Lecture 19: Fault tolerance

Overview
- Replication of data
  - reasons for and benefits
- Fault tolerance
  - consistency of data
    - linearity
    - sequential consistency
  - passive replication
  - active replication
- Availability
  - gossip architecture

Replication of data
- Definition
  - maintaining copies of data on multiple computers
- Requirements
  - replication transparency: clients unaware of existence of multiple copies
  - consistency
- Benefits
  - performance enhancement (sharing workload, data closer to clients)
  - increased availability (independent server failures)
  - fault tolerance (server crashes, Byzantine failures)

Assumptions
- The model
  - asynchronous
  - multicast communication
  - process crash failures only, no network partitions
  - failure detector
- System architecture
  - front end: client interface (request handling)
  - service: several replica managers
    - must be recoverable
    - updates atomic (indivisible)
    - different managers hold copies of different objects

Architecture model for replicated data

Fault-tolerant services
- Function
  - provide a service with correct behaviour despite up to $f$ process failures, as if there was a single copy of data
- For example
  - system responds despite server crash
  - all transactions are implemented and in correct order
  - updates are propagated to all copies
- Ensuring consistency of data
  - linearity
  - sequential consistency
Example

- Fault-tolerant banking system
  - two bank accounts x & y, initially 0
  - two replica managers A & B, updates propagated

Scenario

<table>
<thead>
<tr>
<th>Time</th>
<th>Client 1</th>
<th>Client 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>B: credit(x,1)</td>
<td></td>
</tr>
<tr>
<td>(2)</td>
<td>B fails</td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td>A: credit(y,2)</td>
<td></td>
</tr>
<tr>
<td>(4)</td>
<td>A: balance(y) → 2</td>
<td></td>
</tr>
<tr>
<td>(5)</td>
<td>A: balance(x) → 0</td>
<td>(B credit not arrived)</td>
</tr>
</tbody>
</table>

But… not correct with respect to a single copy
- balance(x) should be 1, since credit(x,1) before credit(y,2)

Linearizability

- Definition:
  for any execution, there is some interleaving of client operations such that
  - the result of operations is correct with respect to a single copy
  - the order of operations is consistent with real times at which they actually occurred during execution

Back to example…
- not linearizable (already linearly ordered in time, so cannot swap)

Problem
- linearizability strong requirement
- difficult to synchronise clocks in asynchronous systems

Sequential consistency

- Definition:
  for any execution, there is some interleaving of client operations such that
  - the result of operations is correct wrt a single copy
  - the order of operations is consistent with the order in which each client executed them (shuffle)

Example: sequentially consistent but not linearizable

<table>
<thead>
<tr>
<th>Time</th>
<th>Client 1</th>
<th>Client 2</th>
<th>Shuffle</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>B: credit(x,1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2)</td>
<td>A: credit(y,2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td>A: balance(y) → 0</td>
<td>(1)</td>
<td></td>
</tr>
<tr>
<td>(4)</td>
<td>A: balance(x) → 0</td>
<td>(2)</td>
<td></td>
</tr>
</tbody>
</table>

Passive replication

- single primary manager, one or more secondary backups
- only primary communicates with front end
- if primary fails, one of the backups promoted

Data consistency
- all requests filtered and sequenced at primary, backups act as slaves
- if clients are allowed to read directly from backups, then obtain sequential consistency

Failures
- if primary fails, replace by one of the backups (OK if surviving managers agree on operations up to that point)
- to tolerate process crashes, need f+1 replica managers
- cannot tolerate Byzantine failures

How it works
- request issued to primary, with unique id
- primary receives requests in order, checks unique ids in case already executed
- each request processed by primary atomically
- if the request is an update, the updated state and unique id is forwarded to all backups which send acknowledgements
- primary sends response to front end when ack received

Assuming primary correct
- all requests processed in order of receipt by primary
- implements linearizability
Active replication

- replica managers work as a group
- front end multicasts requests to group
- managers process requests independently but identically and reply

How it works

- request multicast to group using multicast, with unique id
- requests delivered in the same (total) order to all managers
- every manager processes the request (same requests, same order)
- each replica manager sends response to front end
- front end collects responses from multicast

Thus

- all requests processed independently, in order of receipt
- implements sequential consistency

Consistency

- cannot achieve linearizability (real time order of client requests may differ from the actual total order of processing by managers)
- variants with read-only requests to individual managers also possible

Failures

- replica manager crashes have little effect on performance, front end can count responses
- can sometimes tolerate Byzantine failures [Canetti & Rabin]: need 2f+1 replica managers and digital signatures

Availability

- highly-available services:
  - clients can access the service with reasonable response time, for as much time as possible
  - sequential consistency of data not a priority
    - clients can temporarily put up with stale data
    - updates lazy, as opposed to eager:
      - acceptable level of service while minimising client delay
- examples
  - shared diary service
  - electronic bulletin board

The gossip architecture

- idea
  - replicate data close to the points where groups of clients need it
  - replica managers exchange ‘gossip’ in the background, from time to time
- operations
  - queries (read-only), with timestamps
  - updates (modify, but do not read)
- consistency
  - process updates in causality order, delay queries
  - weaker than sequential consistency

Query & update in a gossip service
Gossip service

- How it works
  - front end sends request to a single replica manager at a time, with timestamp
  - if query, front end/client blocks waiting for reply
  - if update, return to client immediately; updates propagated in the background
  - request processed according to ordering constraints
  - replica managers update each other by sending gossip messages
- Timestamps
  - uses vector timestamps (arrays of logical clocks, e.g. (2,2,5), one per manager)

Replica manager state

- Value
  - value of the state maintained by replica manager
- Value timestamp
  - vector timestamp that represents the updates that are reflected in the value
- Update log
  - logs all update operations, until they become stable
- Replica timestamp
  - represents updates that have been accepted and placed in log
- Executed operation table
  - list of executed updates, to prevent executing twice
- Timestamp table
  - timestamps that arrive in gossip messages

Gossip service

- Operation ids
  - vector timestamps, linearly ordered
- Query
  - based on query and value timestamps decides if all earlier updates received; holds query until then & returns value after execution
- Causal Update
  - checks if not already executed, records it in log; if stable executes it and places it in executed table, and otherwise waits for gossip
- Gossip
  - brings table up-to-date

Summary of gossip architecture

- Purpose
  - high availability at a cost of weaker consistency
- But...
  - distributed design
  - a lot of internal communication (gossip traffic!)
  - cannot be used in real-time situations (too lazy!)
  - does not scale well to large numbers of replica managers (vector timestamps large)
- Simplified design
  - can make most replicas read-only, reduce traffic

Summary

- Fault tolerance
  - passive: primary plus backups, linearizability
  - active: group plus multicast, sequential consistency
  - passive tolerates only crash failures (if f+1 managers)
  - active can tolerate Byzantine failures (need 2f+1 managers)
- Availability
  - gossip architecture
  - weaker form of sequential consistency
  - lazy approach to updates (when needed)