COMP60009 - An Elixir Implementation of the Raft Consensus Algorithm

Rob Wakefield <rgw20>

February 26, 2024

1 Architecture

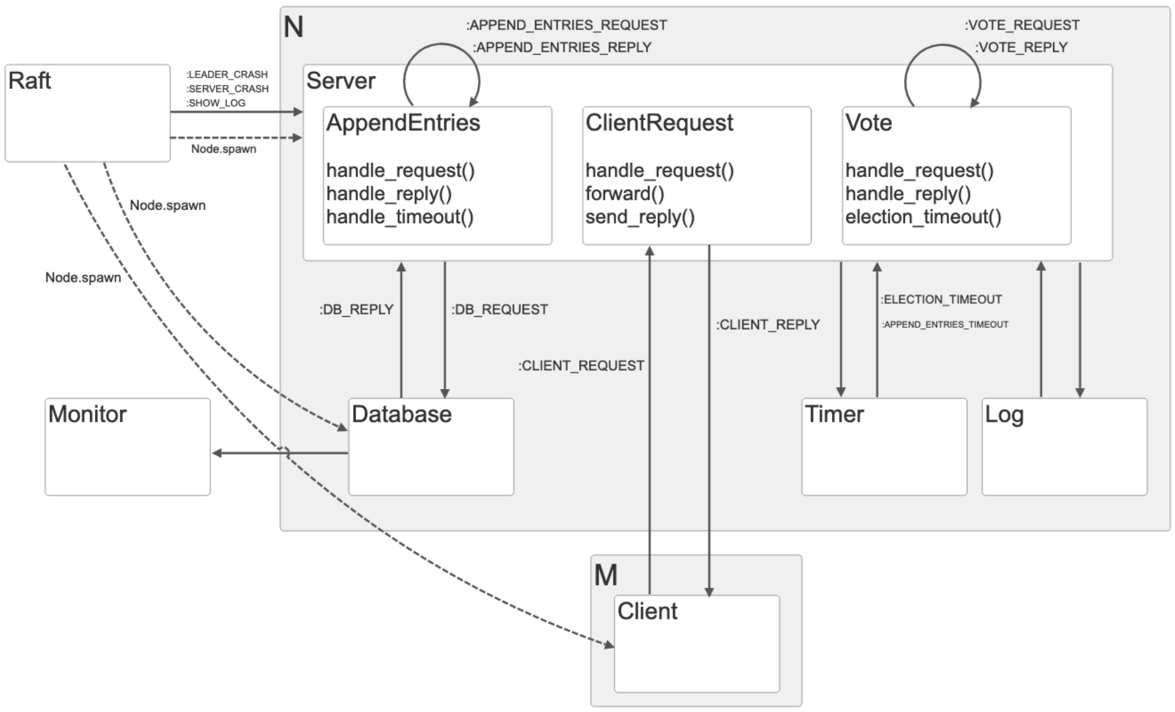


Figure 1: System Architecture

Figure 1 shows a diagram of the architecture of the system. The diagram represents a system with N servers and M clients. The messages sent between modules are labelled and the spawning of elixir nodes are represented by dashed lines.

A normal flow of a typical client request is as follows:

Sender	Receiver	Message
Client	Server	:CLIENT_REQUEST
Server	Server	(N-1) × :APPEND_ENTRIES_REQUEST
Server	Server	(N-1) × :APPEND_ENTRIES_REPLY
Server	Database	:DB_REQUEST
Database	Server	:DB_REPLY
Server	Client	:CLIENT_REQUEST

2 Design and Implementation

The consensus algorithm is implemented as per the original paper [1] with some specific optimisations to utilise the Elixir language.

The AppendEntries requests include an entries map with only the items that the receiving process needs to append to their log, instead of the leader's entire log. This allows for efficient message passing as it should only take 1 request message to bring a previously crashed or slow server up to date and minimises the size of messages sent during heartbeats.

After receiving most messages, the term sent in that message is compared with the servers current term to ensure that messages from older terms are ignored and previously erroneous processes do not interfere with a correct process.

In order to prevent duplicate ClientRequest messages, all previously seen client message ids are stored by the leader process. This allows for followers to forward ClientRequests they receive to the leader despite the client process possibly sending the same message to multiple servers. Whilst there is no leader (e.g. during an election), any ClientRequests received are simply sent back to the server that received them, until a leader is elected. This adds some extra message passing during an election, but allows for robust delivery of ClientRequests to the leader during normal operation.

2.1 Debugging

To aid with debugging the system, a number of *assert* statements were placed at critical points in the system as well as logging statements for major events. In the AppendEntries handler functions, the requests and replies are filtered based on whether they are heartbeats or not. This helps to avoid flooding the debug output with heartbeat messages when the AppendEntries messages are being displayed.

In the information the *monitor* outputs, the balance for *account1* was added in order to quickly check that the databases for each server match each other. And, when there is an error with the database a message is sent from the *monitor* to all servers requesting for them to display their current logs around the erroneous database entry. This helps with understanding why the log is wrong - if it is a wrong ordering or a completely overwritten log for example.

The debugging flags and *param* functions are explained in detail in the *Makefile* and the *README.md*.

