Übung: Transaktionale Systeme

22.04.2010

Jürgen Broß
<juergen.bross@fu-berlin.de>

Organisatorisches

- Aufbau der Übung
  - Fragen zur Vorlesung → Bitte möglichst schon am Mittwoch in Google-Docs-Dokument einfügen
  - Aktuelles Übungsblatt
  - Projektfortschritt
  - Vertiefung des Vorlesungsstoffs
  - Vorbesprechung Übungsblatt

- Übungsbetrieb
  - Ausgabe Übungszettel: Donnerstagnachmittag
  - Abgabe: elektronisch (per Mail) bis Do. 10 Uhr, ausgedruckt in der Übung
  - Erfolgreiche Bearbeitung gilt als aktive Teilnahme
  - Umfang abgestimmt auf Projektarbeit
  - 2er Gruppen

Organisatorisches

- Projektzeitplan:

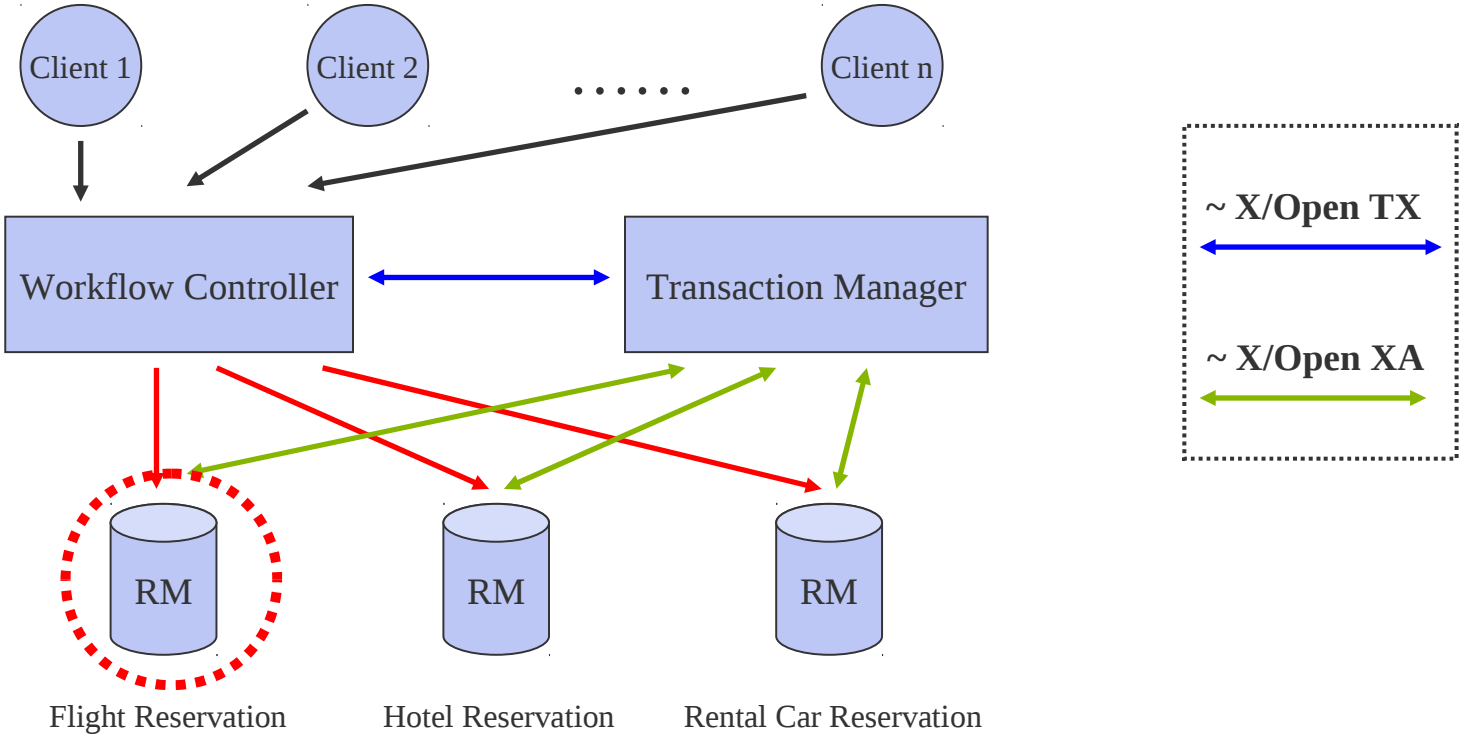
Projektteil	Start	Abgabe	Besprechung
A	22.04.2010	27.05.2010	27.05.2010
B	27.05.2010	01.07.2010	01.07.2010

