TU Braunschweig
Institut für Betriebssysteme
und Rechnerverbund

IBR

Verteilte Systeme

Prof. Dr. Stefan Fischer

Kapitel 9: Transaktionen

Überblick

- Einführung und Motivation
- Transaktionen
- Geschachtelte Transaktionen
- Steuerung des gleichzeitigen Zugriffs auf Ressourcen
  - Locks
  - Optimistischer Ansatz
  - Verwendung von Timestamps
- Verteilte Transaktionen

Einführung

- Wir haben im letzten Kapitel bereits das Konzept des gegenseitigen Ausschlusses betrachtet.
- Ziel: eine bestimmte Ressource soll nur von einem Client zur selben Zeit benutzt werden
- Ganz generell haben Transaktionen dasselbe Ziel: Schutz von Ressourcen vor gleichzeitigem Zugriff
- Transaktionen gehen jedoch noch wesentlich weiter:
  - Es ist möglich, auf mehrere Ressourcen in einer einzigen atomaren Operation zuzugreifen.
  - Diverse Arten von Fehlern können abgefangen werden, so dass Prozesse in den Zustand vor Beginn der Ausführung einer Transaktion zurückgesetzt werden.

Beispiel für dieses Kapitel

- Wir wollen die in diesem Kapitel vermittelten Konzepte anhand eines durchgängigen Beispiel erläutern.
- Es gibt zwei Arten von Ressourcen:
  - Account-Objekte, die Abbuchungen und Einzahlungen gestatten sowie Abfragen und Änderungen des Kontostands
  - Branch-Objekte, die eine Filiale repräsentieren und es gestatten, Konten zu erzeugen, Konten zu suchen und den Gesamtstand aller Konten in dieser Filiale abzufragen.

Schnittstelle der Objekte

Methoden des Account-Objekts

deposit(amount)
deposit amount in the account
withdraw(amount)
withdraw amount from the account
getBalance() -> amount
return the balance of the account
setBalance(amount)
set the balance of the account to amount

Methoden des Branch-Objekts

create(name) -> account
create a new account with a given name
lookUp(name) -> account
return a reference to the account with the given name
branchTotal() -> amount
return the total of all the balances at the branch

Einfache Synchronisation

- Im Account-Objekt müssen die Operationen deposit() und withdraw() *atomar* ausgeführt werden, d.h., sie dürfen in der Ausführung nicht unterbrochen werden.
- In Java lässt sich das auf einfache Weise unter Verwendung des Schlüsselwortes `synchronized` erreichen:
  `public synchronized void deposit(...);`
- Ergebnis: wenn mehrere Threads gleichzeitig dieselbe bzw. eine andere synchronisierte Methode dieses Objekts benutzen wollen, wird nur einer zugelassen; die anderen werden blockiert.

Unterstütztes Fehlermodell

- Lampson führte 1981 ein Fehlermodell, das heute als Grundlage aller Transaktionsalgorithmen verwendet wird. Mit anderen Worten, ein Algorithmus muss folgende Fehler behandeln können:
  - Fehler beim Schreibzugriff auf permanenten Speicher
  - Server-Crash
  - Beliebige Verzögerungen bei der Nachrichtenübertragung

Transaktionen

- Transaktionen bestehen aus einer Folge von Operationen (Anfragen an Server), für die bestimmte Eigenschaften gelten – die ACID-Eigenschaften.
- Beispiel für eine Transaktion in der Bankanwendung: eine Kunde will verschiedene Operationen auf drei Konten a, b und c ausführen. Die Operationen sollen ohne Unterbrechung ausgeführt werden:
- Transaction T:
  a.withdraw(100);
  b.deposit(100);
  c.withdraw(200);
  b.deposit(200);

Die ACID-Eigenschaften

- ACID ist ein von Härder und Reuter (zwei Deutsche, immerhin) vorgeschlagenes Acronym.
- Bedeutung:
  - **A**tomicity: alle Operationen der Transaktion oder keine
  - **C**onsistency: eine Transaktion überführt das System von einem konsistenten Zustand in den anderen
  - **I**solation: jede Transaktion muss von der Ausführung anderer Transaktionen unabhängig bleiben
  - **D**urability: wenn eine Transaktion abgeschlossen ist, müssen die Ergebnisse auf permanentem Speicher gesichert werden

ACID(1): Atomicity und Durability

- Atomicity und durability wird erreicht durch die Verfügbarkeit *wiederherstellbarer Objekte*.
- Wenn ein Server-Prozess während der Abarbeitung einer Transaktion abstürzt und dann ein neuer Prozess gestartet wird, dann muss dieser den alten Zustand der Objekte wieder laden können.
- Wenn die Transaktion abgeschlossen ist, muss das Objekt den neuen Zustand repräsentieren und abgespeichert werden.
- Die beiden anderen Eigenschaften betrachten wir etwas später.

Implementierung

- Für jede Transaktion gibt es einen *Koordinator*, der den Ablauf steuert.
- Verwendung:
  - Der Client startet eine Transaktion, woraufhin der Koordinator eine Transaktions-ID allokiert und zurückgibt.
  - Dann werden die Operationen ausgeführt.
  - Am Ende ruft der Client ein closeTransaction() auf, woraufhin der Koordinator alle Objekte speichert und eine positive oder negative Abschlussmeldung liefert.
  - Der Client kann auch von sich aus die Transaktion abbrechen.
- Die nächste Folie zeigt ein typisches Interface für einen solchen Koordinator (meist auch als Transaction Monitor bezeichnet).

