart werden. Im Spezialfall N = 2 Knoten kann eine weitere Nachricht gespart werden, wenn man dem WORK – Aufruf gleichzeitig die Commit – Koordination mit übergibt. Der Agent trifft dann unmittelbar nach der Ausführung der lokalen Sub – Transaktion die globale Commit – Entscheidung und teilt diese mit einem „DONE“ seinem Koordinator mit. Dann wird nur noch eine ACK – Bestätigung an den neuen Commit – Koordinator zurückgesendet.
Allgemein ist die Effizienz 2 * (N-1) Nachrichten.
Jedoch hat das Verfahren einige Nachteile. Die Operationen geben hier ihr Recht auf Transaktionsabbruch frühzeitig aus, so kann es zu einer langen Phase der Unsicherheit über den Transaktionsausgang kommen, da keiner so genau weiß, ob die Transaktion abgebrochen werden muss, oder nicht. Zum Anderen besteht hier eine hohe Abhängigkeit vom Koordinatorknoten und es ist mehrfaches Logging des READY – Zustands pro Transaktion und pro Knoten möglich.
Durch diese Schwachstellen, ist das 1 – Phasen Commit – Protokoll nur bedingt für die Realisierung eines sicheren Transaktionssystems geeignet. Zwar wird es wie oben schon gesagt hauptsächlich für verteilte Transaktionssysteme verwendet, doch sind die genannten Nachteile ein deutliches Argument gegen den Einsatz in Informationssystemen, die in sicherheitskritischen Bereichen ihre Anwendung haben.

4.2.7 3 Phasen – Commit

Der große Schwachpunkt bei den 2 Phasen – Protokollen ist die Abhängigkeit vom Koordinator, was vor allem bei sicherheitskritischen Anwendungen ein Problem sein kann.
Wenn nämlich der Koordinator während dem Prepared – Zustand ausfällt, blockiert das ganze Transaktionssystem bis es zum Timeout kommt.

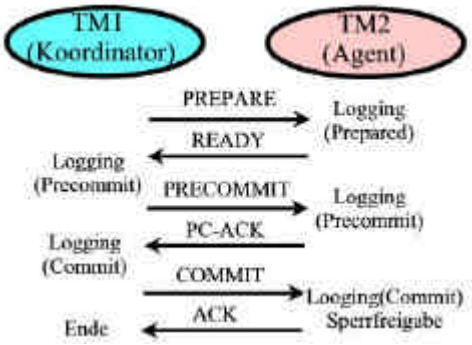

Abb10: 3 – Phasen Commit

Abhilfe schafft hier ein nicht blockierendes Commit – Protokoll. Für die korrekte Funktion dieses Protokolls, auch im Fehlerfall, sind aber zwei Voraussetzungen zu erfüllen:

- Es darf keine Netzwerk – Partitionen geben (Erklärung: siehe unten!)
- Es dürfen höchstens K < N Rechner gleichzeitig ausfallen, wobei K ein Verfahrensparameter ist.

Der ABORT – Fall wird hier wie im 2 Phasen – Protokoll behandelt. Es entsteht nur zusätzlicher Aufwand, wenn alle Agenten am Ende der 1. Phase (identisch zum 2 – Phasen Commit) mit READY stimmen.
In der 2. Phase folgt dann ein PRECOMMIT zu den Agenten sowie deren Antwort, das PRE - ACK. Dies wird mit einem entsprechenden Log – Satz protokolliert. Sobald K der N Agenten mit PRE – ACK ihre Bereitschaft gemeldet haben, wird auf COMMIT entschieden und an alle Agenten verschickt.
Die letzte Phase ist wieder identisch zum 2 – Phasen Protokoll.
Mit dem Precommit sichert der Koordinator den Agenten zu, dass er die Transaktion von sich aus nicht mehr abbricht. Natürlich kann noch der Fall eintreten, dass der Koordinator durch andere Umstände ausfällt. Dann ist noch ein Abbruch möglich. Es folgt dann die Bestimmung eines neuen Koordinators. Dieser erfrägt zuerst den Zustand aller Agenten und setzt die Transaktion am Abbruchpunkt fort.
Natürlich muss man diese höhere Robustheit mit einem entsprechenden Kommunikations-aufwand „bezahlen“: Es sind 6 * (N – 1) Nachrichten sowie 3 * N Schreibvorgänge nötig.
Die oben genannte Forderung, dass es keine Netzwerk Partitionierungen geben darf, ist notwendig, da sonst in unterschiedlichen Partitionen unterschiedliche Koordinatoren für eine Transaktion gewählt werden würden und unterschiedliche Commit – Ergebnisse die Folge wären.
Diese oben erwähnte „Robustheit“ ist ein besonderes Argument, das für die Verwendung eines 3 – Phasen Commits in extrem sicherheitskritischen verteilten Datenbanksystemen spricht.

4.2.2 Leistungsbewertung der Protokolle

Verglichen miteinander sieht die Leistungsfähigkeit der Protokolle, bezogen auf den Nachrichtenbedarf, folgendermaßen aus:

	Allgemein	Beispiel 1: (N = 2, M=0)	Beispiel 2: (N = 10, M = 5)
1- Phasen-Commit	2 * (N – 1)	2	18
Lineares 2 – Phasen-Commit	2 * N – 1	3	19
zentralisiertes/ hierar - chisches 2 – Phasen-Commit	4*(N-1) – 2*M	4	26
3 – Phasen-Commit	6*(N–1) – 4*M	6	34

Es wurde angenommen, dass die Transaktion an N (N > 1) Knoten durchgeführt wurde, wobei an M (M < N) Knoten keine Änderungen erfolgen.

4.3 Synchronisation

Sinn der Synchronisation von parallelen Datenbankanfragen, wie sie in verteilten Datenbanksystemen vorkommen, ist die Wahrung der Konsistenz, wobei keiner der Benutzer etwas vom Mehrbenutzerbetrieb merken soll.
Das Leichteste wäre hier natürlich eine serielle Ausführung der Anfragen, da hier die Konsistenz immer sichergestellt ist. Dies verbietet sich jedoch aus Leistungsgründen.
Ziel der Synchronisation ist es Anomalien wie lost - updates, dirty - reads, und ähnliches zu verhindern und doch den Kommunikationsaufwand gering zu halten. Dazu gibt es drei wichtige Verfahren: Sperrverfahren, Zeitmarken und optimistische Protokolle.

