6.824 2015 Lecture 12: Eventual Consistency, Bayou

Managing Update Conflicts in Bayou, a Weakly Connected Replicated
Storage System Terry, Theimer, Petersen, Demers, Spreitzer, Hauser,
SOSP 95
some material from Flexible Update Propagation for Weakly Consistent
Replication, SOSP 97

Why this paper?
  Eventual consistency is pretty common
    git, iPhone sync, Dropbox, Amazon Dynamo
  Why do people like eventual consistency?
    fast read/write of local copy (no primary, no paxos)
    disconnected operation
  What goes wrong?
    doesn't look like "single copy" (no primary, no paxos)
    conflicting writes to different copies
    how to reconcile them when discovered?
  Bayou has the most sophisticated reconciliation story

Paper context:
  Early 1990s (like Ficus)
  Dawn of PDAs, laptops, tablets
    H/W clunky but clear potential
    Commercial devices did not have wireless
  Devices might be off or not have network access
    This problem has not gone away!
    iPhone sync, Dropbox sync, Dynamo

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 we satisfied with central server?
 I want to use scheduler on disconnected iPhone &c
   So need DB replica in each node.
   Modify on any node, as well as read.
 Periodic connectivity to net.
 Periodic direct contact with other calendar users (e.g. bluetooth).

Straw man 1: merge DBs.
 Similar to iPhone calendar sync, or file sync.
 May need to compare every DB entry -- lots of time and net b/w.
 Still need a story for conflicting entries, i.e. two meetings at same time.
   User may not be available to decide at time of DB merge.
   So need automatic reconciliation.

Idea for conflicts: update functions
  Application supplies a function, not a new value.
  Read current state of DB, decide best change.
  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 apply update functions to DB replica
  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 nodes' DBs should be identical

Idea: ordered update log
  Ordered log of updates at each node.
  Syncing == ensure both nodes have same updates in log.
  DB is result of applying update functions in order.
  Same log => same order => same DB content.

How can nodes agree on update order?
  Update ID: <time T, node ID>
  T is creating node's wall-clock time.
  Ordering updates a and b:
    a < b if a.T < b.T or (a.T = b.T and a.ID < b.ID)

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: staff at 10:00
  B: hiring at 10:00
  This is what A/B user will see before syncing.

Now A and B sync with each other
  Each sorts new entries into its log, order by time-stamp
  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
  Big point: the log holds the truth; the DB is just an optimization
  We will optimize roll-back in a bit

Displayed meeting room calendar entries are "tentative"
  B's user saw hiring at 10, then it changed to hiring at 11

Will update order be consistent with wall-clock time?
  Maybe A went first (in wall-clock time) with <10,A>
  Node clocks unlikely to be perfectly synchronized
  So B could then generate <9,B>
  B's meeting gets priority, even though A asked first
  Not "externally consistent"

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!

Lamport logical clocks for causal consistency
  Want to timestamp events s.t.
    if node observes E1, then generates E2, then TS(E2) > TS(E1)
  So all nodes will order E1, then E2
  Tmax = highest time-stamp seen from any node (including self)
  T = max(Tmax + 1, wall-clock time) -- to generate a timestamp
  Note properties:
    E1 then E2 on same node => TS(E1) < TS(E2)
    BUT
    TS(E1) < TS(E2) does not imply E1 came before 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 always 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!
  Would be nice if updates were eventually "stable"
    => no changes in update order up to that point
    => results can never again change -- you know for sure when your meeting is
    => don't have to roll back, re-run committed updates

Bad idea: a fully decentralized "commit" scheme
  Proposal: <10,A> is stable if all nodes have seen all updates w/ TS <= 10
  Have sync always send in log order -- "prefix property"
  If you have seen updates w/ TS > 10 from *every* node
    Then you'll never again see one < <10,A>
    So <10,A> is stable
  Why doesn't Bayou do this?

How does Bayou commit updates, so that they are stable?
 One node designated "primary replica".
 It marks each update it receives with a permanent CSN.
   Commit Sequence Number.
   That update is committed.
   So a complete time stamp is <CSN, local-time, node-id>
   Uncommitted updates come after all committed updates.
 CSN notifications are synced between nodes.
 The CSNs define a total order for committed updates.
   All nodes will eventually agree on it.

Will commit order match tentative order?
  Often.
  Syncs send in log order (prefix property)
    Including updates learned from other nodes.
  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, gets CSN=5.
  Later A syncs w/ primary, gets CSN=6.
  When C syncs w/ primary, order will change to W2 W1
    <5,20,B> W1
    <6,10,A> W2
  So: committing may change order.

Committing allows app to tell users which calendar entries are stable.
  A stable meeting room time is final.

Nodes can discard committed updates.
  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?
 Suppose I've discarded all updates with CSNs.
 I keep a copy of the stable DB reflecting just discarded entries.
 When I propagate to node X:
   If node X's highest CSN is less than mine,
     I can send him my stable DB reflecting just committed updates.
     Node X can use my DB as starting point.
     And X can discard all CSN log entries.
     Then play his tentative updates into that DB.
   If node X's highest CSN is greater than mine,
     X doesn't need my DB.
 In practice, Bayou nodes keep the last few committed updates.
   To reduce chance of having to send whole DB during sync.

How to sync?
  A sending to B
  Need a quick way for B to tell A what to send
  Committed updates easy: B sends its CSN to A
  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"
    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,X>, &c
  This is a version vector -- it summarize log content
    It's the "F" vector in Figure 4
    A's F: [X:40,Y:20]
    B's F: [X:30,Y:20]

How could we cope with a new server Z joining the system?
  Could it just start generating writes, e.g. <-,1,Z> ?
  And other nodes 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 Z"

  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
    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

  Let's step back

  Is eventual consistency a useful idea?
    Yes: people want fast writes to local copies
    iPhone sync, Dropbox, Dynamo, Riak, Cassandra, &c

  Are update conflicts a real problem?
    Yes -- all systems have some more or less awkward solution

  Is Bayou's complexity warranted?
    I.e. log of update functions, version vectors, tentative operations
    Only critical if you want peer-to-peer sync
      I.e. both disconnected operation AND ad-hoc connectivity
    Only tolerable if humans are main consumers of data
    Otherwise you can sync through a central server (iPhone, Dropbox)
    Or read locally but send updates through a master (PNUTS, Spanner)

  But there's are good ideas for us to learn from Bayou
    Update functions for automatic application-driven conflict resolution
    Ordered update log is the real truth, not the DB
    Logical clock for causal consistency
```