Distributed Software Development
Distributed Transactions

Chris Brooks

Department of Computer Science
University of San Francisco

Transactions

- Features of transactions
- Serial equivalence
- Locking and deadlock
- Distributed transactions
- Two-phase commit
- Distributed deadlock

Transacti

- A transaction is a sequence of operations between a client and a server.
- Goal: make sure that:
  - Objects remain in a consistent state
  - System is tolerant to crash failures
  - Transaction effects are independent of other transactions
  - Transactions are either completed or not started.

Example

- As an example, we'll look at an interface to a banking system.
- We'd like to be able to do the following operations on accounts:
  - deposit(amt)
  - withdraw(amt)
  - getBalance()
  - setBalance(amt)
- We'd also like the following operations to be available for branches:
  - CreateAccount(name)
  - lookUpAccount(name)
  - totalAccounts()

Example transaction

- A transaction may involve several operations, each of which changes the state of a different object:
  - Transaction T:
    1. alexAcct.withdraw(100)
    2. nancyAcct.deposit(100)
    3. nancyAcct.widthdraw(200)
    4. brooksAcct.deposit(200)
  - We can't stop in the middle, lose any of the operations, or do them in the wrong order.

Committing and Abor

- A transaction may be either committed or aborted.
- When all operations are complete and the transaction is ready to be accepted, it is *committed.*
  - Written to permanent storage
  - After this point, it cannot be undone
- If the server decided that a transaction cannot be processed (undone by the client, or it will leave the system in an inconsistent state), it is *aborted.*
  - All operations are undone

ACID

- The desirable features of a transactional DBMS are sometimes referred to as ACID
  - Atomicity
  - Consistency
  - Isolation
  - Durability

Atomicity

- Atomicity is sometimes referred to as “all-or-nothing”.
- Either a transaction completes successfully, and all effects are applied to all objects, or it has no effect at all.
- Either all withdraws and deposits are made, or none of them are.

Consiste

- A transaction must move the system from consistent state to consistent state.
- For example, the sum of all the accountBalances must always be equal to the branch’s totalAccounts
- Depending on the application, at database may have other constraints
  - No negative balances on an account
  - No post-dated transactions
- If the system is in an inconsistent state after a transaction, the transaction must be undone, so as to restore consistency.

Isolation

- The intermediate effects of a transaction must not be visible to other transactions.
  - In our example, Nancy’s bank account balance briefly went up by $100. (the money was then transferred to Brooks’ account)
  - No other process or transaction should see that balance.
- In other words, to the outside world, a transaction must appear as a single operation.

Isolation

- How to provide isolation?
- Perform all transactions in a single thread
  - Works, but doesn’t scale.
- Use locks to control concurrent access.
  - Better, although now we need to detect (and undo) deadlocks.

Durab

- After a transaction has been processed, its effects are saved to permanent storage.
- The transaction will never be undone or lost, even in the presence of a server crash.
- This typically also requires some guarantees about the nature of the permanent storage.

Common problems in transaction processing

- If we assume that a server will process multiple transaction simultaneously (in separate threads), problems can occur.
  - Lost updates
  - Inconsistent retrievals

Lost updates

- 'Lost updates' refer to the problem of one transaction overwriting the result of another transaction.
- For example:
  - Consider transactions T and U. T wants to transfer 10% of the balance in account B from A to B. U wants to transfer 10% of the balance in account B from C to B. B starts at $200.
  - Transaction T:
    - balance1 = B.getBalance()
    - B.setBalance(balance1 * 1.1)
    - A.withdraw(balance1 / 10)
  - Transaction U:
    - balance2 = B.getBalance()
    - B.setBalance(balance2 * 1.1)
    - C.withdraw(balance1 / 10)

Lost upd

- At the end, the balance in account B should be $242.
- But what if the operations happen in this order:
  - (T) = balance1 = B.getBalance() ($200)
  - (U) = balance2 = B.getBalance() ($200)
  - (U) = b.setBalance(balance2 * 1.1)($220)
  - (T) = b.setBalance(balance1 * 1.1)($220)
  - (T) = A.withdraw(balance1 / 10)
  - (U) = C.withdraw(balance2 / 10)
- We lose one of the updates, due to the fact that the second setBalance is working with stale data.

Inconsistent retrievals

- Consider transaction T: transfer $100 from account A to B. Transaction U: get branchTotal():
- Transaction T:
  - A.withdraw(100)
  - B.deposit(100)
- Transaction U:
  - getBranchTotal()

Inconsistent retrievals

- If the operations are performed in this order, we get an inconsistent retrieval:
  - (T) A.withdraw(100)
  - (U) getBranchTotal()
  - (T) B.deposit(100)
- The bank's total will appear to be $100 less than it should.

