Bayou

CS380L: Mike Dahlin

November 12, 2003

Wherever you go, there you are.

1 Preliminaries

1.1 Review

1.2 Outline

- System model
- 3 key ideas
  - Update propagation
  - Conflict detection and resolution
  - Consistency for clients
- Other questions

1.3 Preview

Security

2 System model and goals

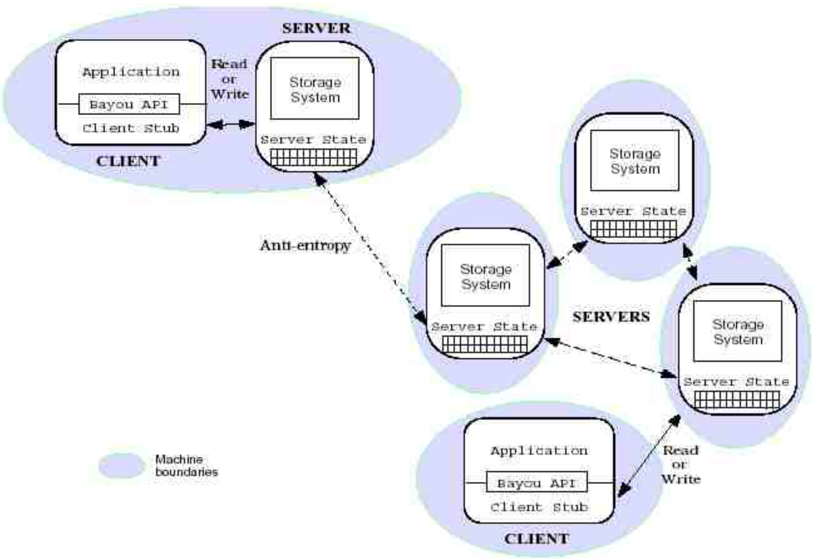

Figure 1. Bayou System Model

- Many machines
- Each stores copy of DB
- Write-any, read-any model
  - $\rightarrow$ consistency is big issue
  - Alternatives
    * Central server
    * Central server + caching + lock manager
    * Quorum (notice – central server is special case of quorum)

3 Update Propagation

3.1 Vector clock

Basic building block in distributed systems – want to capture causality – did event A happen causally before event B?

- E.g., when machine M generated event B, had it seen event A?

Solution: Each machine maintains a *vector clock*

1. $V_i[i]$ = Number of events that have occurred so far at P_i
2. if $V_i[j] = k$ then P_i knows that k events have occurred at P_j

Invarients:

- Get #1 by incrementing $V_i[i]$ at each local event
- Get #2 by sending $V_i[]$ to j whenever i sends j information j then sets $V_j[k] = max(V_i[k], V_j[k]) \forall k$

3.2 log

- Store all local and remote updates in log
- Invariant: Sorted by $(acceptTime, serverId)$

Goal: If everyone has the same log, then they agree on all objects' values.

3.3 Basic protocol

```
write(data){       // Write data at server S
  w.data = data;
  w.acceptStamp = V_S[S]
  w.serverID = S;
  V_S[S]++;
  writeLog.append(S);
}

send_anti_entropy(S, R){ // Server S updates receiving server R
  get V_R[] from R
```

```
  w = first write in S.writeLog



  while(w){






    if(V_R[w.serverID] < w.acceptStamp){
      send w to R
    }
    w = S.writeLog.next();
  }
}

recv_anti_entropy(R, S){ // Server R receives update for S
  send V_R[] to S




  w = recv();
  while(w != eof){







    R.writeLog.insertSorted(w); // Sorted by (acceptTime, serverID)
    V_R[w.serverID] = w.acceptStamp;
Q1: V_R[R] = max(V_R[R], w.acceptStamp+1);

    w = recv();
  }
}
```

- *prefix property* – if log contains write (serverID, acceptStamp), it also contains all writes (serverID, acceptStamp') where acceptStamp' < acceptStamp
- Question 1: How is *causal order* in log maintained?
  - Need to make sure all subsequent writes land *after* all data read
  - $\rightarrow$ acceptStamp of all subsequent writes must be larger than acceptStamp of any writes I have received
- Question 2: When are writes "stable"?
  - Notice that writes can be re-ordered. This can cause "conflicts." E.g., you can have a write that takes $100 out of your checking account that is legal until an earlier write arrives that takes your

checking account down to $10.