Das *Koordinator* -Interface

openTransaction() -> trans;
starts a new transaction and delivers a unique TID *trans*. This identifier will be used in the other operations in the transaction.

closeTransaction(trans) -> (commit, abort);
ends a transaction: a *commit* return value indicates that the transaction has committed; an *abort* return value indicates that it has aborted.

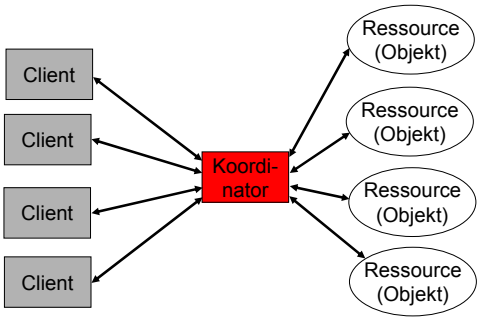
abortTransaction(trans);
aborts the transaction.

Beispiel: der Java `TransactionManager`

- Findet sich in javax.transaction.
- Ausschnitt aus der Schnittstelle:

`void`	`begin()` Create a new transaction and associate it with the current thread.
`void`	`commit()` Complete the transaction associated with the current thread
`Transaction`	`getTransaction()` Get the transaction object that represents the transaction context of the calling thread
`void`	`rollback()` Roll back the transaction associated with the current thread
`Transaction`	`suspend()` Suspend the transaction currently associated with the calling thread and return a Transaction object that represents the transaction context being suspended.
`void`	`resume(Transaction tobj)` Resume the transaction context association of the calling thread with the transaction represented by the supplied Transaction object.

Zusammenspiel der Objekte



Implementierung der Transaktions-ID

- Bei jeder Operation, die ein Client für eine Transaktion durchführt, muss er die Transaktions-ID angeben.
- Mögliche Implementierungen:
  - Angabe als zusätzlicher Parameter in den Operationen, z.B.
    deposit(trans, amount)
  - In Middleware wird die ID üblicherweise implizit bei allen entfernten Aufrufen angegeben, z.B. beim CORBA Transaction Service. Der Anwendungsprogrammierer muss sich darum nicht kümmern.

Ende einer Transaktion

- Eine Transaktion kann in zwei Zuständen enden:
  - Committed: die Transaktion wurde erfolgreich beendet, alle Objekte wurden erfolgreich geschrieben
  - Aborted: ein Fehler ist aufgetreten, die Transaktion wird nicht erfolgreich beendet, alle Objekte behalten ihre Zustände von vor dem Beginn der Transaktion
- Fehler werden vom Client oder vom Server ausgelöst.

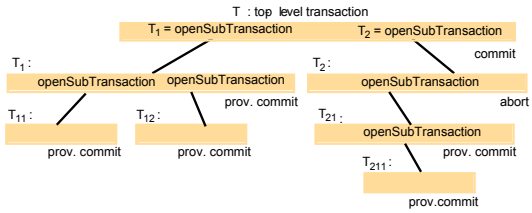
Beispiele für Transaktionshistorien

Successful	*Aborted by client*		*Aborted by server*
openTransaction	*openTransaction*		*openTransaction*
operation	*operation*		*operation*
operation	*operation*		*operation*
⋮	⋮	server aborts transaction →	⋮
operation	*operation*		*operation ERROR reported to client*
closeTransaction	*abortTransaction*		

Geschachtelte Transaktionen

- Geschachtelte Transaktionen erweitern das bisherige (flache) Transaktionsmodell, indem sie gestatten, dass Transaktionen aus anderen Transaktionen zusammengesetzt sind.
- Die Transaktion auf der höchsten Ebene wird als *top-level transaction* bezeichnet, die anderen als *subtransactions*.
- Die Verwendung geschachtelter Transaktionen bringt ein neues Problem mit sich: *was passiert, wenn eine Teiltransaktion abgebrochen wird?*

Beispiel einer geschachtelten Transaktion



Vorteile geschachtelter Transaktionen

- Zusätzliche Nebenläufigkeit:
  - Subtransactions auf der selben Hierarchieebene können nebenläufig ausgeführt werden
  - Bankenbeispiel: die Operation branchTotal() muss für sämtliche Konten die Methode getBalance() aufrufen. Man könnte jeden dieser Aufrufe als Untertransaktion starten
- Unabhängiges Commit oder Abort:
  - Dadurch werden Transaktionen potentiell robuster (hängt von der Anwendung ab)
  - Die Elterntransaktion muss jeweils entscheiden, welche Folge ein Abort der Untertransaktion haben soll.

Regeln für das Commitment geschachtelter Transaktionen

- Eine Transaktion darf nur abgeschlossen werden, wenn ihr Untertransaktionen abgeschlossen sind.
- Wenn eine Untertransaktion abschließt, entscheidet sie unabhängig, entweder provisorisch zu comitten oder endgültig abzubrechen.
- Wenn eine Elterntransaktion abbricht, werden auch alle Subtransaktionen abgebrochen.
- Wenn eine Subtransaktion abbricht, entscheidet die Elterntransaktion, was weiter geschieht.
- Wenn eine Elterntransaktion commited ist, dann können alle provisorisch committeten Untertransaktionen ebenfalls committet werden.

