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

• At each synchronization period, server P picks a random server Q and exchanges updates

• Three types:
  – *push*: P → Q
  – *pull*: P ← Q
  – *push-pull*: P ↔ Q

• Entire databases are exchanged
  – no timestamp information to determine what’s new
  – optimization: use checksums to see if they’re different
Epidemics

• Established theory of how diseases spread by epidemics
  – adapt it
  – primary difference: infection is good
  – we control the “disease”
Epidemic Theory

• Assume a fixed population of size $n$
• For now assume homogeneous spreading
  – anybody can infect anybody else with equal probability
• Assume $k$ members already infected
• Assume infection occurs in rounds
• In computer terms:
  – a site holding an update it is willing to share is *infective*
  – a site that has not yet received an update is *susceptible*
  – a site that has received an update, but unwilling to share is *removed*
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 $p_i$ 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
      \[ p_{i+1} = p_i (1 - \frac{1}{n})^{n(1-p_i)} \approx \frac{p_i}{e} \]
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?
Rumor Mongering

- Server P randomly selects Q to push updates
- If Q already has seen the updates of P, then P may “lose interest” (become *removed*)
  - ... with probability $1/k$
Rumor Mongering

• Analysis

s: % servers that are susceptible
i: % servers that are infective

differential equations from epidemic theory:

\[
\frac{ds}{dt} = -si \quad \frac{di}{dt} = si - \frac{1}{k} (1 - s)i
\]

for large n:

\[
i(s) = \frac{k + 1}{k} (1 - s) + \frac{1}{k} \log s
\]

i(s) is zero when:

\[
s = e^{-(k+1)(1-s)}
\]

for k=1, 20% will miss the rumor
for k=2, 6% will miss it
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
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
Bayou

Client moves to other location and (transparently) connects to other replica

Replicas need to maintain client-centric consistency

Wide-area network

Distributed and replicated database

Portable computer

Read and write operations
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
Application-specific Conflicts

• Concurrent writes to the same object might not conflict
  – e.g., two updates reserving the same room for different times
• Writes to different objects may conflict
  – e.g., one update for scheduling the projector and the other one the meeting room
• Conventional techniques decide false conflicts or false non-conflicts
• Bayou: Account for application semantics
  – conflict detection with help of application
Conflict Detection & Resolution

- ** Dependency check**
  - a condition over the state of the database
  - included in every “write”
    - together with “expected result”
  - shared-calendar example:
    - condition: Is there another meeting at the same room at the same time?

- **A merge procedure** is used if a conflict is detected
  - shared-calendar example:
    - **resolution**: reschedule to alternate time
  - produces new update
A Bayou “write”

• Processed at each replica:

```c
Bayou_Write(update, dep_check, mergeproc) {
    IF (DB_EVAL(dep_check.query) ≠ dep_check.expected_result)
        resolved_update = EXECUTE(mergeproc);
    ELSE
        resolved_update = update;
    DB_EXEC(resolved_update);
}
```
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)
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
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)
Dealing with the Constraints

• Clients keep track of dependencies
  – read set: set of IDs for writes that are relevant to session reads
  – write set: set of IDs for writes performed in session

• Server S maintains $DB(S, t)$
  – ordered sequence of writes received by server up to time $t$
Read Your Writes

• Whenever write is accepted by a server, client adds write ID to write set
• Before each read from server S at time t, client must check that
  write set $\subseteq DB(S, t)$
Monotonic Reads

• Before each read from server S at time t, client must check that read set $\subseteq DB(S, t)$

• 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)
  – $V_s[x] = \text{logical time of most recent (by logical time) write ID received from server } x \text{ by server } S$
• 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 = ∞ initially (while tentative)
  - primary propagates updated write IDs
  - once received, order is permanent (and total)

\[
\begin{array}{cccccc}
\text{committed} & c_0 & c_1 & \cdots & c_n & t_0 & t_1 & t_2 & \cdots & t_i & t_{i+1} & \text{tentative}
\end{array}
\]
Log Truncation

- When can it be done?

\[
\begin{array}{cccc}
\text{committed} & \cdots & \text{tentative} \\

c_0 & c_1 & \cdots & c_n & t_0 & t_1 & t_2 & \cdots & t_i & t_{i+1}
\end{array}
\]

This part is fixed and represented by database contents

This part is not fixed: its order can change