- We will talk more about conflicts later, but for now, think of Coda – what happens if two different nodes update the same datum?
- A stable write is one that can not be reordered (e.g., all previous writes are stable too.)
- A prefix of the log is stable.
- What prefix is stable?
- *Answer 1:* all writes with w.acceptStamp $< \min_{\forall k}(V_R[k])$
- What is wrong with answer 1?
- *Answer 2:* "Commit sequence number"
  - One node acts as "primary" – it is allowed to assign commit sequence number to writes
  - When it receives a write, it assigns w.csn = CSN++;
  - Everyone sorts log by (CSN, acceptStamp, serverID)

New protocol (with CSN)

```
write(data){       // Write data at server S
  // As above
}



send_anti_entropy(S, R){ // Server S updates receiving server R
  get V_R[] and CSN_R from R

  w = first committed write w/ CSN larger than CSN_R      // *
  if(w == NULL){                                          // *
    w = first tentative write;                            // *
  }                                                       // *

  while(w){
    if(w.CSN > CSN_R && w.acceptStamp <= V_R[w.serverID]){// **
      // R has write but does not know it is committed    // **
      assert(w.CSN != NULL);                              // **
      SendCommitNotification(R, w.acceptStamp, w.serverID, w.CSN); // **
    }                                                     // **
    else if(w.acceptStamp > V_R[w.serverID]){
      send w to R
    }
    w = S.writeLog.next();
  }
}




recv_anti_entropy(R, S){ // Server R receives update for S
  send V_R[] and CSN to S                                   // *
```

```
  w = recv();
  while(w != eof){
    if(w.CSN != NULL){ // Recv committed write                   // **
      // If this is a write I already know about that has been committed // **
      // Q: What should this ``if'' be?                           // **
      // A: (V_R[w.serverID] >= w.acceptStamp)                    // **
      if( WHAT ){ // A commit of a write I already saw            // **
        wold = r.writeLog.remove(w.serverId, w.acceptStamp); // **
        w.data = wold.data;                                       // **
      }                                                           // **

      R.writeLog.insertSorted(w);                                 // **
      // Q: What else do I have to do?                            // **
      V_R[w.serverID] = max(w.acceptStamp, V_R[s.serverID]); // **
      V_R[R] = max(V_R[R], w.acceptStamp+1);                      // **
      CSN = max(CSN, w.CSN);

    }
    else{ // Recv tentative write
      R.writeLog.insertSorted(w);


      // Q: What else do I have to do?
      V_R[w.serverID] = w.acceptStamp;
      V_R[R] = max(V_R[R], w.acceptStamp+1);
    }
    w = recv();
  }
}
```

- Question 3: What if I want to read the database?
  - Could scan through the log...
  - Keep checkpoint on disk – corresponds to committed writes
  - Keep in-memory state – corresponds to committed + tentative writes
  - Allow read(key) and readCommitted(key)
- Also note: insertSorted just got easier
  - Roll back to earliest write you will accept
  - Roll forward as new writes arrive
  - Good news: Everything is sequential traversal through log (O(N))
  - Better news: simpler to build (than on-disk reliable priority queue).