ACID (2): Isolation und Consistency

- Die Operationen aller Transaktionen müssen so synchronisiert werden, dass Isolation und Consistency erreicht werden.
- Einfachste Variante: Serielle Ausführung der Transaktionen
- Warum ist das nicht wünschenswert?
- Die Performance des Servers wäre sehr schlecht, mögliche nebenläufige Ausführung von Transaktionen würde nicht berücksichtigt.
- Zur Maximierung der Leistung wird deshalb versucht, die Nebenläufigkeit zu maximieren.
- Zwei Transaktionen dürfen nebenläufig ausgeführt werden, wenn diese Ausführung denselben Effekt hat wie die sequentielle Ausführung – die Ausführung heißt dann *serially equivalent*.

Concurrency Control

- Concurrency Control ist deshalb die wichtigste Aufgabe des Transaktionsmanagers.
- Aufgabe: finde möglichst nebenläufige Ablaufpläne für Transaktionen, ohne das serielle Äquivalentskriterium zu verletzen.
- Es geht also im wesentlichen darum, miteinander in Konflikt stehende Operationen korrekt einzuplanen.
- Zwei typische Probleme, die vermieden werden müssen:
  - Lost update
  - Inconsistent retrieval

Das “lost update”-Problem

Transaction *T*:	**Transaction *U*:**
balance = b.getBalance(); *b.setBalance(balance*1.1);* *a.withdraw(balance/10)*	*balance = b.getBalance();* *b.setBalance(balance*1.1);* *c.withdraw(balance/10)*
balance = b.getBalance(); $200	
	balance = b.getBalance(); $200
	*b.setBalance(balance*1.1);* $220
*b.setBalance(balance*1.1);* $220	
a.withdraw(balance/10) $80	
	c.withdraw(balance/10) $280

Korrektes Interleaving von *T* und *U*

Transaction *T*:		**Transaction *U*:**	
balance = b.getBalance()		*balance = b.getBalance()*	
*b.setBalance(balance*1.1)*		*b.setBalance(balance*1.1)*	
a.withdraw(balance/10)		*c.withdraw(balance/10)*	
balance = b.getBalance()	$200		
*b.setBalance(balance*1.1)*	$220		
		balance = b.getBalance()	$220
		*b.setBalance(balance*1.1)*	$242
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$278

Das “inconsistent retrievals”-Problem

Transaction V:		**Transaction *W*:**	
a.withdraw(100)		*aBranch.branchTotal()*	
b.deposit(100)			
a.withdraw(100);	$100		
		total = a.getBalance()	$100
		total = total+b.getBalance()	$300
		total = total+c.getBalance()	
b.deposit(100)	$300	⋮	

Korrektes Interleaving von *V* und *W*

Transaction *V*:		**Transaction *W*:**	
a.withdraw(100);		*aBranch.branchTotal()*	
b.deposit(100)			
a.withdraw(100);	$100		
b.deposit(100)	$300		
		total = a.getBalance()	$100
		total = total+b.getBalance()	$400
		total = total+c.getBalance()	
		...	

Konflikt zwischen Operationen

- Was bedeutet es, wenn zwei Operationen zueinander im Konflikt stehen?
- Ihr kombinierter Effekt hängt von der Reihenfolge ab, in der sie ausgeführt werden.
- Mit diesem Begriff lässt sich serielle Äquivalenz formaler definieren:
- *Zwei Transaktionen sind genau dann seriell äquivalent, wenn alle Paare von miteinander in Konflikt stehenden Operationen der beiden Transaktionen auf allen betroffenen Objekten in derselben Reihenfolge ausgeführt werden.*

Arten von Operationen

- Wir betrachten im folgenden vereinfachend
  - Read-Operationen (keine Änderung von Daten)
  - Write-Operationen (Änderung von Daten)

Operations of different transactions		*Conflict*	*Reason*
read	*read*	No	Because the effect of a pair of *read* operations does not depend on the order in which they are executed
read	*write*	Yes	Because the effect of a *read* and a *write* operation depends on the order of their execution
write	*write*	Yes	Because the effect of a pair of *write* operations depends on the order of their execution

Beispiel

- Gegeben seien zwei Transaktionen wie folgt:
  - T: x=read(i); write(i,10); write(j,20);
  - U: y=read(j); write(j,30); z=read(i);
- Ist der folgende Ablauf seriell äquivalent? Warum bzw. warum nicht?

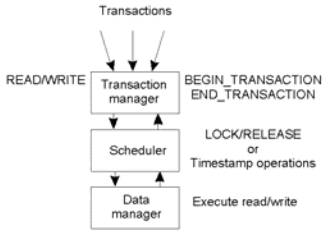
Transaction *T*:	**Transaction *U*:**
x = read(i)	
write(i, 10)	
	y = read(j)
	write(j, 30)
write(j, 20)	
	z = read (i)

Nicht äquivalent!
T greift auf i vor U zu, aber U greift auf j vor T zu.

(Beachte: der Zugriff auf einzelne Objekte ist jeweils serialisiert!)

Implementierung von Concurrency Control

- General organization of managers for handling transactions.



