Replicated Key-Value Store

Aadi Swadipto Mondal Mohil Patel Rahul Uday Chakwate
Project Group 2/8

1 WISC AFS DESIGN

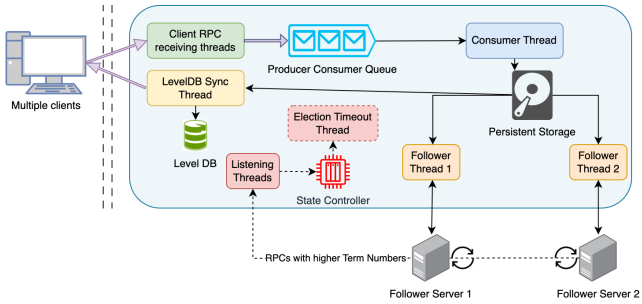

Figure 1: Leader design

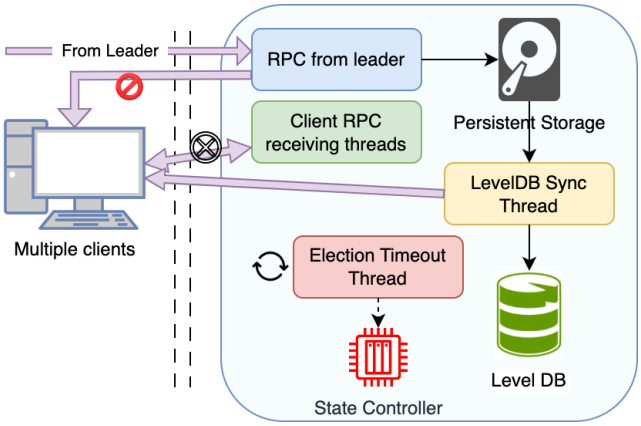

Figure 2: Follower design

1.0.1 Key Design Principles. Our replicated distributed key-value database is designed using Raft. The following points describe the key design designs of our distributed database.

(1) Our entire system is designed using polling-based ideology. There is no signaling between threads (like condition variables). Each thread runs independently, polls and updates shared (persistent) states, and decides actions based on their values. Threads yield whenever they have no work to allow other threads to run.
(2) All persistent states are updated atomically (using locks and atomic instructions).
(3) Log entries are lazily synced with followers (Leader maintains a separate follower thread for each follower for performing log sync).
(4) Client uses a token system to track “put” requests (always served by the leader).
(5) Election procedure is modified to ensure the candidate with the most updated log wins.

1.0.2 Leader Design. The leader design is described in figure 1.

(1) Leader receives client "put" requests using a multi-threaded GRPC server and en-queues them in a multi-producer, single-consumer queue. In our case, we have used lock-less queues to improve performance.
(2) A separate consumer thread de-queues entries from the queue and writes them in a persistent log (we use a log file to make it persistent).
(3) For each follower, a separate follower thread reads those logs and tries to sync them with its designated follower. If there is a mismatch in the previous log entry, this follower thread decrements the last-sync-index till the logs match for both and then syncs the rest of the leader logs.
(4) For every log entry, follower threads update a shared state for the majority count. Once a majority is received, it updates the last-commit index.
(5) A LevelDB sync thread reads the last-commit index and updates the state machine (LevelDB) if there are any committed logs left to apply on the state machine. It also sends a positive acknowledgment to the client once those logs are applied.
(6) Leader also responds to "get" requests from the client once it has at least one entry of its term committed.
(7) GRPC server listens for Request-Vote and Append-Entry RPCs with higher term numbers. On receipt, it steps down from leader to follower and sends negative acknowledgments for client requests in the queue.

