Replicated Database: cRaft++

Project group 9:
Deep Jiten Machchar - machchhar@wisc.edu
Vishnu Ramadas - vramadas@wisc.edu
Selvaraj Anandaraj - anandaraj@wisc.edu

1 Introduction and Design

We have implemented a replicated database using the raft consensus algorithm [1]. We used C++ as the language of our choose and avail of gRPC to communicate between servers.
On the server, we maintain a custom key-value store implementation and log client commands persistently. Our implementation forks threads for every RPC request in the order they arrive so as to make communication concurrent and self-contained. These threads die when their assigned job is done. A master thread takes care of synchronizing between all request execution threads. We also overlap execution time with communication on followers by spawing them on different threads. Any append entry request on follower first checks if there is a consensus and then returns immediately before executing while the execution thread independently executes committed entries. We maintain a volatile version of the log in main-memory that contains un-executed entries and delete them once they are executed. This software cache helps us reduce both disk accesses and the memory footprint. On the client side, any request to a server returns the leader ID. Therefore, if the client accesses the wrong leader, it can retry the same request to the correct server. In case the client-server link is broken, the client retries to a different server in a round-robin fashion until it gets a working link, and then it gets redirected to any future leader.

2 Results

To evaluate our implementation, we used a system with three server nodes and one client node on the Utah Cloudlab Cluster. Our system support only one node failure since more nodes failing will result in no election or consensus majorities. The metrics we measured are the latencies of client get and put operations, client request throughputs, and the availability of the system. The results of our experiments are captured below.

- Latency -
  To calculate latency of requests, we time 10,000 get and put requests from the client to each node (when they were leader)

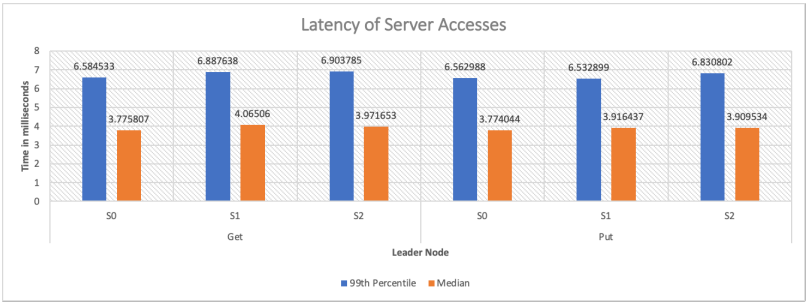

Figure 1: Read/write latencies for all servers

- System Throughput -
  Since the client requests are synchronous in nature, we calculate the system throughput using the latency numbers

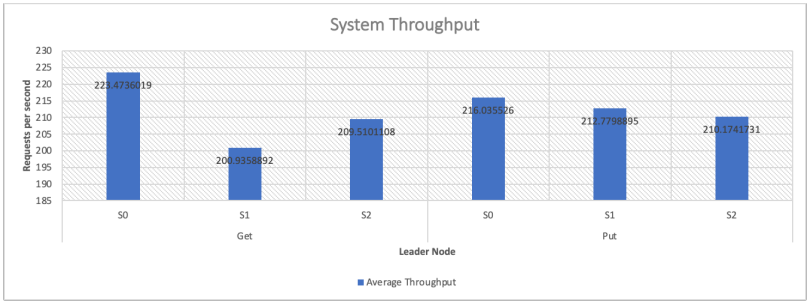

Figure 2: Read/Write throughput (requests/second)

- Availability -
  To measure availability, we first measure the leader election time. Since the system cannot respond to client requests until a leader is elected and stable, this is equivalent to MTTR. We measure this by starting a stream of requests to a stable leader system before crashing the leader. As the client retries to a different server once its RPC requests fail, it continually monitors the system until a leader is elected. We measure this interval to be 6.88 sec for an election timeout of 3 sec. We plot availablity varying MTBR 25 sec between failure to 500 sec between failures. Since the average election time is constant between failures, we stick to a theoretical plot. We also varied the election timeout values and found that this changes the leader election times. We include the plots for three such election times, 6.88 sec (election timeout = 3 sec), 7.27 sec (election timeout = 3.5 sec), and 7.89 sec (election timeout = 4 sec).

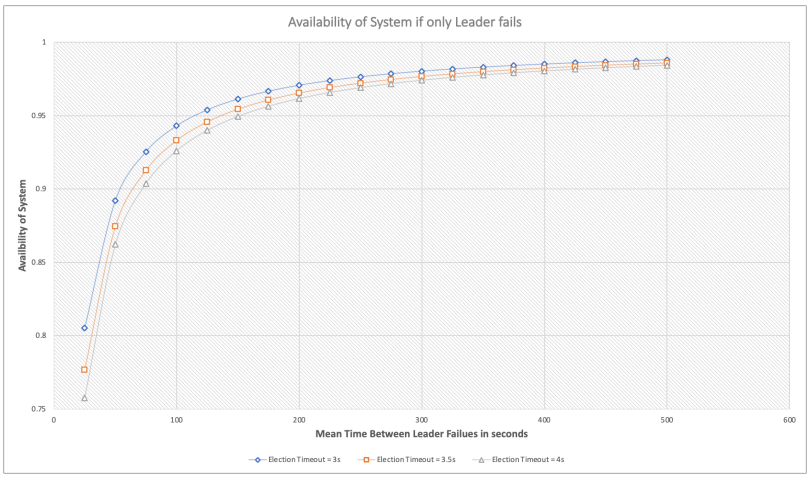

Figure 3: Availability by varying election timeouts

- Execution optimization on followers -
  To quantize the benefit of overlapping execution of followers with communication, we added a delay in the execution at follower. We reason that since the key-value requests are trivial and won't account for enough compute time, we added synthetic a delay of 5ms and measured round trip latencies. We observed that the latencies don't change much and remain between 4-5ms. If there would have been no pipelined implementation, the time would have instead increased by 5ms. This also helps overcome performance degradation due to stragler servers if it is slow in executing the commands.

3 Conclusion

We implemented the Raft consensus protocol on a key-value storage system using C++. We went through the paper [1] and Ongaro's thesis [2] multiple times to deepen our understanding of the protocol. We understood it further during implementation and bug fixing. In the end, we built the replicated database that handles various leader crash scenarios successfully.

References

[1] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. USA: USENIX Association, 2014, p. 305–320.

[2] O. Diego, "Consensus: Bridging theory and practice," ://github.com/ongardie/dissertationreadme, 2014.