6.824 2018 Lecture 17: Eventual Consistency, Bayou

"Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System" by Terry, Theimer, Petersen, Demers, Spreitzer, Hauser, SOSP 95. And some material from "Flexible Update Propagation for Weakly Consistent Replication" SOSP 97 (sections 3.3, 3.5, 4.2, 4.3).

Why are we reading this paper?
  It explores an important and interesting problem space.
  It uses some specific techniques worth knowing.

Big points:
  * Disconnected / weakly connected operation is often valuable.
    iPhone sync, Dropbox, git, Amazon Dynamo, Cassandra, &c
  * Disconnected operation implies eventual (weak) consistency.
    And it takes work (i.e. ordering) to even get that.
  * Disconnected writable replicas lead to update conflicts.
  * Conflict resolution generally has to be application-specific.

Technical ideas to remember:
  Log of operations is equivalent to data.
  Log helps eventual consistency (merge, order, and re-execute).
  Log helps conflict resolution (write operations easier than data).
  Causal consistency via Lamport-clock timestamps.
  Quick log comparison via version vectors.

Paper context:
  Early 1990s
  Dawn of PDAs, laptops, tablets
    Clunky but clear potential
  They wanted devices to be useful regardless of connectivity.
    Much like today's smartphones, tablets, laptops.

Let's build a conference room scheduler
  Only one meeting allowed at a time (one room).
  Each entry has a time and a description.
  We want everyone to end up seeing the same set of entries.

Traditional approach: one server
  Server executes one client request at a time
  Checks for conflicting time, says yes or no
  Updates DB
  Proceeds to next request
  Server implicitly chooses order for concurrent requests

Why aren't authors satisfied with a central server?
  They want full disconnected operation.
    So need DB replica in each device.
    Modify on any device, as well as read.
    "Sync" devices to propagate DB changes (Bayou's anti-entropy).
  They want to be able to use point-to-point connectivity.
    Sync via bluetooth to colleague in next airplane seat.

Why not merge DB records? (Bayou doesn't do this)
 Allow any pair of devices to sync (synchronize) their DBs.
 Sync could compare DBs, adopt other device's changed records.
 Need a story for conflicting entries, e.g. two meetings at same time.
   User may not be available to decide at time of DB merge.
   So need automatic reconciliation.

There are lots of possible conflict resolution schemes.
  E.g. adopt latest update, discard others.
  But we don't want people's calendar entries to simply disappear!

Idea for conflicts: update functions
  Application supplies a function, not just a DB write.
  Function reads DB, decides how best to update DB.
  E.g. "Meet at 9 if room is free at 9, else 10, else 11."
    Rather than just "Meet at 9"
  Function can make reconciliation decision for absent user.
  Sync exchanges functions, not DB content.

Problem: can't just run update functions as they arrive
  A's fn: staff meeting at 10:00 or 11:00
  B's fn: hiring meeting at 10:00 or 11:00

  X syncs w/ A, then B
  Y syncs w/ B, then A
  Will X put A's meeting at 10:00, and Y put A's at 11:00?

Goal: eventual consistency
  OK for X and Y to disagree initially
  But after enough syncing, all devices' DBs should be identical

Idea: ordered update log
  Ordered log of update functions at each device.
  Syncing == ensure both devices have same log (same updates, same order).
  DB is result of applying update functions in order.
  Same log => same order => same DB content.
  Note we're relying here on equivalence of two state representations:
    DB and log of operations.
    Raft also uses this idea.

How can all devices agree on update order?
  Assign a timestamp to each update when originally created.
  Timestamp: <T, I>
  T is creating device's wall-clock time.
  I is creating device's ID.
  Ordering updates a and b:
    a < b if a.T < b.T or (a.T = b.T and a.I < b.I)

Example:
 <10,A>: staff meeting at 10:00 or 11:00
 <20,B>: hiring meeting at 10:00 or 11:00
 What's the correct eventual outcome?
   the result of executing update functions in timestamp order
   staff at 10:00, hiring at 11:00

What DB content before sync?
  A's DB: staff at 10:00
  B's DB: hiring at 10:00
  This is what A/B users will see before syncing.

