CS514: Intermediate Course in Operating Systems

Professor Ken Birman
Vivek Vishnumurthy: TA

Today

- Transactions in large, complex settings:
  - Nested Transactions
  - "Transactions" in WebServices.
- Then touch on some related issues
  - Need for 2-phase commit
  - Availability limitations of the transactional model.

Large complex systems

- They will often have many components
- Operations may occur over long periods of time
- We'll need to ensure all-or-nothing outcomes but also need to allow high levels of concurrency

Concerns about transactions

- While running a transaction acquires locks
  - Other transactions will block on these locks hence the longer a transaction runs the more it cuts system-wide concurrency
- Some subsystems may not employ transactional interfaces
- Application may be a "script", not a single program

Transactions on distributed objects

- Idea was proposed by Liskov's Argus group
- Each object translates an abstract set of operations into the concrete operations that implement it
- Result is that object invocations may "nest":
  - Library "update" operations, do
  - A series of file read and write operations that do
  - A series of accesses to the disk device

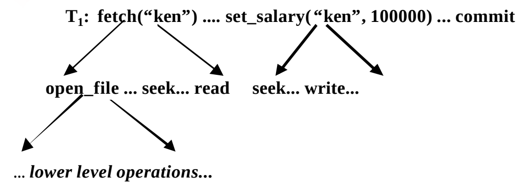
Nested transactions

- Call the traditional style of flat transaction a "top level" transaction
  - Argus short hand: "actions"
- The main program becomes the top level action
- Within it objects run as nested actions

Arguments for nested transactions

- It makes sense to treat each object invocation as a small transaction: begin when the invocation is done, and commit or abort when result is returned
  - Can use abort as a "tool": try something; if it doesn't work just do an abort to back out of it.
  - Turns out we can easily extend transactional model to accommodate nested transactions
- Liskov argues that in this approach we have a simple conceptual framework for distributed computing

Nested transactions: picture



Observations

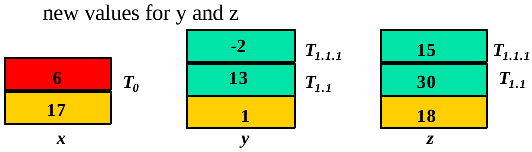
- Can number operations using the obvious notation
  - T_1, $T_{1.2.1}$.....
- Subtransaction commit should make results visible to the parent transaction
- Subtransaction abort should return to state when subtransaction (not parent) was initiated
- Data managers maintain a stack of data versions

Stacking rule

- Abstractly, when subtransaction starts, we push a new copy of each data item on top of the stack for that item
- When subtransaction aborts we pop the stack
- When subtransaction commits we pop two items and push top one back on again
- In practice, can implement this much more efficiently!!!

Data objects viewed as "stacks"

- Transaction T_0 wrote 6 into x
- Transaction T_1 spawned subtransactions that wrote new values for y and z



Locking rules?

- When subtransaction requests lock, it should be able to obtain locks held by its parent
- Subtransaction aborts, locks return to "prior state"
- Subtransaction commits, locks retained by parent
- ... Moss has shown that this extended version of 2-phase locking guarantees serializability of nested transactions

Commit issue?

- Each transaction will have touched some set of data managers
  - Includes those touched by nested sub-actions
  - But not things done by sub-actions that aborted
- Commit transaction by running 2PC against this set
- We'll discuss this in upcoming lectures but

2-Phase commit: Reminder

- Goal is simply to ensure that either
  - All processes do an update, or
  - No process does the update
- For example, at the end of a transaction we want all processes to commit or all to abort
- The "two phase" aspect involves
  1. Asking: "Can you commit transaction t_x?"
  2. Then doing "Commit" or "Abort"

Experience with model?

- Some major object oriented distributed projects have successfully used transactions
- Seems to work only for database style applications (e.g. the separation of data from computation is natural and arises directly in the application)
- Seems to work only for short-running applications (Will revisit this issue shortly!)

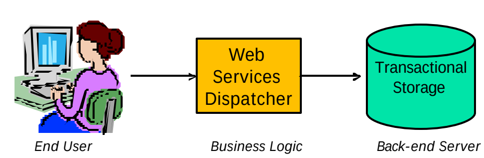
Web Services

- Supports nested transaction model but many vendors might opt for only flat transactions
- Also provides a related model called *business transactions*
  - Again, application accesses multiple objects
  - Again, each access is a transaction
  - But instead of a parent transaction, we use some form of script of actions and compensating actions to take if an action fails

Transactions in Web Services

- Imagine a travel agency that procures air tickets, hotel stays, and rental cars for traveling customers.
- And imagine that the agency wants to automate the whole process.
  - Where all *partners* expose WS interfaces
- This process can be very lengthy.
- And typically spans multiple "sub-processes", each in a different administrative domain.
- What to do when say the agency could find air-tickets and hotel accommodation,but no rental car?