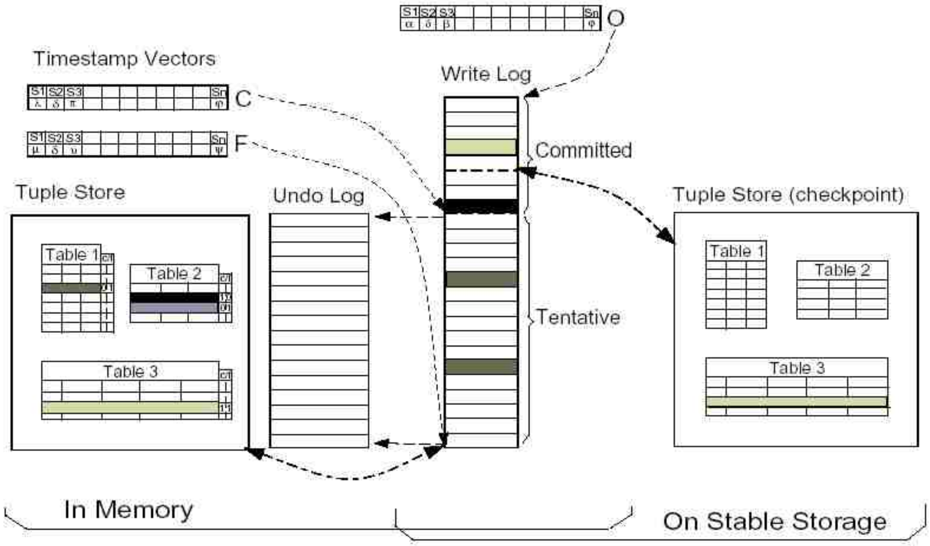

Figure 4. Bayou Database Organization

- Data structures
  - *memTupleStore*: In-memory tuple store
  - *checkpoint*: On-disk checkpoint tuple store; all committed writes
  - *committedLog*: On-disk committed write log
  - *tentativeLog*: On-disk tentative write log
  - *For simplicity: slightly different than paper*
    * roll back by reverting to checkpoint (rather than undo log)
    * memoryTupleStore = shadow copy of on-disk checkpoint (contains only recent writes) → read:

```
read(id)
  r = memTupleStore.read(id);
  if(r == null){
    r = checkpoint.read(id);
  }
  return r;
```

    * DA: Log merge costs O(age of oldest uncommitted write) instead of O(age of oldest write received)

```
recv_anti_entropy(R, S){ // Server R receives update from S
  send V_R[] and CSN to S
  w = recv();
  while(w != eof){                // ***  Don't want to
    recvd.append(w);              // ***  roll back for each
    w = recv();                   // ***  individual write!
  }
  R.oldTentativeLog = R.tentativeLog;          // *
  R.tentativeLog = new Log();                  // *
  R.memTupleStore = newTS();                   // *
  while(recvd.hasMore()){
    w = recvd.next();
    if(w.CSN != NULL){  // Recv committed write
```

```
      if(V_R[w.serverID] >= w.acceptStamp){ // Already seen it
        wold = R.tentativeLog.remove(w.serverId, w.acceptStamp);
        w.data = wold.data;
      }
      R.committedLog.append(w);                // **
      R.checkpoint.update(w);                  // **
      V_R[w.serverID] = max(w.acceptStamp, V_R[s.serverID]);
      V_R[R] = max(V_R[R], w.acceptStamp+1);
      CSN = max(CSN, w.CSN);
    }
    else{ // Integrate tentative write
      while(R.oldTentativeLog.head() BEFORE w){ // ***
        o = R.oldTentativeLog.removeHead()      // ***
        R.tentativeLog.append(o);               // ***
        R.memTupleStore.update(o);              // ***
      }
      R.tentativeLog.append(w);                 // ****
      R.memTupleStore.update(w);                // ****
      V_R[w.serverID] = w.acceptStamp;
      V_R[R] = max(V_R[R], w.acceptStamp+1);
    }
    while(R.oldTentativeLog.head() != NULL){   // ***
      o = R.oldTentativeLog.removeHead()       // ***
      R.tentativeLog.append(o);                // ***
      R.memTupleStore.update(o);               // ***
    }
  }
}
```

- Implementation issues (exercise for reader...)
  - As noted above: Using undo log might be more efficient. How to change code to handle?
  - Transactional update of multiple disk structures (e.g., “tenativeLog.remove()” etc.)
  - Defensive programming: what if sender sends unexpected stuff (or stuff is just delayed/reordered in network? (More generally: what if a node is broken and sends corrupted stuff?)
- Question 4: Log truncation
  - Any node can truncate a stable prefix from its log at any time
    - Typically: limit log size
  - What are the new issues?
    - How do you detect that someone you are talking to has truncated his log too far?
    - What do you do then?
  - How should it work?
- Question 5: Add/remove servers
  - What are the new issues?
  - How should it work?