Algorithmen zur Concurrency Control

- Es geht also nun darum, einen Ablaufplan für zueinander in Konflikt stehende Operationen zu finden.
- Drei gängige Ansätze:
  - Locking
  - Optimistic concurrency control
  - Timestamp ordering

Locking

- Älteste und am weitesten verbreitete Form der Concurrency Control
- Einfachste Variante: exklusive Locks
  - Wenn ein Prozess Zugriff auf eine Ressource (ein Datenobjekt) benötigt, bittet er den Scheduler (über den Transaction Manager) um ein exklusives Lock
  - Wenn er es erhalten hat und seine Arbeit anschließend beendet hat, gibt er das Lock wieder zurück
- Aufgabe des Schedulers: Vergabe der Locks in einer Weise, dass nur serielle äquivalente Schedules entstehen
- Beispiel nächste Seite: U muss auf T warten

Transactions *T* und *U* mit exklusiven Locks

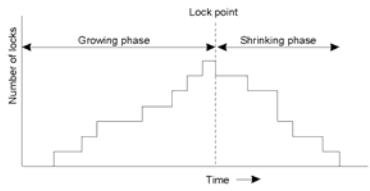
Transaction *T*:		**Transaction *U*:**	
balance = b.getBalance()		*balance = b.getBalance()*	
*b.setBalance(bal*1.1)*		*b.setBalance(bal*1.1)*	
a.withdraw(bal/10)		*c.withdraw(bal/10)*	
Operations	Locks	Operations	Locks
openTransaction			
bal = b.getBalance()	lock *B*		
*b.setBalance(bal*1.1)*		*openTransaction*	
a.withdraw(bal/10)	lock A	*bal = b.getBalance()*	waits for *T*'s lock on *B*
closeTransaction	unlock *A,B*	•••	
			lock *B*
		*b.setBalance(bal*1.1)*	
		c.withdraw(bal/10)	lock C
		closeTransaction	unlock *B,C*

Two-Phase Locking

- Bekannter Algorithmus, von dem bewiesen ist, dass er seriell äquivalente Schedules erstellt, wenn sich alle Transaktionen daran halten.
- Ablauf:
  - Bei Erhalt einer Operation op(T,x): Prüfung ob Konflikt mit anderen Operationen, für die schon ein Lock vergeben ist. Falls ja, wird op(T,x) verzögert, falls nein, bekommt T das Lock für x.
  - Der Scheduler gibt niemals ein Lock für x ab, außer der Data Manager bestätigt die Ausführung der Operation.
  - Wenn der Scheduler erst einmal ein Lock für T abgegeben hat, wird er niemals ein neues Lock für irgendein Datum für T reservieren. Jeder Versuch von T in dieser Hinsicht bricht T ab.

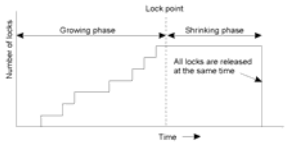
Locks beim Two-Phase Locking

- „Normales“ Two-phase locking: zuerst werden alle Locks erworben und dann nach und nach abgegeben.



Striktes Two-Phase Locking

- In dieser Variante werden alle Locks erst wieder abgegeben, wenn die Transaktion beendet ist.
- Vorteile:
  - Von anderen Transaktionen gelesene Werte sind auf jeden Fall endgültig (d.h. kein späteres Abort mehr möglich)
  - Lock Management kann unabhängig von der Transaktion durchgeführt werden (bei normalem Two-Phase Locking muss die Transaktion entscheiden, ob ein Lock freigegeben werden kann.
- 2PL und S2PL können zu Deadlocks führen.



Granularität

- Üblicherweise besitzt ein Datenserver natürlich eine größere Menge von Objekten
- Einfachste Variante: ein einziges Lock für alle Daten, Beispiel: ein Lock für alle Accounts
- Offensichtlich keine gute Lösung
- Generell schränken exklusive Locks die mögliche Nebenläufigkeit zu sehr ein, wenn man die Tabelle auf Folie 9-29 betrachtet.
- Besseres Schema: viele konkurrierende Lese- oder genau ein Schreibzugriff („many readers/single writer“)

Shared Locks

- Verbesserung: bei reinen read-Operationen können *shared locks* verwendet werden, die von mehreren Transaktionen geteilt werden
- Wenn nötig, können *read locks* zu *write locks* werden (*promotion*), allerdings nur, wenn sie nicht *shared* sind.
- Regeln für die Vergabe:

For one object		*Lock requested*	
		read	*write*
Lock already set	*none*	OK	OK
	read	OK	wait
	write	wait	wait

Shared Locks in 2PL

1. When an operation accesses an object within a transaction:
   (a) If the object is not already locked, it is locked and the operation proceeds.
   (b) If the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.
   (c) If the object has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.
   (d) If the object has already been locked in the same transaction, the lock will be promoted if necessary and the operation proceeds. (Where promotion is prevented by a conflicting lock, rule (b) is used.)
2. When a transaction is committed or aborted, the server unlocks all objects it locked for the transaction.

Mögliche Implementierung eines Locks

```
public class Lock {
        private Object object;              // the object being protected by the lock
        private Vector holders;             // the TIDs of current holders
        private LockType lockType;          // the current type
        public synchronized void acquire(TransID trans,   LockType aLockType ){
                while(/*another transaction holds the lock in conflicing mode*/) {
                        try {
                                wait();
                        }catch ( InterruptedException e){/*...*/ }
                }
                if(holders.isEmpty()) { // no TIDs  hold lock
                        holders.addElement(trans);
                        lockType  = aLockType;
                } else if(/*another transaction holds the lock, share it*/ ) ){
                        if(/* this transaction not a holder*/)
                                holders.addElement(trans);
                } else if (/* this transaction is a holder but needs a more exclusive
lock*/)
                                lockType.promote();
                }
        }
```


Mögliche Implementierung eines Locks

```
public synchronized void release(TransID trans ){
                holders.removeElement(trans);       // remove this holder
                // set locktype to none
                notifyAll();
        }
}
```

Deadlocks

- Ein Deadlock ist ein Zustand, in dem jedes Mitglied einer Gruppe von Transaktionen darauf wartet, dass ein anderes Mitglied ein Lock freigibt.
- Je feiner die Granularität bei der Concurrency Control, desto geringer die Gefahr von Deadlocks.
- Frage: kann man Deadlocks erkennen bzw. verhindern?

