

# CmpE 494: Service-Oriented Architectures and Web Services

## Transactions

Based largely on:

*Service-Oriented Computing: Semantics, Processes, Agents*

– Munindar P. Singh and Michael N. Huhns, Wiley, 2005

# Motivation

As services are employed for serious purposes, they will inevitably update information resources

- Can we be sure that such updates preserve integrity?
- What about when multiple services need to work together?
- What about when the services are involved in a long-lived activity?

# Transactions: 1

- A transaction is a computation (i.e., program *in execution*, not the source code or binaries) that accesses and possibly modifies a database:
  - Can be interleaved with other transactions
  - But guarantees certain properties

The purpose of the transaction concept is to avoid the problems that may arise from interleaving

# Transactions: 2

- *Operation*: action on a data item
- *Transaction*: set of operations performed in some partial order according to the specifying program

*A transaction makes a set of operations appear as one logical operation*

If programmers guarantee that their transactions are individually correct, then the DBMS guarantees that all interleaved executions are also correct

# ACID Properties

- These formalize the notion of a transaction behaving as one operation
  - (Failure) *Atomicity*—all or none—if failed then no changes to DB or messages
  - *Consistency*—don't violate DB integrity constraints: execution of the op is correct
  - *Isolation* (Atomicity)—partial results are hidden
  - *Durability*—effects (of transactions that "happened" or committed) are forever

# Transaction Lifecycle

A transaction goes through well-defined stages in its life (always terminating)

- inactive
- active (may read and write: this is where the entire business logic occurs)
- precommit (no errors during execution; needed for mutual commitment protocols)
- failed (errors)
- committed (the DBMS decides this)
- forgotten (the DBMS reclaims data structures)

# Schedules

- Schedules are histories of computations showing all events of interest
- Schedule of  $T_1 \dots T_n$  has ops of  $T_1 \dots T_n$  in the same order as within each  $T_i$ , but *interleaved* across  $T_i$  to model concurrency
  - Includes active transactions
  - Typically a partial order among events
- Two challenges
  - What are the bad schedules?
  - How can the DBMS prevent them?

# Conflict

## Order-sensitivity of operations

- Two operations of different transactions, but on the same data, *conflict* if
  - Their mutual order is significant, i.e., determines at least one of the following:
    - The final value of that item read by future transactions
    - The value of the item as read by present transactions
  - Consider the flow of information or the nonflow of information

# Conflict

Consider the following example:

- $T1 = r1(z); w1(z); c1$
- $T2 = r2(z); w2(z); c2$
- Two example schedules:
  - $S1 = r1(z); w1(z); c1; r2(z); w2(z); c2$
  - $S2 = r1(z); r2(z); w1(z); w2(z); c1; c2$
- Conflict?

# Serial Schedules

Transactions are wholly before or after others (i.e., one by one)

- Clearly, we must allow for service requests to come in slowly, one-by-one
- Thus, under independence of transactions (assuming each transaction is correct), serial schedules are obviously correct

# Serializable Schedules

- Interleaved schedules are desirable
  - Why?
- Those *equivalent* to some serial schedule. Here equivalent can mean
  - *Conflict* equivalent —all pairs of conflicting ops are ordered the same way
  - *View* equivalent—all users get the same view

# Achieving Serializability

- *Optimistically*: Let each transaction run, but check for serializability before committing
- *Pessimistically*: Use a protocol, e.g., locking, to ensure that only serializable schedules are realized

Generally, the pessimistic approach is more common

# Recoverable Schedules

In which a transaction commits after all transactions it *read from* have committed

- In terms of the ACID properties, what is the risk in allowing a nonrecoverable schedule?

# Avoid Cascading Aborts

In which a transaction does not read from uncommitted transactions

- What is the risk in allowing such reads?