4.3.1 Synchronisation durch Sperrverfahren

Hier erwirbt das Datenbankverwaltungssystem vor dem Zugriff auf ein Objekt für die Transaktion eine Sperre. Dabei darf aber eine Transaktion nicht mehrere Sperren für ein Objekt erwerben. Im einfachsten Fall gibt es dabei Lese- (R-) Sperren oder Schreib-(X-) Sperren (exclusive). So kann es zwar zwei R-Sperren auf ein Objekt geben, niemals aber zwei X-Sperren. Dabei ist das Protokoll strikt zweiphasig und hält die Sperren bis zum Ende der Transaktion, um die Serialisierbarkeit zu erfüllen. Die Verwaltung kann dabei zentral, auf einem dedizierten Server, oder verteilt erfolgen. Die zentrale Verwaltung hat den Vorteil, dass stets der aktuelle Synchronisationsstand bekannt ist und die Deadlock – Behandlung wie im Einrechnerfall erfolgen kann, nachteilig ist aber, dass es zu Engpässen (Leistung und Verfügbarkeit) kommen kann und keine Knotenautonomie gegeben ist. Auch ist ein hoher Kommunikationsaufwand nötig, die bei verteilter Synchronisation wegfällt, da jeder Rechner seine Dateipartitionen selbst verwaltet. Hier erfolgt die Sperrfreigabe im Rahmen des Commit – Protokolls, jedoch existiert das Problem der globalen Deadlock – Behandlung (dazu später mehr).

4.3.2 Synchronisation durch Zeitmarken

Das Zeitmarkenverfahren ist nur in einer verteilten Umgebung sinnvoll. Hier wird die Serialisierbarkeit an Hand von Zeitstempeln an den Datenobjekten umgesetzt. Das verteilte Protokoll ergibt sich dadurch, dass die Überprüfung stets am Speicherort der Dateien vorgenommen wird. Zum einen entfällt dadurch jegliche Kommunikation, zum anderen gibt es keine Deadlocks. Im einfachsten Fall bekommt jede Transaktion zu Beginn der Transaktion eine Zeitmarke. Entweder kann diese Vergabe über einen dedizierten Rechner erfolgen, was aber viele Nachteile (Autonomie, Kommunikation) hat, oder aber es wird lokal ein zweiteiliger Zeitstempel bestehend aus Rechner - ID und lokaler Uhrzeit generiert. Dazu ist

eine Zeitsynchronisation aller Knoten nötig um monoton steigende Zeitstempel zu bekommen.
Für die Umsetzung gilt, dass alle Änderungen von älteren Transaktionen ausgeführt sein müssen, bis eine Aktuelle erfolgen darf, damit diese alle Änderungen sehen kann. Jedoch dürfen keine jüngeren Transaktionen gesehen werden. Falls Fehler auftreten (eine „jüngere" wird vor dem Ende einer „älteren" Transaktion ausgeführt), werden alle beteiligten Transaktionen zurückgesetzt und bekommen einen neuen Zeitstempel. Das Problem dabei ist, dass Transaktionen, die lange dauern eventuell ewig warten müssen und nie zur Ausführung kommen.

4.3.3 Synchronisation durch optimistische Protokolle

In diesem Fall gilt die Annahme, dass Konflikte zwischen Transaktionen selten auftreten und das Verhindern dieser Konflikte damit unnötiger Aufwand wäre. Es ist also zunächst ein beliebiges paralleles Arbeiten auf der DB möglich. Erst am Ende wird geprüft, ob Konflikte aufgetreten sind. Die Ausführung einer Transaktion gliedert sich damit in 3 Phasen: Lesephase, in der die eigentliche Transaktionsverarbeitung in einem exklusiv zugreifbaren Transaktionspuffer durchgeführt wird, der Validierungsphase, die am Ende der Transaktion folgt und in der überprüft wird, ob eventuell Konflikte mit anderen Transaktionen aufgetreten sind und am Ende die Schreibphase, die nur ausgeführt wird, wenn die Validierung erfolgreich war. Dabei wird durch Logging jedoch zuerst die Wiederholbarkeit der Transaktion sichergestellt. Dann erfolgt das Einbringen der Änderungen in die Datenbank. In der Validierungsphase werden Konflikte durch Rücksetzen der beteiligten Transaktionen aufgelöst. Das gibt zwar eine höhere Anzahl von Rücksetzungen, aber keine Deadlocks. Dieses Konzept wird vor allem für verteilte Datenbanksysteme verwendet.

4.4 Deadlocks

Deadlocks stellen eine Gefahr dar, die z.B. durch die Verwendung von Sperrverfahren verursacht werden können. Sie sind eine zyklische Wartebeziehung zwischen 2 oder mehr Transaktionen. Falls es in verteilten Systemen zwischen Transaktionen auf verschiedenen Rechnern zu Verklemmungen kommt, kann es zu einem globalen Deadlock kommen.

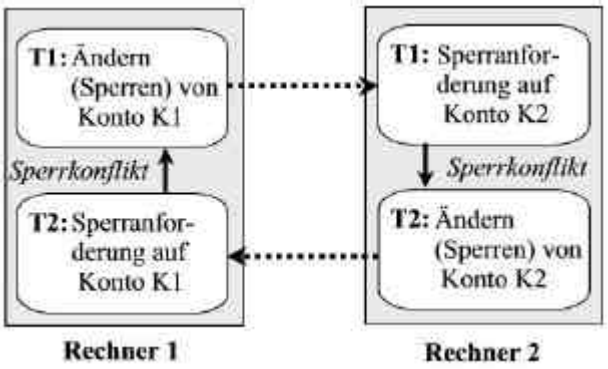

Abb11: Beispiel für einen globalen Deadlock

Im Allgemeinen gibt es vier Hauptstrategien, sowie einige hybride Verfahren zur Deadlock – Behandlung:

4.4.1 Deadlock – Verhütung

Dieses Verfahren soll Deadlocks verhüten, ohne dass Maßnahmen während der Abarbeitung der Transaktion nötig sind. Dies kann z.B. dadurch erreicht werden, dass eine Transaktion bereits alle Sperren am Beginn einer Transaktion anfordern muss. Dabei können Verklemmungen verhindert werden, indem jede Transaktion ihre Sperren in einer bestimmten Reihenfolge anfordert. Allerdings ist hier die Menge der gesperrten Objekte oft viel zu groß (am Beginn sind meist nur Obermengen der zu referenzierenden Objekte bekannt) und auch der Kommunikationsaufwand ist erheblich.
Dieser Ansatz hat somit keine praktische Relevanz.

