CS 138: Replication and Gossip
In what will be our general scenario, we assume we have a collection of objects. These objects are replicated and placed under the control of replica managers. It may be that each replica manager has a copy of each object, or perhaps some objects are on some replica managers but not others. Clients somehow interact with the replica managers, requesting operations to be performed on the objects. We expect to have some form of consistency in the effect of these operations on the objects.

This material is taken from the text, Section 18.3, starting on page 775.
This example, taken from the text (page 776) shows the effect of a simplistic replication scheme in which a replica manager first performs a client’s requested operation, then, sometime later, updates the other RMs. In this case, client 1 first connects to RM B and sets x’s balance to $1. RM B crashes before it updates RM A. Client 1 then connects to RM A and set y’s balance to $2. Client 2 then connects to RM A and queries y’s balance, which is reported as $2. It then queries x’s balance, which is reported as 0 (since RM A never received the update).
Linearizability (also called strict consistency) means that the system behaves essentially as if there is no replication and there is just one replica manager managing all objects. The precise definition is a bit more complicated (next slide).
Linearizability Definition

- Clients perform sequences of operations
  - each operation consists of request, arguments, and result
- A system is linearizable iff
  - for any execution of the system, the operations of all the clients can be put into a sequence such that
    - the sequence could have taken place in a system with only one replica manager
    - the operations in the sequence are partially ordered by the real times of their actual occurrences

Note that the example of slide 3 is not linearizable — it could not have been put into a sequence that could have taken place in a system with just one RM.
Another Example

<table>
<thead>
<tr>
<th>Client 1</th>
<th>Client 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setBalance_{B}(x, 1)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>getBalance_{A}(y)\rightarrow 0</code></td>
</tr>
<tr>
<td></td>
<td><code>getBalance_{A}(x)\rightarrow 0</code></td>
</tr>
<tr>
<td><code>setBalance_{A}(y, 2)</code></td>
<td></td>
</tr>
</tbody>
</table>

This example is also not linearizable: there is no sequence respecting the real-time ordering of operations that could have taken place on a system with a single RM. But if we relax the restriction that we must respect the real-time ordering and respect just causal orderings (in this case, just the ordering on each client), then this does become plausible. It makes sense if both of client 2’s operations occurred before client 1’s operations.
Sequential Consistency

- Clients perform sequences of operations
  - each operation consists of request, arguments, and result
- A system is **sequential consistent** iff
  - for any execution of the system, the operations of all the clients can be put into a sequence such that
    - the sequence could have taken place in a system with only one replica manager
    - the operations in the sequence are partially ordered by their order in each client
In passive replication, one RM is the primary and the others are backups. Note that the system is linearizable.
Passive Replication Sequence

1) Request: client issues request to primary
2) Coordination: primary takes each request atomically, in order received
3) Execution: primary executes request and stores response
4) Agreement: if request is an update, primary sends request to backups
5) Response: once all backups respond, primary sends response to client
What should happen if the primary fails?
Raft to the Rescue

- New primary is elected
- Clients communicate with it
In active replication, each client uses totally ordered, reliable multicast to send its operations to each RM. The client waits for responses from all before it performs its next operation. Though the system is not linearizable, it is sequential consistent.
Now we discuss the Gossip system, described in “Providing High Availability Using Lazy Replication,” by R. Ladin et al., appearing in ACM Transactions on Computer Systems, Vol. 10, No. 4, November 1992. A copy can be found at http://portal.acm.org/citation.cfm?id=138877. This material is also covered in Coulouris, Dollimore, Kindberg, and Blair in Section 18.4.1 (starting on page 782). The latter presentation is considerably easier to follow than the former.
Scenario

- Distribution-list service
  - multiple, geographically distributed servers
  - replicated database
  - operations
    - post a message
    - add a user
    - ostracize a user
Note that neither passive replication nor active replication is satisfactory.
In this case, the “relaxed” approach of maintaining causal orders doesn’t work. A total order is necessary. The down side of this is that doing so may require more overhead, more elapsed time, or both.
Ostracizing a User

- Cody defects to CS1951E
- No longer trusted to receive confidential CS138tas email
- Must be removed from list immediately (if not sooner) at all servers
  - urgent!
Desired Features

- Causal ordering
  - needed for updates
- Forced ordering
  - both causal and total order
  - needed for adding JCarberry
- Immediate ordering
  - forced ordering with minimal delay
  - needed for ostracizing Cody
• Clients normally communicate with one RM
  – but it might be busy
    - communicate with another
    - communicate with many
Rough Outline
(Causal Ordering)

• Query
  – client sends request to one or more RMs
  – respond when causally possible
• Update
  – client sends request to one or more RMs
  – update and respond when causally possible
  – propagate changes to others via “gossip” messages
    - not specified how this is done
    - allows many possibilities
Unless otherwise indicated, timestamps are from vector clocks. Note that the causal constraints are rather weak: it’s important only that the query result be causally consistent with what the client knows, and not with events unknown to the client. This allows the distributed system to function even if partitioned into multiple pieces.
Update (1)

- Client sends
  - request (u.op)
  - causal dependencies
    - u.prev = client.ts
Update (2)