- Teilaufgaben auf Übungszetteln
- Entwicklungsumgebung:
  - Java 6
  - Eclipse
  - JUNIT 4
  - Java RMI

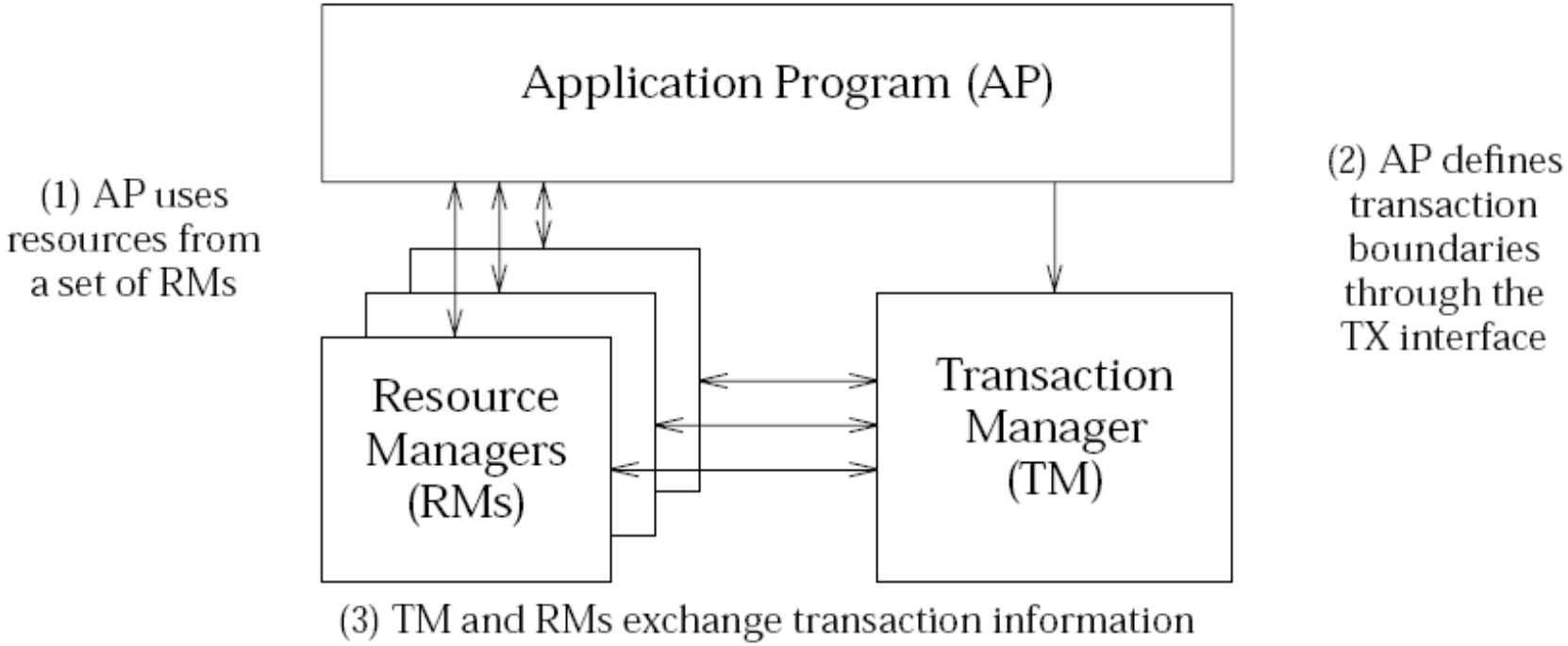
Projekt: Überblick

- Ziel: Verteiltes transaktionales Reisebuchungssystem

  →Eine Reise besteht aus Flügen, Hotels, Mietwagen
  →Einzelne Leistungen werden bei verschiedenen Anbietern gebucht