3-Tier Model (reminder)



Transaction Hierarchy in WS

- Basic unit is the *activity* : a computation executed as a set of scoped operations.
- Top-level process is "Business Activity"
  - May run for a long time, so holding locks on resources until commit is not viable.
  - Have to expose results of uncommitted business activities to concurrently executing activities.

Transaction Hierarchy in WS

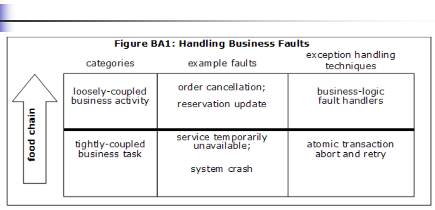
- Small lower-level interactions are called Atomic Transactions
  - Short; executed within limited trust domains.
  - Satisfy ACID properties.
- Imagine a tree structure here (similar to nested txs)

Fault-tolerance

- We know how faults are handled in atomic transactions.
- What about faults in Business Activities?
  - Say Business Transaction B contains atomic transactions A1 and A2, and A1 fails and A2 succeeds – need to “undo” A2 after it had committed
- Issue: since we aren’t using nested transactions, how can we obtain desired all-or-nothing outcome?

Compensating actions

- Idea is to write a form of script
  - If <action succeeds> then <next step>
  - Else <compensate>
- The compensation might undo some actions much as an abort would, but without the overheads of a full nested transaction model
- (Model has also been called “sagas”)



The WS-Coordination Spec.

- A standard that describes how different Web Services work together reliably.
- The coordination framework contains the Activation, Registration and Coordination Services…

Some Terminology

- The Coordination type identifies what kind the activity is (Atomic Transaction/ Business Activity)
- Each message sent by a participant contains a CoordinationContext for message to be understood:
  - Has an activity identifier (unique for each activity)
  - A pointer to the registration service used by the participant.
  - The coordination type.

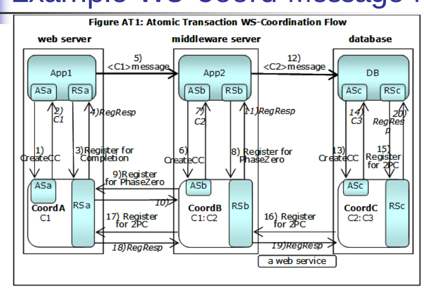
The Coordinator

- Activation Service: used to create activities
  - Participants specify the coordination type
  - Activation Service returns the CoordinationContext that's used in later stages.
- Registration Service: used by participants to register with (respective) coordinator for a given coordination protocol.
- Coordination Protocol Services: A set of these for each supported coordination type.

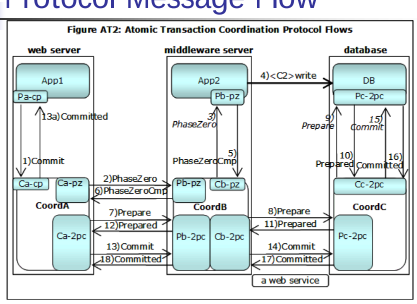
WS-Transaction

- Specifies protocols for each coordination type.
- Atomic Transactions
  - Completion, PhaseZero, 2PC, etc.
- Business Transactions
  - BusinessAgreement, BusinessAgreementWithComplete

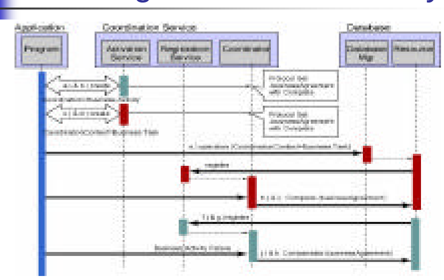
Example WS-Coord Message Flow



Protocol Message Flow



Handling a Business Activity



Transactions in WS – Resources

- http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnglobspec/html/ws-coordination.asp
- http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnglobspec/html/ws-transaction.asp
- http://www-128.ibm.com/developerworks/library/ws-wstx1/
- http://www-128.ibm.com/developerworks/library/ws-wstx2/

Recap

- We've considered two mechanisms for applying transactions in complex systems with many objects
  - Nested transactions, but these can hold locks for a long time
  - Business transactions, which are a bit more like a command script
- In remainder of today's talk look at transactions on replicated data

Reliability and transactions

- Transactions are well matched to database model and recoverability goals
- Transactions don't work well for non-database applications (general purpose O/S applications) or availability goals (systems that must keep running if applications fail)
- When building high availability systems, encounter replication issue

Types of reliability

- Recoverability
  - Server can restart without intervention in a sensible state
  - Transactions do give us this
- High availability
  - System remains operational during failure
  - Challenge is to replicate critical data needed for continued operation