Serial equivale

- We would like to have an interleaving of operations that produces the same effect as if the transactions had been performed one at a time.
- This is called *serial equivalence*
- For example:
  - (T) = balance1 = B.getBalance() ($200)
  - (T) = b.setBalance(balance1 * 1.1)($220)
  - (U) = balance2 = B.getBalance() ($200)
  - (U) = b.setBalance(balance2 * 1.1)($220)
  - (T) = A.withdraw(balance1 / 10)
  - (U) = C.withdraw(balance2 / 10)
- This ordering is serially equivalent to doing each transaction separately.

Conflicting operations

- The trick to achieving serial equivalence is to identify operations that conflict with each other.
  - Read/write and write/write (read/read is OK)
  - For two transactions to be serially equivalent, all pairs of conflicting operations must be executed in the same order on all objects they both access.

Conflicting operations

- For example:
  - T: x=read(i); write(i,10); write(j,20)
  - U: read(j); write(j,30); z=read(i)
- To have serial equivalence, one of the following conditions must hold:
  - T accesses i before U does and T accesses j before U does
  - U accesses I before T does and U accesses J before T does

Loc

- The most common way to achieve serial equivalence is through the use of locks.
- Before a client transaction uses an object, it requests an associated lock.
- The object cannot be used until the lock is acquired.
- To ensure serial equivalence, a process may not acquire new locks within a transaction once a lock has been released.
  - This is called two-phase locking.
  - There is a “lock-growing phase” and a “lock-shrinking” phase

Locking example

- T: begin transaction
- (T) = balance1 = B.getBalance() ($200) T locks B
- (T) = b.setBalance(balance1 * 1.1)($220)
- U begins transaction
- (T) = A.withdraw(balance1 / 10)T locks A
- (U) = balance2 = B.getBalance() ($200) B is locked - U must wait.
- T ends transaction: A and B unlocked.
- (U) = b.setBalance(balance2 * 1.1)($220) U locks B
- (U) = C.withdraw(balance2 / 10) U locks C
- U ends transaction: B and C unlocked.

Locking

- To prevent “dirty reads” and “premature writes”, all locks must be held until the transaction is committed.
- This is called strict two-phase locking.
- “dirty read” - one transaction sees a value that is part of another transaction that is later aborted.
  - T: b1 = a.getBalance() ($100)
  - T: a.setBalance(b1 + 10) ($110)
  - U: b2 = a.getBalance() ($110)
  - U: a.setBalance(b2 + 20) ($130)
  - U committed.
  - T aborted - U cannot be undone.

Loc

- To prevent “dirty reads” and “premature writes”, all locks must be held until the transaction is committed.
- This is called strict two-phase locking.
- “premature writes” - an aborted operation is reset to the wrong value.
- Some systems will store the ’before image’ with a write and roll back to that in the event of an abort
  - A: initial balance: $100
  - T: a.setBalance(105) (’before image’: 100)
  - U: a.setBalance(110) (’before image’: 105)
  - U commits.
  - T aborts: A reset to 100. (should remain 110)
- What if T aborts, then U aborts (balance will be 105, should be 100)

Locks

- There is a tradeoff between the number of locks a system has and the degree of concurrency allowed
  - This is called the *granularity* of the system
- Goal: allow many objects the ability to read an object, but only one the ability to write the object.
  - Provide read locks and write locks.

Deadlocks

- Whenever locks are used, a system can reach *deadlock*.
  - Two or more processes are each waiting for a lock held by the other process.
  - Neither can proceed until the other gives up its lock.
  - Neither can give up its lock until it proceeds.

Deadlock exan

- T wants to transfer $100 from A to B. U wants to transfer $50 from B to A.
  - T: A.deposit(100) - write lock on A.
  - U: B.deposit(50) - write lock on B.
  - T: B.withdraw(100) - wait for write lock for B
  - U: A.withdraw(50) - wait for write lock on A.
- Since neither process will release its locks without committing or aborting, we are in deadlock.

Preventing deadlock

- One simple (but not efficient) method to prevent deadlock is to require a transaction to (atomically) acquire all locks at the beginning.
  - Restricts access to shared resources
  - May not be possible to predict at the beginning of a transaction, as in interactive applications.
- Can also specify a static order.

