CS514: Intermediate Course in Computer Systems

Lecture 38: April 23, 2003

Nested transactions and other weird implementation issues…

Transactions Continued

- We saw
  - Transactions on a single server
  - 2 phase locking
  - 2 phase commit protocol
- Set commit to the side (much more we can do with it)
- Today stay focused on the model
  - So-called “nested” transactions
  - Issues of availability and fault-tolerance

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, does
  - A series of file read and write operations, which do
  - A series of accesses to the disk device

Argus Language

- Structured, like Java, around objects
- But objects can declare “persistent” state
  - Stored on a disk
  - Each time this object is touched, state must be retrieved and perhaps updated
- Language features include begin, commit, abort
- “Can’t commit” an example of an exception Argus can throw

Argus premises

- Idea was that distributed computing needs distributed programming tools
- Languages are powerful tools
- Reliability inevitably an issue
  - Can we successfully treat the whole system as a set of O-O applications?
  - Can one imagine a complex networked system built using a single language?
- Note that in Java we don't get similar transactional features
  - Do these *need* to be in the language?

What about WS_TRANSACTION?

- As written, the specification doesn’t seem to allow nesting
- But if web services become common it would be very hard to avoid!
  - E.g. suppose a web service system does a DNS update.
  - Is the DNS update “part” of the transaction?

Transactional “creep”

- Once you open the door to transactions it is hard to stop half way
  - Some researchers have done just this… they implement part of the transactional model but not all
  - But more often you need to implement the whole model everywhere and then “relax” it.
  - Argus does this

Nested transactions

- In a world of objects, each object operates by
  - operations on other objects
  - primitive operations on data
- If objects are transactional, how can we handle “nested” transactions?
- Argus extends the transactional model to address this case

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

CS514

- 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

CS514

T_1: **fetch("ken") .... set_salary("ken", 100000) ... commit**

open_file ... seek... read **seek... write...**

… ***lower level operations...***

Observations

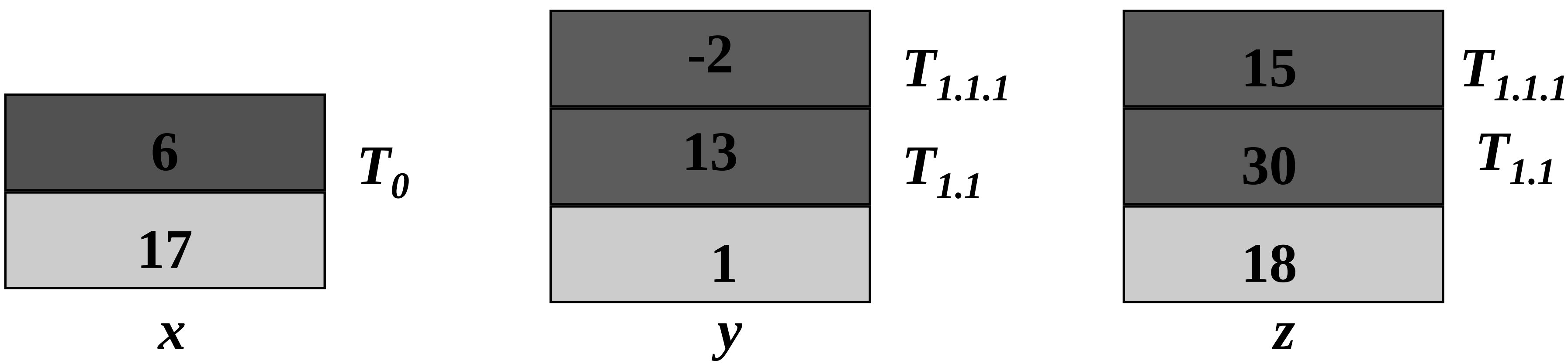
- Can number operations using the obvious notation
  - T_1, $T_{1.2.1}$.....
- Subtransaction commit should make results visible to the parent transaction
- Subtransaction abort should return to state when subtransaction (not parent) was initiated
- Data managers maintain a stack of data versions

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

Commit issue

- Each subtransaction will need to run its own 2PC protocol (at first glance)
  - After all, each of them has accessed one or more data managers
  - And each “acts” like a transaction!
- This obviously can be very expensive

Extending two-phase commit

- Too costly to run a commit for each nested level…
- … so Liskov suggests an “optimistic” scheme:
  - Only runs final top-level commit as a full-fledged protocol
  - If conflict arises, then “climb the tree” to a common ancestor node with regard to conflicting lock and ask if the subtransaction that holds the lock committed or aborted

Other major issues

- In object oriented systems, the transactional model can be unnatural:
  - No way for objects that run concurrently to cooperate with each other (model suggests that all communication must be through the database! "I write it, you read it")
  - Externally visible actions, like "start the motor", are hard to represent in terms of data or objects in a database
- Long-running transactions will tend to block out short-running ones, yet if long-running transaction is split into short ones, we lose ACID properties

Argus responses?