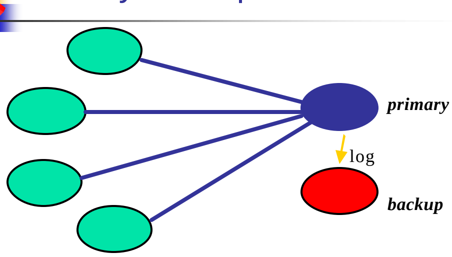
Replicating a transactional server

- Two broad approaches
  - Treat replication as a special situation
    - Leads to a primary server approach with a "warm standby"
    - Most common in commercial products
  - Just use distributed transactions to update multiple copies of each replicated data item
    - Very much like doing a nested transaction but now the components are the replicas
    - We'll discuss this kind of replication in upcoming lectures

Server replication

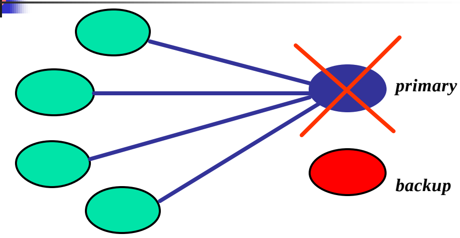
- Suppose the primary sends the log to the backup server
- It replays the log and applies committed transactions to its replicated state
- If primary crashes, the backup soon catches up and can take over

Primary/backup



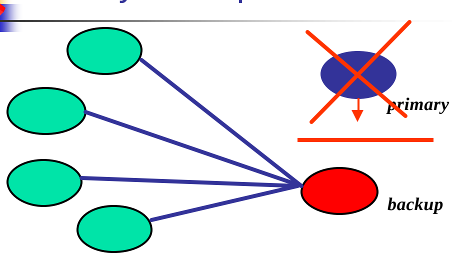
Clients initially connected to primary, which keeps backup up to date. Backup tracks log

Primary/backup



Primary crashes. Backup sees the channel break, applies committed updates. But it may have missed the last few updates!

Primary/backup

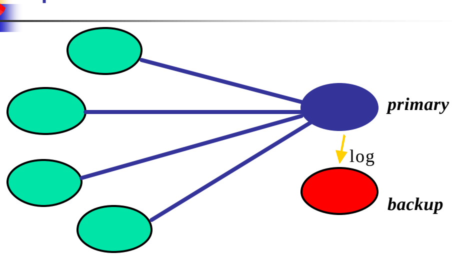


Clients detect the failure and reconnect to backup. But some clients may have "gone away". Backup state could be slightly stale. New transactions might suffer from this

Issues?

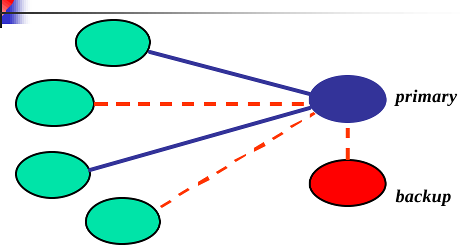
- Under what conditions should backup take over
  - Revisits the consistency problem seen earlier with clients and servers
  - Could end up with a "split brain"
- Also notice that still needs 2PC to ensure that primary and backup stay in same states!
  - Either want both to reflect a committed transaction, or (if the transaction aborted), neither to reflect it

Split brain: reminder



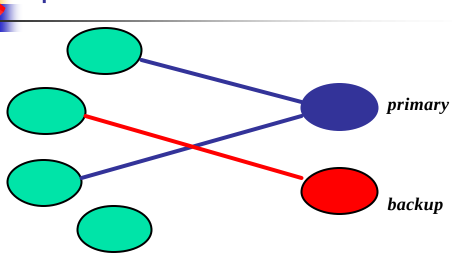
Clients initially connected to primary, which keeps backup up to date. Backup follows log

Split brain: reminder



Transient problem causes some links to break but not all. Backup thinks it is now primary, primary thinks backup is down

Split brain: reminder



Some clients still connected to primary, but one has switched to backup and one is completely disconnected from both

Implication?

- A strict interpretation of ACID leads to conclusions that
  - There are no ACID replication schemes that provide high availability
  - We'll see more on this issue soon...
- Most real systems evade the limitation by weakening ACID

Real systems

- They use primary-backup with logging
- But they simply omit the 2PC
  - Server might take over in the wrong state (may lag state of primary)
  - Can use hardware to reduce or eliminate split brain problem

How does hardware help?

- Idea is that primary and backup share a disk
- Hardware is configured so only one can write the disk
- If server takes over it grabs the "token"
- Token loss causes primary to shut down (if it hasn't actually crashed)

Reconciliation

- This is the problem of fixing the transactions impacted by loss of tail of log in a failure
- Usually just a handful of transactions
  - They committed but backup doesn't know because it never saw a commit record
  - Someday, primary recovers and discovers the problem
    - Need to apply the missing ones
    - Also causes cascaded rollback
    - Worst case may require human intervention
- Similar to compensation in Web Services

Summary?

- We looked at a variety of situations in which transactions touch multiple objects
  - ...because of nesting
  - ... because of complex business applications
  - ... because of primary/backup replication
- We left one major stone unturned:
  - Replicated data in the sense of process groups, often with goal of higher availability
  - We'll explore this in the next few lectures