Now A and B sync with each other
  Each sorts new entries into its log, order by timestamp
  Both now know the full set of updates
  A can just run B's update function
  But B has *already* run B's operation, too soon!

Roll back and replay
  B needs to to "roll back" DB, re-run both ops in the right order
  The "Undo Log" in Figure 4 allws efficient roll-back

Big point: the log holds the truth; the DB is just an optimization

Now DBs will be eventually consistent.
  If everyone syncs enough,
  and no-one creates new updates,
  every device will have the same ordered log,
  and everyone's DB will end up with identical content.

We now know enough to answer The Question.
  initially A=foo B=bar
  one device: copy A to B
  other device: copy B to A
  dependency check?
  merge procedure?
  why do all devices agree on final result?

Will update order be consistent with wall-clock time?
  Maybe A went first (in wall-clock time) with <10,A>
  Device clocks unlikely to be perfectly synchronized
  So B could then generate <9,B>
  B's meeting gets priority, even though A asked first

Will update order be consistent with causality?
  What if A adds a meeting,
    then B sees A's meeting,
    then B deletes A's meeting.
  Perhaps
    <10,A> add
    <9,B> delete -- B's clock is slow
  Now delete will be ordered before add!

So: design so far is not causally consistent.

Causal consistency means that if operation X might have caused
or influenced operation Y, then everyone should order X before Y.

Bayou uses "Lamport logical clocks" for causal consistency
Want to timestamp writes s.t.
if device observes E1, then generates E2, then TS(E2) > TS(E1)
So all devices will order E1, then E2
Lamport clock:
Tmax = highest timestamp seen from any device (including self)
T = max(Tmax + 1, wall-clock time) -- to generate a timestamp
Note properties:
E1 then E2 on same device => TS(E1) < TS(E2)
BUT
TS(E1) < TS(E2) does not imply E1 came before or caused E2

Logical clock solves add/delete causality example
When B sees <10,A>,
B will set its Tmax to 10, so
B will generate <11,B> for its delete

Irritating that there could be a long-delayed update with lower TS
That can cause the results of my update to change
User can never be sure if meeting time is final!
Entries are "tentative"
Would be nice if each update eventually became "stable"
=> no changes in update order up through that point
=> effect of write function now fixed, e.g. meeting time won't change
=> don't have to roll back, re-run committed updates
We'd like to know when a write is stable, and tell the user

Idea: a fully decentralized "commit" scheme (Bayou doesn't do this)
<10,A> is stable if I'll never see a new update w/ TS <= 10
Once I've seen an update w/ TS > 10 from *every* device
I'll never see any new TS < 10 (sync sends updates in TS order)
Then <10,A> is stable
Why doesn't Bayou use this decentralized commit scheme?

Idea: Bayou's "primary replica" to commit updates.
One device is the "primary replica".
Primary sees updates via sync in the ordinary way.
Primary marks each received update with a Commit Sequence Number (CSN).
That update is committed.
So a complete timestamp is <CSN, logical-time, device-id>
Uncommitted updates come after all committed updates
i.e. have infinite CSN
CSN notifications are synced between devices.

Why does the commit / CSN scheme eventually yield stability?
Primary assigns only increasing CSNs.
Device logs order all updates with CSN before any w/o CSN.
So once an update has a CSN, the set of previous updates is fixed.

Will commit order match tentative order?
Often.
Syncs send in log order ("prefix property")
Including updates learned from other devices.
So if A's update log says
<-,10,X>
<-,20,A>
A will send both to primary, in that order
Primary will assign CSNs in that order
Commit order will, in this case, match tentative order

Will commit order *always* match tentative order?
No: primary may see newer updates before older ones.
A has just: <-,10,A> W1
B has just: <-,20,B> W2
If C sees both, C's order: W1 W2
B syncs with primary, W2 gets CSN=5.
Later A syncs w/ primary, W1 gets CSN=6.
When C syncs w/ primary, C will see order change to W2 W1
<5,20,B> W2
<6,10,A> W1
So: committing may change order.