4.4.2 Deadlock – Vermeidung

Erkennung potentieller Deadlocks im Voraus. Bei diesem Verfahren ist eine Laufzeitunterstützung nötig. Die Vermeidung erfolgt dann meist durch Zurücksetzen. Z.B. kann man in einem einfachen Ansatz die in Konflikt geratenen Transaktionen einfach abbrechen. Leider verursacht dieser Ansatz zu viele Zurücksetzungen und wird nicht verwendet. Gebräuchlicher sind Verfahren, die Zeitmarken bzw. Zeitintervalle verwenden. Zum einen kann hier ein dem Zeitmarkenverfahren bei der Synchronisation sehr ähnliches Verfahren verwendet werden. Der einzige Unterschied ist, dass kein Abbruch erfolgt, wenn keine Sperre aufgetreten ist. Da dieses Verfahren lokal arbeitet, entsteht keine Kommunikation, z.T. allerdings Rücksetzungen, ohne dass ein Deadlock aufgetreten ist.
Das obige Verfahren arbeitet mit statischen Zeitmarken. Zum anderen gibt es auch ein dynamisches Verfahren, dass erst dann eine Zeitmarke bestimmt, wenn ein Zeitmarke bestimmt, wenn der erste Konflikt mit einer Transaktion aufgetreten ist. Die Zeitmarken werden so gewählt, dass zumindest der erste Konflikt ohne Rücksetzung überstanden wird, was dadurch erreicht wird, dass im Konfliktfall nur die ältere (bzw. jüngere) Transaktion warten darf. So ist hier keine Zyklusbildung möglich und es sinkt die Anzahl der Rücksetzungen gegenüber dem obigen Verfahren.
Auch ist die Vergabe von Zeitintervallen möglich. Jede Transaktion startet zunächst mit unendlichem Zeitrahmen. Bei jedem Konflikt wird das Intervall verkleinert. Somit ergibt sich eine Rücksetzung erst bei leerem Zeitintervall. Es ergibt sich zwar eine erhöhte Komplexität bei der Realisierung, das Ausmaß der Rücksetzungen ist jedoch geringer als bei allen vorigen Verfahren. (WOUND/WAIT) / (WAIT/DIE) – Verfahren

4.4.3 Timeout

Sehr einfaches und vom Kommunikationsaufwand her billiges Verfahren, dass eine Transaktion einfach zurücksetzt und ihre Sperren aufhebt, falls eine bestimmte Zeitspanne überschritten wird. Schwierig ist hier jedoch die Wahl des Timeout – Wertes: Einerseits, soll langes Warten im Fall einer Sperre vermieden werden, andrerseits muss großen Transaktionen genug Zeit zur Ausführung gegeben werden.
In der Praxis hat sich gezeigt, dass dieses Verfahren nur bei geringer Konflikthäufigkeit praktikabel ist, weil es sonst zu viele Rücksetzungen verursacht. Es wird jedoch in vielen kommerziellen Datenbanksystemen eingesetzt.

4.4.4 Deadlock – Erkennung

Dieses Verfahren arbeitet mit Graphen. Es werden alle aktiven Transaktionen explizit in einen Wartegraphen eingetragen. Ergibt sich dabei ein Zyklus, ist eine Verklemmung aufgetreten.

Diese wird aufgelöst, indem eine oder mehrere Transaktionen, die den Zyklus verursacht haben, zurückgesetzt werden. Dieses Verfahren ist sehr gut für zentralisierte Datenbanksysteme. In verteilten Umgebungen entsteht jedoch ein sehr hoher Kommunikationsaufwand, zum anderen sind die Warteinformationen bezüglich externen Transaktionen zwischen verschiedenen Rechnern meist veraltet. Hier schafft auch die Verwendung eines zentralen Rechners, der nur den Wartegraphen aktuell hält, wenig Abhilfe.
In der Praxis hat sich gezeigt, dass dieses Verfahren sehr viele Rücksetzungen durchführt, obwohl keine Deadlocks vorlagen. Wohl auch durch seine sehr aufwändige Implementierung wird es selten verwendet.

5. Transaktionssysteme

In diesem Kapitel sollen noch kurz zwei Ausprägungen von verteilten Transaktionssystemen vorgestellt werden, um einen Einblick in die Anwendungsgebiete zu bekommen. Dabei soll der Schwerpunkt auf die Transaktionsverwaltung und –verarbeitung gelegt werden.

5.1 Überblick (Middleware, Realisierungen)

Als *Middleware* bezeichnet man oft die Softwareschicht, die zwischen dem eigentlichen Betriebssystem und dem verwendeten Datenbanksystem liegt. Zur Middleware zählt man unter anderem Datenbankverwaltungssysteme, Kommunikationsservices, Transaktionsmanager oder Authentifizierungsdienste. Auch die TP – Monitore *(„Transaction Processing“)* von denen gerade im Zusammenhang mit verteilten Datenbanksystemen immer wieder gesprochen wird, sind der Middleware zuzuordnen, jedoch bringen Sie einen weitaus größeren Funktionsumfang mit.
Eigentlich bezeichnet man als Middleware ja alle Systemdienste, die die Interoperabilität und Portabilität fördern und unterstützen. Leider ist dies in der Praxis noch nicht der Fall, da die Standardisierung einer gemeinsamen Lösung noch nicht weit genug fortgeschritten ist. Deshalb müssen oft Gateways als Überbrückung zwischen den einzelnen Systemen eingesetzt werden, die die Transaktionen in die jeweils richtige Form übersetzen. Hier kann jedoch ein sehr hoher administrativer Aufwand nötig werden.
Im Wesentlichen gibt es drei große Realisierungsansätze zur verteilten Transaktions – verarbeitung. Alle Systeme haben als Voraussetzung, dass die Verteilungseinheit ganze logische Datenbanken sind.

5.2 TP - Monitore

Eine transaction processing (TP) application ist ein Programm, das administrative (Verwaltungs-) Funktionen verrichtet. Dieses operiert im Auftrag eines online Anwenders, wobei es auf die gemeinsame Datenbank zurückgreift. Die TP-Anwendungen werden unterstützt vom TP-System, in das folgende Produkte integriert sind:

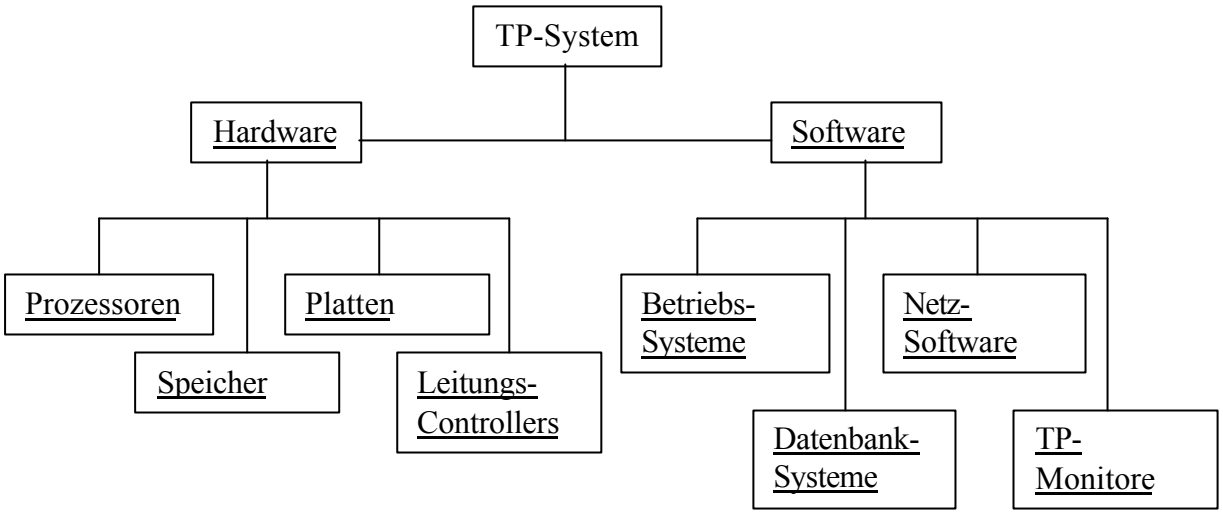

Abb12: TP-System

Der hier integrierte TP-Monitor trägt seinerseits viel zur Integration dieser Bauteile bei, indem er den Fluss der Transaktions-Anforderungen (Requests) koordiniert. Es besteht nämlich die Möglichkeit, dass eine Transaktion auf mehrere Datenbanksysteme zugreift. Dies erfordert weitere Koordination, die oft mit Hilfe des 2-Phasen-Commit-Protokolls (s.u.) des TP-Monitors oder des Betriebssystems geleistet wird.

Eine Benutzer-orientierte Anfrage benötigt unter Umständen die Ausführung mehrerer Transaktionen. So ist es zum Beispiel bei der Durchführung einer Bestellung nötig, die Bestellung einzugeben, den Transport anzufordern und die Rechnung auszustellen. Diese drei Transaktionen sollen aber gewöhnlich als eine Transaktion behandelt werden. Typische TP-Anwendungen schliessen verhältnismässig wenige Transaktionen ein, d.h. manchmal weniger als 10 bis gewöhnlich weniger als ein paar hundert. Wie schon erwähnt benötigt eine Transaktion jedoch einiges an Ressourcen:

- bis zu einigen Dutzend Plattenzugriffe
- bis zu einigen Millionen Anweisungen in Maschinensprache
- bis zu ein paar Dutzend Netzwerk-Messages

Heutzutage besitzen die grössten TP-Systeme allerdings bis zu 1000 Platten und können einige 1000 Transaktionen pro Sekunde unter Höchstlast bewerkstelligen.

5.2.1 TP-Monitor-Architektur

Die Hauptfunktion eines TP-Monitors ist die bereits erwähnte Koordinierung des Flusses der Transaktions-Anfragen zwischen den Terminals und den Anwendungsprogrammen. Dazu werden die Software-Komponenten eines TP-Systems vom TP-Monitor auf eine bestimmte Art und Weise strukturiert. Die Aktivitäten jeder Komponente werden von Einrichtungen unterstützt, die der TP-Monitor anbietet (siehe \ref{TP-Monitor-Funktionen}). Die Mehrheit der TP-Anwendungen entspricht diesem Aufbau, gemäss dem folgende Schritte für jedes Terminal ausgeführt werden:

1. Mit dem Benutzer des Terminals interagieren, um den Transaktions-Input zu sammeln. Dies geschieht üblicherweise mit Hilfe von Formularen und Menüs.

2. Umwandeln des Transaktions-Inputs in eine **standard format request message**.

3. Beginnen der Transaktion.

4. Untersuchen des Kopfteils (*header*) der Anfrage (umgewandelter Input) um seinen Typ zu ermitteln.

5. Durchführen der zum Anfragetyp gehörigen Anwendung, welches der Reihe nach die Datenbanksysteme und andere Anwendungsprogramme aufrufen kann.

6. Nachdem die Anwendung zu Ende ist, erfolgt das *commit* der Transaktion.

7. übermitteln des Outputs der Transaktion an das Terminal.

Der TP-Monitor teilt Anwendungen in Bauteile auf, die diese sieben Schritte ausführen.

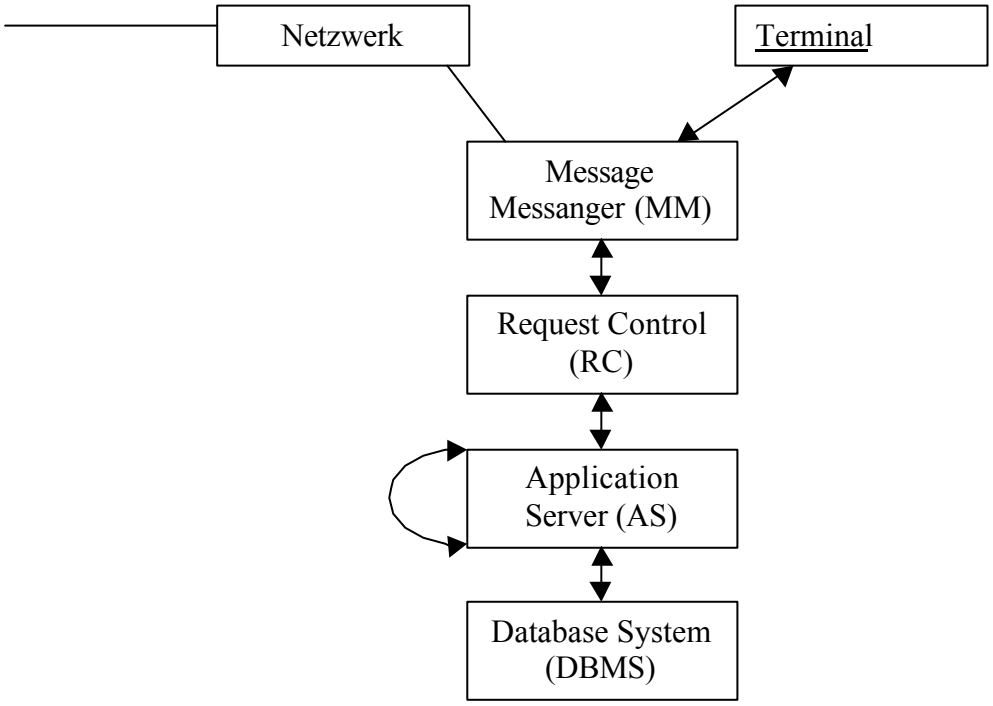

Abb13: TP-Monitor-Diagramm

5.2.2 TP-Monitor-Funktionen

5.2.2.1 Der Message Manager (MM)

Der Message Manager muss mit einer Vielfalt von Terminaltypen interagieren, um den Input zu sammeln und den Output darzustellen. Diese Vielfalt entsteht dadurch, dass die Technologie bei den Terminals häufig wechselt. Deshalb besitzen heutige TP-Systeme eine Mischung von Terminaltypen, wie z.B. character-at-a-time-, und screen-at-a-time-Terminals oder PCs. Aber auch Industrie-spezifische Varianten wie Bar-Code-Leser, automatische

Schaltermaschinen und Verkaufsterminals. Wird hier von Terminals gesprochen, so sind alle diese Typen darunter zu verstehen.