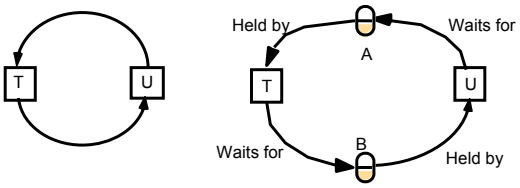
Beispiel für ein Deadlock

Transaction *T*		Transaction *U*	
Operations	**Locks**	**Operations**	**Locks**
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)			
•••	waits for *U*'s	*a.withdraw(200);*	waits for *T*'s
	lock on *B*	•••	lock on *A*
•••		•••	
•••		•••	

Wait-For-Graph

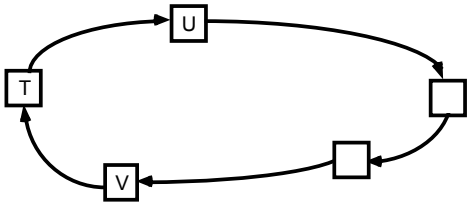
- Ein Wait-For-Graph wird verwendet, um die Wartebeziehungen zwischen Transaktionen graphisch zu beschreiben.
- Dabei
  - repräsentieren die Knoten Transaktionen und
  - die Kanten die Wartebeziehungen – es gibt eine Kante von Knoten T zu Knoten U, wenn T darauf wartet, dass U ein Lock freigibt .
- Variante: stelle auch die Locks bzw. Objekte dar, auf die eine Transaktion wartet.

Beispiel: Wait-For-Graph für 9-44



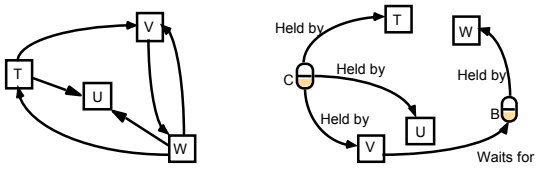
Allgemein: Zyklus im Wait-For-Graph

- Es gilt: wenn im Wait-for-Graph ein Zyklus existiert, dann ist das System in einem Deadlock



Ein weiteres Szenario

- T, U und V teilen ein read lock für c, während W ein write lock für b besitzt, auf das V zugreifen möchte.
- Jedes Objekt wartet nur auf ein Objekt, trotzdem ist V in zwei Zyklen.
- Lösung: breche V ab, dann werden die Zyklen aufgelöst



Verhindern von Deadlocks

- Einfache Lösung: erwerbe am Anfang der Transaktion die Locks für alle benötigten Objekte
- Verhindert Deadlocks, ist aber zu restriktiv
- Außerdem kann es z.B. bei interaktiven Transaktionen sein, dass am Anfang nicht bekannt ist, welche Objekte benötigt werden.
- Eher selten verwendete Lösung

Erkennen von Deadlocks

- Deadlocks werden am Wait-For-Graph erkannt: enthält er einen Zyklus, ist das System in einem Deadlock.
- Dazu kann der entsprechende Standard-Graphen-Algorithmus verwendet werden.
- Es muss dann eine Transaktion herausgesucht werden, deren Abbruch zum Brechen des Zyklus führt.
- Möglichkeit: verwende Timeouts.
  - Ein Lock einer Transaktion ist für eine bestimmte Zeit unverletzlich; danach kann es gebrochen werden
  - Eine Transaktion mit einem gebrochenem Lock wird abgebrochen.

Beispiel: Timeout-Entscheidung

Transaction T		Transaction U	
Operations	**Locks**	**Operations**	**Locks**
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)			
•••	waits for *U*'s lock on *B*	*a.withdraw(200);*	waits for T's lock on *A*
	(timeout elapses)	•••	
	T's lock on *A* becomes vulnerable, unlock *A*, abort T	•••	
		a.withdraw(200);	write lock *A* unlock *A*, *B*

Optimistic Concurrency Control

- Erkenntnis: Locking hat einige gravierende Nachteile:
  - Locks müssen immer verwendet werden, auch wenn es gar nicht nötig wäre (z.B. bei reinen Reads) → unnötiger Overhead
  - Gefahr von Deadlocks
  - Reduzierung der Nebenläufigkeit durch spätes Abgeben von Locks
- Alternative: verwende den optimistischen Ansatz, dass es sehr selten Konflikte geben wird
- Alle Transaktionen arbeiten, als wären sie die einzigen.
- Nur, wenn ein Konflikt auftritt, muss am Ende eine der Transaktionen abgebrochen werden und deren Ergebnisse verworfen werden.

Phasen der Transaktion

1. Working Phase:
   die Transaktion besitzt eine eigene Kopie der notwendigen Daten und arbeitet auf diesen
2. Validation Phase
   Nach Abschluss der Operationen der Transaktion wird überprüft, ob es Konflikte mit anderen Transaktionen gibt. Wenn ja, müssen diese Konflikte gelöst werden
3. Update Phase
   Wurde die Transaktion validiert, werden die Daten permanent gemacht.

Validierung von Transaktionen

- Natürlich werden auch hier die Konfliktregeln verwendet, um die serielle Äquivalenz mit allen überlappenden Transaktionen zu garantieren.
- Überlappend: Transaktionen, die noch nicht abgeschlossen waren, als die neue Transaktion startete.
- Transaktionen werden durchnummeriert in der Reihenfolge ihres Eintritts in die Validierungsphase. Das heisst T_i geht vor T_j, wenn $i<j$.

