CS 138: Replication and Gossip
Scenario

Client

Replica Manager

Replica Manager

Client

Replica Manager

Client
Example

<table>
<thead>
<tr>
<th>Client 1</th>
<th>Client 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>setBalance(_B)(x, 1)</td>
<td></td>
</tr>
<tr>
<td>setBalance(_A)(y, 2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getBalance(_A)(y)→2</td>
</tr>
<tr>
<td></td>
<td>getBalance(_A)(x)→0</td>
</tr>
</tbody>
</table>
Linearizability

![Diagram showing the concept of linearizability with a network of clients and replica managers]
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
## 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><code>setBalance_A(y, 2)</code></td>
<td><code>getBalance_A(x)→0</code></td>
</tr>
<tr>
<td><code>getBalance_A(y)→0</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>getBalance_A(x)→0</code></td>
</tr>
</tbody>
</table>
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
Passive Replication
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
Passive Replication
Raft to the Rescue

- New primary is elected
- Clients communicate with it
Active Replication
Gossip
Scenario

• Distribution-list service
  – multiple, geographically distributed servers
  – replicated database
  – operations
    - post a message
    - add a user
    - ostracize a user
Posting a Message

- Client posts a message
  - contacts nearest server
  - wants quick confirmation
- However ...
  - server could crash at any moment
    - message should be replicated at other servers
    - causality constraints must be satisfied
  - but these are pretty weak
Adding a User

- Two different people want to register for the CS138 list as “JCarberry”
- One asks Jeff, the other asks Jon
- Jeff and Jon each attempt to add his person
  - simultaneously
- First one to reach the server succeeds, second one fails
- But there are multiple servers
  - each contacts a different server
- Want the same total order at all servers
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
Query Model

Client

Distributed Service

Replica Manager

Value

TS

Replica Manager

TS

App

front end

App

front end

Client
Query

• Client sends query q
  – request (q.op)
  – causal dependencies
    - q.prev = client.ts

• RM i responds
  – receives query
  – holds it until
    - q.prev ≤ rm_i.val.ts
  – returns value and timestamp (rm.val.ts)

• Client
  – updates its own timestamp
    - client.ts = merge (client.ts, rm.val.ts)
Update Model

Distributed Service

Replica Manager

Replica Manager

Replica Timestamp

TS

Value

Log

TS

App

front end

Client

App

front end

Client
Update (1)

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

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

• RM b receives gossip
  – merges rm\textsubscript{a}.log into rm\textsubscript{b}.log
  – rm\textsubscript{b}.replica.ts = merge(rm\textsubscript{a}.replica.ts, rm\textsubscript{b}.replica.ts)
  – while there exists request r in rm\textsubscript{b}.log such that
    r.u.prev \leq rm\textsubscript{b}.val.ts && r.processed == false
    • r.processed = true
    • update val (by applying r.u.op)
    • rm\textsubscript{b}.val.ts = merge(rm\textsubscript{b}.val.ts, r.TS)
Forced Updates (1)

- Need a causal order that’s also total
  - all clients go through same RM
Forced Updates (2)

• What if primary crashes?
  – elect new primary
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
Updated Update Model

Distributed Service

Client

App
front end
TS

Replica Timestamp

Invalid CID

TS
Value
Log

Replica Manager

Replica Manager
Another Problem?

• Won’t logs and invalid CID 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
- $r$.node is the node that created the log entry $r$
- $r$.ts is the vector timestamp assigned to the log entry by $r$.node
- $r$.ts[$r$.node] is the logical time on $r$.node when the entry was created
- $r$ may be removed from $i$’s log when:
  - $\forall k: r$.ts[$r$.node] $\leq$ $r$m$_k$.replica.ts[$r$.node]
Bounding Logs (2)

- How does RM i know rmₖ.replica.ts[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
    - rmᵢ.ts_table
    - ∄k rmᵢ.ts_table[k] ≤ rmₖ.replica.ts
  - RM i may remove log entry r when:
    - ∄k: r.ts[r.node] ≤ rmᵢ.ts_table[k][r.node]
Trimming the Invalid CID\textquotesingle s 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 \( \delta \)
  - takes into account clock skew, etc.
Yet More Machinery …

• General idea
  – all equivalent update messages terminated by TA a must be received by $a.t$ + $\delta$

• 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 $m.t$ + $\delta$ < replica’s local time
  – discard TA $a$ from log if it appears in all logs and $a.t$ + $\delta$ < replica’s local time
    - no other instances of updates terminated by $a$ are still circulating