Der Message Manager verrichtet 4 Hauptfunktionen:

- Er formatiert Anfragen
- Er verwaltet Formulare
- Er erklärt Input für gültig
- Er stellt Output dar

Um den Request-Controller von den verschiedenartigen Schnittstellen abzuschirmen, die von den verschiedenen Terminaltypen geliefert werden, überführt der Message Manager jeden Input, der eine Transaktion verlangt in die *standard format request message*, oder einfacher: in einen Request. Das Format der Anfrage wird durch den TP-Monitor festgelegt. Es enthält einen normierten header, der für alle Anwendungen, die den TP-Monitor benutzen, gleich ist. Weiterhin enthält er die Adresse des Terminals, den Namen des Anwenders an diesem Terminal und den Namen des Anfrage-Typs. Das Format enthält noch einen Rumpf, der die Parameter für die Transaktion enthält. Dieser wird durch die Anwendung festgelegt.

Die Komponente des Message Managers, die nur für die Umwandlung der Terminal-Anfrage verantwortlich ist, heisst Forms Manager (FM). Er stellt dem Anwendungsprogrammierer einen Editor zur Verfügung, mit dem dieser Formulare definieren und modifizieren kann. Formulare bestehen aus einer Reihe von Feldern, welche wiederum Merkmale, wie z.B. eine Marke oder einen Datentyp besitzen. Auch besitzen die Felder eine Darstellung auf dem Bildschirm oder auf Windows. Mit Hilfe des Compilers werden die Formulare in eine *translation table* und in eine *record definition* umgewandelt. Die translation table wird vom Laufzeit-System des Forms Manager dazu gebraucht, das Terminal-spezifische Format in das standard request format zu übersetzen. Record definition ist eine höhersprachige Deklaration für das standard request format des Formulares, das zur Aufnahme in ein Anwendungsprogramm, welches das Formular benutzt, geeignet ist.

Eine weitere Aufgabe des Forms Managers ist es, den Output darzustellen, der bei der Bearbeitung der Anfrage geliefert wird. Dies kann eine Ausgabe auf dem Bildschirm sein, aber auch den Befehl an einen Bankautomaten Bargeld auszugeben.

Die Bestätigung des Inputs fällt ebenfalls in den Verantwortungsbereich des Message Managers. Er kontrolliert, ob der jeweilige Input vom geeigneten Typ ist. Es soll z.B. kein Buchstabe in einem Zahlenfeld stehen oder zulässige Wertbereiche sollen nicht überschritten werden. Während der Durchführung der *date validation* muss ein Message Manager gewöhnlich nicht in gemeinsamen Datenbanken lesen, welche durch die Transaktionen auf den neuesten Stand gebracht wurden. Diese Tatsache gestattet es dem Message Manager, effizient in engem Bezug zum Terminal zu arbeiten und mit Abstand zur gemeinsamen Datenbank.

Bei einem Message Manager schreibt der Anwendungsprogrammierer nur die Formulardefinitionen und data-validation-Routinen. Der TP-Monitor erledigt den Rest, also Compilierung der Formulardefinitionen und die Laufzeit-Umwandlung jedes Formulars in einer Anfrage.

Zusätzlich zu dieser Vor-Bearbeitung von Anfragen verrichtet der Message Manager gewöhnlich einige Sicherheitsfunktionen:

- Beglaubigung jedes Benutzers durch die überprüfung eines Passwords.

- Einbau des Identifieres vom Benutzer in jede Anfrage, die den Benutzer erreicht.

- Zugriffskontrolle dadurch, dass er nur Menüeingänge darstellt, die den Anfrage-Typen entsprechen, auf die der Benutzer zugreifen darf, abhängig von dem Terminal, das gerade verwendet wird.

- Da das Betriebssystem nichts über Anfrage-Typen weiss und das Datenbanksystem typischerweise nicht das Teminal kennt, von welchem die Anfrage stammt, muss diese Funktion ebenfalls vom TP-Monitor verrichtet werden.

5.2.2.2 Request Control (RC)

Die jeweilige Anfrage, die von einem Message Manager aufgebaut wurde, wird einer Request Control zur Bearbeitung übergeben. Die Request Control ist dann verantwortlich dafür, dass die Anfrage an einen Application Server (AS) versandt wird, der die Anfrage bearbeiten kann. Die einzige Aufgabe des Anwendungsprogrammierers ist es, eine Tabelle zu liefern, die jedem Anfragen-Typ einen geeigneten Application Server zuordnet. Die Request Control macht dann den Rest. Sie sucht im Anfragen-Header nach dem Symbol des Anfragen-Typs und bildet es im Identifier des zuständigen Application Servers ab. Diesem Application Server übergibt die Request Control dann die Parameter, die sie aus dem Anfragen-Rumpf herausgeholt hat.

Die Request Control kann ihre Aufgabe auf zwei verschiedene Arten erfüllen:

RC-to-AS-mapping

Die Abbildung des symbolischen Anfragen-Typs in den Application Server-Indentifier sollte dynamisch sein. Dies ist hilfreich für die Fehlertoleranz, da es dem System erlaubt einen Anfragen-Typ schnell in einen anderen Application Server-Identifier umzuschreiben, falls der ursprüngliche Application Server versagt. Auch gestattet es dem Systemmanager, Anfragen auf verschiedene Application Server umzuverteilen, z.B. aus Leistungsgründen.

Eine einfache Möglichkeit, diese dynamische Abbildung zu implementieren, ist eine Tabelle, die sich lokal befindet zur Request Control. Gibt es nun mehr als ein Exemplar der Request Control, so kann jedes Exemplar eine unterschiedliche Kopie deser Tabelle besitzen.. Das führt nun zu folgendem Problem: Verwalten verschiedene Request Controls verschiedene Anfragen-Typen, so könnte bei einer Request Control eine Anfrage eintreffen, mit dem die Request Control nichts anfangen kann.

Es gibt 2 Standard-Lösungen:

- Der jeweilige Message Manager weiss, welche Request Control welchen Anfragen-Typ bearbeiten kann, und versendet die Anfrage dementsprechend richtig.

- Jede Request Control weiss, welche Request Controls den jeweiligen Anfragen-Typ bearbeiten können. Der Message Manager übergibt die Anfrage nun an irgendeine

Request Control. Diese befördert die Anfrage dann, falls nötig, zur passenden Request Control

Einige Systeme unterstützen einen *global name service*, wo Namen in ein Attribut-Wert-Tupel abgebildet werden können. Der Service ist von jedem Knoten aus erreichbar. Bringt man so Anfragen-Typen und Request Control-Identifier in Verbindung, so können wegen der globalen Zugriffsmöglichkeit entweder die Message Manager oder die Request Controls die Verantwortung für korrekte Versendung der Anfrage übernehmen. Bei den 2 Standard-Lösungen ist dies nicht der Fall, dort kann das nur entweder der Message Manager oder die Request Control.