Serialisierbarkeit einer Transaktion

- Der Validierungstest für eine Transaktion T_v basiert auf den Konflikten mit den überlappenden Transaktionen T_i.
- Um die Serialisierbarkeit herzustellen, müssen die folgenden Regeln erfüllt werden:

T_v	T_i	*Rule*
write	*read*	1. T_i must not read objects written by T_v
read	*write*	2. T_v must not read objects written by T_i
write	*write*	3. T_i must not write objects written by T_v and T_v must not write objects written by T_i

Validierungsstrategien

- Bei Rückwärtsvalidierung werden die Regeln mit den Transaktionen überprüft, die vorher in die Validierungsphase eingetreten sind.
- Bei Vorwärtsvalidierung werden die Transaktionen einbezogen, die später begonnen haben und noch aktiv sind.
- Beide Techniken haben ihre Anwendungen, meist ist jedoch Vorwärtsvalidierung vozuziehen.

Timestamp Ordering

- Idee: jede Transaktion bekommt einen *eindeutigen* Zeitstempel *ts(T)* (ermittelt mit Hilfe von Lamport-Uhren)
- Jede Operation einer Transaktion besitzt diesen selben Zeitstempel.
- Außerdem besitzt jedes Datenobjekt einen Lesezeitstempel $ts_{RD}(x)$ und einen Schreibezeitstempel $ts_{WR}(x)$.
- $ts_{RD}(x)$ enthält den Wert $ts(T_l)$, wobei T_l die Transaktion ist, die als letzte lesend auf x zugegriffen hat.
- Für $ts_{WR}(x)$ gilt dasselbe bzgl. der letzten schreibenden Transaktion. Schreibzugriffe sind zunächst tentativ, bis zum commit.

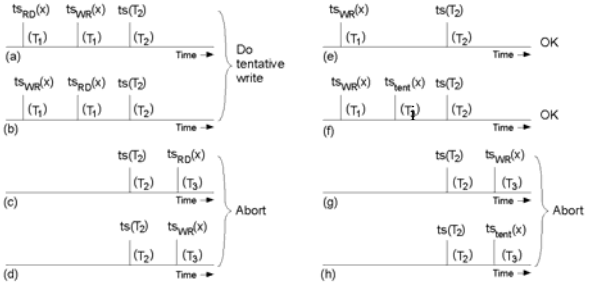
Konfliktlösung

- Situation 1: der Scheduler erhält ein read(t,x) mit Zeitstempel ts.
  - Wenn ts< $ts_{WR}(x)$, dann wurde die letzte Write-Operation nach dem Start von T durchgeführt → T wird abgebrochen
  - Wenn ts> $ts_{WR}(x)$, dann darf das read stattfinden. Setze außerdem $ts_{RD}(x)$ auf max(ts, $ts_{RD}(x)$).
- Situation 2: der Scheduler erhält ein write(T,x) mit Zeitstempel ts.
  - Wenn ts< $ts_{RD}(x)$, dann wird T abgebrochen, da eine jüngere Transaktion x gelesen hat. T ist sozusagen zu spät dran.
  - Wenn aber ts>$ts_{RD}(x)$, dann darf der Wert von x geändert werden, da keine jüngere Transaktion den Wert gelesen hat. Setze außerdem $ts_{WR}(x)$ auf max(ts, $ts_{WR}(x)$).

Beispiel

- Drei Transaktionen T1, T2, T3
- T1 wurde abgeschlossen, bevor T2 und T3 begannen, d.h., alle read- und write-Zeitstempel stehen auf ts(T1).
- T2 und T3 werden nebenläufig ausgeführt, mit ts(T2)<ts(T3).
- Betrachten wir einige Beispielsituationen (nächste Folie):
  - T2 schreibt x, ohne dass T3 bereits zugegriffen hätte (a) und (b)
  - T2 schreibt x, aber T3 hat vorher gelesen (c) oder geschrieben (d)
  - T2 liest x ohne Konflikt (e). In (f) muss T2 warten, bis Ti committed hat.
  - T2 liest x, aber T3 hat schon comitted (g) bzw. ist gerade dabei (h) → abort T2

Beispiel



Vergleich der Concurrency-Control-Techniken

- Locking
  - + Gut geeignet für Transaktionen mit vielen Updates
  - – Bei dominierenden Read-Operationen ist Timestamp besser
- Optimistische Concurrency Control
  - + Deadlock-frei
  - + Maximale Ausnutzung von Nebenläufigkeit, deshalb sehr gut bei konfliktfreien Situationen
  - – Transaktionen müssen manchmal wiederholt werden (vor allem schwierig unter hoher Last)
- Timestamps
  - + Deadlock-frei
  - – Bricht Transaktionen auch ab, wenn es bei Locking nicht nötig wäre

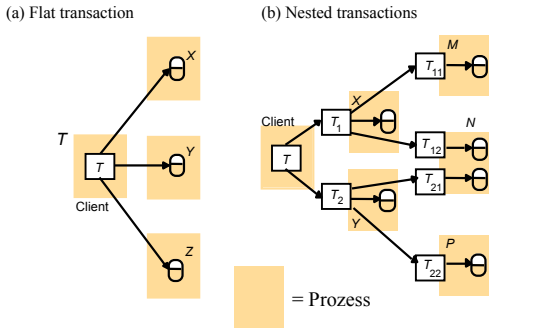
Verteilte Transaktionen

- Bisher haben wir Transaktionen betrachtet, die auf Objekte auf einem einzigen Server zugreifen.
- Oft jedoch werden die Objekte bzw. Operationen über mehrere Server verteilt sein
- Beispiel: Buchung einer Reise
  - Flug
  - Hotel
  - Bustransfers
  - Tagesausflüge

