Transactional data processing

INF 5040 autumn 2008

lecturer: Roman Vitenberg

Introduction

- Servers can offer concurrent access to the objects/data the service encapsulates
- Application frequently needs to perform sequences of operations as undivided units
  - => atomic transactions
- The server can offer persistent storage of objects/data
  - => motivation for continued operation after a server process has failed
- Service can be provided by a group of servers
  - => distributed transactions

Transactional service

- Offers access to resources via transactions
  - Cooperation between clients and transactional servers
- Operations of transactional services

OpenTransaction() $\rightarrow$ *TransId*
CloseTransaction(TransID) $\rightarrow$ *{commit, abort}*
AbortTransaction (TransID) $\rightarrow$ *{}*

- All operations between OpenTransaction and CloseTransaction are said to be performed in a transactional context

INF5040, Roman Vitenberg 3

Completing a transaction

- *Commit point* for transaction T
  - All operations in T that access the server database are successfully performed
  - The effect of the operations is made permanent (typically by recording them in a log)
- We say that transaction T is "committed"
  - The service (or the database system) has put itself under an obligation
  - The results of T are made permanent in the database

INF5040, Roman Vitenberg 4

Desirable properties of transactions

- *Failure atomicity* (all-or-nothing semantics)
  - The effect is atomic even if the server fails
- Two common implementations:
  - Private copy
  - Log file
- Log file:
  - Updates are written directly to the database
  - Log file includes an undo record
    - Transaction id, operation type (read/write), previous value, new value
  - If committed, write commit in log
  - If abort, roll back transaction

Desired properties of transactions

- *Isolation*
  - Intermediate results of a transaction must be invisible to other transactions
  - => need for synchronization (concurrency control)
  - Sequential execution
    - Ensures isolation but ruins the performance
  - Serializable execution ("serial equivalence")
    - The effect of transactions in an interleaved execution must be as if the transactions were executed in some sequential order
      - The data read as part of the transactions
      - The eventual state of the database (all data values)
    - Ensured by concurrency control algorithms

Implementing isolation

- Concurrency: interleaving of operations from different transactions
  - Better system utilization
  - Shorter response time
- Interleaving of operations may potentially cause problems
  - The problem of lost updates
  - The problem of visible intermediate results (inconsistent retrieval or "dirty read")
  - The problem of premature write
  - The problem of cascading aborts

The problem of lost updates

x: database element
T1: x = x + 1000
T2: x = x + 50

Concurrent execution		Value in the database
T1:	read(x)	500
	x = x +1000	
T2:	read(x)	500
	x = x + 50	
T1:	write(x)	1500
T2:	write(x)	550

The performed update of T1 disappears

Visible intermediate results (inconsistent retrieval)

T1: transfer of 100 from A to B
T2: calculates A + B

Execution (schedule)

T1: read(A)
read(B)
A=A-100
write(A)
T2: read(A)
read(B)
sum= A + B
T1: B=B+100
write(B)

T2 sees a semi-updated database with the new value of A but old value of B.

Visible intermediate results ("premature write")

X: database element
T1: x = x + 1000
T2: x = x + 50

Execution			Value in the database
T1:	read(x)	⇐	500
	x = x +1000		
	write(x)	⇒	1500
T2:	read(x)	⇐	1500
	x = x + 50		
	write(x)	⇒	1550
	commit T2		
T1:	abort T1		

T2 bases its update on a temporary value of x ("dirty read").
The transactions that has produced this value aborts
=> Failure in the execution of T2: **not recoverable!!**
=> T2 must delay its commit until T1 has terminated

Problem of cascading aborts

X: database element
T1: x = x + 1000
T2: x = x + 50

Execution			Database value
T1:	read(x)	⇐	500
	x = x +1000		
	write(x)	⇒	1500
T2:	read(x)	⇐	1500
	x = x + 50		
	write(x)	⇒	1550
T1:	abort		

T2 bases the update on a temporary value of x and waits with performing `commit`. The transaction that has produced that value (T1) aborts
=> Failure in the execution of T2

=> T2 must abort

If other transactions have seen T2's temporary values
=> Those must abort too

This situation is called **cascading aborts**

Prevent **cascading aborts:** Transactions can only read data objects from transactions that have already performed `commit`.

