6.824 2018 Lecture 10: Distributed Transactions

```
Topics:
  distributed transactions = concurrency control + atomic commit

what's the problem?
  lots of data records, sharded on multiple servers, lots of clients
  [diagram: clients, servers, data sharded by key]
  client application actions often involve multiple reads and writes
    bank transfer: debit and credit
    vote on an article: check if already voted, record vote, increment count
    install bi-directional links in a social graph
  we'd like to hide interleaving and failure from application writers
  this is a traditional database concern
    today's material originated with [distributed] databases
    but the ideas are used in many distributed systems

example situation
  x and y are bank balances -- records in database tables
  x and y are on different servers (maybe at different banks)
  x and y start out as $10
  client C1 is doing a transfer of $1 from x to y
  client C2 is doing an audit, to check that no money has been lost
  C1:             C2:
  add(x, 1)       tmp1 = get(x)
  add(y, -1)      tmp2 = get(y)
                  print tmp1, tmp2

what do we hope for?
  x=11
  y=9
  C2 prints 10,10 or 11,9

what can go wrong?
  unhappy interleaving of C1 and C2's operations
    e.g. C2 executes entirely between C1's two operations, printing 11,10
  server or network failure
  account x or y doesn't exist

the traditional plan: transactions
  client tells the transaction system the start and end of each transaction
  system arranges that each transaction is:
    atomic: all writes occur, or none, even if failures
    serializable: results are as if transactions executed one by one
    durable: committed writes survive crash and restart
  these are the "ACID" properties
  applications rely on these properties!
  we are interested in *distributed* transactions
    data sharded over multiple servers

the application code for our example might look like this:
  T1:
    begin_transaction()
    add(x, 1)
    add(y, -1)
    end_transaction()
  T2:
    begin_transaction()
    tmp1 = get(x)
    tmp2 = get(y)
    print tmp1, tmp2
    end_transaction()

a transaction can "abort" if something goes wrong
  an abort un-does any record modifications
  the transaction might voluntarily abort, e.g. if the account doesn't exist
  the system may force an abort, e.g. to break a locking deadlock
  some servers failures result in abort
  the application might (or might not) try the transaction again

distributed transactions have two big components:
  concurrency control
  atomic commit

first, concurrency control
  correct execution of concurrent transactions
```

```
The traditional transaction correctness definition is "serializability"
  you execute some concurrent transactions, which yield results
    "results" means new record values, and output
  the results are serializable if:
    there exists a serial execution order of the transactions
    that yields the same results as the actual execution
  (serial means one at a time -- no parallel execution)
  (this definition should remind you of linearizability)

You can test whether an execution's result is serializable by
  looking for an order that yields the same results.
  for our example, the possible serial orders are
    T1; T2
    T2; T1
  so the correct (serializable) results are:
    T1; T2 : x=11 y=9 "11,9"
    T2; T1 : x=11 y=9 "10,10"
  the results for the two differ; either is OK
  no other result is OK
  the implementation might have executed T1 and T2 in parallel
    but it must still yield results as if in a serial order

what if T1's operations run entirely between T2's two get()s?
  would the result be serializable?
  T2 would print 10,9
  but 10,9 is not one of the two serializable results!
what if T2 runs entirely between T1's two adds()s?
  T2 would print 11,10
  but 11,10 is not one of the two serializable results!

Serializability is good for programmers
  It lets them ignore concurrency

two classes of concurrency control for transactions:
  pessimistic:
    lock records before use
    conflicts cause delays (waiting for locks)
  optimistic:
    use records without locking
    commit checks if reads/writes were serializable
    conflict causes abort+retry, but faster than locking if no conflicts
    called Optimistic Concurrency Control (OCC)

today: pessimistic concurrency control
next week: optimistic concurrency control

"Two-phase locking" is one way to implement serializability
  2PL definition:
    a transaction must acquire a record's lock before using it
    a transaction must hold its locks until *after* commit or abort

2PL for our example
  suppose T1 and T2 start at the same time
  the transaction system automatically acquires locks as needed
  so first of T1/T2 to use x will get the lock
  the other waits
  this prohibits the non-serializable interleavings

details:
  each database record has a lock
  if distributed, the lock is typically stored at the record's server
    [diagram: clients, servers, records, locks]
    (but two-phase locking isn't affected much by distribution)
  an executing transaction acquires locks as needed, at the first use
    add() and get() implicitly acquires record's lock
    end_transaction() releases all locks
  all locks are exclusive (for this discussion, no reader/writer locks)
  the full name is "strong strict two-phase locking"
  related to thread locking (e.g. Go's Mutex), but easier:
    explicit begin/end_transaction
    DB understands what's being locked (records)
    possibility of abort (e.g. to cure deadlock)

Why hold locks until after commit/abort?
  why not release as soon as done with the record?
  example of a resulting problem:
```

```
    suppose T2 releases x's lock after get(x)
    T1 could then execute between T2's get()s
    T2 would print 10,9
    oops: that is not a serializable execution: neither T1;T2 nor T2;T1
  example of a resulting problem:
    suppose T1 writes x, then releases x's lock
    T2 reads x and prints
    T1 then aborts
    oops: T2 used a value that never really existed
    we should have aborted T2, which would be a "cascading abort"; awkward

Could 2PL ever forbid a correct (serializable) execution?
  yes; example:
    T1        T2
    get(x)
              get(x)
              put(x,2)
    put(x,1)
  locking would forbid this interleaving
  but the result (x=1) is serializable (same as T2;T1)

Locking can produce deadlock, e.g.