• RM i responds
  – receives update
    - assigns timestamp
      • \( \text{rm}_i.\text{replica.ts}[i] += 1 \)
      • \( \text{TS} = \text{u.prev}; \text{TS}[i] = \text{rm}_i.\text{replica.ts}[i] \)
    - puts in log
      • \( <u, i, \text{TS}> \) (update, node, timestamp)
    - returns TS
    - when \( \text{u.prev} \leq \text{rm}_i.\text{val.ts} \)
      • updates val (by applying \( u.\text{op} \))
      • \( \text{rm}_i.\text{val.ts} = \text{merge(rm}_i.\text{val.ts}, \text{TS}) \)

Note that the client application can continue on, after sending the update to the client front-end. It’s the front-end that waits for the TS result.
Update (3)

- Client
  - updates its own timestamp
    - client.ts = merge (client.ts, TS)
Gossipping

- RM a initiates gossip
  - sends to RM b:
    - contents of log (rmₐ.log)
    - replica timestamp (rmₐ.replica.ts)
- RM b receives gossip
  - merges rmₐ.log into rmₜ.log
  - rmₜ.replica.ts = merge(rmₐ.replica.ts, rmₜ.replica.ts)
  - while there exists request r in rmₜ.log such that
    r.u.prev ≤ rmₜ.val.ts && r.processed == false
      - r.processed = true
      - update val (by applying r.u.op)
      - rmₜ.val.ts = merge(rmₜ.val.ts, r.TS)
Forced Updates (1)

- Need a causal order that's also total
  - all clients go through same RM

- Diagram showing App, Client, Primary Replica Manager, Backup Replica Manager, and Distributed Service.
Forced Updates (2)

- What if primary crashes?
  - elect new primary

- Diagram:
  - App
  - front end
  - Client
  - Primary Replica Manager
  - Backup Replica Manager
  - Distributed Service
Immediate Updates (1)

- Primary requests logs and replica timestamps
  - backups respond and stop processing queries
  - updates are accepted but not executed
Immediate Updates (2)

- Backups respond with logs and timestamps
  - primary stops processing queries and updates
  - processes logs and timestamps
Immediate Updates (3)

- Primary sends back updated log records
  - backups process immediately
Immediate Updates (4)

- Backups acknowledge updates
Immediate Updates (5)

- After half the backups respond, primary commits and responds to client
  - half the backups + primary = majority
Problem?

- What if client sends update request to multiple RMs?
  - multiple copies of the request are propagated
  - all are executed
  - probably aren’t idempotent
Solution

- Client assigns unique ID (CID) to each request
- RMs keep track of CIDs of completed requests
  - completed requests go to invalid CIDs list
  - check list before doing a request
    - don’t perform requests that have already been performed

CID = call identifier.
Another Problem?

- Won’t logs and invalid CIDs lists grow without bound?
  - yes ...
Bounding Logs (1)

• Each log entry \( r \) must be kept on RM \( i \) until it is present on all RMs
  – so that gossip from \( i \) will inform other RMs
• \( r\text{-node} \) is the node that created the log entry \( r \)
• \( r\text{-ts} \) is the vector timestamp assigned to the log entry by \( r\text{-node} \)
• \( r\text{-ts}[r\text{-node}] \) is the logical time on \( r\text{-node} \) when the entry was created
• \( r \) may be removed from \( i \)'s log when:
  – \( \forall k: r\text{-ts}[r\text{-node}] \leq rm_k\text{-replica.ts}[r\text{-node}] \)
Bounding Logs (2)

- How does RM i know \( \text{rm}_k \cdot \text{replica.ts}[\text{r.node}] \) ?
  - gossip messages contain replica timestamps
    - timestamps on logs
  - each RM keeps a table of the most recent timestamps obtained from all other RMs
    - \( \text{rm}_i \cdot \text{ts\_table} \)
    - \( \forall k \, \text{rm}_i \cdot \text{ts\_table}[k] \leq \text{rm}_k \cdot \text{replica.ts} \)
  - RM i may remove log entry r when:
    - \( \forall k: \, \text{r.ts}[\text{r.node}] \leq \text{rm}_i \cdot \text{ts\_table}[k][\text{r.node}] \)
Trimming the Invalid CID List

- When can an entry be removed?
  - when it will never be received again
- Assuming perfect communication, how can you tell?
  - you can’t: client’s front-end might send same update to multiple RMs
  - what’s more, communication might not be perfect
- More machinery needed ...
More Machinery …

• Client front-end puts (real-time) timestamps on all update requests

• After successful transmission of last transmission of an update, it sends “that’s all” (TA) message to at least one RM
  – contains CID of update and (real-time) timestamp
    - timestamp of TA is later than that of updates
  – RM puts it in log (and includes it in gossips)

• Assume maximum real time required for any RM i to notify RM j of new info via gossip is δ
  – takes into account clock skew, etc.
Yet More Machinery ...

- General idea
  - all equivalent update messages terminated by TA a must be received by $a.timestep + \delta$

- Details
  - discard CID c from invalid CID list if its TA is in log and no update records for c in log
    - all RMs have seen c
  - ignore updates if m.time + \delta < replica’s local time
  - discard TA a from log if it appears in all logs and $a.timestep + \delta <$ replica’s local time
    - no other instances of updates terminated by a are still circulating