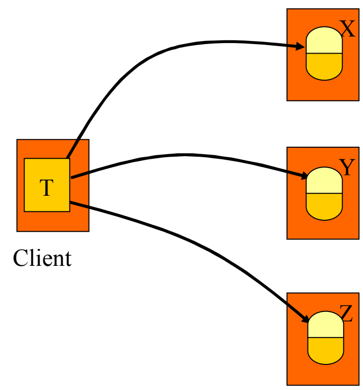
INF5040, Roman Vitenberg 11

Summary:
Desirable properties of transactions

- *Atomicity*: All-or-nothing semantics
- *Consistency*: Ensures that the data is manipulated correctly. Generally assumed to be responsibility of the programmer
- *Isolation*: Transaction does not make its own updates visible to other transactions before it has performed "commit". Implemented by concurrency control methods
- *Durability:* When a transaction has performed "commit", its effect in the database is never lost due to later a failure.
- Collectively called ACID properties ...

INF5040, Roman Vitenberg 12

Distributed transactions



Client transaction that invokes operations on multiple servers

INF5040, Roman Vitenberg 14

Component roles

- Distributed system components that are involved in a transaction can have a role as:

- Transactional client
- Transactional server
- Coordinator

INF5040, Roman Vitenberg 16

Coordinator

- Plays a key role in managing the transaction
- The component that handles begin/commit/abort operations
- Allocates globally unique transaction identifiers
- Includes new servers in the transaction (`Join` operation) and monitors all the participants
- Typical implementation
  - The first server that the client contacts (by invoking `OpenTransaction`) becomes a coordinator for the transaction

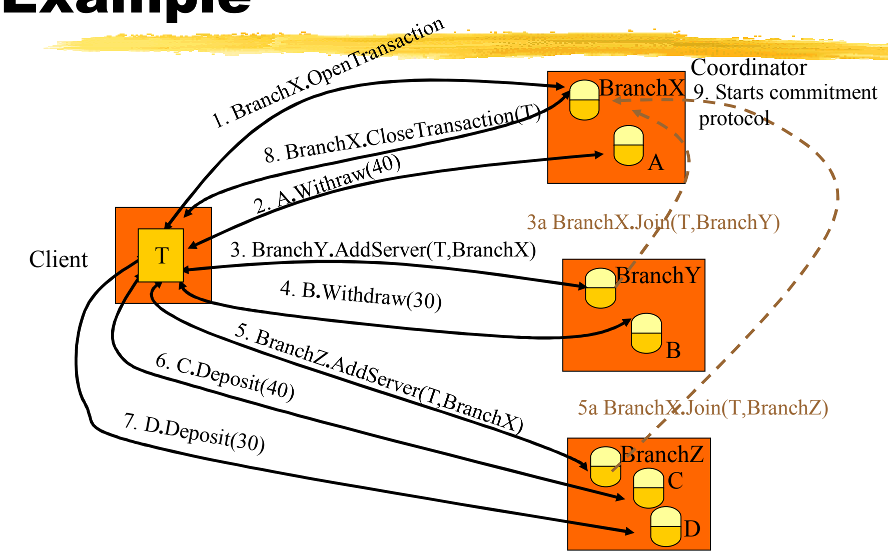
Transactional server

- Serves a proxy for each resource that is accessed or modified under transactional control
- Transactional server must know its coordinator
  - via parameter in the `AddServer` operation
- Transactional server registers its participation in the transaction via the coordinator
  - By invoking the `Join` operation at the coordinator.
- Transactional server must implement a transaction protocol (such as two-phase commit - 2PC)

Transactional client

- Sees the transaction only through coordinator
  - Invokes operations at the coordinator
    - Open Transaction
    - CloseTransaction
    - AbortTransaction
- The implementation of the transaction protocol (such as 2PC) is transparent for the client

Example



The non-blocking atomic commit problem (intuition)

- Multiple autonomous distributed servers
- Prior to committing the transaction, all the transactional servers must verify that they can locally perform commit
- If any server cannot perform commit, all the servers must perform abort

The non-blocking atomic commit problem (formal)

- Uniform agreement
  - All processes that decide, decide on the same value
  - Decisions are not reversible
- Validity
  - Commit can only be reached if all processes vote for commit
- Non-triviality
  - If all voted commit and there are no (suspicions of) failures, then the decision must be commit
- Termination
  - If after some time there are no more failures, then eventually all live processes decide

2-PC protocol

- One-phase protocol is insufficient
  - Does not allow a server to perform unilateral abort
    - E.g., because of a deadlock
- Rationale for two phases
  - Phase one: agreement
  - Phase two: execution

