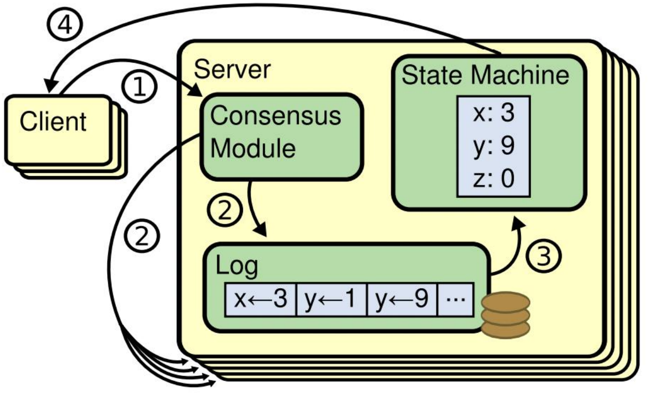
Replicated Database: cRaft++

CS 739 Project 2

Selvaraj Anandaraj, Deep Jiten Machchhar, Vishnu Ramadas

cRaft++ : A Raft based Replicated Database

- Database with Get/Put Key-Value support
- Have a strong leader at any instant
- Implements leader election
- Strongly consistent
- Persistent state machine
- Ensures safety and availability and understandability



cRaft++ : A Raft based Replicated Database

Client

- Uses a returned active leader to access next time
- Does a round-robin try on hitting a failed server

Server

- Maintains the key-value store and log persistently
- Detects leader failures and initiates election automatically
- Modularized C++ implementation utilizing its features

cRaft++ : A Raft based Replicated Database

What's new?

- Multi-threaded execution of RPC's
- Pipelining log entry execution with RPC response
- Log cache on leaders to reduce disk access

Test cases?

- Ensure log consistency checks
- Ensure the election and commit rule
- Ensure log rollback and pruning

Multi-Threaded Execution of RPC’s

- cRaft++ uses RPCs for server-server log replication, heartbeats and elections
- No concept of sequential RPCs across servers
- Each fellow server has its own thread for each RPC type
- Dependencies for strong consistency distilled for maximum multi-threading
- Have self-contained *master-subordinate* threads orchestrating each other
- Threads smart enough to kill itself based on state changes

Pipelining Execution with RPC Response

- cRaft++ overlaps RPC response with execution on followers
- Leader gets instant *ack* on a consistent log append
- Helps overlap computation with communication
- Increases compute and network utilization
- Benefit from the fact that followers state machine execution needed for client

Log Cache

- cRaft++ uses a volatile log for intermediate accesses
- Volatile log is a system-maintained SW cache to avoid disk accesses
- Motivation: Need to access log for execution, consistency checks, log rollback, etc.
- Would be smart to make this log in memory until needed
- Lifetime of a volatile log entry is from Log Append to Entry to execution (which is optimistically enough!)

Test Case 1:

Check Leader Election, Replication and Election Rule

- Bring all servers up, wait for a leader!
- Transact to the leader to build and replicate log
- Crash leader, wait for a new leader, resurrect the dead
- Transact to the new leader
- Check for synchronized log replication

- Demo

Test Case 2:

Check Log Consistency Check and Log Bring-up

- Bring all servers up, wait for a leader!
- Transact to the leader to build and replicate log
- Crash follower
- Transact to the new leader
- Bring back the follower, log needs to be built again!
- Check for synchronized log replication

- Demo

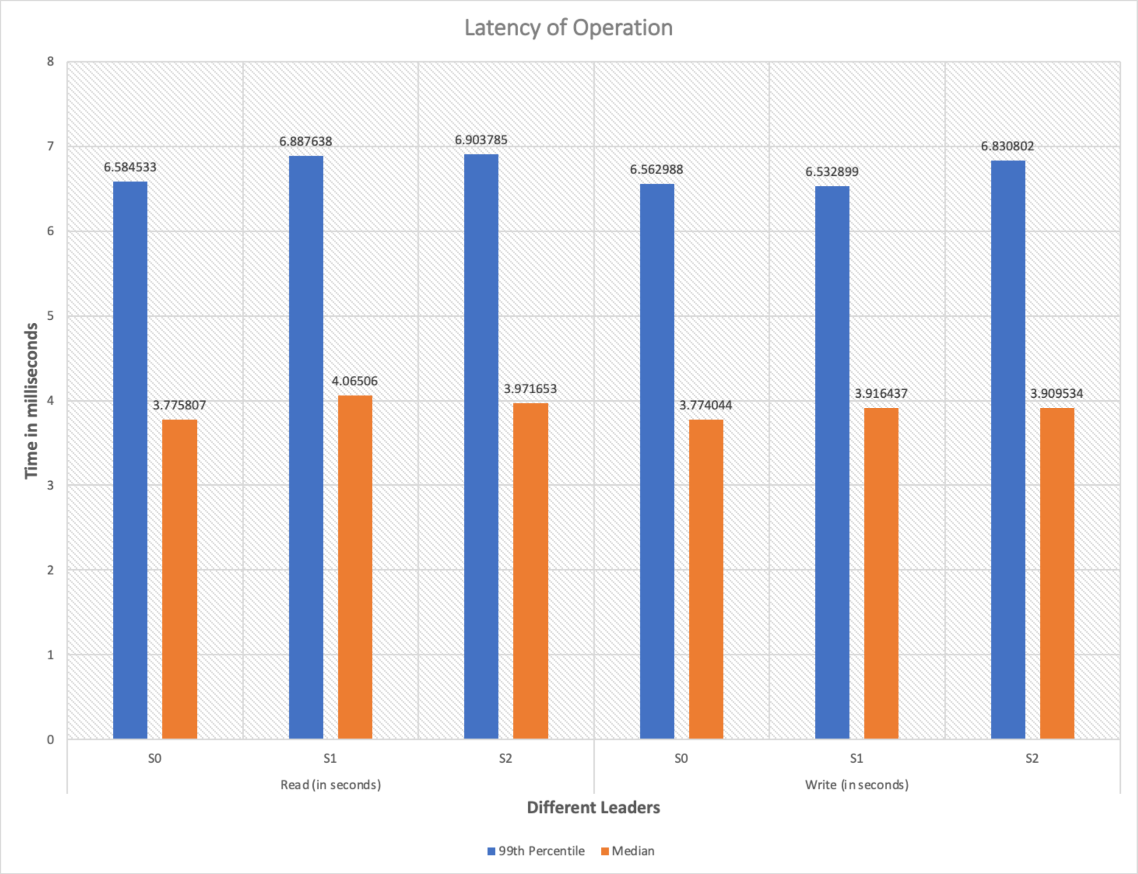
Test Case 3:

Check Log Commit Rule

- Bring all servers up, wait for a leader!
- Transact to the leader to build and replicate log (TX 1)
- Crash leader (Server A)
- Transact to the new leader (Server B, TX 2)
- Crash the new leader (Server B)
- Resurrect the former leader which becomes follower (Server A)
- At this point, commit rule will stop TX2 to commit in S A
- Transact to the new leader (Server C), now TX2 will commit in A

- Demo

Results: Server-wise Read/Write Latency

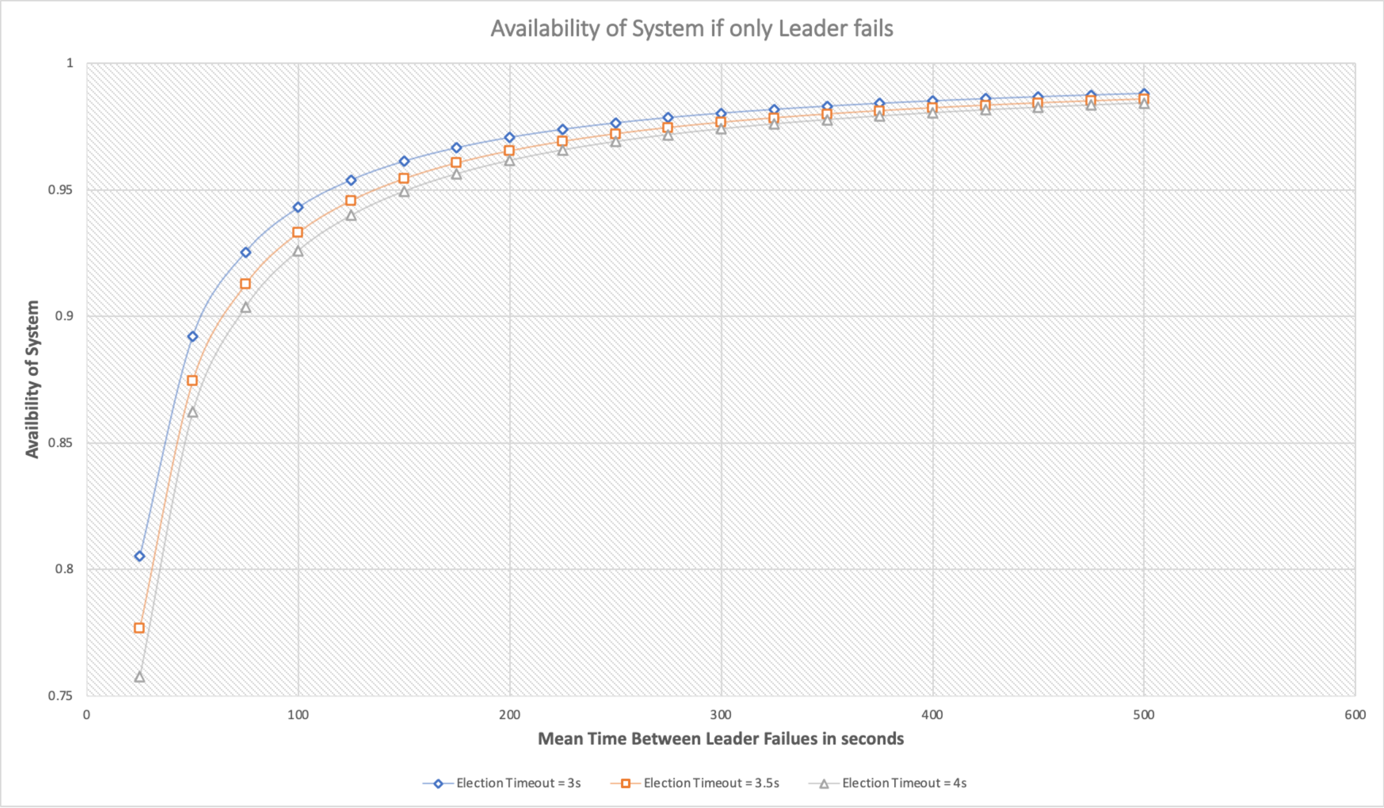


Results

- Execution time on follower servers is less than 2 milliseconds; This is the time we save through overlapping compute with communication

- Heartbeat should be less than Election Timeout by 50 millisecond to work seamlessly.
  Hypothesis: We believe more leader elections to take place if Election Timeout and Heartbeat timeout are close.

Results: System Availability



Thank you!