Bayou: A Weakly Connected Replicated Storage System

Robert Grimm
New York University
Altogether Now: The Three Questions

- What is the problem?
- What is new or different?
- What are the contributions and limitations?
A replicated storage system

- Designed with mobile computing in mind
  - Supports read/write anywhere
  - Makes very limited assumptions about connectivity

- Provides eventual consistency
  - Exposes both tentative and stable data

- Is not transparent to applications
  - Writes are <update, dependency check, merge procedure>’s

- Is centered around an epidemic anti-entropy protocol
  - One-way operation between pairs of servers
  - Propagation of writes
  - Constrained by accept order
The Target Environment

- A worst-case scenario…
  - Mobile computers
  - Expensive connection time
  - Frequent disconnections
  - Computers never connected simultaneously

- Though, Bayou also works in less bad environments
  - Considerable flexibility in setting anti-entropy policies
    - When to reconcile
    - With which replicas to reconcile
    - When to truncate the write-log
    - From which servers to create new replicas
Let’s Start from Scratch: Calendaring as an Example

- Two main issues related to consistency
  - Ordering of operations
  - Detection and resolution of conflicts
- The traditional solution: Lots of clients, one server
  - Ordering: One copy, server picks order
  - Conflicts: Server checks for conflicts, returns errors
- So, why not use this approach?
  - Local access on personal devices
  - Intermittent connectivity with Internet
  - Intermittent connectivity with other users (Infrared, Bluetooth, …)
Straw Man: Swap/Sync Databases

- May be resource intensive
  - Might require lots of network bandwidth
- Hard to ensure consistency
  - There is no notion (of ordering) of operations
  - It is hard to *automatically* detect conflicts and resolve them
  - Problem: Viewing DB as collection of bits
    - Represents a snapshot in time
  - Solution: View DB in terms of updates
    - Operational: Read, think, make change
    - Well-ordered: Ensure that all replicas converge on same snapshot
Towards a More Good Solution

- Maintain an ordered list of updates for each node
  - Enter the write log
- Make sure every node has the same updates
- Make sure every node applies updates in same order
  - Accept order, causal accept order, total order
- Make sure that updates are deterministic
  - No access to local time, server name, rand(), …

- Now, a sync does not merge databases, but lists
  - Much easier than merging of collections of bits
Observation: We very much care about ordering
  - Even for tentative operations
Read your writes: W→R
  - E.g., change password, log in
Monotonic reads: W→R1→R2
  - E.g., meetings stay in calendar, listed emails readable
Write follows read: W1→R1→W2 → W1→W2
  - E.g., newsgroup reply appears after original post
Monotonic writes: W1→W2
  - E.g., last text file edit survives
An Example Write

Bayou_Write(
    update = {insert, Meetings, 12/18/95, 1:30pm, 60min, “Budget Meeting”},
    dependency_check = {
        query = “SELECT key FROM Meetings WHERE day = 12/18/95
        AND start < 2:30pm AND end > 1:30pm”,
        expected_result = EMPTY},
    mergeproc = {
        alternates = [{12/18/95, 3:00pm}, {12/19/95, 9:30am}];
        newupdate = {};
        FOREACH a IN alternates {
            # check if there would be a conflict
            IF (NOT EMPTY (SELECT key FROM Meetings WHERE day = a.date
                            AND start < a.time + 60min AND end > a.time))
                CONTINUE;
            # no conflict, can schedule meeting at that time
            newupdate = {insert, Meetings, a.date, a.time, 60min, “Budget Meeting”};
            BREAK;
        }
        IF (newupdate = {})) # no alternate is acceptable
            newupdate = {insert, ErrorLog, 12/18/95, 1:30pm, 60min, “Budget Meeting”};
        RETURN newupdate;
    }
)

- Also need timestamp: <local time, accepting node>
  - Lamport clock for causal accept order
Propagating Writes

- Unidirectional, peer-to-peer synchronization
  - By wired/wireless network, floppy disk, USB keychain,…
- Updates may appear out of (total) order
  - E.g., <701, A>, <770, B>; node B receives <701, A>
  - Need to be merged into log
    - Undo newer updates (e.g., <770, B>)
    - Insert just received updates
    - Replay the log
- User’s view of data (calendar) may change
  - But when everybody has seen all writes, everybody will agree
We Like Short Logs...
Step 1: How about Commitment?

- We need to know when everybody has seen a write
  - Lamport clocks preserve causal order, but don’t provide global consensus
- We need a notion of commitment
  - For entry X to be committed, everyone must agree on
    - The total order of all previous writes
    - The fact that X is next in this total order
    - The fact that all tentative entries follow after X
  - “Any mechanism that stabilizes the position of a write in the log can be used.”
Bayou’s Civilized Commitment Procedure

- Each data collection has one *primary* replica (why?)
  - Commits all writes for that collection
  - Marks each write with a commit sequence number (CSN)
    - Timestamp really is <CSN, local time, accepting node>
  - Propagates commitments during anti-entropy

- How to ensure that CSN order observes causal accept order?
  - Local time actually is Lamport (logical) time
  - Everybody propagates updates in order
  - As a result, primary sees updates in causal order and commits them in that order
We Like Short Logs...
Step 2: Let’s Throw Writes Away!

- Truncating the log
  - Tentative writes must never be discarded
    - May have to be undone and redone (due to reordering)
  - Committed writes *may* be discarded
    - But other replicas may not yet have seen them
    - So, keep some amount of history

- But, where did the truncated writes go?
  - We don’t just have a log, but also an actual database
    - Also contains tentative writes
    - But all committed entries are marked as such (flag bit)
    - We track omitted sequence number (OSN)
Let’s Throw Writes Away! (cont.)

- During anti-entropy, we may have to send DB (!)
  - If receiver’s CSN is smaller than sender’s OSN
    - I.e., if receiver’s head of log is before sender’s tail of log
  - Sender’s DB provides new starting point for receiver
    - Receiver discards committed writes
    - Receiver and send continue with rest of anti-entropy
Some More Details

- Replicas can be added and removed dynamically
  - Addition/removal is relative to another replica
    - Replicas named relative to that replica (preserves causal order)
- Access control performed at granularity of database
  - Based on public/private key cryptography
  - Checked by accepting and by committing replica
    - Accepting replica first-line defense against unauthorized access
    - Committing replica definitive authority
What Do You Think?