```
6.824 2015 Lecture 20: Two-Phase Commmit

Topics:
  distributed commit, two-phase commit
  distributed transactions
  Argus -- language for distributed programming

Distributed commit:
  A bunch of computers are cooperating on some task, e.g. bank transfer
  Each computer has a different role, e.g. src and dst bank account
  Want to ensure atomicity: all execute, or none execute
    "distributed transaction"
  Challenges: crashes and network failures

We've seen the fundamental problem before
  What to do if *part* of a distributed computation crashes?
  IVY/Treadmarks had no answer
  MR/Spark could re-execute *part* of computation, for big data
  What about for small updates?

Example:
  calendar system, each user has a calendar
  want to schedule meetings with multiple participants
  one server holds calendars of users A-M, another server holds N-Z
  [diagram: client, two servers]
  sched(u1, u2, t):
    begin_transaction
      ok1 = reserve(u1, t)
      ok2 = reserve(u2, t)
      if ok1 and ok2:
        commit
      else
        abort
    end_transaction
  the reserve() calls are RPCs to the two calendar servers
  We want both to reserve, or both not to reserve.
  What if 1st reserve() returns true, and then:
    2nd reserve() returns false (time not available)
    2nd reserve() doesn't return (lost RPC msg, u2's server crashes)
    2nd reserve() returns but then crashes
    client fails before 2nd reserve()
  We need a "distributed commit protocol"

Idea: tentative changes, later commit or undo (abort)
  reserve_handler(u, t):
    if u[t] is free:
      temp_u[t] = taken -- A TEMPORARY VERSION
      return true
    else:
      return false
  commit_handler():
    copy temp_u[t] to real u[t]
  abort_handler():
    discard temp_u[t]

Idea: single entity decides whether to commit
  to prevent any chance of disagreement
  let's call it the Transaction Coordinator (TC)
  [time diagram: client, TC, A, B]
  client sends RPCs to A, B
```

```
  on end_transaction, client sends "go" to TC
  TC/A/B execute distributed commit protocol...
  TC reports "commit" or "abort" to client

We want two properties for distributed commit protocol:
  TC, A, and B start in state "unknown"
    each can move to state "abort" or "commit"
    but then each never changes mind
  Correctness:
    if any commit, none abort
    if any abort, none commit
  Performance:
    (since doing nothing is correct...)
    if no failures, and A and B can commit, then commit.
    if failures, come to some conclusion ASAP.

We're going to develop a protocol called "two-phase commit"
  Used by distributed databases for multi-server transactions
  And by Spanner and Argus

Two-phase commit without failures:
  [time diagram: client, TC, A, B]
  client sends reserve() RPCs to A, B
  client sends "go" to TC
  TC sends "prepare" messages to A and B.
  A and B respond, saying whether they're willing to commit.
    Respond "yes" if haven't crashed, timed out, &c.
  If both say "yes", TC sends "commit" messages.
  If either says "no", TC sends "abort" messages.
  A/B "decide to commit" if they get a commit message.
    I.e. they actually modify the user's calendar.

Why is this correct so far?
  Neither can commit unless they both agreed.
  Crucial that neither changes mind after responding to prepare
    Not even if failure

What about failures?
  Network broken/lossy
  Server crashes
  Both visible as timeout when expecting a message.

Where do hosts wait for messages?
  1) TC waits for yes/no.
  2) A and B wait for prepare and commit/abort.

Termination protocol summary:
  TC t/o for yes/no -> abort
  B t/o for prepare, -> abort
  B t/o for commit/abort, B voted no -> abort
  B t/o for commit/abort, B voted yes -> block

TC timeout while waiting for yes/no from A/B.
  TC has not sent any "commit" messages.
  So TC can safely abort, and send "abort" messages.

A/B timeout while waiting for prepare from TC
  have not yet responded to prepare
  so can abort
  respond "no" to future prepare
```

```
A/B timeout while waiting for commit/abort from TC.
  Let's talk about just B (A is symmetric).
  If B voted "no", it can unilaterally abort.
  So what if B voted "yes"?
  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".

If B voted "yes", it must "block": wait for TC decision.

What if B crashes and restarts?
  If B sent "yes" before crash, B must remember!
    --- this is today's question
  Can't change to "no" (and thus abort) after restart
  Since TC may have seen previous yes and told A to commit
  Thus:
    B must remember on disk before saying "yes", including modified data.
    B reboots, disk says "yes" but no "commit", must ask TC.
    If TC says "commit", copy modified data to real data.

What if TC crashes and restarts?
  If TC might have sent "commit" or "abort" before crash, TC must remember!
    And repeat that if anyone asks (i.e. if A/B/client didn't get msg).
    Thus TC must write "commit" to disk before sending commit msgs.
  Can't change mind since A/B/client have already acted.

This protocol is called "two-phase commit".
  What properties does it have?
  * All hosts that decide reach the same decision.
  * No commit unless everyone says "yes".
  * TC failure can make servers block until repair.

What about concurrent transactions?