# Strict Schedules

In which an item can't be *read or written* until the previous transaction to write that item has committed (the aborted ones having been factored out)

- Compare with ACA
- This allows us to UNDO by restoring the before image

# Rigorous Schedules

In which an item can't be *read or written* until the previous transaction to read or write that item has committed

- Compare with strict schedules

# Locks

Lock item  $x$  while using item  $x$

- Binary: at most one party may lock  $x$ 
  - `lock(x)`: acquire the lock
    - computation hangs until `lock(x)` returns, i.e., the lock is acquired
  - `unlock(x)`: relinquish the lock
- Yields mutual exclusion, but restrictive

# Multimode Locks

- When one party has an exclusive lock, no other party may have an exclusive or a shared lock
  - Shared-lock(x) needed for read(x)
  - Exclusive-lock(x) needed for write(x); exclusive means no other concurrent lock can exist on x
  - Can be upgraded (read to write) or downgraded (write to read)
- Can we guarantee serializability with locks?
- Considers S2 again

# Two-Phase Locking

Locks on different data items must be related

- 2PL considers two phases in the life of a transaction
  - Growing phase: acquire but not release locks
  - Shrinking phase: release but not acquire locks
- Guarantees serializability, but can deadlock
- *Strict 2PL* releases all locks at once when the transaction commits or rolls back
  - Ensures strict schedules (also ACA by definition)
- Conservative 2PL: acquire locks at start; won't deadlock but may starve a transaction

# Distributing ACID Transactions

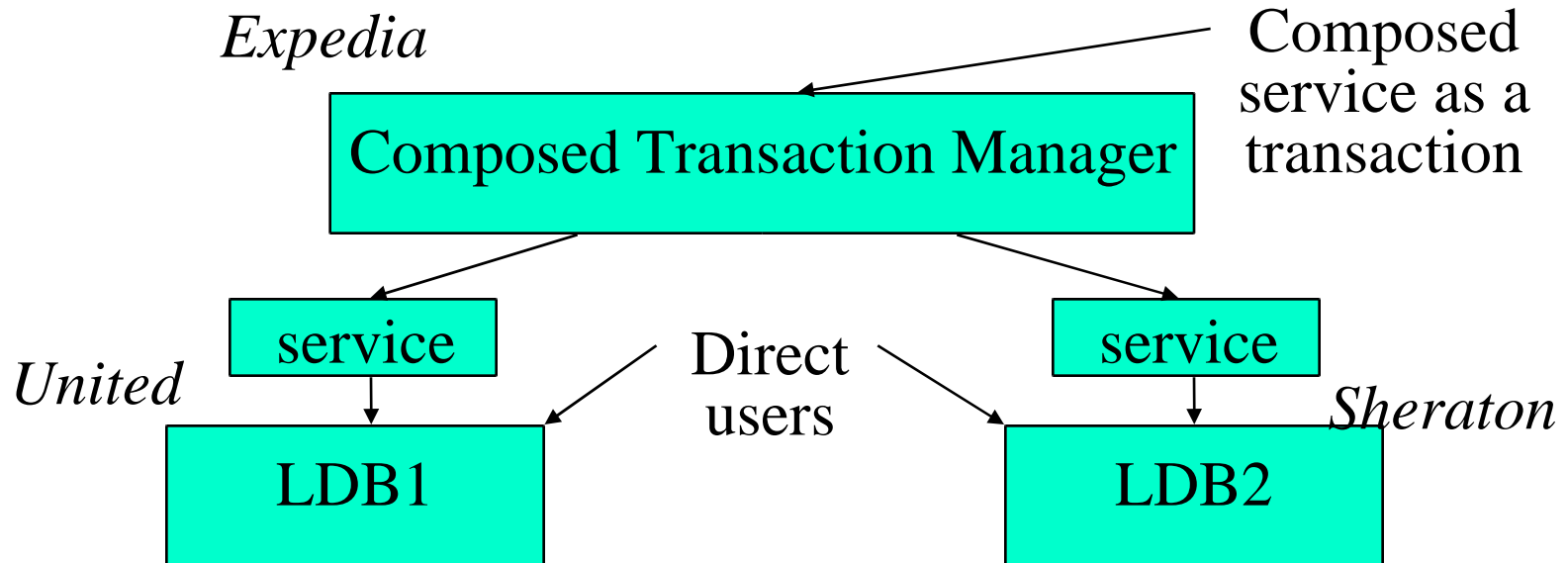
- ACID transactions are applicable for
  - Brief, simple activities (few updates; seconds, at most)
  - On centralized architectures
  - Flat transactions
- Working together requires distributed transactions (nested transaction)
- What would ACID properties mean for distributed transactions?