- Added
  - A top-action construct
    - Application can step outside of this transaction
  - Supported additional ways of locking objects
    - User can design sophisticated locking schemes
  - Features for launching concurrent activities in separate threads
    - They run as separate actions

Experience with model?

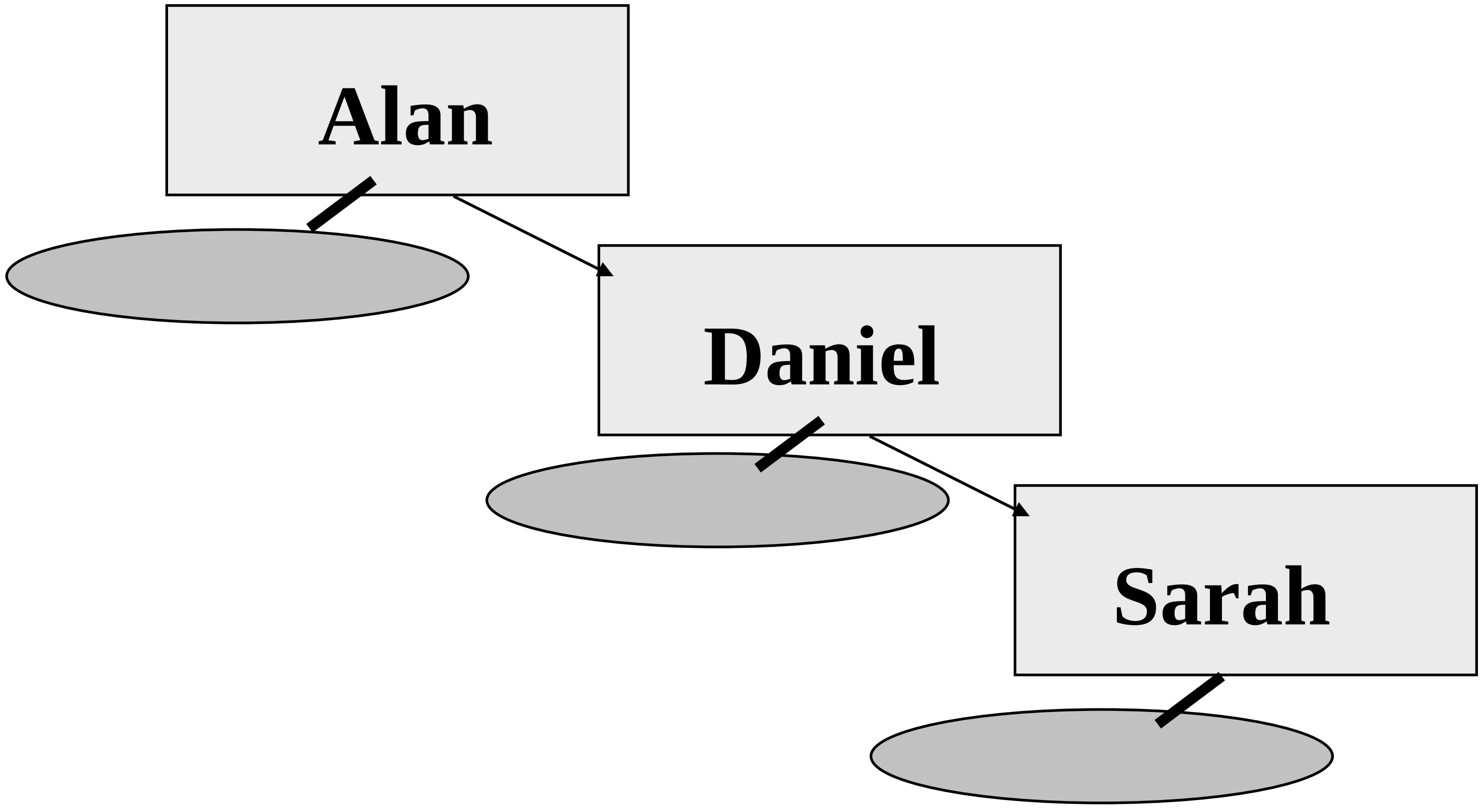
- Some major object oriented distributed projects have successfully used transactions
- Seems to work only for database style applications (e.g. the separation of data from computation is natural and arises directly in the application)
- Seems to work only for short-running applications

