CS 138: Epidemics and Bayou
Topics

- Update Dissemination
- Bayou and Weak Consistency
Room Reservation

- Room database maintained on one or more servers
- Clients use Gossip to make updates
  - causal ordering
    - different outcomes at different replicas
  - forced ordering
    - identical outcomes at all replicas
    - what about conflicts?
  - immediate ordering
    - notify client about conflicts right away
Weak Consistency

- Replicas are eventually consistent with one another
Synchronization of Content

- Consensus
  - requires quorum
- Simpler approach
  - disseminate updates to all servers
  - each server determines proper order on its own
Update Dissemination

- Reliable multicast
  - doesn’t scale well
- Hierarchical dissemination
  - via a spanning tree
- Randomly
Anti-entropy works as follows: periodically, a server P randomly chooses a server Q and then the two exchange databases. Three modes of anti-entropy are possible. In push mode, P pushes its database to Q, which adds to its database everything in P that it doesn’t have. In the pull mode, the reverse is done. Push-pull mode works in both directions.

Epidemics

- Established theory of how diseases spread by epidemics
  - adapt it
  - primary difference: infection is good
  - we control the “disease”
We look at the fundamentals of epidemic theory in order to analyze the basic performance of the anti-entropy algorithms. Assume that any server can infect any other server with equal probability. Also assume that $k$ members are already infected and infections occur in rounds: at each round, each server randomly picks another server. For a simple epidemic, sites are either infective or susceptible and exchanges occur in rounds.
Time to Infect

- Number of rounds necessary to infect whole population grows $O(\log n)$
  - for push: $\log_2(n) + \ln(n) + O(1)$ (large $n$)
Push vs. Pull

- Which is better initially?
  - push
- If number of infected members is large; pull is better
  - let \( p \) be the probability of a site’s remaining susceptible after the \( i \)th cycle of anti-entropy
    - for pull, a site remains susceptible after the \( i+1 \)st cycle if it was susceptible after the \( i \)th cycle and it contacted a susceptible site in the \( i+1 \)st cycle.
      \[
      p_{i+1} = p_i^2
      \]
      quickly goes to zero for small \( p_i \)
    - for push, a site remains susceptible after the \( i+1 \)st cycle if it was susceptible after \( i \)th cycle and no infectious site chose to contact it in the \( i+1 \)st cycle
      \[
      p_{i+1} = p_i (1 - \frac{1}{n})^{n(1-p_i)} \approx \frac{p_i}{e}
      \]

Note: for the last equation, the first parenthesized term is the probability of a site choosing to exchange with a site other than the one in question. The exponent is the expected number of infectious sites. Thus we have the probability of a site’s being susceptible after the \( i \)th cycle times the probability that all the infectious sites chose to contact other sites.

If we compare the performance of the pull and push approaches; we observe that (for a very large number of servers) the pull approach converges (i.e., infects the entire population) faster than push does. Push-pull performs similar to pull.
Exchange Updates, Not Databases

- Each node keeps list of recent updates (list of "infections")
  - propagates these to others at each contact
    - receiver adds updates to its list
  - when do items get removed from list?
A form of non-simple epidemic is called **rumor mongering**.
The math is taken from epidemic theory. The final equation tells us how many are still susceptible when there are no more infectious nodes. Thus these will never get the updates.
Rumor Mongering with Anti-Entropy

- Spread updates using rumor mongering
  - some nodes will not get updates
  - but exchanges are cheap
- Run anti-entropy infrequently
  - guarantees all nodes will get updates
  - but exchanges are expensive
Recall that we have no time stamps, so it cannot be determined if an update is new or old.

Removing Data

- Spread deletions just like updates
- What if an old copy of update $x$ arrives after delete $x$ has been processed?
  - $x$ is added again
  - not good ...
Death Certificates

• Solution: hold on to fact that x has been deleted
  – propagate death certificates
  – like updates, but record a void

• How long do nodes hold on to death certificates?
  – a finite period
    - never long enough …
• Retain a few dormant death certificates
  – will be repropagated as death certificates if old update shows up
Papers on Bayou include:


Bayou Overview

- Read anywhere/Update anywhere
- Dissemination via epidemic protocols
- Application-defined conflict detection and resolution
- Client-specified and -checked consistency constraints
Scenario 1

- 138 TAs on a Mars mission
- 167 TAs on a Europa mission
- Each group wants to book CIT 368 for a celebratory meeting when they return
- Each TA maintains own calendar on laptop
- Each group agrees to meet at 4pm on May 29
- Speed of conflict detection limited by speed of light
- In the meantime, they need to make other plans
Coping

- Calendar software grants each a reservation immediately, but marked tentative
- Total order (of reservation requests) determined later
  - one group is first and gets 368
  - the other group does not
    - they provide alternative rooms/times to be used in conflict resolution, which is done automatically
There are many scenarios in which conventional version vector-based
techniques fail to detect the conflicts correctly. In some applications,
concurrent writes to the same object may not conflict as expected.
Furthermore, concurrent writes to different objects may conflict. Bayou
handles such cases by having the application define its own notion of conflict.
Conflict detection is accomplished by using *dependency checks* — each write operation also supplies a dependency check that specifies a simple condition defined over the state of the database. The dependency check also defines what the application expects to see as a result when the condition gets evaluated. If indeed the expected result is observed then Bayou decides that there is no conflict, else it decides that there is a conflict.