  We realy want atomic distributed transactions,
    not just single atomic commit.
  x and y are bank balances
  x and y start out as $10
  T1 is doing a transfer of $1 from x to y
  T1:
    add(x, 1)  -- server A
    add(y, -1) -- server B
  T2:
    tmp1 = get(x)
    tmp2 = get(y)
    print tmp1, tmp2

Problem:
  what if T2 runs between the two add() RPCs?
  then T2 will print 11, 10
  money will have been created!
  T2 should print 10,10 or 9,11

The traditional approach is to provide "serializability"
  results should be as if transactions ran one at a time in some order
  either T1, then T2; or T2, then T1

Why serializability?
  it allows transaction code to ignore the possibility of concurrency
```

```
    just write the transaction to take system from one legal state to another
    internally, the transaction can temporarily violate invariants
      but serializability guarantess no-one will notice

  One way to implement serializabilty is with "two-phase locking"
    this is what Argus does
    each database record has a lock
    the lock is stored at the server that stores the record
      no need for a central lock server
    each use of a record automatically acquires the record's lock
      thus add() handler implicitly acquires lock when it uses record x or y
    locks are held until *after* commit or abort

  Why hold locks until after commit/abort?
    why not release as soon as done with the record?
    e.g. why not have T2 release x's lock after first get()?
      T1 could then execute between T2's get()s
      T2 would print 10,9
      but that is not a serializable execution: neither T1;T2 nor T2;T1

  2PC perspective
    Used in sharded DBs when a transaction uses data on multiple shards
    But it has a bad reputation:
      slow because of multiple phases / message exchanges
      locks are held over the prepare/commit exchanges
      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

  Paxos and two-phase commit solve different problems!
    Use Paxos to high availability by replicating
      i.e. to be able to operate when some servers are crashed
      the servers must have identical state
    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
    Paxos does not ensure that all servers do something
      since only a majority have to be alive

  What if you want high availability *and* distributed commit?
    [diagram]
    Each "server" should be a Paxos-replicated service
    And the TC should be Paxos-replicated
    Run two-phase commit where each participant is a replicated service
    Then you can tolerate failures and still make progress
    This is what Spanner does (for update transactions)

  Case study: Argus

  Argus's big ideas:
    Language support for distributed programs
      Very cool: language abstracts away ugly parts of distrib systems
      Aimed at services interacting via RPC
    Clean handling of RPC and server failure
      Transactional updates via 2PC
      So crash results in entire transaction un-done, not partial update
    Easy persistence ("stable"):
      Ordinary variables automatically persisted to disk
      Automatic crash recovery
    Easy concurrency control:
```

```
    Multiple clients means multiple distributed transactions
    Automatic locking of language objects

The overall design story seems very sensible
  Starting point: you want to handle RPC failures cleanly
  Clean failure handling means Argus needs transactions
  Transaction roll-back means Argus must manage program objects
  Crash recovery means Argus must handle persisting program objects

Picture
  "guardian" is like an RPC server
    has state (variables) and handlers
  "handler" is an RPC handler
    reads and writes local variables
  "action" is a distributed atomic transaction
  action on A
    A RPC to B
      B RPC to C
    A RPC to D
  A finishes action
    prepare msgs to B, C, D
    commit msgs to B, C, D

The style is to send RPC to where the data is
  Not to fetch the data
  Argus is not a storage system

Look at bank example
  page 309 (and 306): bank transfer

Points to notice
  stable keyword (programmer never writes to disk &c)
  atomic keyword (programmer almost never locks/unlocks)
  enter topaction (in transfer)
  coenter (in transfer)
  RPCs are hidden (e.g. f.withdraw())
  RPC error handling hidden (just aborts)

what if deposit account doesn't exist?
  but f.withdraw(from) has already been called?
  how to un-do?
  what's the guardian state when withdraw() handler returns?
    lock, temporary version, just in memory

what if an audit runs during a transfer?
  how does the audit not see the tentative new balances?

if a guardian crashes and reboots, what happens to its locks?
  can it just forget about pre-crash locks?

subactions
  each RPC is actually a sub-action
  the RPC can fail or abort w/o aborting surrounding action
  this lets actions e.g. try one server, then another
  if RPC reply lost, subaction will abort, undo
    much cleaner than e.g. Go RPC

is Argus's implicit locking the right thing?
  very convenient!
  don't have to worry about forgetting to lock!
  (though deadlocks are easy)
```

```
    databases work (and worked) this way; it's a sucessful idea

  is transactions + RPC + 2PC a good design point?
    programmability pro:
      very easy to get nice fault tolerance semantics
    performance con:
      lots of msgs and disk writes
      2PC and 2PL hold locks for a while, block if failure

  is Argus's language integration the right thing?
    i.e. persisting and locking language objects
    it looks very convenient (and it is)

  why didn't more systems pick up on Argus' language-based approach?
    Java RMI is perhaps the closest in common use
    perhaps people prefer to build distributed systems around data
      not around RPC
      e.g. big web sites are very storage-centric
      database provides transactions, persistence, &c
      tables, records, and queries are more powerful than Argus' data
    maybe there is a better language-based scheme waiting to be found
```