CS 138: Epidemics and Bayou
Topics

• Client-based consistency
• Epidemic algorithms
• Bayou and Weak Consistency
Room Reservation

- Room database maintained on one or more servers
- Clients Gossip to make updates
  - everything is linearizable
- Problem solved ...
Room Reservation (2)

• 138 TAs schedule room for help session
• 167 TAs also schedule room for help session
• But it’s spring break …
  – 138 TAs are on flight to Rio de Janeiro
    - at least one has copy of reservation DB
  – 167 TAs are on flight to Tahiti
    - at least one has copy of reservation DB
  – can’t detect conflicts en route
    - cheap airfares — no internet on plane
  – Gossip can’t cope
Weak Consistency

- Replicas are eventually consistent with one another
  - assuming no conflicts
Synchronization of Content

• Consensus
  – requires quorum

• Simpler approach
  – each node periodically contacts a random node
    - update it (or vice versa)
  – doesn’t require quorum
  – does it work?
Anti-Entropy

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

• Three types:
  – push: $P \rightarrow Q$
  – pull: $P \leftarrow Q$
  – push-pull: $P \leftrightarrow Q$

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

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
  – if susceptible node encountered, restart rumor mongering
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
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
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
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:

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)
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)
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 (of its not holding)
  – you read your calendar at one server, then go to another and don’t see a recent update that was present at the first server, then go to another server and see it, then go to another and don’t see it …
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

• Example
  – you update a header file in a source-code directory on one server, then go to another and update a source-code file; someone else now tries to compile
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
  – you change your password, then move to another server and your new password hasn’t taken effect
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 \( x \) that was read

• Example:
  – consider a loosely connected bulletin-board service (such as Usenet). You would like it to be the case that responses to queries are seen after the queries at all sites
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 $\subseteq \text{DB}(S, t)$
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
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
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
  \[
  \text{write set} \subseteq \text{DB}(\text{S}, \ t)
  \]

• After each write is accepted by server, client adds write ID to its write set
A Practical Implementation

• Implement write IDs as \(<\text{server}, \text{logical time}\>\)
• 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)
Log Truncation

- When can it be done?

\[ c_0 \quad c_1 \quad \cdots \quad c_n \quad t_0 \quad t_1 \quad t_2 \quad \cdots \quad t_i \quad t_{i+1} \]

- committed
- tentative

This part is fixed and represented by database contents

This part is not fixed: its order can change