Example of a poor fit

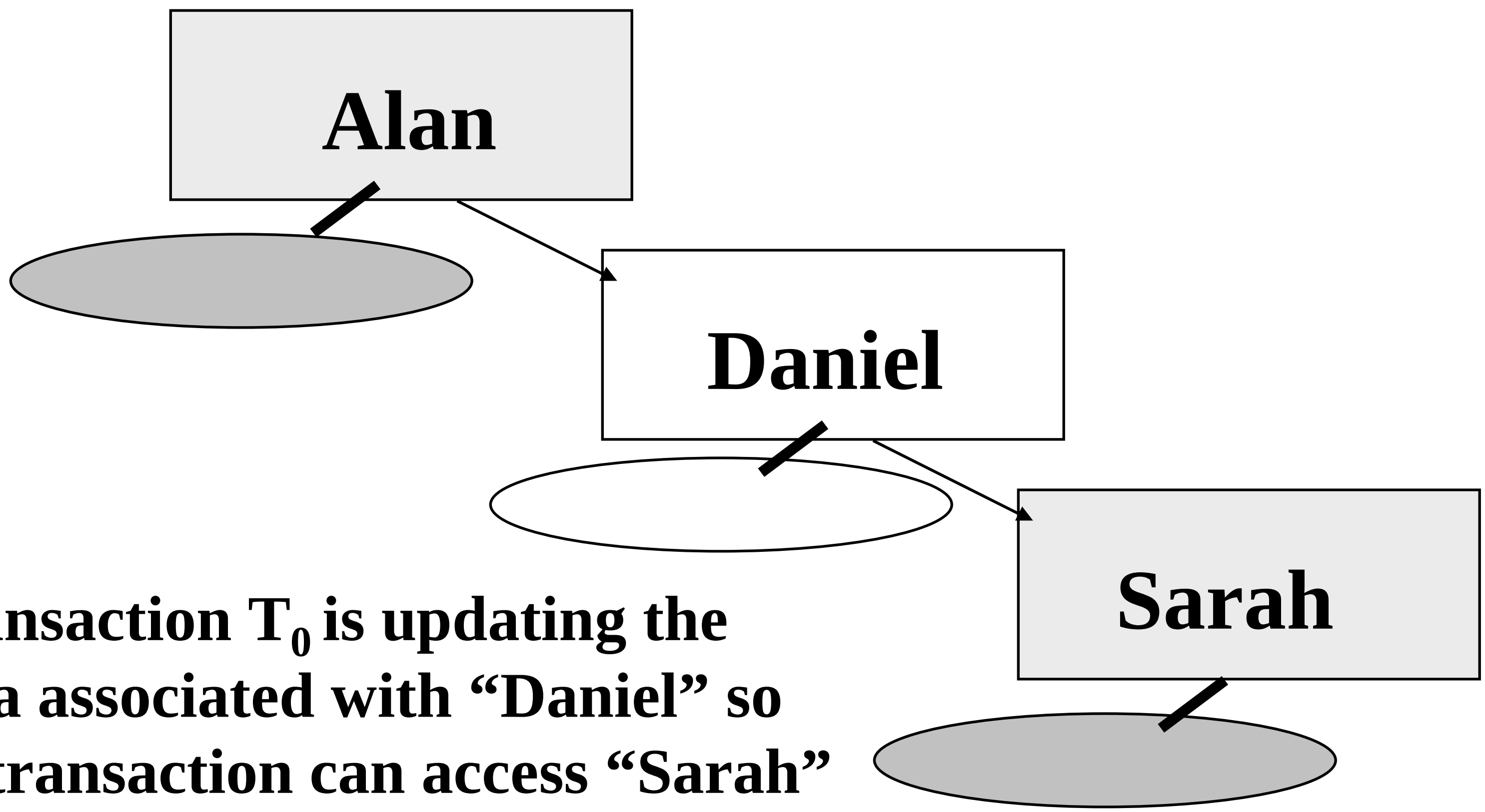
- Transactions don't work well for a "directory" that has a linked list of indexed records
- Problem is that long-running transaction may leave records it walks past locked
- Technically, this is correct, but in practice it kills performance
- Many suggestions but they require sophistication. Even the simplest structures demand this!