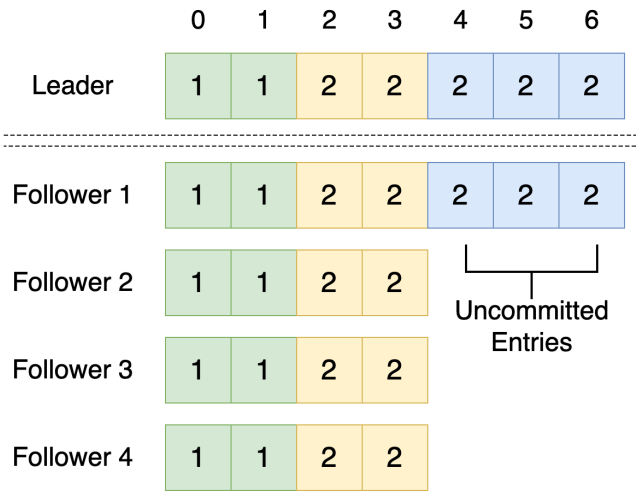

Figure 3: Election

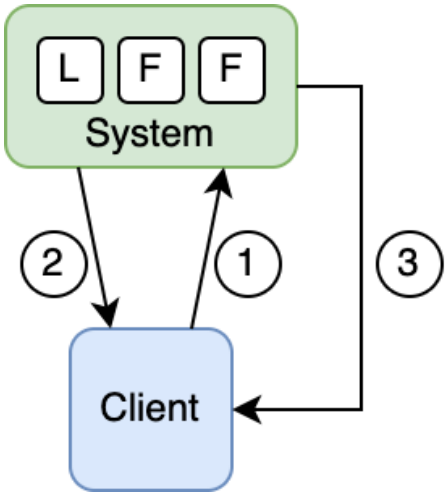

Figure 4: Client Token System

1.0.3 Follower Design. The follower design is described in figure 2.

(1) GRPC servers listen for Append-Entry and Request-Vote RPCs.
(2) On receipt of an Append-Entry RPC, it validates the previous log. If validation fails, it sends failure to the leader. On success, it writes the log in persistent storage.
(3) Append-Entry RPC also updates the last-commit index. We ensure that we do not decrement the last-commit index. If the leader tries to sync a log entry below the last-commit index, the follower asserts that the present entry and the RPC entry are exactly the same.
(4) In case of a log overwrite, the follower sends a negative acknowledgment to the client for the overwritten entry.
(5) Similar to the leader, a LevelDB sync thread updates the state machine and sends a positive acknowledgment to the client.
(6) An election thread wakes up at periodic intervals to check if any heartbeat was received. If no heartbeat was received, the follower increments term and becomes a candidate.

1.0.4 Election Procedure. Apart from the standard election requirements in Raft, we also introduced an additional condition for a candidate to become a leader. A candidate cannot become a leader if some server denies the vote to it. This is to ensure minimum log overwrites occur so that clients do not receive failure for requests.

An example is shown in Figure 3, let's assume that the leader gets into a network partition after syncing log entries 4-6 only in Follower 1 in a 5-server system. Followers, 2-4 can still become a leader in the next election term, i.e. term 3. If that happens, log entries 4-6 will be overwritten by the new leader eventually sending negative acknowledgments for those 3 entries.

Apart from this, we follow the standards of Raft Election. The election thread detects a timeout (in absence of a leader) and sends Request-Vote RPCs. Any denial of vote, failure to achieve a majority, or receipt of Append-Entry and Request-Vote RPCs with higher term numbers leads to stepping down again to a follower. If the candidate successfully becomes the leader, it restarts the consumer and follower threads. LevelDB sync thread continues to sync the log when the last-commit index becomes more than the last-sync index.

1.0.5 Client Token System. The procedure for a client to communicate with Raft Servers involves multiple steps as shown in figure 4.

(1) **Step 1:** The client sends a request to the leader of the raft system. The request contains the client ID and request number (tracked by the client).
(2) **Step 2:** The leader then acknowledges receipt of the request. Although, this may not be successful if the leader's queue is already full.