RC-to-AS-binding

Um sich mit einem Application Server zu verbinden, muss eine Request Control den Identifier des Application Servers benutzen. Die Beschaffenheit dieser Verbindung ist vom Betriebssystem und der Architektur des Kommunikationsnetzes festgelegt. Sind z.B. die Request Control und der Application Server im gleichen Adressraum verbunden, könnte ein Application Server-Identifier eine externe Referenz sein. In verschiedenen Adressräumen, welche durch eine *communications session* verbunden sind, könnte der Application Server-Identifier eine Netzwerkadresse sein, die benötigt wird, um eine Verbindung für diese Sitzung herzustellen.

Hier kann der global name service dazu verwendet werden, die Abbildung der Anfrage-Typen in Application Server-Identifier zu speichern. Da diese Abbildung wieder global erreichbar ist, können die Application Server direkt darauf zugreifen, ohne Request Controls als Vermittler zu benötigen. Der TP-Monitor muss nun die Request Controls nicht mehr von Application Servers unterscheiden, d.h. der Begriff der Request Control verschwindet. Die Anwendungen behalten jedoch die Unterscheidung bei, d.h. einige Application Servers bilden Anfrage-Typen auf Application Server-Identifier ab (sie verrichten also die Request Control-Funktion), andere führen das Anwendungsprogramm für den Anfrage-Typ aus.

Durch diese Strukturierung wird die Anzahl der Bindungen auf ein Minimum reduziert. So verbindet sich jeder Message Manager mit einem oder allerhöchstens mit einigen wenigen Application Servers, die die Request Control-Funktion verrichten. Und die Application Server, die diese Funktion nicht verrichten, verbinden sich nur mit solchen, die dies tun. Ohne diese Struktur müssten sich alle Message Manager mit allen Application Servern oder alle Application Server untereinander verbinden.

Hat ein Applicaton Server einen Auftrag beendet, gibt er möglicherweise das Ergebnis an die Request Control zurück, welche es typischerweise zum entsprechenden Message Manager weitergibt. Daher muss die Request Control in der Lage sein, jede Anfrage zu dem Message Manager zurückzuschreiben, der ihn gesendet hat. Der Identifier des entsprechenden Message Managers ist gewöhnlich im Anfragen-Header gespeichert.

Gewöhnlich ist die Request Control auch für die Klammerung der Transaktion verantwortlich. Dazu gibt sie den Befehl \emph{BOT} aus, bevor irgendein Application Server aufgerufen wird. Wenn die Transaktion beendet ist, gibt die Request Control den Befehl *commit* oder *abort* aus.

5.2.2.3 Application Server (AS)

Jeder Application Server besteht aus einem oder mehreren Anwendungsprogrammen, welche typischerweise Zugriff auf eine gemeinsame Datenbank haben. Zusätzlich zur Verknüpfung von Applikation Servern mit Request Controls, stellt der TP-Monitor auch Application Server zur Verfügung, die Zugriff auf das Terminal haben, das die jeweilige Anfrage geliefert hat.

5.2.3 System-Management und Recovery

Ein TP-Monitor stellt System-Management-Operationen zur Verfügung, um die Menge von Message Managern, Request Controls und Application Server-Prozessen zu verwalten. Hierzu unterhält er eine Beschreibung der Konfiguration der Prozesse im System. Diese Beschreibung umfasst unter anderem

- die Terminals und Formulare, die jedem Message Manager zugeordnet sind
- die Sicherheitsmerkmale von Benutzern
- den Satz von Anfrage-Typen, die von der jeweiligen Request Control behandelt werden
- die Menge von Programmen, die von dem jeweiligen Application Server verwaltet werden

In einem verteilten System enthält die Beschreibung auch die Namen der Knoten, an denen der jeweilige Prozess ausgeführt wird. Ein Systemmanager kann Prozesse generieren und zerstören, sie zwischen Knoten verlagern und die Menge von Formularen und Programmen, die vom jeweiligen Prozess verwendet werden, abändern.

Der TP-Monitor kann die Leistung des laufenden Systems messen und diese Information dem Systemmanager übergeben. Der Systemmanager kann diese Information, die Transaktions-Raten, Antwortzeiten, usw. enthält, verwenden um die Antwortzeit und den Durchsatz zu verbessern.

Die Kenntnis der Message Manager-Request Control-Application Server-Konfiguration ist wertvoll bei der Behandlung von Fehlern. Versagt etwa ein Knoten (Rechner), kann der TP-Monitor die Message Manager des Knotens an einem anderen Knoten wiederherstellen, der Zugriff auf die gleiche Menge von Terminals hat (z.B. ein Knoten, der mit dem gleichen LAN verbunden ist). Ausserdem können die Request Controls und Application Server des fehlerhaften Knotens an einem anderen Knoten wiederhergestellt werden, sofern der Rechner die entsprechenden Programme laden kann und Reservekapazitäten besitzt, um die Prozesse durchzuführen. Mit Hilfe der Konfigurationsbeschreibung kann der TP-Monitor diese Tätigkeit ohne menschlichen Eingriff verrichten. Er gebraucht dabei entweder vordefinierte Aushilfskonfigurationen oder er konstruiert eine durchführbare Konfiguration neu. Das Ganze geschieht im Rahmen von Recovery-Massnahmen.

Dieses Recovery wird mit Hilfe des Transaktions-Konzepts durchschaubar. Versagt ein Prozess, brechen die Transaktionen ab, die in diesem Prozess operierten. Hat der TP-Monitor den fehlerhaften Prozess, möglicherweise an einem anderen Knoten, wiederhergestellt, starten Anfragen, die von der abgebrochenen Transaktion bearbeitet wurden, neu. Ist das Recovery schnell, erkennt der Benutzer diesen Fehler lediglich als langsame Antwortzeit. Zudem wird dieses Recovery beinahe vollständig von TP-Monitor und Transaktions-Mechanismen vollendet. Anwendungsprogrammierung ist gar nicht oder kaum nötig.

5.3 Verteilte Transaktionssysteme

Hier kooperieren mehrere Datenbankverwaltungssysteme(DBVS) zur Bearbeitung von Datenbank – Operationen. Diese verteilte Verarbeitung kann jedoch auch außerhalb von DBVS mit TP – Monitoren umgesetzt werden.
Die eigentliche Transaktionsverarbeitung kann hier entweder zwischen gleichberechtigten Verarbeitungsrechnern oder in Client – Server Systemen erfolgen. Dabei kann die Verteilung der Transaktionen auf drei Arten geregelt werden. Zum einen gib es die programmierte Verteilung und die Verteilung von Datenbank – Operationen, andrerseits das Transaction – Routing.

