```
6.824 2020 Lecture 12: 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: 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 databases
    but the ideas are used in many distributed systems

the traditional plan: transactions
  programmer marks beginning/end of sequences of code as transactions

example transactions
  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
  T1 and T2 are transactions
    T1: transfer $1 from x to y
    T2: audit, to check that no money is lost
  T1:             T2:
  begin_xaction   begin_xaction
    add(x, 1)       tmp1 = get(x)
    add(y, -1)      tmp2 = get(y)
  end_xaction       print tmp1, tmp2
                  end_xaction

what is correct behavior for a transaction?
  usually called "ACID"
    Atomic -- all writes or none, despite failures
    Consistent -- obeys application-specific invariants
    Isolated -- no interference between xactions -- serializable
    Durable -- committed writes are permanent
  we're interested in ACID for distributed transactions
    with data sharded over multiple servers

What does serializable mean?
  you execute some concurrent transactions, which yield results
    "results" means both output and changes in the DB
  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!
what if x's server does the increment but y's server fails?
  x=11 y=10 is not one of the serializable results!

Why serializability is popular
  An easy model for programmers
    They can write complex transactions while ignoring concurrency
  It allows parallel execution of transactions on different records

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, or y's balance is <= 0
  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 (to provide isolation/serializability)
  atomic commit (to provide atomicity despite failure)

first, concurrency control
  correct execution of concurrent transactions

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
    called Optimistic Concurrency Control (OCC)
  pessimistic is faster if conflicts are frequent
  optimistic is faster if conflicts are rare

today: pessimistic concurrency control
next week: optimistic concurrency control (FaRM)

"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 until the first completely finishes
  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_xaction() 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_xaction
      DB locks automatically, on first use of each record
      DB unlocks automatically, at transaction end
      DB may automatically abort 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

  Two-phase 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 a transaction

  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)

  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 can do their part, but aren't sure if the other will?

  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

  The setting
    Data is sharded among multiple servers
    Transactions run on "transaction coordinators" (TCs)
    For each read/write, TC sends RPC to relevant shard server
      Each is a "participant"
      Each participant manages locks for its shard of the data
    There may be many concurrent transactions, many TCs
      TC assigns unique transaction ID (TID) to each transaction
```

```
    Every message, every table entry tagged with TID
    To avoid confusion

Two-phase commit without failures:
  [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 gets to the end of the transaction.
  TC sends PREPARE messages to A and B.
  If A is willing to commit,
    A responds YES.
    then A is in "prepared" state.
  otherwise, A responds NO.
  Same for B.
  If both A and B say YES, TC sends COMMIT messages to A and B.
  If either A or B says NO, TC sends ABORT messages.
  A/B commit if they get a COMMIT message from the TC.
    I.e. they write tentative records to the real DB.
    And release the transaction's locks on their records.
  A/B acknowledge COMMIT message.

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 (despite crash)!
  Because A might have received a COMMIT and committed.
  So B must be able to commit (or not) even after a reboot.

Thus participants must write persistent (on-disk) state:
  B must remember on disk before saying YES, including modified data.
  If B reboots, and 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.
  Thus TC must write COMMIT to disk before sending COMMIT msgs.
  And repeat COMMIT if it crashes and reboots,
    or if a participant asks (i.e. if A/B didn't get COMMIT msg).
  Participants must filter out duplicate COMMITs (using TID).

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.

Note:
  The commit/abort decision is made by a single entity -- the TC.
```

```
    This makes two-phase commit relatively straightforward.
    The penalty is that A/B, after voting YES, must wait for the TC.

  When can TC completely forget about a committed transaction?
    If it sees an acknowledgement from every participant for the COMMIT.
    Then no participant will ever need to ask again.

  When can participant completely forget about a committed transaction?
    After it acknowledges the TC's COMMIT message.
    If it gets another COMMIT, and has no record of the transaction,
      it must have already committed and forgotten, and can acknowledge (again).

  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.

  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 participant 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]
    The TC and servers should each be replicated with Raft
    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 Spanner uses this arrangement
```