X/Open DTP Modell



Projekt: Teil A

- Implementierung eines Lock-Managers (LM)
- Implementierung eines Resource Managers (RM)
- Anforderungen:
  - RM verwaltet die Entitäten Flights, Hotels, Cars, Customers, Reservations (spezialisiert auf Reisebuchungen)
  - Unterstützt ACID-Transaktionen:
    - Atomicity: per „Schattenspeicher-Mechanismus“
    - Isolation: 2PL, (LockManager zu implementieren)
    - Durability: Einfacher Recovery-Mechanismus
  - Paralleler Zugriff von verschiedenen, entfernten Klienten
    → Isolation
    → RMI

Projekt: Teil A

- Lock Manager:
  - Interface ist gegeben
  - Verwaltet Lese-/Schreibsperren auf Datenbankobjekten (identifiziert über eindeutigen Bezeichner)
  - Deadlock-Erkennung durch Wartegraphen
  - Pro Resource Manager eine Lock Manager Instanz
- Schnittstelle:
  - *lock(int xid, String strData, LockType lockType) throws DeadLockException*
    - strData := eindeutiger Bezeichner für zu sperrendes Datenbankobjekt
    - lockType := LockType.READ | LockType.WRITE (ENUM)
    - blockierende Operation
    - implizites lock upgrade
  - *unlockAll(int xid)*

Projekt: Teil A

- (monopolistisches;-)) Datenmodell:
  - **FLIGHTS** (String flightNum, int price, int numSeats, int numAvail)
    - Es existiert nur eine Fluglinie „WorldAir“, alle Sitze in einem Flug haben den gleichen Preis
  - **HOTELS** (String location, int price, int numRooms, int numAvail)
    - Alle Hotelzimmer in einem Hotel haben den gleichen Preis, location ist PK → nur ein Hotel pro Ort
  - **CARS** (String location, int price, int numCars, int numAvail)
    - Alle Mietwagen an einem Ort haben den gleichen Preis, location ist PK → nur ein Mietwagenanbieter pro Ort
  - **CUSTOMERS** (String custName)
  - **RESERVATIONS** (String custName, int resvType, String resvKey)
    - resvType (1=FLIGHT, 2=HOTEL, 3=CAR)

Projekt: Teil A

- Resource Manager Interface

1. **Administrative Schnittstelle**:
   - addFlight, addRooms, addCars, newCustomer
   - deleteFlight, deleteRooms, deleteCars, deleteCustomer

1. **Reservierungs-Schnittstelle**:
   - reserveFlight,…

1. **Anfrage-Schnittstelle**:
   - queryFlight, queryFlightPrice, …

1. **Transaktions-Schnittstelle**:
   - start, commit, abort

1. ***Test-Schnittstelle***
   - shutdown, dieNow,…

Projekt: Teil A

- Transaktionen im Resource Manager
  - Aufruf von start() beginnt neue Transaktion und liefert eindeutige Transaktionskennung (XID)
  - Alle nachfolgenden Operationen auf dem RM werden mit der XID parametrisiert → Zugehörigkeit zu einer Transaktion
  - Beispiel:

```
int xid = rm.start
//If it‘s cheap enough and there are seats available
if(rm.queryFlightPrice(xid, ‘347‘) < 300 &&
   rm.queryFlight(xid, ‘347‘) > 0){

   rm.reserveFlight(xid, ‘John‘, ‘347‘);
}
rm.commit(xid);
```

Projekt: Teil A

- Schritt 1: Lock Manager
  - Nutze die Klassen aus dem *java.util.concurrent* Paket
  - Insbesondere hilfreich: *java.util.concurrent.locks*

  - Nichterhalt einer Sperre blockiert die anfordernde Transaktion (bzw. den anfordernden Thread)
  - Lockupgrade von Read- auf Write-Lock möglich (wenn keine anderen Leser)
  - Aufgabe aller Sperren gleichzeitig zum Ende einer Transaktion (eignet sich für rigoroses 2PL)