  T1      T2
  get(x)  get(y)
  get(y)  get(x)
The system must detect (cycles? lock timeout?) and abort one of the transactions

The Question: describe a situation where Two-Phase Locking yields
higher performance than Simple Locking. Simple locking: lock *every*
record before *any* use; release after abort/commit.

Next topic: distributed transactions versus failures

how can distributed transactions cope with failures?
  suppose, for our example, x and y are on different "worker" servers
  suppose x's server adds 1, but y's crashes before subtracting?
  or x's server adds 1, but y's realizes the account doesn't exist?
  or x and y both do their part, but aren't sure if the other did?

We want "atomic commit":
  A bunch of computers are cooperating on some task
  Each computer has a different role
  Want to ensure atomicity: all execute, or none execute
  Challenges: failures, performance

We're going to develop a protocol called "two-phase commit"
  Used by distributed databases for multi-server transactions
  We'll assume the database is *also* locking

Two-phase commit without failures:
  the transaction is driven from the Transaction Coordinator
  [time diagram: TC, A, B]
  TC sends put(), get(), &c RPCs to A, B
    The modifications are tentative, only to be installed if commit.
  TC sees transaction_end()
  TC sends PREPARE messages to A and B.
  If A (or B) is willing to commit,
    respond YES.
    then A/B in "prepared" state.
  otherwise, respond NO.
  If both say YES, TC sends COMMIT messages.
  If either says NO, TC sends ABORT messages.
  A/B commit if they get a COMMIT message.
    I.e. they write tentative records to the real DB.
    And release the transaction's locks on their records.

Why is this correct so far?
  Neither A or B can commit unless they both agreed.

What if B crashes and restarts?
  If B sent YES before crash, B must remember!
  Because A might have received a COMMIT and committed.
  So B must be able to commit (or not) even after a reboot.

Thus subordinates must write persistent (on-disk) state:
  B must remember on disk before saying YES, including modified data.
  If B reboots, disk says YES but no COMMIT, B must ask TC, or wait for TC to re-send.
```

```
  And meanwhile, B must continue to hold the transaction's locks.
  If TC says COMMIT, B copies modified data to real data.

What if TC crashes and restarts?
  If TC might have sent COMMIT before crash, TC must remember!
    Since one worker may already have committed.
  And repeat that if anyone asks (i.e. if A/B didn't get msg).
  Thus TC must write COMMIT to disk before sending COMMIT msgs.

What if TC never gets a YES/NO from B?
  Perhaps B crashed and didn't recover; perhaps network is broken.
  TC can time out, and abort (since has not sent any COMMIT msgs).
  Good: allows servers to release locks.

What if B times out or crashes while waiting for PREPARE from TC?
  B has not yet responded to PREPARE, so TC can't have decided commit
  so B can unilaterally abort, and release locks
  respond NO to future PREPARE

What if B replied YES to PREPARE, but doesn't receive COMMIT or ABORT?
  Can B unilaterally decide to abort?
    No! TC might have gotten YES from both,
    and sent out COMMIT to A, but crashed before sending to B.
    So then A would commit and B would abort: incorrect.
  B can't unilaterally commit, either:
    A might have voted NO.

So: if B voted YES, it must "block": wait for TC decision.

Two-phase commit perspective
  Used in sharded DBs when a transaction uses data on multiple shards
  But it has a bad reputation:
    slow: multiple rounds of messages
    slow: disk writes
    locks are held over the prepare/commit exchanges; blocks other xactions
    TC crash can cause indefinite blocking, with locks held
  Thus usually used only in a single small domain
    E.g. not between banks, not between airlines, not over wide area
  Faster distributed transactions are an active research area:
    Lower message and persistence cost
    Special cases that can be handled with less work
    Wide-area transactions
    Less consistency, more burden on applications

Raft and two-phase commit solve different problems!
  Use Raft to get high availability by replicating
    i.e. to be able to operate when some servers are crashed
    the servers all do the *same* thing
  Use 2PC when each subordinate does something different
    And *all* of them must do their part
  2PC does not help availability
    since all servers must be up to get anything done
  Raft does not ensure that all servers do something
    since only a majority have to be alive

What if you want high availability *and* atomic commit?
  Here's one plan.
  [diagram]
  Each "server" should be a Raft-replicated service
  And the TC should be Raft-replicated
  Run two-phase commit among the replicated services
  Then you can tolerate failures and still make progress
  You'll build something like this to transfer shards in Lab 4
  Next meeting's FaRM has a different approach
```