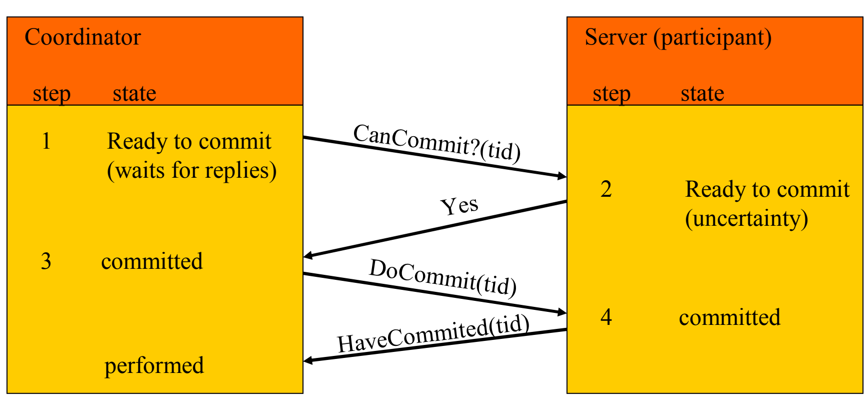
Phase one: agreement

- Coordinator asks all servers if they are able to perform commit (`CanCommit?(T)` call)
- Server response:
  - **Yes**: will perform commit if the coordinator requests, but the server does not know yet if it will perform commit
    - Determined by the coordinator
  - **No**: the server performs immediate abort of the transaction
- Servers can unilaterally perform abort, but they cannot unilaterally perform commit

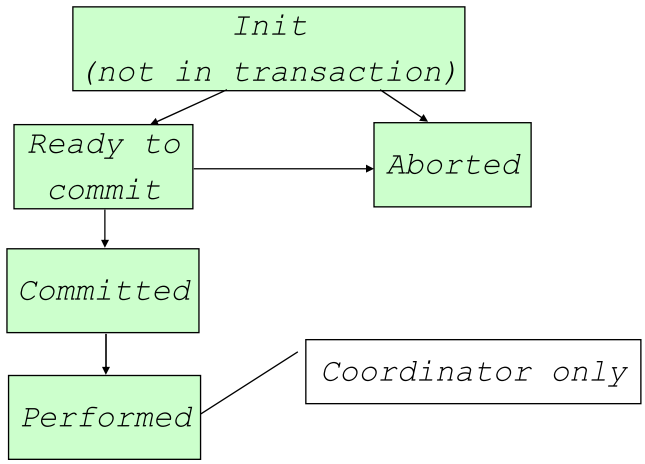
Phase two: execution

- Coordinator collects all replies from the servers, including itself, and decides to perform
  - commit, if all replied **Yes**
  - abort, if at least one replied **No**
- Coordinator propagates its decision to the servers
- All participants perform
  - `DoCommit(T)` call if the decision is commit
  - `AbortTransaction(T)` call otherwise
- If the decision is commit, the servers notify the coordinator right after they have performed `DoCommit(T)`
  - call `HaveCommited(T)` back on the coordinator

The 2PC protocol



2PC state diagram



2PC: when a previously failed server recovers

	Coordinator	Participant
Init	Nothing	Nothing
Ready	AbortTransaction	GetDecision(T)
Committed	Sends DoCommit(T)	Sends HaveCommitted(T)
Performed	Nothing	

2PC: when a process detects a failure

- What happens if a coordinator or a participant does not receive a message it expects to receive?
- For a participant in the "Ready" state
  - Figure out the state of other participants
  - What if all remaining participants are in the "Ready" state?
- This is known as blocking
  - There are more advanced protocols (3PC) that block in fewer cases
    - Impose higher overhead during normal operation
    - 2PC is the most widely used protocol
  - If the network might partition, blocking is unavoidable

Summary

- Two-phase commit
  - Phase one: agreement
  - Phase two: execution
- CORBA Transaction Service
  - Implements 2PC
  - Requires resources to be "transaction-enabled"
- Transactions and EJB
  - programmatic & declarative transactions
  - Container provides support for distributed transactions
    - based on CORBA OTS and X/Open XA protocol
  - EJB container/server implements Java Transaction API (JTA) and Java Transaction Service (JTS)
- Extended transaction models & OASIS BTP
  - B2B transactions