  - Abgabe: jar-Datei mit einzelner Klasse *lockmgr.LockManager*, die das *lockmgr.ILockManager* Interface implementiert
  - Zeitraum: 2 Wochen, nach erster Woche Vorstellung des Konzepts

Projekt: Teil A

- Schritt 2: Basis RM
  - Ignoriere Atomarität, Persistenz
  - Alle Daten werden im Speicher gehalten
    → z.B. Hashtable für jede Relation (Flights, …)
    →PK ist Schlüssel für Hashtable
    →Zeile in Relation ist Eintrag in Hashtable
  - Reservierungen evtl. mit Kunden Tabelle verknüpfen
    →Hashtable mit Kundennamen als Schlüssel und Liste von Reservierungen als Einträge
  - Beachte Einfachheit des Datenmodells:
    addCars(T1, ‘Berlin‘, 4, 100€)
    addCars(T2, ‘Berlin‘, 7, 80€)
    →11 Autos á 80€ in Berlin
  - Zeitraum: 1/2 Woche

Projekt: Teil A

- Schritt 3: Atomarität
  - Gewährleistet durch Schattenspeicherkonzept
  - Führe die Updates einer Transaktion auf einer Kopie der Datenbasis aus
    →z.B. separate Hashtabellen
  - Bei commit:
    → Bringe Updates in die eigentliche Datenbasis (im Speicher) ein.
    → Schreibe Datenbasis in separate Datei auf Hintergrundspeicher.
    (stetiger Wechsel zwischen zwei Dateien)
    → In atomarem Schritt ändere globalen Zeiger so, dass die neue Datei als aktive Datenbasis markiert ist
  - Bei abort:
    → Verwerfe die separaten Hashtabellen (im Speicher)
  - Zeitraum: 1/2 Woche

→Shadowspeicher (Originalliteratur)
http://www.inf.fu-berlin.de/lehre/WS06/DBS-Tech/Reader/shadowStorage.pdf

Projekt: Teil A

- Schritt 4: Isolation
  - Implementierung von 2PL (rigoros) mithilfe des Lock Managers
    →für jede Operation auf RM Lese- oder Schreibsperre anfordern
    →bei commit/abort alle Lese-/Schreibsperren aufgeben
  - Granularität der Locks so fein wie möglich für eine Operation
    →maximale Parallelität
    →möglichst keine Table-Locks
  - Lock Manager erkennt Deadlocks durch Wartegraph
    →DeadLockException
    →betroffene Transaktion abbrechen (abort)
  - Zeitraum: 1 Woche

Projekt: Teil A

- Schritt 5: Dauerhaftigkeit
  - Persistenz und Recovery
  - Zustand des Resource Managers auf Hintergrundspeicher schreiben:

    →aktive Datenbasis
    →Zustand von Transaktionen (start, commit, abort)
    →XIDs
    →Zeiger auf aktive Datenbasis

  - Recovery, falls RM nicht korrekt heruntergefahren wurde (dieNow(), dieBeforePointerSwitch(), dieAfterPointerSwitch())

    →Welche Transaktionen waren vor Absturz bekannt?
    →Welchen Zustand hatten diese Transaktionen?

  - Zeitraum: 1 Woche

Projekt: Teil A

- Test-Schnittstelle:
  - shutdown():
    → warten, bis alle Transaktionen beendet sind
    → keine neuen zulassen
    → „korrekten“ Zustand hinterlassen, so dass keine Recovery beim nächsten Start notwendig ist
  - dieNow(): sofortiges System.exit();
  - dieAfter/BeforePointerSwitch():
    → setze Flag, so dass System.exit() während der nächsten Commit-Operation aufgerufen wird
    →direkt vor bzw. direkt nach dem Ändern des Zeigers auf aktive Datenbasis

Projekt: Teil A

Projektstart: **22.04.2010**

Projektabgabe: **27.05.2010**

- Bis nächste Woche:
  - → Projektbeschreibung gründlich anschauen
  - → Konzept für Lock Manager (ggf. Implementierung)
  - → Fragen aufschreiben!

→ theoretische Aufgaben bearbeiten