Directory represented as linked list

CS514



Directory represented as linked list



Transaction T_0 is updating the data associated with “Daniel” so no transaction can access “Sarah”

Examples of Proposed Work-Arounds

- Bill Weihl suggests that the list here was “over ordered”: for our purposes, perhaps we should view it as an unordered set. Then we can hunt for a record, e.g. “Sarah”, even if some other record is locked
- Here we use the Argus top-level transaction mechanisms, but these are tricky to work with

Problems with top-level actions

- The top-level action was created from a transaction
  - It may have updated objects first
  - But it hasn't committed yet
- Which versions of those objects should the top-level action see?
  - Argus: it sees the updated versions
  - If the transaction that launched it aborts, the top-level action becomes an "orphan"
    - In some sense it finds an inconsistent world!

Problems with top-level actions

- Top-level actions can also leak information about the non-committed transaction to the outside world
- And it may be desired that we coordinate the commit of the resulting set of actions, but Argus has no way to do this

Difficulty with Work-Arounds?

- Even very simple data structures must be designed by experts. Poorly designed structures will not support adequate concurrency.
- Implication is that data structures live extracted in libraries, and have complex implementations.
- But real programmers often must customize data structures and would have problems with this constraint

What About nested / distributed cases

- Both are very costly to support: factor of 10 to 100 performance loss relative to non-distributed non-nested transactional model
- Many developers now seek support for these features when they purchase products
- Few actually use these features except in very rare situations!

Experience in real world?

CS514

- Argus and Clouds proposed transactions on general purpose Corba-style objects
- Neither was picked up by industry; costs too high
- Encina proposed transactional environment for UNIX file I/O
- This became a very important OLTP product, but was adopted mostly in database-style applications

Whither WS_TRANSACTION?

- Right now, focus is on a single transactional application talking to a single data manager
- Hope is that "business transaction" concept can cover remaining needs

Business transactions

- Think of an insurance document
  - I submit my claim
  - Bill Smith checks it over and routes it to fact verification
  - Claire Williams pulls my credit history and that of the other person in the accident. She routes it to adjustments
  - Jim Browne estimates damage…

Business transactions

- We can’t do this as a single transaction
  - The paperwork might need a month to wind its way through the system
  - Along the way it visits many subsystems…
- Leads to the view that perhaps we should represent a chain of transactions in an explicit way

Business transactions

- Model as
  - A chain of desired actions (transactional programs that run on servers)
  - And a chain of “compensating actions” to undo effect (optional)
- Allows us to push the transaction forward or, perhaps, backwards

WS_TRANSACTION

- The hope is that with business transactions, can cope with the kinds of issues Argus solves with nested transactions and top-level actions
- But the jury is far from in!

File systems

- An example of a “server” in which transactions were applied…
  - … but with mixed success!
- Goals
  - Improve robustness after a crash
  - Provide better isolation for concurrently active applications

File systems that use transactions?

- QuickSilver (Schmuck) exploits atomic commit for recoverability, but not serializability aspects
- Hagmann reported positive experiences with a group-commit scheme in Xerox file system
- Encina extends file system to offer efficient support for transactions on file-structured database objects

Summary of experiences?

- File systems and transactions are only a “so-so” match
- It can be done…
  - But the model becomes a peculiar, partial one
  - Very tricky to learn to use
  - Neither fish nor fowl

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
- Web Services will crash into this problem soon

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
  - Just use distributed transactions to update multiple copies of each replicated data item
    - We already know how to do this, with 2PC
    - Each server has “equal status”
  - Somehow treat replication as a special situation
    - Leads to a primary server approach with a “warm standby”

Replication with 2PC

- Our goal will be "1-copy serializability"
  - Defined to mean that the multi-copy system behaves indistinguishably from a single-copy system
  - Considerable form and theoretical work has been done on this
- As a practical matter
  - Replicate each data item
  - Transaction manager
    - Reads any single copy
    - Updates all copies