If there is a conflict, then a *merge procedure* (i.e., a *conflict resolution procedure*), also supplied by the application, is executed to resolve the conflict. The merge procedure produces a new alternate update that is acceptable by the application and that would not create a conflict with the current database state (i.e., an update for which the dependency check will produce the expected result).
The slide shows the basic format of a Bayou write.
Example: Write and Reconcile in a Shared Calendar

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} }

FOREACH a IN alternates {
    /* check if feasible, produce newupdate */
    if(newupdate = { }) /* no feasible alternate */
        newupdate = { insert, ErrorLog, “Update” }
    return(newupdate)
}

The slide shows an example Bayou write, including the dependency check and the merge procedure.
Coping with Weak Consistency

- Updates will eventually reach all servers and all will be consistent with one another
  - but what happens in the meantime?
Example 1

• Account database replicated on a number of servers
• You update your password on one
• When you try to log in, a different server is contacted and your new password doesn’t work

• Want to contact a server that, at the least, has seen all your writes
Read Your Writes

- A data store is said to provide read-your-writes consistency, if the following condition holds
  - the effect of a write operation by a process on data item x will always be seen by a successive read operation on x by the same process
Example 2

- You look at your email just before boarding a flight
- You refuse to pay for in-flight WiFi
- You arrive at your destination and look at your mail again, but you contact a server that hasn’t seen the updates that the departure-city server had seen

- There’s no point contact a server that’s not at least as up-to-date as the last one contacted
Monotonic Reads

- A data store is said to provide monotonic-read consistency if the following condition holds:
  - if a process reads the value of a data item x any successive read operation on x by that process will always return that same value or a more recent value

Note that “process” here means a client who is connecting to multiple servers.
Example 3

- It’s 1990
- You’re using Usenet, a loosely connected bulletin-board service
- You see responses to posts before you see the actual posts
- This results in confusion, name calling, and worse
- For the sake of world peace, want others not to see responses until after they’ve seen earlier posts
 Writes Follow Reads

- A data store is said to provide writes-follow-reads consistency, if the following holds:
  - a write operation by a process on a data item x following a previous read operation on x by the same process is guaranteed to take place on the same or a more recent value of x than was read
Example 4

• You’re contributing software to an open-source project
• You modify the header files on one server
• You modify the source files on another
• Someone else tries to compile
  – they get the old header file but the new source

• You’re accused of writing bad code …
Monotonic Writes

• In a monotonic-write consistent store, the following condition holds:
  – a write operation by a process on a data item x is completed before any successive write operation on x by the same process
Bayou

- Updates propagated via epidemic protocols
  - some combination of rumor mongering and anti-entropy
- Weak consistency
  - requirements given from the client’s perspective
  - client can choose desired (weak) consistency model
    - (next several slides)
We sketch an approach to an implementation of the constraints. Our first few ideas aren’t entirely practical, but will lead to an approach that is.
If a server can’t be found that satisfies the restriction of the second bullet, then the application must be notified that read-your-writes cannot be maintained.
Monotonic Reads

- Before each read from server S at time t, client must check that
  \[ \text{read set} \subseteq \text{DB}(S, t) \]
- After each read, add to read set the writes that the read depended on
Writes Follow Reads

- After each read, add to read set the writes that the read depended on
- Before each write at time \( t \), client must check that
  \[ \text{read set} \subseteq \text{DB}(S, t) \]
Monotonic Writes

- Before each write at time $t$, client must check that
  \[ \text{write set} \subseteq \text{DB}(S, t) \]
- After each write is accepted by server, client adds write ID to its write set
Anti-Entropy Constraints

- Required of servers to handle *monotonic* writes and writes follow reads
  - new writes accepted by a server from a client are ordered after existing writes
  - during anti-entropy exchanges:
    - if server S1 sends write W2 to server S2, then any W1 ordered before W2 on S1 is also sent to S2
A Practical Implementation

• Implement write IDs as <server, logical time>
• Servers maintain version vectors (essentially vector clocks)
  – \( V_s[x] \) = logical time of most recent (by logical time) write ID received from server \( x \) by server \( S \)
• Clients maintain two session vectors, updated like vector clocks
  – read vector: writes relevant to session reads
  – write vector: session writes
Anti-Entropy Implementation

- Each server keeps log of all writes, ordered by write ID
- On anti-entropy exchange (sender S; receiver R):
  - S sends R all writes unknown to R
    - uses version vectors
Ordering

- Initial write order is tentative
- Designated primary server determines total order of writes: commit sequence number (CSN)
  - write ID = <CSN, server ID, logical time>
    - CSN = $\infty$ initially (while tentative)
  - primary propagates updated write IDs
  - once received, order is permanent (and total)

```
... c_0 c_1 ... c_0 t_0 t_1 t_2 ... t_i t_{i+1}
```

committed ←←→ tentative
Log Truncation

• When can it be done?

This part is fixed and represented by database contents

This part is not fixed: its order can change