# Closed-Nested Distributed Transactions

ACID transactions can be implemented in distributed settings

- Consider two or more subtransactions, conceptually organized as a tree
  - Ensure atomicity through two-phase commit (2PC)
  - Ensure isolation so that results are not exposed till the global transaction commits
- Why would this ever be useful for SOC?

# Transactions over Composed Services



- As an instance of autonomy, each LDB
  - Retains full control on execution even if in conflict with CTM
  - Decides what (control) information to release
- Assume each party (CTM, LDB1, LDB2) ensure serializability

# Compositional Serializability

Transactions throughout the system should be serializable

- CTM ensures that the composed transactions are serializable
- This doesn't guarantee compositional serializability, because of *indirect* conflicts:
  - CTM does T1: r1(a); r1(c)
  - CTM does T2: r2(b); r2(d)
  - LDB1 does T3: w3(a); w3(b)
  - LDB2 does T4: w4(c); w4(d)
  - Since T1 and T2 are read-only, they are serializable.
  - LDB1 sees S1=r1(a); c1; w3(a); w3(b); c3; r2(b); c2
  - LDB2 sees S2=w4(c); r2(d); c2; r1(c); c1; w4(d); c4
  - Each LDB has a serializable schedule; yet jointly they put T1 before and after T2
- Notice we would have lots of potential compositions, so the problem is worse

# Strawman 1: Tickets

Compositional serializability fails because of local conflicts that the CTM does not see

- Fix by always causing conflicts--whenever two composed transactions execute at a site, they must conflict there. Indirect conflicts become local conflicts visible to the LDB
  - Make each composed transaction increment a ticket at each site
- Downside:
  - Causes all local subtransactions of a transaction to go through a local hotspot
  - Composed transactions are serialized, but only because many are aborted!

# Strawman 2: Rigorous Scheduling

Hold read and write locks till end (no tickets)

- Check that this prevents the bad example
- The CTM must delay all commits until all actions are completed
  - possible only if allowed by LDB
  - requires an operation-level interface to LDB
- Downside:
  - Causes all sites to be held up until all are ready to commit
  - Essentially like the 2PC approach

# Possible Methodology

- When no cross-service constraints apply, local serializability is enough
- Split data into local and shared partitions
  - LDB controls local data
  - CTM controls shared (local transactions can read, but write only via CTM)
- Downside: doesn't work in all cases
  - All shared data is managed through a special service
  - Only for the most trivial compositions

# Compositional Atomicity

- Can succeed only if the services restrict their communication autonomy through service-level agreements
- Otherwise,
  - If services do not release their prepare-to-commit state
  - They may not participate in a mutual commit protocol

# Compositional Deadlock

Easy to construct scenarios in which a deadlock is achieved.

- Assume LDB1 and LDB2 use 2PL. If a deadlock is formed
  - Solely of upper-level transactions, then the CTM may detect it
  - Of a combination of local and upper transactions, then
    - CTM won't know of it
    - LDBs won't share control information

# Compositional Atomicity & Durability

- What happens when a CT fails? Each service individually ensures atomicity and durability of its local transactions
  - With 2PC, CTM can guarantee that all or none commit
  - Otherwise, traditional techniques do not apply – we need different, application-specific expectations, involving weaker atomicity and durability

# Weaker Atomicity and Durability

May possibly be achieved via:

- *redo*: rerun the writes from log
- *retry*: rerun all of a subtransaction
- *compensate*: semantically undo all ***other*** subtransactions

# Beyond ACID Transactions

- Composed services feature in business processes, which
  - Are complex, i.e., long-running, failure-prone, multisite, with subtle consistency requirements
  - Cooperate with other processes, involving computational and human tasks
  - Respect autonomy and heterogeneity of components

# Extended Transactions

Extended transaction models relax the ACID properties by modifying various features

- Allowing open nesting, wherein partial results are revealed (newer, non-ACID)
- Atomicity, e.g., contingency procedures, to ensure “all”
- Consistency restoration, e.g., via compensation, to ensure “none”
- Constraints among subtransactions, such as
  - Commit dependencies
  - Abort dependencies

Ultimately, a transaction must be atomic (albeit in a relaxed sense)

# Compensation

- Something that semantically undoes the effect of a transaction
- Common in business settings, where you would often have (imperfectly) complementary pairs of activities
  - Deposit and withdraw
  - Reserve and cancel
  - Ship and return
  - Pay and refund