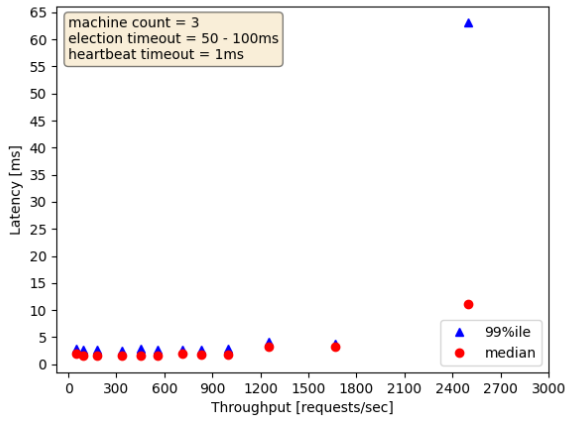

Figure 5: RTT latency (99 Percentile and Median) against throughput

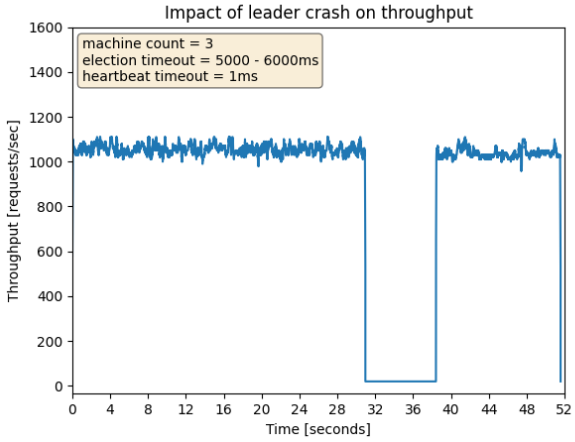

Figure 6: Impact of leader crash on throughput

(3) When entries are written in the log, the client ID and the request number (sent by the client) are also written in the log entry.
(4) **Step 3:** For "put" requests, the client will receive a positive or negative acknowledgment at some point in the future.
(5) For "get" requests, a successful response can only be given if the leader has committed at least one entry of its term.

The client API offers both blocking and non-blocking put requests, being blocked until acknowledged or a timeout occurs. Additionally, the client caches the leader's ID after the first successful request and updates it in the event of a failure to reach the leader.

2 PERFORMANCE

In this section, we measure the performance of RAFT. In the first experiment, we plot the round trip time (RTT) latency of our system against throughput as shown in Figure 5. For this experiment, we use the server count of 3 with a reduced election timeout of 50-100ms and heartbeat timeout of 1ms in order to gain performance. To vary the throughput, we put sleep in between the "put" requests to achieve the desired throughput. In the figure, We plot both the median as well as 99th percentile latency. As expected, as the throughput increases, load on the system increases which increases the latency of the system.

In another experiment, we visualize the variation of throughput upon leader crash and re-election. To get a better overview of this process, we increase the election timeout to 5 - 6 secs and maintain a constant throughput of around 1000 requests per second. In Figure 6, we plot the running average of the throughput in a window of 0.1 sec (as against 1 sec window shown in the presentation). We also let the system achieve a steady state of processing around 1000 "put" requests per second before recording the readings. As seen in the figure, when the leader crashes, the system stops serving "put" requests and the throughput drops to zero. It remains zero for the period of re-election, which takes around 6-7 seconds. The client request fails or receives negative acknoeledgement during this period. When a new leader is elected, the system quickly reaches back to serving around 1000 requests per second.

3 MEMBERSHIP CHANGE PROTOCOL

In our project, we have implemented a membership change protocol as shown in fig 7. The current implementation only supports member addition and not member removal.

As shown in fig 7, member addition system works as follows:

(1) New Machine puts an add new member request to the leader.
(2) Leader stops accepting *put* requests from the client.

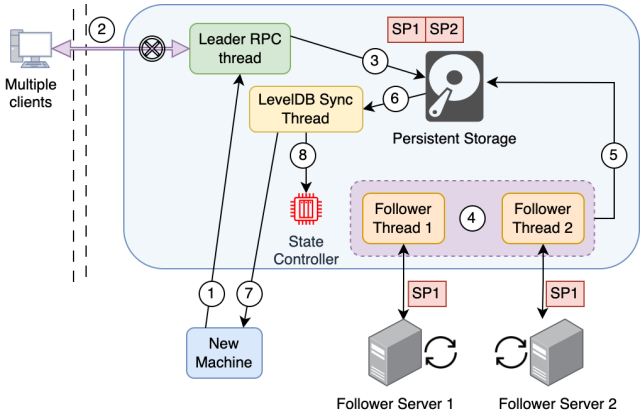

Figure 7: Membership Change Protocol

(3) Leader inserts two special entries *SP1* and *SP2* in it's log.
(4) Follower threads in the leader sync the *SP1* first, and after getting majority for *SP1* they sync *SP2*.
(5) On receiving majority follower thread updates the last committed index.
(6) The sync thread, on the increase in the last committed index, sees if *SP2* was just committed, it will increase the machine count in the leader. Additionally, followers will increment their machine count on commit of *SP1*.
(7) Leader replies to the new machine to join the cluster
(8) Leader steps down, and a new election happens.