4 Admin

5 Conflict detection and resolution

5.1 Problem

- Recall from Coda – write/write conflict
- Same issue in Bayou
- 2 questions: (1) how do you detect conflicts? (2) how do you resolve them?
- 2 answers: (1) generic rules, (2) Bayou extensions and generalizations

5.2 Generic rules

5.2.1 Generic write/write conflict detection

Note: this is not what Bayou does

- Question: How would you detect write/write conflict as you replay log?
  - Hint: I think you may want to add some additional information to each write
  - One solution:
    1. Include in each write the version number just overwritten (e.g., “previous serverID”, “previous acceptOrder)
    2. Include serverID and acceptOrder in tuple store and checkpoint
    3. When applying write to checkpoint or tuple store, compare the write’s “previous version number” to the tuple store’s version number. If they mismatch, you have a write-write conflict.

5.2.2 Generic write/write conflict resolution

- What can you do if you detect such a conflict?
  - Last write wins?
  - Omit that write? (What if other writes to other locations depend on this one?)
  - Omit all subsequent writes from that server? (When can you start accepting again? Perhaps you can accept writes that are causally after the conflict? Non-trivial. May omit lots of writes...)
  - Omit transaction:
    * Associate a transactionID with each write
    * If any write in a transaction is in conflict, omit all writes from that transaction
    * Notice:
      · Transaction cannot “commit” (become durable) until stable
      · Subsequent transactions may fail (same write-write conflict)
      · What about write/read conflicts?
- Oracle: 14 “rules” that can be spedified for each write
  - Last write wins
  - Omit conflicting writes (log error)
  - Preferred server wins

  - Preferred user wins
  - Larger value wins
- Bayou: Application-specifc detection and resolution

5.2.3 Generic ways to reduce number of conflicts

- Synchronize often
  - Gray "The dangers of replication and a solution"
    * $P(collision) \propto disconnectionDuration^2$
    * intuition:
      · Suppose 2 nodes each randomly write k of N locations
      · On my first write k/N chance that my write conflicts with one of yours
      · On my second write, k/N chance that my write conflicts
      · ...
      · Expected number of collisions across all writes $\approx k^2/N$
- Causal order tricks
  - Example:
    suppose nodes A and B each commit two writes during a period of disconnection

A	B
1A	1B
2A	2B

    There are six legal CAUSAL serializations (causal: 1a preceeds 2a and 1b preceeds 2b)

    1a 1b 2a 2b
    1a 1b 2b 2a
    1a 2a 1b 2b
    1b 1a 2b 2a
    1b 1a 2a 2b
    1b 2b 1a 2a

  - If we assume that both nodes started with the same acceptOrder values, and that they increment their logical clocks by one at each write, and these are the only writes, and if we assume a¡b, then we get

    1a 1b 2a 2b

  - But the others are all possible by running clocks at different rates at different nodes (and we are allowed to run logical clocks at arbitrarily different rates between connections.)
  - One interesting option: relate acceptOrder logical clock to timeOfDay clock
  - $\rightarrow$ final order corresponds (mostly) to actual time of day when writes committed
  - One particularly interesting variation is a CLUSTER logical clock (a la Vahdat SOSP2001) "serialization order where writes accepted by the same partition during a particular interval cluster together"

1a 2a 1b 2b

- Yu and Vahdat [sosp 2001] show that this ”CLUSTER” order ”dominates” CAUSAL and ALL
- (definition: ’Serialization order S dominates serialization order S’ if, for a given workload and fault-load, the dominating algorithm using S can commit a write W whenever the dominating algorithm using S’ can commit the write.’
- Question: how would you implement this?
  - One answer: break logical clock into “interval” and “acceptNumber”
  - On synchronization, set interval to max(B.interval+1, A.interval+1) and set “acceptNumber” to 0
  - On write, acceptNumber++
  - Sort by (interval, serverID, acceptNumber)

