Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System
Bayou

- **assumptions:**
  - read/write anywhere
  - DB fully replicated
  - disconnected operation

- **goals**
  - read anywhere/write anywhere
  - weak connectivity
  - eventual consistency
  - app-specific conflict resolution

- **methods**
  - optimism
  - anti-entropy sessions
  - session semantics
  - per-update dependency checks and merge procedures
  - committed vs tentative updates
  - security
System Model
Apps

- **Apps:**
  - **Scheduler**
    - users can specify multiple possible meeting times
  - **Bib database**
    - collisions on bib entry labels
    - duplicate entries

- **If:**
  - conflict detection far in future

- **Then:**
  - Need app-specific
    - conflict detection
    - resolution (merge procedures)
Fun Stuff

- **Dependency Checks**
  - Better than version vectors because:
    - read/write conflicts
    - arbitrary, multi-item constraints
  - Example 1: scheduler
    - checks to see if the requested time filled
  - Example 2: bank
    - Need to transfer $100 from A to B. If app checked first and saw $150, traditional opt approach would be to check for $150 before committing
    - Bayou can formalize real requirement, i.e., $A \geq \$100$

```c
Bayou_Write (update, dependency_check, mergeproc) {
    IF (DB_Eval (dependency_check.query) <> dependency_check.expected_result) {
        resolved_update = Interpret (mergeproc);
    } ELSE {
        resolved_update = update;
        DB.Apply (resolved_update);
    }
}
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;
    )
)
Replica Consistency

- **Eventual consistency**
  - writes must be committed in same order everywhere

- **New writes tentative**
  - ordered by local server timestamp
  - TS is mono-increasing: \(<TS,\text{server ID}>\)
  - immediately applied!
    - must have undo
    - must have redo

- **Piecewise determinism:**
  - no dep on anything but replica contents.
Write Stability

Write is **stable** when it has been executed for last time at that server.

- **How to determine stability?**
  - matrix clocks
  - timestamps
  - cheat

- **Primary commit**
  - one server responsible for final ordering of all updates
  - ordering?
    - not clear
    - hopefully consistent w/ timestamp order, but maybe not if some servers disconnected
Applying Sets of Writes

Receive_Writes (writeset, received_from) {
    IF (received_from = CLIENT) {
        # Received one write from the client, insert at end of WriteLog
        # first increment the server’s timestamp
        logicalclock = MAX(systemclock, logicalclock + 1);
        write = First(writeset);
        write.WID = {logicalclock, myServerID};
        write.state = TENTATIVE;
        WriteLog_Append(write);
        Bayou_Write(write.update, write.dependency_check, write.mergeproc);
    } ELSE {
        # Set of writes received from another server during anti-entropy,
        # therefore writeset is ordered
        write = First(writeset);
        insertionPoint = WriteLog_IdentifyInsertionPoint(write.WID);
        TupleStore_RollbackTo(insertionPoint);
        WriteLog_Insert(writeset);
        # Now roll forward
        FOREACH write IN WriteLog AFTER insertionPoint DO
            Bayou_Write(write.update, write.dependency_check, write.mergeproc);
            # Maintain the logical clocks of servers close
            write = Last(writeset);
            logicalclock = MAX(logicalclock, write.WID.timestamp);
    }
}
Details

- **“O” vector**
  - TS’s of last tossed (‘omitted’) writes
  - works because writes tossed in order
  - writes from any server are propagated and committed in TS order

- **Two more vectors**
  - “C” : max TS’s of committed writes
  - “F” : max TS’s of tentative writes
  - used for anti-entropy, not conflict detection
Can we toss committed writes?
Access Control

- **We are:**
  - potentially disconnected
  - and still need to make progress

- **Outline:**
  - public-key crypto
  - both clients/servers have signed certificates for rights
  - delegation certificates
  - revocation
Flexible Update Propagation for Weakly Consistent Replication
Basic Anti-entropy

- **Servers assign accept-stamps**
  - provide total order for writes accepted by a server, and
  - partial order “accept-order” over all writes

- **Anti-entropy**
  - one-way op between servers
  - consists of propagation of writes
  - write propagation constrained by accept-order
    - helps maintain “prefix property”
  - partner choice undefined
Anti-Entropy

anti-entropy(S,R) {
    Get R.V from receiving server R
    # now send all the writes unknown to R
    w = first write in S.write-log
    WHILE (w) DO
        IF R.V(w.server-id) < w.accept-stamp THEN
            # w is new for R
            SendWrite(R, w)
            w = next write in S.write-log
        END
    }
Anti-Entropy W/ Commits

anti-entropy(S,R) {
    Get R.V and R.CSN from receiving server R
    # first send all the committed writes that R does not know about
    IF R.CSN < S.CSN THEN
        w = first committed write that R does not know about
        WHILE (w) DO
            IF w.accept-stamp <= R.V(w.server-id) THEN
                # R has the write, but does not know it is committed
                SendCommitNotification(R, w.accept-stamp, w.server-id, w.CSN)
            ELSE
                SendWrite(R, w)
            END
        END
        w = next committed write in S.write-log
    END
    END
    w = first tentative write
    # now send all the tentative writes
    WHILE (w) DO
        IF R.V(w.server-id) < w.accept-stamp THEN
            SendWrite(R, w)
            w = next write in S.write-log
        END
    }
}
Anti-Entropy W/ Commits and Truncation

