Distributed Software Development

Transactions

Chris Brooks

Department of Computer Science
University of San Francisco

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

- 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.

- 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()

- 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.

- 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

17-5: ACID

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

17-6: 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.

17-7: Consistency

- 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.

17-8: 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.

17-9: 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.

17-10: Durability

- 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.

17-11: Collision 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

17-12: 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)

17-13: Lost updates

- 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.

17-14: 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()

17-15: 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.

17-16: Serial equivalence

- 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.