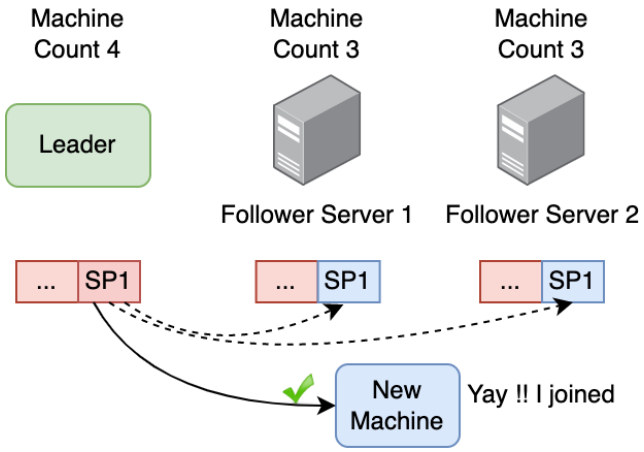

Figure 8: Why two entries?

In the above procedure, we use two special entries *SP1* and *SP2* rather than just a single entry. Fig 8 shows why this is necessary. If we only use one special entry, there is a possibility that *SP1* gets committed in the leader but not in the followers. Due to *SP1's* commit, the leader will increase its machine count, but if the leader fails at this point and one of the followers with uncommitted *SP1* is elected. Then we would elect a new leader with an older machine count even though the new machine has been sent an acknowledgment to join.

4 TESTING CORRECTNESS

Testing for correctness is hard in Raft. There are many edge cases, and it is hard to test them by hand. So to automate the testing, inspired by Netflix's Chaos Monkey project, we wrote our random partitioner. With the help of Linux's *iptables* command, we create a virtual firewall to self-isolate the node. This way, this node will be partitioned from the rest of the system. After every few seconds, the tool randomly decides (with some probability passed as a parameter) whether to partition the node or not. In this way, we designed a randomized testing environment where each raft node runs this tool (which we call gracious monkey). We ran the testing using this tool and showed that our system works properly (i.e., the logs match on all the machines at the end) when the time limits are in the order of seconds. For smaller time limits, we sometimes face some issues.

Additionally, we did extensive manual testing for many possible different cases. We tested cases like leader crash, follower crash, and network partition, among many others. And while programming the system, we used a fail-fast programming style with many assert statements to check for basic properties like the last commit index should never decrease, committed indexes should match, during state transition we can't jump from follower to leader directly, etc.

5 TAKEAWAYS

5.0.1 Hard crashes can lead to metadata corruption. In our system, many of the atomic metadata updates change multiple files. Due to this behavior, a hard crash in between this atomic update may leave some files in an incorrect state as their metadata was not updated. Additionally, if such a system restarts, it can propagate incorrect metadata to the rest of the system too. This propagation can lead to the whole system's corruption. To handle this, we can design the system using a single file atomic update or transactions for file updates. This

way, it would be possible to revert the file updates if the transaction did not finish properly.

5.0.2 Testing Correctness for Raft is hard. As discussed in the previous section, Raft has many edge cases possible, and testing them manually is impossible. Manual testing with human responsiveness is not feasible for a millisecond or lower time scales. So this inspired us to design the automated randomized testing as discussed in the previous section. Lastly, we sometimes saw GRPC getting stuck when we did random partitioning using Linux's *iptables* command. So there could be a bug in the GRPC code too.

5.0.3 Polling based systems are easier to design. In our design approach, we have used a polling-based system instead of an event-triggered-based one. Although polling-based systems can be inefficient compared to the event-based system, Raft only has a small number of threads, mainly in the leader machine, so this should not have a considerable performance impact. And the significant benefit we get from a polling-based system is lesser deadlock possibilities and lock contention, which can help improve performance and correctness.