Detecting deadlock

- A more common approach is to have the process responsible for administering locks detect deadlock and force transactions to abort.
- We can represent processes as nodes and lock dependencies as edges in a *wait-for graph*
- When a lock is requested and must be waited on, an edge is added.
- When a lock is freed, the edge is removed.
- If the graph has a cycle, then we have deadlock.
- The transaction at one of the nodes in the cycle can then be aborted.
  - Choosing which transaction to abort can be tricky

Distributed Transacti

- So far, we've focused on how to provide concurrent transactions within a single server.
- *Distributed transactions* involve multiple servers that must maintain a consistent state.
- If a transaction is committed at one server, it must commit at all servers.
- If a transaction is aborted at one server, it must abort at all servers.

Nested transactions

- In a centralized server, we can allow *nested transactions*
  - These are transactions that are composed of other transactions.
  - Equivalent to modules or subroutines
- Transaction T can be decomposed into T1 and T2
- T1 is further composed into T11 and T12
  - These are processed and committed or aborted

Nested transactions

- A transaction can choose whether to commit or abort based on what happens to subtransactions.
  - There may be an alternative way to accomplish the subtransaction.
- Nested transactions can potentially allow for greater parallelism
- For example, getBranchTotal can nest into a separate getBalance for each account.

Committing in nested transacti

- Committing in a nested transaction follows these rules
  - All of its subtransactions have committed or aborted.
  - A subtransaction can commit provisionally or abort. (Aborting is final)
  - When a parent aborts, all children must abort.
  - When a child aborts, the parent can decide whether to continue or not.
  - When the top-level transaction commits, all children who have provisionally committed may commit, provided none of their ancestors have aborted.

Flat and Nested Distributed transactions

- In a flat transaction, the client invokes objects stored on multiple servers.
- Requests are processed sequentially, as in a centralized system
  - The client finishes with server A before starting to interact with server B.
- In a nested transaction, the top-level transaction spawns subtransactions on different servers.
- These subtransactions can run concurrently

Coordinators

- Distributed transactions are managed via a coordinator.
  - This is a process within each server
  - Each server can act as a coordinator for separate transactions
- Client sends an openTransaction message to a server, and receives a unique transactionID
- That server is now the coordinator for that transaction.
- Servers that manage objects needed by the transaction are called *participants*
- Coordinator tracks all participants.
- It is responsible for managing the commit protocol.

Two-phase con

- Recall that, in order to achieve atomicity, all servers must commit or all must abort.
- How can multiple servers know whether or not they should commit?
- The two-phase commit protocol solves this problem, even when servers crash or messages are lost.

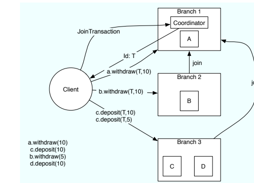
Two-phase commit

- Phase 1: The coordinator asks each participant whether it is able to commit.
  - A participant may not change its vote to abort once it has voted to commit.
  - This means that a participant can't vote until it knows that it can complete its portion of the transaction.

Two-phase commit

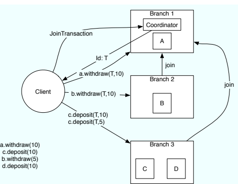
- Phase 2: Once all participants have voted, if any participant voted to abort, the doAbort message is sent to all participants.
- If all participants voted to commit, the doCommit message is sent to all participants.
- Clients each inform the coordinator that they have committed.
- Once all participants have replied, the client is informed that the transaction has been processed.

Two-phase con



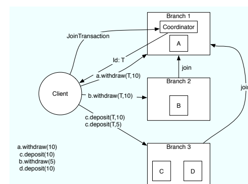
- Client wants to perform a transaction involving four accounts at three branches.
- Sends openTransaction to coordinator at A and receives a transaction ID.
- Includes transaction ID with each request to other branches.
- As each branch is contacted, they send a join message to A.

Two-phase commit



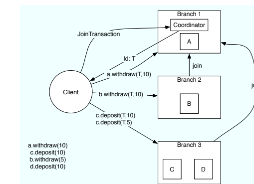
- If the client wants to abort, it sends a doAbort message to the coordinator.
- This is forwarded to all group members, who immediately abort.

Two-phase commit



- If the client wants to commit, it sends a doCommit message to the coordinator.
- The coordinator sends a canCommit message to all participants.
- When they are ready to commit, they send back yes.
- If they must abort, they send back no.

Two-phase con



- Once all votes have been collected, if there are any aborts, the coordinator sends doAbort to all participants.
- Otherwise, a doCommit is sent to all participants.
- Once a participant commits, it returns haveCommitted to the coordinator.
- When all haveCommit messages are received, the client is updated and the transaction closed.