5.3 Bayou: Custom detection and resolution

- Issues:
  - How to detect conflicts – sometimes write/write is OK; sometimes write/read is not OK
  - How to resolve conflicts – abort this one (and all that depend) v. fix this one
- Basics
  - Transactional updates – each “write” can contain multiple updates
  - Application-specific detection and resolution procedures

```
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);
}
```

Figure 2. Processing a Bayou Write Operation

```
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;}
)
```

Figure 3. A Bayou Write Operation

- This allows one write to replace another in the log
- Detects write/read, write/write, etc. conflicts (on arbitrary subsets of database)
- Still get convergence (as long as detection and resolution procedures are deterministic)
- Note: even more general rules possible and sometimes desirable (e.g., "apply all credits during a day before applying any debits for the day")

6 Consistency for clients

- Scenario
  - Suppose you want to access database but don't want to keep a replica
  - Contact one server, start processing
  - That server fails $rightarrow$ switch to a different one
  - What if the new one is inconsistent with the old one?
- How could you tell?
  - Option 1: $V_{new}[k] > V_{old}[k] \forall k$ – too restrictive?
  - Option 2: Talk to any server
  - Option 3: Talk to any server that has at least all of the writes I read
  - Option 4: Talk to any server that has at least all writes I wrote
  - ...
- Bayou: 4 "session" constraints
  - Read-your-writes – "any writes made within a session are visible to reads within the session."

- * if read R follows write W in a session and R is performed at server S at time t, then W is included in DB(S,t)
- Monotonic reads – if a process reads the value of a data item x, any successive read operation on x by that process will return that same value or a more recent valu
- Writes Follow Reads – "ensures that traditional Write/Read dependencies are preserved in the ordering of Writes at all servers...in every copy of the database, Writes made during the session are ordered after any writes whose effects were seen by previous reads in the session"
  - * if read R1 preceeds write W2 in a session, and R1 is performed at server S1 at time t1, then for any server S2, if W2 is in DB(S2) then any W1 in RelevantWrites(S1, t1, R1) is also in DB(S2) and WriteOrder (W1, W2)
  - * Note: this affects users outside of the session
- Monotonic Writes – "writes must follow previous writes within the session"
  - * if write W1 preceeds write W2 in a session, then for any server S2, if W2 in DB(S2) then W1 is also in DB(S2) and WriteOrder(W1, W2)

- Providing these guarantees
  - Session manager keeps
    - * read-set – set of WIDs for the writes that are relevant to session reads
    - * write-set – set of WIDs for those writes performed in the session
  - What to check

Guarantee	session state updated on	Session state checked on
Read Your Writes	Write	Read
Monotonic Reads	Read	Read
Writes Follow Reads	Read	Write
Monotonic Writes	Write	Write

  - Given read-set, write-set, and $V_{server}[]$, what check do you do on reads and writes?
  - Step 1: compress read-set and write-set to version vectors $R_{session}[]$ and $W_{session}[]$
    - * $R_{session}[k]$ = latest acceptNumber of any WID by server k in read-set
    - * $W_{session}[k]$ = latest acceptNumber of any WID by server k in write-set
  - Step 2: What operations on each event?
    - * Read your writes (on a read) server S's $V_S[]$ must *dominate* $W_{session}[k]$
    - * Monotonic reads (on a read) server S's $V_S[]$ must *dominate* $R_{session}[k]$
    - * Writes follow reads (on a write) server S's $V_S[]$ must *dominate* $R_{session}[k]$
    - * Monotonic writes (on a write) server S's $V_S[]$ must *dominate* $R_{session}[k]$

7 Other questions

- Why is client consistency different from server-server consistency
  - vis Coda: primary replicas v. secondary replicas
  - reads follow writes, etc v. causal, etc.
- Compare to p2p systems (even weaker consistency – why?)