5.3.1 Programmierte Verteilung

Die Verteilung wird hier auf der Ebene der Anwendungsprogramme realisiert. Der Client greift auf externe Anwendungen zu, um auf dort vorliegende Datenbanken zuzugreifen. Das Verteilungsgranulat ist hier das Teilprogramm und unterstützt die Heterogenität und die Knotenautonomie sehr gut (An jedem Knoten kann ein eigenes logisches Datenbanksystem vorliegen, wobei keine Kooperation stattfindet). Auch die Administration ist sehr einfach, da keine Rechte für einzelne Datenbankobjekte vergeben werden müssen, sondern nur Rechte für lokale Teilanwendungen. Die Transparenz kann durch einen TP - Monitor mit Name Server Funktion, zum leichten Auffinden der Programme, gewährleistet werden.
Dieses Konzept wird von den meisten TP – Monitoren unterstützt und hat vor allem den Vorteil, dass es relativ leicht realisierbar ist.

5.3.2 Verteilung von Datenbank – Operationen

Dies ist entspricht einem Fernzugriff auf Datenbanken, die Anwendungen kooperieren also mit mehreren Datenbanksystemen, um auf die entsprechenden Datenbanken zuzugreifen.
Hier ist das Verteilungsgranulat eine Datenbank – Operation, d.h. eine Operation darf nur die Daten eines Rechners referenzieren. Hier wird aber eine höhere Kommunikation nötig. Besonders Leseoperationen, die eine große Ergebnismenge zurückliefern, können hier sehr kommunikationsintensiv sein. Die Anwendungsprogrammierung ist hier auch sehr komplex, da praktisch keine Verteilungstransparenz unterstützt wird. Zwar lässt sich durch logische Namen in den Datenbank – Operationen eine gewisse Ortstransparenz erreichen, jedoch ist die Datenunabhängigkeit und die Unterstützung für heterogene Umgebungen nicht gut.

5.3.3 Transaction Routing

In diesem Fall werden ganze Transaktionsaufträge zwischen den Verarbeitungsrechnern verteilt. Wenn hier keine feinere Unterteilung unterstützt wird, muss sichergestellt werden, dass eine Transaktion immer komplett auf einem Rechner ausgeführt werden kann. Damit muss auch sichergestellt sein, dass der TP – Monitor die Programmzuordnungen kennen muss. Dieses Verfahren wird nur von wenigen TP – Monitoren unterstützt, unter anderem von CICS (IBM) und hat eine eher untergeordnete Bedeutung.

Abschließend bleibt zu sagen, dass bei verteilten Transaktionssystemen der Administrations-aufwand relativ hoch ist, da viele Rechner administriert werden müssen. Enorm wichtig ist die transaktionsbasierte Verarbeitung, da sie eine weitgehende Fehlerisolation Anwendungen gegenüber ermöglicht und die Konsistenz der Daten auch im Fehlerfall sicherstellt.

Durch die hohe Verfügbarkeit, die mit verteilten Datenbanksystemen erreicht werden kann und die gute Möglichkeit zur geographischen Verteilung, ist dieses Konzept gut zur Schaffung von zusätzlicher Sicherheit in Informationssystemen geeignet.

Beispiele für die Realisierung verteilter Transaktionssysteme bzw. Standardisierungsansätze:

- X/Open Distributed Transaction Processing(DTP): eine Standardisierung von Systemschnittstellen zur verteilten Transaktionsbearbeitung
- Remote Database Access (RDA): ISO – OSI Kommunikationsstandard zur Verteilung von Datenbank – Operationen in heterogenen Umgebungen.
- ODBC bzw. IDAPI: APIs um Clientsystemen den Zugriff auf mehrere heterogene Datenbankverwaltungssysteme zu ermöglichen.

5.4 Föderative Datenbanksysteme(FDBS)

Föderative Mehrrechner Datenbanksystem sind nichts anderes als eine Menge von autonomen und evtl. heterogenen logischen Datenbanksystemen, die in einem begrenzten Umfang kooperieren. Durch diese Zusammenarbeit ist es jetzt möglich in einer DB-Operation auf mehrere Datenbanken zuzugreifen. Außerdem werden dadurch Einschränkungen bezüglich der Autonomie in der Transaktionsverwaltung in logischen Datenbanksystemen ausgeräumt.
Die Architektur basiert weitgehend auf mehreren logischen Datenbanksystemen, über denen eine Zusatzebene liegt, die das eigentliche FDBS realisiert.

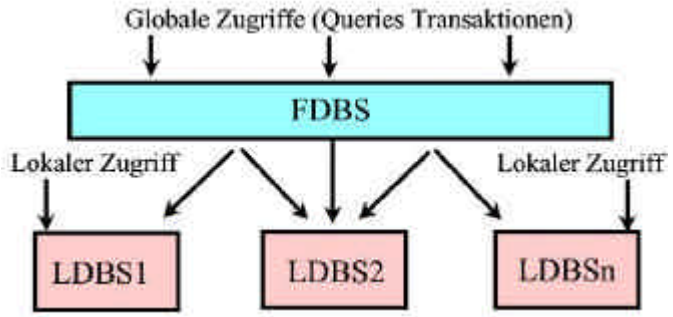

Abb14: Grobarchitektur der föderativen Datenbanksysteme

Die FDBS – Schicht übernimmt dabei die Zerteilung der in einer gemeinsamen Anfragesprache gestellten globalen Anfragen, die mehrere Datenbanksysteme betreffen, in lokal ausführbare Teilanfragen, die von einem Datenbanksystem abgearbeitet werden können. Diese Zusatzebene übernimmt auch Aufgaben wie die Transaktionsverwaltung und kann entweder zentral auf einem dedizierten Knoten oder verteilt auf allen Datenbank – Knoten realisiert sein.

5.4.1 Transaktionsverwaltung in FDBS

Auch die Transaktionsverwaltung erfolgt in diesem System zweistufig: Zum einen gibt es auf jedem Knoten eine lokale Transaktionsverwaltung, in der FDBS – Zusatzschicht ist aber zusätzlich eine globale Transaktionsverwaltung implementiert. Diese lokale Verwaltung eines Knoten soll dabei mit der globalen Transaktionsverwaltung, aber auch mit der lokalen Verwaltung anderer Knoten kooperieren. Dabei wird nicht nach lokalen Transaktionen und globalen Subtransaktionen unterschieden, die auf einem Knoten ausgeführt werden. Dies wirf Probleme auf: Das Datenbanksystem nimmt am Ende einer globalen Subtransaktion bereits das Commit vor, so dass diese Änderungen sofort sichtbar und dauerhaft werden, obwohl gar nicht sichergestellt ist, ob die anderen globalen Subtransaktionen ebenfalls mit einem

COMMIT beendet wurden. Hier kann das globale Verwaltungssystem nur versuchen durch Kompensationstransaktionen die falschen Änderungen auszugleichen. Darüber hinaus wirft das Vorhandensein von globaler und lokaler Transaktionsverwaltung Synchronisations-probleme auf und die globale Serialisierbarkeit ist nicht gewährleistet. Um diese Probleme zu minimieren und um den Einsatz von identischen Commit - Protokollen zu ermöglichen ist eine Beschränkung der Heterogenität nötig.