Nested two-phase commit

- We can extend two-phase commit to work with nested transactions.
- When a subtransaction is started, a coordinator for that subtransaction is selected.
- If one of the participants in that subtransaction aborts, the coordinator can choose whether to abort, based on the specifics of the transaction.
  - Example: withdraw fees from all accounts. If one account becomes overdrawn, that subtransaction is aborted, but others may proceed.
- If the parent is aborted, the subtransactions must abort.

Failure in Two-phase commit

- Message loss is dealt with via timeouts.
- Server failure is dealt with by writing progress out to permanent storage.
- As operations are processed, they are logged.
  - A replica is then able to start up and recreate progress.
- Couloris has a great deal of detail on the way logs can be managed to do recovery.

Locking and Dead

- In a distributed transactional system, a lock manager that is local to the resource manages access to that resource.
- Remote processes request access to the locks as usual.
- Since there are many independent lock managers, deadlock can arise.
  - Transaction T locks variable X on server A.
  - Transaction U locks variable Y on server B.
  - Now each transaction wants to modify the other variable.

Distributed Deadlock

- In a centralized system, we would detect deadlock by finding a cycle in the wait-for graph.
- This is the simplest approach in a distributed system
  - All lock managers send their information to a central authority.
  - All the usual problems with this approach apply.

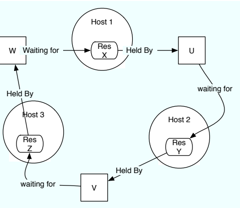
Edge chasing

- When a server that is acting as a lock manager receives a request for a locked resource, it adds an edge to its local wait-for graph.
  - For example, say server X controls access to database A.
  - Initially, transaction T wants to execute A.withdraw(10)
  - X grants T the lock to A.
  - Then transaction U wants to execute A.deposit(20)
  - A is locked, so U $\rightarrow$ T is added to X's wait-for graph.

Edge cha

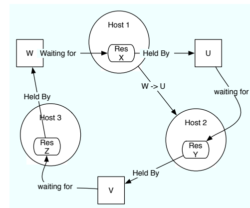
- When a transaction begins waiting for a lock, the coordinator is informed.
- When a process receives a request for a lock, and another transaction is holding that lock, it checks with the coordinator to see if the other process is waiting.
- if it is, it sends a probe to the server administering that lock.
  - Includes the edge in the wait-for graph
- When the probe is received, if there are waiting processes, those edges are added and the probe is forwarded.
- As edges are added, cycles are checked for.

Edge Chasing Example



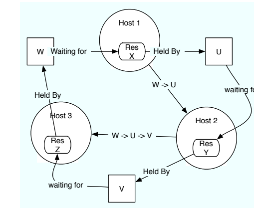
- Initially, transaction U holds Resource X and is waiting for Resource Y
- Transaction V holds Resource Y and is waiting for Resource Z.
- Transaction W holds Resource Z.
- Let's say W requests Resource X.

Edge Chasing Example



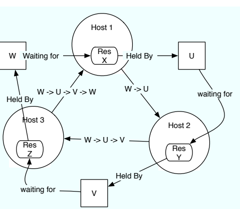
- Server host1 checks to see if any other transactions are waiting for its resource.
- W is waiting for resource X.
- U holds Resource X and is waiting on Resource Y.
- Host 1 sends the message W → U to Host 2

Edge Chasing Exam



- Host 2 knows that U is waiting for Resource Y
- It also knows that Resource Y is held by transaction V
- It checks and finds that V is waiting for the lock to Resource Z.
- It sends the message W → U → V to Host 3.

Edge Chasing Example



- Host 3 knows that Resource Z is held by W.
- It checks and finds that W is waiting on Resource X.
- It sends X the message W → U → V → W to Host 1.
- Host 1 detects a cycle in this graph and knows that there is a distributed deadlock.
- In this case, we wound up building the entire wait-for graph, but in larger systems this might not be the case.

Breaking deadlock

- In order to break the deadlock, a transaction must be aborted.
  - Which one?
- Transactions are given globally unique priority identifiers.
  - Coordinator gives these out when the transaction begins.
- Lowest priority is aborted,
- This way, all servers agree on who to abort.

Summ

- Processing transactions takes more care than message passing
  - Atomicity requirement
- Locking can provide serial equivalence
- Detection of deadlock is needed.
- In a distributed system, two-phase commit can process transactions.
- Detection of distributed deadlock is an issue.
  - Edge chasing can detect distributed deadlock.