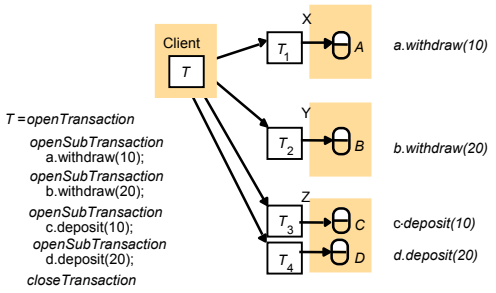
Struktur verteilter Transaktionen

- Verteilte Transaktionen können wiederum flach oder geschachtelt sein.
- Bei einer flachen verteilten Transaktion greift der Client nacheinander auf die beteiligten Server zu.
- Bei der Verwendung von Subtransaktionen kann von der Parallelität der verschiedenen Server Gebrauch gemacht werden.

Struktur verteilter Transaktionen



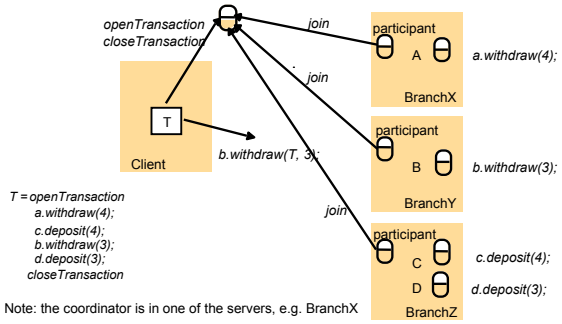
Bankenbeispiel



Koordination der Server

- Zur korrekten Abwicklung der Transaktion müssen die Server ihre Aktionen koordinieren.
- Dazu wird für jede Transaktion ein Koordinator bestimmt (typischerweise in einem der Server).
- Zum Start der Transaktion sendet der Client ein openTransaction() an den Koordinator. Dieser liefert dann eine eindeutige Transaktions-ID zurück.
- Der Koordinator entscheidet, ob eine verteilte Transaktion abgebrochen oder korrekt beendet wird.
- Er kennt alle Teilnehmer, die wiederum alle ihn kennen.
- Zur Kooperation wird ein Commit-Protokoll verwendet.

Beispiel für die Koordination



Note: the coordinator is in one of the servers, e.g. BranchX

Commit-Protokolle

- Commit-Protokolle gibt es seit den frühen 70ern; das berühmte Two-Phase-Commit wurde 1978 von Jim Gray vorgestellt (Turing-Preisträger 1999)
- Aufgabe: Atomarität der verteilten Transaktion herstellen
- One-Phase Commit-Protokolle sind zu einfach; sie erlauben es den beteiligten Sub-Transaktionen nicht, einen Abbruch an den Koordinator zu melden, der ja dann wieder die anderen informieren müsste.

Das Two-Phase-Commit-Protokoll (2PC)

- Das Protokoll beginnt erst, wenn der Client die Transaktion beendet.
- Aufteilung der Phasen
  - Phase 1: Der Koordinator fragt alle Beteiligten, ob sie zum Commit bereit sind.
  - Phase 2: Der Koordinator teilt den Beteiligten die Entscheidung (commit/abort) mit und fordert sie auf, die entsprechenden Maßnahmen zu treffen.
- Betrachten wir die Operationen des Protokolls.

Operationen von 2PC

canCommit?(trans)-> Yes / No
Call from coordinator to participant to ask whether it can commit a transaction. Participant replies with its vote.

doCommit(trans)
Call from coordinator to participant to tell participant to commit its part of a transaction.

doAbort(trans)
Call from coordinator to participant to tell participant to abort its part of a transaction.

haveCommitted(trans, participant)
Call from participant to coordinator to confirm that it has committed the transaction.

getDecision(trans) -> Yes / No
Call from participant to coordinator to ask for the decision on a transaction after it has voted *Yes* but has still had no reply after some delay. Used to recover from server crash or delayed messages.

Ablauf des Protokolls

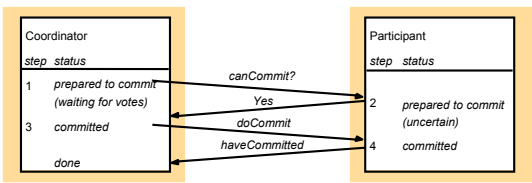
Phase 1 (voting phase):

1. The coordinator sends a *canCommit*? request to each of the participants in the transaction.
2. When a participant receives a *canCommit*? request it replies with its vote (*Yes* or *No*) to the coordinator. Before voting *Yes*, it prepares to commit by saving objects in permanent storage. If the vote is *No* the participant aborts immediately.

Phase 2 (completion according to outcome of vote):

3. The coordinator collects the votes (including its own).
   (a) If there are no failures and all the votes are *Yes* the coordinator decides to commit the transaction and sends a *doCommit* request to each of the participants.
   (b) Otherwise the coordinator decides to abort the transaction and sends *doAbort* requests to all participants that voted *Yes*.
4. Participants that voted *Yes* are waiting for a *doCommit* or *doAbort* request from the coordinator. When a participant receives one of these messages it acts accordingly and in the case of commit, makes a *haveCommitted* call as confirmation to the coordinator.