FDBS erhöhen die Sicherheit für ein Informationssystems, da sie eine sehr gute geographische Verteilung der Datenbanksysteme zulassen. Durch die Kooperation der einzelnen DB – Teile untereinander, ist jedoch die Verfügbarkeit geschwächt: Für eine korrekte Funktion, müssen immer alle Datenbanken laufen.

6. Anwendungsbeispiel

Wir wollen uns die Überweisung eines Geldbetrags zwischen den Konten zweier Banken Bk1 und Bk2 betrachten. Dabei werden wir uns im ersten Beispiel die programmierte Verteilung (-> 3.2.1) und im zweiten Beispiel die Realisierung mittels der Verteilung von DB – Operationen anschauen.

Programmierte Verteilung	Verteilung von DB – Operationen
Einlesen der Eingabedaten(Konten K1,K2; Banken Bk1, Bk; Geldbetrag Geldbetr) BEGIN – Transaction	*Einlesen der Eingabedaten(Konten K1,K2; Banken Bk, Bk2; Geldbetrag Gb)* BEGIN – Transaktion
	Verbindung mit dem Datenbank-system von Bank Bk1 herstellen CONNECT to R1.DB1
Aufruf der Funktion zum Abbuchen von Geld von einem Konto CALL Debit(Bk1, K1, Geldbetr,..)	*Den Kontostand durch Abbuchung aktualisieren* UPDATE ACCOUNT SET BALANCE = BALANCE – Geldbetr WHERE ACC_NO = „K1“
	Mit dem Datenbanksystem von Bank Bk2 verbinden CONNECT to R2.DB2
Aufruf der Funktion zum Zubuchen eines Betrages auf ein Konto CALL Credit(Bk2, K2, Geldbetr,..)	*Den Kontostand durch Zubuchung aktualisieren* UPDATE …
Transaktion beenden COMMIT – Transaction	*Transaktion beenden* COMMIT – Transaction
Evtl. Nachricht ausgeben	*Evtl. Nachricht ausgeben* *Verbindung abbauen (DISCONNECT)*

Hier sieht man sehr gut, dass bei der programmierten Verteilung das Granulat ganze Funktionen sind, während man hingegen auf der anderen Seite mit einzelnen DB – Operationen arbeiten muss. Das zweite Beispiel ist auch vom Programmierungsaufwand her

wesentlich aufwendiger, da der Benutzer immer zuerst eine Verbindung zum DB – System aufbauen muss und sich diesem gegenüber authentifizieren muss. Im ersten Fall wird das alles vom Teilprogramm erledigt. Auch ist das erste Beispiel weniger anfällig gegenüber Schema - änderungen.

7. Abschließende Betrachtung

Die hier vorgestellten Konzepte und Protokolle tragen mit Sicherheit einen großen Teil zur Sicherheit von Informationssystemen bei. Durch die Erfüllung der ACID – Bedingungen, wie es für Transaktionen gefordert wird, ist die Integrität der Daten sichergestellt. Auch die Verfügbarkeit der Daten ist durch das Verteilungskonzept je nach Implementierung relativ gut gelöst, da die Systemkomponenten, die zur Realisierung eines verteilten Transaktionssystems nötig sind, nicht zwingend an einem Ort räumliche Nähe zu einander haben müssen. Auch kann die Datensicherung durch Replikationsverfahren gut abgedeckt werden, wie wir in Vortrag 5 gehört haben.
Der Punkt der Vertraulichkeit der Daten wird über die Rechtevergabe des Transaktions- bzw. Datenbanksystems geregelt und hängt im Allgemeinen nicht von der Transaktions-verarbeitung ab. Auch einen Mechanismus zur Sicherstellung der Authentizität kann das System nicht bieten, dafür sind andere Komponenten verantwortlich.
Man kann nach der Betrachtung aller Punkte zu dem Schluss kommen, dass die Verteilung des Transaktionen durchaus einen Sicherheitsgewinn für die verwalteten Daten bringt, schon allein durch die Möglichkeit der räumlichen Trennung der Daten und die Erfüllung der ACID – Eigenschaften.
Nächster Vortrag: Rechnercluster…

6. Abbildungsverzeichnis

Abb1: Zustandsübergänge eine Transaktion 6
Abb2: Verzahnte Ausführung zweier Transaktionen und das erstellte Log 9
Abb3: Zwei Recoveryarten 10
Abb4: Modell Koordinator – Agent 13
Abb5: Zentralisiertes 2 – Phasen Commit 14
Abb6: Zustandsübergänge von Koordinator und Agent 15
Abb7: Lineares 2 – Phasen Commit 15
Abb8: Hierarchisches 2 – Phasen Commit 16
Abb9: Vergleich 1- Phasen und 2 – Phasen Commit 18
Abb10: 3 – Phasen Commit 19
Abb11: Beispiel für einen globalen Deadlock 21
Abb12: TP-System 24
Abb13: TP-Monitor-Diagramm 25
Abb14: Grobarchitektur der föderativen Datenbanksysteme 31

7. Literaturverzeichnis

A.Kemper, A.Eickler, Oldenbourg Verlag, „Datenbanksysteme eine Einführung“, 1999

J.Gray, A.Reuter, Morgan Kaufmann Publishers, „Transaction Processing: Concepts and Techniques“, 1993

E.Rahm, Addison-Wesley, „Mehrrechner-Datenbanksysteme: Grundlagen der verteilten und parallelen Datenbankverarbeitung“, 1994

E.Rahm, Vieweg Verlag, „Hochleistungs-Transaktionssysteme: Konzepte und Entwicklungen moderner Datenbankarchitekturen, 1993

R.Ben-Natan, McGraw-Hill New York, „CORBA: A guide to common object request broker architecture”, 1995

C.Eckert, Oldenbourg Verlag, „IT – Sicherheit: Konzepte – Verfahren – Protokolle“, 2001

B.Mitschang, Uni Stuttgart, „Transaktionssysteme, parallele und verteilte Datenbanksysteme“, WS 01/ 02, http://www.informatik.uni-stuttgart.de/ipvr/as/lehre/vorlesungen/TRSWS0102.html

M.Greulich, Diplomarbeit TU München, „Diskussion und Implementierung des von der OMG definierten Transaktions -services auf Basis des Transaktionsmonitors UTM“, 1997 http://www11.informatik.tu-muenchen.de/publications/da/greulich97/da.html

B.Fischer, Diplomarbeit TU München, „Entwurf und Implementierung eines Objektmodells für das Management von TP – Monitoren“, 1996, http://www.nm.informatik.uni-muenchen.de/common/Literatur/MNMPub/Diplomarbeiten/fisc96/HTML-Version/main.html

Bayerische Landesbank, Schulungsunterlagen München, „MQSeries – ein kurzer Überblick“

Bayerische Landesbank, Genia GmbH Mainz, „CICS“