CSCI 1380 : Day 16 : lazy replication
Last week

Lazy replication (passive replication)

This week

Lazy replication (Gossip, Vector clocks, Motivating)

Practical Consensus (practical active/passive replication)
Passive

Active

Active, all apps are
converged

Passive, all apps are
converged

Key issues: lots of messages
& high overhead

Active network latency = 40 - 150ms

Passive network latency = 4ms

Data centers/clusters

Requires code to be deterministic

Code can be non-deterministic

Wide area network (Internet)

Geo distributed database storage

Write area network (Intravnet)
Application (Lazy replication / Causal ordering)

1. read
2. write (read data)

App

1. email/messages
2. forums
3. social network

Causal relationship: you can only respond to what you need or what you received.
The Gossip Protocol is designed to efficiently disseminate information among nodes in a network. The protocol is characterized by its fixed number of nodes and the associated complexity in terms of message passing.

- **Complexity Analysis:**
  - The number of messages sent is $O(N(N-1))$, where $N$ is the number of nodes.
  - This complexity is crucial in understanding the scalability of the protocol.

- **Protocol Mechanics:**
  - Each node periodically picks a random node to gossip with.
  - Updates are sent to the selected node.
  - Sleep periods are introduced to manage the frequency of gossips, ensuring efficient propagation of information.

- **Graphical Representation:**
  - A graph shows the network topology, with nodes and edges indicating the communication links.
  - Each node represents a participant in the Gossip Protocol, and the edges signify the exchange of information.

- **Phase Transition:**
  - The protocol can be divided into phases, where each phase is characterized by a different mode of operation.
  - The number of epochs or rounds is a function of these phases, summing up to a phased exponential pattern.

By understanding these aspects, one can appreciate the efficiency and robustness of the Gossip Protocol in distributed systems.
query → get
update → set

1. FE picks & interacts with one RM before you update; you must first read
   - you get a VC from the RM
2. when you update then you provide VC with the key & value
3. A read (or query) in Lazy RM returns Value & VC.

Update in Lazy RM:
1. update VC in request (increment appropriate index)
2. store the req in local KV store
3. responds to FE with the new VC

reqs/reqs/messages
1. you need to first/ receive
2. before you can respond
1. get KV + VC from local store
2. compare the VC in req & the VC from local store
   
   if req.VC > local.VC
   wait for gossip msgs
   
   else return value + new VC

As long as one BM is alive ⇒ you can update always
Update replication with Gossip

Gossip msg in lazy replication include "update logs"

Parallel but not identical to AppendEntries in Raft

Because

1. Gossip / randomness factor
2. Updates return immediately with "Ok"

Data loss can happen
Ordering

Linearizability = Total ordering + real time + FIFO

Lazy ! = linearizable ordering

① some updates can get lost from
② edge case = conflicting updates
③ VC to determine ordering
different orderings can happen to
unrelated events

Sequential = Total + FIFO
fast because of minimal/no agree

slow because you need notes to agree

weak

Eventually consistent (no guarantee)

Eventually consistent (vector clocks)

Causal+

Lazy replication

Reads has key level agreement

Eventually given sufficient time & no changes to system the nodes will agree

Is passive replication

Active replication

All nodes agree

Linearizable

Sequential

Strong
Summary

Gossip

Lazy Replication

* Motivating examples

* Query (get) requests

* Update (set) requests

* Propagating update log btw RM's

* Availability Guarantees

Ordering & Consistency

(waits until RM's local VC is as up to date as VC in get)

(returns instantaneously with new VC)