CS 138: Replication and Gossip
Today

- Different consistency models
  - Linearizability, sequential

- Different replication models
  - Active, passive, lazy

- Gossip Protocol
  - Models: Immediate, forced, causal
  - Challenges: bounding logs, trimming CIDS
Linearizability

Client

Replica Manager

Replica Manager

Replica Manager

Client
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
  – Vector clocks is not sufficient – need total ordering
### 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>getBalance(_A)(y) → 2</td>
<td></td>
</tr>
<tr>
<td>getBalance(_A)(x) → 0</td>
<td></td>
</tr>
</tbody>
</table>

B crashes
Another Example

<table>
<thead>
<tr>
<th>Client 1</th>
<th>Client 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>setBalance_B(x, 1)</td>
<td>getBalance_A(y) → 0</td>
</tr>
<tr>
<td></td>
<td>getBalance_A(x) → 0</td>
</tr>
<tr>
<td>setBalance_A(y, 2)</td>
<td></td>
</tr>
</tbody>
</table>

B crashes
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

• You can use logical clocks here
Passive Replication

- Client
- Replica Manager
- Replica Manager
- Replica Manager
- Client
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

Client

Replica Manager

Replica Manager

Replica Manager

Client
Raft to the Rescue

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

Diagram showing a network of clients and replica managers, where each client is connected to replica managers, and the replica managers are interconnected.
Active Replication Sequence

1) **Request**: client issues request to all
2) **Coordination**: multicast ensure deliver to all
3) **Execution**: each replica executes
4) **Agreement**: no agreement needed
5) **Response**: each replica responds to client
Passive Replication
• Total ordering
• Requires Raft like protocol

Active Replication
• Casual ordering
• Requires total ordered multicast
Gossip

From “Providing High Availability Using Lazy Replication”
Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat
Gossip

- A special form of multicast
  - Each node periodically sends updates to a subset of nodes
  - The subset is chosen at random each time
Provides Different Models of Consistency

- Causal ordering
- Forced ordering
  - both causal and total order
- Immediate ordering
  - forced ordering with minimal delay
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
Example

• I send a message to Rodrigo
  – via the Providence replica
• Rodrigo reads message
  – via Providence replica
• Rodrigo sends response
  – via Rio de Janeiro replica
• I read response
  – still in Providence (sigh)

• Louisa sends Atty message
  – via San Francisco replica
• Atty reads message
  – via Miami replica
• Atty sends response
  – via Grand Cayman replica
• Louisa reads response
  – via Honolulu replica
Adding a User

- Two different people want to register for the CS138 list as “JCarberry”
- One asks Max, the other asks Haris
- Max and Haris each attempt to add their 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

• Carlos defects to CS166
• No longer trusted to receive confidential CS138tas email
• Must be removed from list immediately (if not sooner) at all servers
  – urgent!
Provides Different Models of Consistency

• Causal ordering
  – needed for exchanging messages

• Forced ordering
  – both causal and total order
  – needed for adding JCarberry

• Immediate ordering
  – forced ordering with minimal delay
  – needed for ostracizing Carlos
• Clients normally communicate with one RM
  – but it might be busy
    - communicate with another
    - communicate with many
Rough Outline
(Causal Ordering)

• Query (getting a value)
  – client sends request to one or more RMs
  – respond when causally possible

• Update (setting a value)
  – 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

Distributed Service

Replica Manager

Replica Manager

TS

Value

Replica Manager

Client

App

front end

TS
Query

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

• RM i responds
  – receives query
  – holds it until it catch up, i.e., it’s timestamp is up-to-date
    - 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

TS
Value
Log
Replica Manager

Replica Timestamp

App

front end

TS

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
      - $\text{rm}_i.$replica.ts[i] += 1
      - $\text{TS} = \text{u.prev}; \text{TS}[i] = \text{rm}_i.$replica.ts[i]
    - puts in log
      - $(\text{u}, \text{i}, \text{TS})$ (update, node, timestamp)
    - returns TS
    - when $\text{u.prev} \leq \text{rm}_i.$val.ts
      - updates val (by applying $\text{u.op}$)
      - $\text{rm}_i.$val.ts = merge($\text{rm}_i.$val.ts, $\text{TS}$)
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_a.log)
    - replica timestamp (rm_a.replica.ts)

• RM b receives gossip
  – merges rm_a.log into rm_b.log
  – rm_b.replica.ts = merge(rm_a.replica.ts, rm_b.replica.ts)
  – while there exists request r in rm_b.log such that
    r.u.prev ≤ rm_b.val.ts && r.processed == false
      • r.processed = true
      • update val (by applying r.u.op)
      • rm_b.val.ts = merge(rm_b.val.ts, r.TS)
A special form of multicast
  - Each node periodically sends updates to a subset of nodes
  - The subset is chosen at random each time

How do we get forced updates?
  - Recall we want total ordering
Forced Updates (1)

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

![Diagram]

- App
- front end
- Client
- App
- front end
- Client
- Primary Replica Manager
- Distributed Service
- Backup Replica Manager
- Backup Replica Manager
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 assigns timestamp to update
- Primary sends update to backups

![Diagram showing the components of a distributed service with a primary manager and backup replica managers connected to the primary and client through an app front end.]
Immediate Updates (4)

- Backups acknowledge updates
Immediate Updates (5)

- After half the backups respond, primary commits (updates val) and responds to client
  - half the backups + primary = majority
Immediate Updates (6)

- Primary sends log to each backup (gossips)