How Bayou syncs (this is anti-entropy)?
  A sending to B
  Need a quick way for B to tell A what to send
  Prefix property simplifies syncing (i.e. sync is always in log order)
    So it's meaningful for B to say "I have everything up to ..."
  Committed updates are easy:
    B sends its highest CSN to A
    A sends log entries between B's highest CSN and A's highest CSN
  What about tentative updates?
  A has:
    <-,10,X>
    <-,20,Y>
    <-,30,X>
    <-,40,X>
  B has:
    <-,10,X>
    <-,20,Y>
    <-,30,X>
  At start of sync, B tells A "X 30, Y 20"
    I.e. for each device, highest TS B has seen from that device.
    Sync prefix property means B has all X updates before 30,
      all Y before 20
  A sends all X's updates after <-,30,X>,
    all Y's updates after <-,20,Y>, &c
  "X 30, Y 20" is a version vector -- it summarizes log content
    It's the "F" vector in Figure 4
    A's F: [X:40,Y:20]
    B's F: [X:30,Y:20]

It's worth remembering the "version vector" idea
  used in many systems
  typically a summary of state known by a participant
  one entry per participant
    meaning "I have seen all updates from Pi through update number Vi"

Devices can discard committed updates from log.
  (a lot like Raft snapshots)
  Instead, keep a copy of the DB as of the highest known CSN.
  Roll back to that DB when replaying tentative update log.
  Never need to roll back farther.
    Prefix property guarantees seen CSN=x => seen CSN<x.
    No changes to update order among committed updates.

How do I sync if I've discarded part of my log?
 (a lot like Raft InstallSnapshot RPC)
 Suppose I've discarded all updates with CSNs.
 I keep a copy of the stable DB reflecting just discarded entries.
 If syncing to device X, and its highest CSN is less than mine:
   Send X my complete DB.
 In practice, Bayou devices keep the last few committed updates.
   To reduce chance of having to send whole DB during sync.

How could we cope with a new server Z joining the system?
  Could it just start generating writes, e.g. <-,1,Z> ?
  And other devices just start including Z in VVs?
  If A syncs to B, A has <-,10,Z>, but B has no Z in VV
    A should pretend B's VV was [Z:0,...]

What happens when Z retires (leaves the system)?
  We want to stop including Z in VVs!
  How to announce that Z is gone?
    Z sends update <-,?,Z> "retiring"
  If you see a retirement update, omit Z from VV
  How to deal with a VV that's missing Z?
  If A has log entries from Z, but B's VV has no Z entry:
    e.g. A has <-,25,Z>, B's VV is just [A:20, B:21]
    Maybe Z has retired, B knows, A does not
    Maybe Z is new, A knows, B does not
  Need a way to disambiguate: Z missing from VV b/c new, or b/c retired?

Bayou's retirement plan
  Z joins by contacting some server X
  Z's ID is <Tz,X>
    Tz is X's logical clock as of when Z joined
  X issues <-,Tz,X>:"new server ID=<Tz,X>"

How does ID=<Tz,X> scheme help disambiguate new vs forgotten?

  Suppose Z's ID is <20,X>
  A syncs to B
    A has log entry from Z <-,25,<20,X>>
    B's VV has no Z entry -- has B never seen Z,
      or already seen Z's retirement?
  One case:
    B's VV: [X:10, ...]
    10 < 20 implies B hasn't yet seen X's "new server Z" update
  The other case:
    B's VV: [X:30, ...]
    20 < 30 implies B once knew about Z, but then saw a retirement update

In a few lectures: Dynamo, a real-world DB with eventual consistency