```plaintext
anti-entropy(S,R) {
    Request R.V and R.CSN from receiving server R
    #check if R's write-log does not include all the necessary writes to only send writes or
    #commit notifications
    IF (S.OSN > R.CSN) THEN
        # Execute a full database transfer
        Roll back S's database to the state corresponding to S.O
        SendDatabase(R, S.DB)
        SendVector(R, S.O) # this will be R's new R.O vector
        SendCSN(R, S.OSN) # R's new R.OSN will now be S.OSN
    END
    #now same algorithm as in Figure 2, send anything that R does not yet know about
    IF R.CSN < S.OSN THEN
        w = first committed write that R does not yet know about
        WHILE (w) DO
            IF w.accept-stamp <= R.V(w.server-id) THEN
                SendCommitNotification(R, w.accept-stamp, w.server-id, w.CSN)
            ELSE
                SendWrite(R, w)
            END
            w = next committed write in S.write-log
        END
    END
    w = first tentative write in S.write-log
    WHILE (w) DO
        IF R.V(w.server-id) < w.accept-stamp THEN
            SendWrite(R, w)
        w = next write in S.write-log
    END
}
Extensions

- **Transportable media**
  - parameters CSN and V define minimum state receiver must have in order to use

- **Session guarantees**
  - writes must be "causally ordered"
    - A precedes B iff A was already known to the server that received B from a client
    - scalar logical clock
## Features enabled by specific anti-entropy design components

<table>
<thead>
<tr>
<th>Feature</th>
<th>Design Choices</th>
<th>One-way Peer-to-Peer</th>
<th>Operation-based</th>
<th>Partial Propagation Order</th>
<th>Causal Propagation Order</th>
<th>Stable Log Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arbitrary Communication Topologies</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arbitrary Policy Choices</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Low-bandwidth Networks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Incremental Progress</td>
<td></td>
<td>❅</td>
<td>❅</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eventual Consistency</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>❅**</td>
</tr>
<tr>
<td>Aggressive Storage Management</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Use of Transportable Media</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>❅</td>
</tr>
<tr>
<td>Light-weight Dynamic Replica Sets</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Per Update Conflict Management</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>❅</td>
</tr>
<tr>
<td>Session Guarantees</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Policy Choices

- when to reconcile
  - periodically, manual, system trigger

- who to reconcile with
  - network characteristics, up-to-dateness of replicas, truncations

- how aggressively to truncate write log
  - eh.

- who to create new server from
  - up-to-dateness, identifier length?
Creation and Retirement Writes

- Bayou server $S_i$ creates itself by sending creation write to another server $S_k$
  - gives $S_i$ name $<T_{k,i}, S_k>$
  - tells others of $<T_{k,i}, S_k>$'s existence

- Disappearing servers issue "retirement writes"

- What do we know if target of anti-entropy, $S_L$, doesn't have an entry for some server $<T_{k,i}, S_k>$?
  - either $S_L$ hasn't heard of $<T_{k,i}, S_k>$, or knows that it is gone
  - tell by looking $S_L.V[S_k]$
**My Summary**

- **Dynamite stuff**
  - flexibility on all levels
  - lots of knobs to be tweaked

- **Downsides**
  - not at all clear that merge procs etc. that useful
  - primary copy is a cop-out