Observation

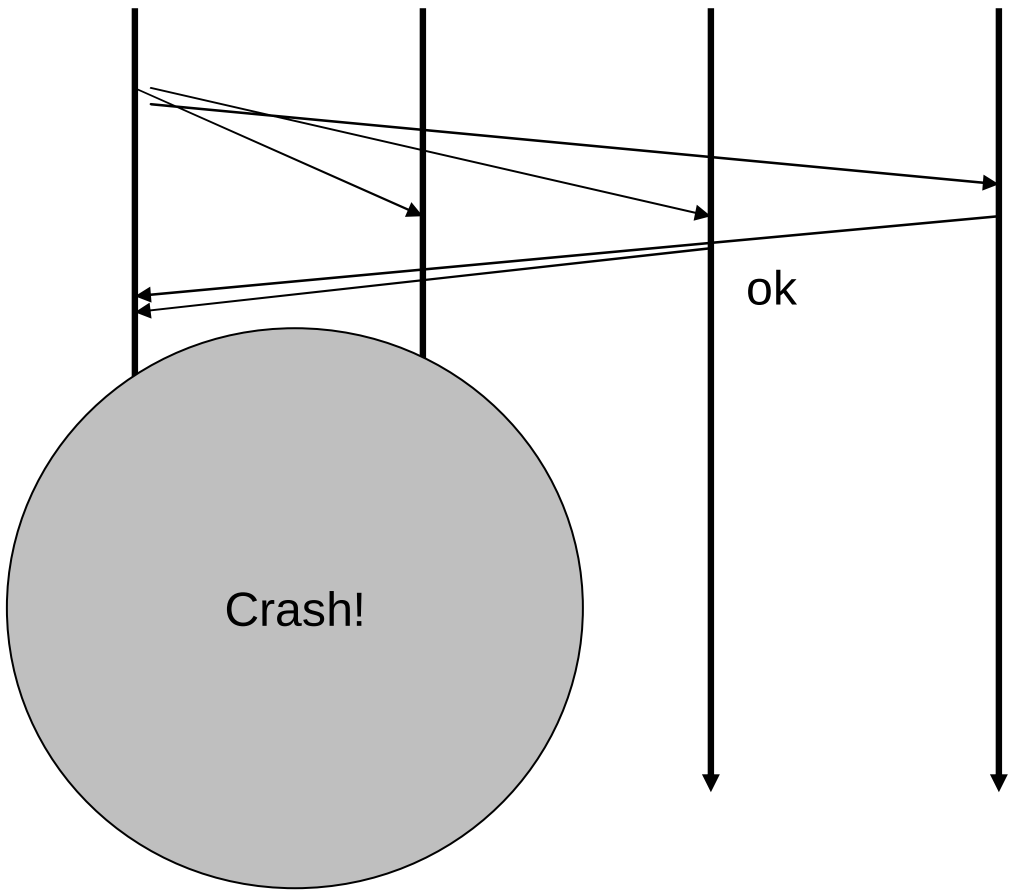
- Notice that transaction manager must know where the copies reside
- In fact there are two models
  - Static replication set: basically, the set is fixed, although some members may be down
  - Dynamic: the set changes while the system runs, but only has operational members listed within it
- Today stick to the static case

Replication and Availability

- A series of potential issues
  - How can we update an object during periods when one of its replicas may be inaccessible?
  - How can 2PC protocol be made fault-tolerant?

2PC and a crash

CS514



Ok to commit?

… ok

Now what?

Two failures leave system hung!

- If coordinator and one participant fail
  - And the coordinator could, itself, be a participant
- We can’t figure out the outcome!
- Some work was done on “3PC” to work around this
  - But it turned out that FLP result applies
  - Basically, the problem can’t be solved

Usual responses?

- Quorum methods:
  - Each replicated object has an update and a read quorum
  - Designed so Q_u+Q_r > # replicas and $Q_{u+}Q_u$ > # replicas
  - Idea is that any read or update will overlap with the last update

Quorum example

- X is replicated at {a,b,c,d,e}
- Possible values?
  - $Q_u = 1$, $Q_r = 5$ (violates $Q_U+Q_u > 5$)
  - $Q_u = 2$, $Q_r = 4$ (same issue)
  - $Q_u = 3$, $Q_r = 3$
  - $Q_u = 4$, $Q_r = 2$
  - $Q_u = 5$, $Q_r = 1$ (violates availability)
- Probably prefer $Q_u=4$, $Q_r=2$

Things to notice

- Even reading a data item requires that multiple copies be accessed!
- This could be *much* slower than normal local access performance
- Also, notice that we won't know if we succeeded in reaching the update quorum until we get responses
  - Implies that any quorum replication scheme *still* needs a 2PC protocol to commit
  - In effect: high availability is just not possible!

Summary

- Transactions are an easy answer to the reliability problem, but a slippery slope
  - With one server, one transaction manager, short transaction, the technique works well
  - With multiple servers, 2PC is slow
  - We can't guarantee availability
  - Nested transactions create huge complexity
  - Concurrency control is trickier than expected
- All in all, transactions are a very risky prospect. Web services will soon rediscover this!