Grafische Darstellung

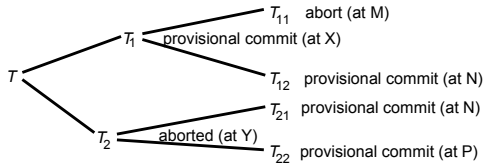


2PC mit geschachtelten Transaktionen

- Etwas komplexere Situation:
  - Es gibt weitere Koordinatoren, die die Untertransaktionen verwalten
  - Untertransaktionen können selbständig entscheiden, ob ein ABORT oder COMMIT stattfindet, ABER: der jeweilige Koordinator kann die Transaktion auch bei einem Abbruch einer Untertransaktion erfolgreich abschließen
- Untertransaktionen führen deshalb zunächst ein provisorisches Commit aus.

Beispiel (zu Folie 10-64 (b))

- T21 und T22 werden im Endeffekt abgebrochen, da T2 abgebrochen wird.
- Wenn T entscheidet, trotzdem zu committen, dann kann auch T1 und damit T12 committen.
- Es wird dann ein 2PC zwischen allen provisorisch committeten Teilnehmern ausgeführt.

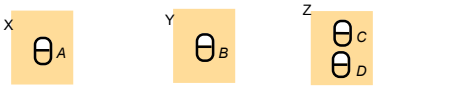


Distributed Deadlocks

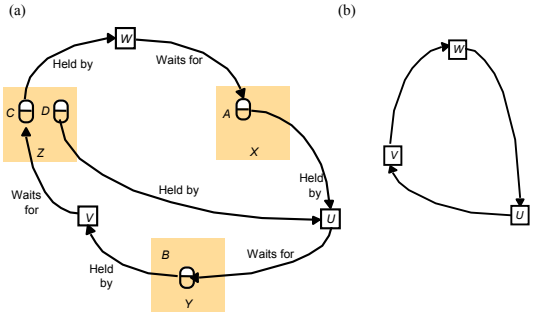
- In verteilten Transaktionen wird auch das Problem der Deadlocks noch einmal eine Stufe schwieriger → verteilte Deadlocks können entstehen.
- Verteilte Deadlocks können u.U. nicht am lokalen Wait-For-Graphen erkannt werden.
- Vielmehr muss ein globaler Graph untersucht und dieser dann auf Zyklen untersucht werden.

Beispiel für verteiltes Deadlock

U		V		W	
d.deposit(10)	lock *D*				
		b.deposit(10)	lock *B* at *Y*		
a.deposit(20)	lock *A* at *X*				
				c.deposit(30)	lock *C* at *Z*
b.withdraw(30)	wait at *Y*				
		c.withdraw(20)	wait at *Z*		
				a.withdraw(20)	wait at *X*



Der globale Wait-For-Graph



Konstruktion des globalen Graphen

- Der globale Wait-For-Graph wird aus den lokalen Graphen konstruiert.
- Einfachste Lösung:
  1. Der zentrale Koordinator sammelt die lokalen Graphen und konstruiert den Graphen.
  2. Anschließend untersucht er ihn auf Deadlocks.
  3. Er fällt eine Entscheidung und informiert die betreffenden Server über die Notwendigkeit eines Transaktionsabbruchs.
  4. Gehe zu 1
- Nicht immer gut wegen der üblichen Probleme (Bottleneck, single point of failure)

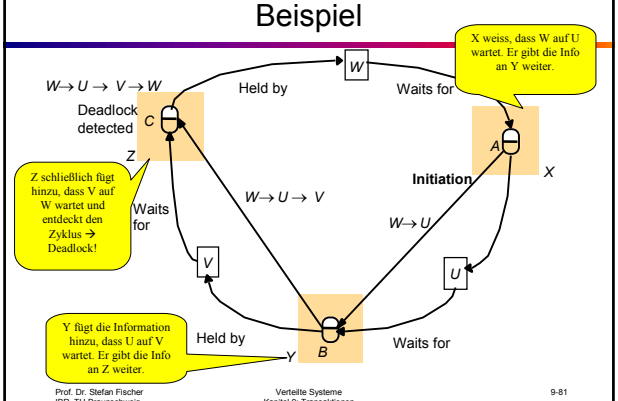
Phantom-Deadlocks

- Ein Phantom-Deadlock ist ein entdecktes Deadlock, das in der Realität jedoch gar nicht existiert.
- Woher kann das kommen?
- Genau von dem Hauptproblem verteilter Systeme, dass sich ein Zustand zu einem gegebenen Zeitpunkt nicht feststellen lässt.
- Das heisst, der Wait-For-Graph kann eine Mischung aus älteren und neueren Daten sein; ein bisher belegtes Lock kann inzwischen schon wieder freigegeben sein.
- Kann nicht passieren in 2PL

Verteilte Ermittlung des Graphen

- Es gibt verteilte Algorithmen zur Ermittlung des globalen Graphen
- Algorithmus „Edge Chasing“
- Idee:
  - Konstruiere den Graphen nicht, aber gebe den einzelnen Servern Informationen über einige seiner Kanten
  - Die Server versuchen Zyklen zu finden, indem sie „probes“ entlang der Pfade des Graphen durch das System schicken.
  - Eine solche Nachricht enthält Informationen über bekannte Pfade des Graphen.

Beispiel



Weitere Literatur

- P. Lewis et al: *Databases and Transaction Processiong – An Application-Oriented Approach*, Addison-Wesley, 2001.