3 Evaluation

3.1 System Specifications

The machine used for evaluation of the Raft implementation is an Apple MacBook Pro with M1 Pro CPU and 16GB RAM running MacOS Sonoma 14.1. The CPU has 8 high-performance cores and 2 high-efficiency cores.

3.2 Performance under Varying Loads

The configuration is set up to run the system for 30 seconds with varying number of servers and clients, at which time the *monitor* will display the total number of database updates performed by all the servers. After 2 runs at each combination of servers and clients, the average number of database updates per second is calculated.

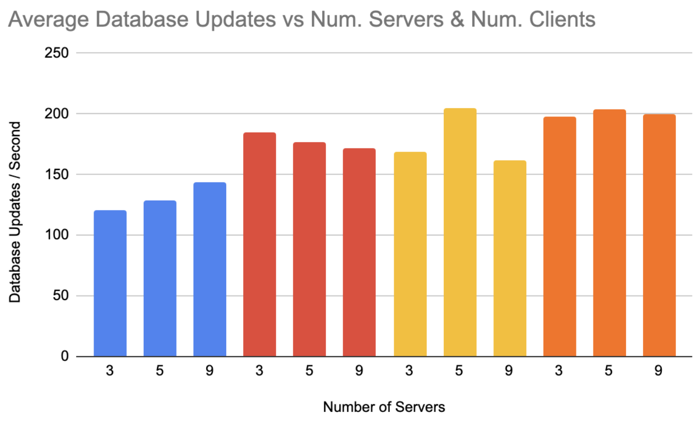


Figure 2: Database Updates vs Number of Servers. Number of clients increases from left to right as [1, 3, 5, 9]

01_9servers_9clients.txt: Figure 2 shows that when there are 3 or more clients, increasing the number of servers in the system does not increase the throughput of database updates - in fact for the case with 5 clients the average number of database updates per second decreases. This could be due to the increasing number of heartbeat requests/replies that must be sent/received congesting the system.

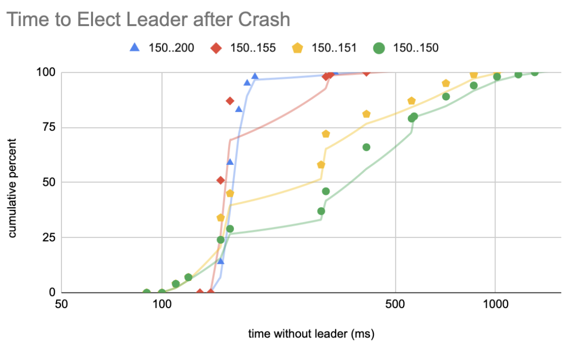
The number of servers in the system has the biggest impact when there is only 1 client. This is unexpected as the client only sends client requests to the leader so the client sending rate is saturated no matter how many servers are in the system. However, the number of heartbeat requests and replies would increase and it would be expected that these would cause some delays in the system and lower the average number of database updates per second. So, the increase in database updates as the number of servers increases may only be explained by the variability of running the tests on a non-isolated machine that has other applications running.

3.3 Split Vote Elections

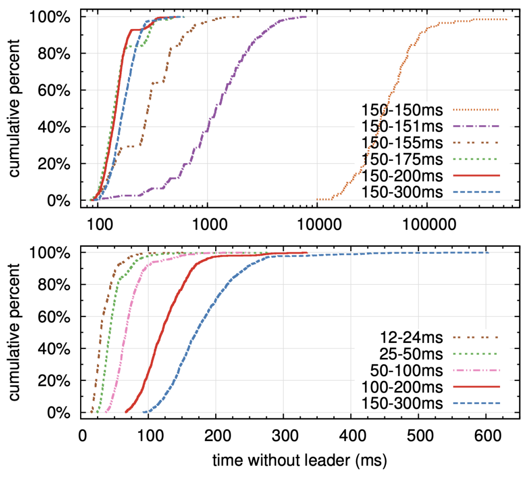
In order to try to replicate the Raft Consensus paper's findings (Figure 3b) about the effect of election timeout range on the time taken for a new leader to be elected after the current leader crashes, a *param function* called *split_vote* sets the number of servers to 5 and disables ClientRequests. The election timeout range is set to various ranges from 150-150ms to 150-200ms, with the time between heartbeats (also the time for an AppendEntries timeout) set to half of the minimum election timeout (75ms). The leader is set to crash every 500ms and an AppendEntries "heartbeat" request is sent out just before the crash to simulate the worst case as mentioned in the paper.

A debug option *time* is set which outputs the time of a crash and of a new leader becoming elected in milliseconds. An example output for the election timeout range 150..155ms is included in **04_split_vote_150_155.txt**. This data is then trimmed to include 100 samples and processed to find the cumulative percent of the 100 samples that have found a leader within a given time.

The result of this experiment is shown in Figure 3a. When compared to the reference graph from the Consensus Paper, there is clearly a similarity between the shape and distribution of the cumulative percent traces for each range. In our graph, the datapoints are less frequent due to the 100 samples versus the paper's 1000, but the trend of the 150..150ms and 150..151ms data points have significantly longer time without a leader than the other ranges as we would expect. The wider ranges are also similar to the paper's findings with steeper gradients and lower average time without a leader.



(a) Effect of election timeout Range on time to elect a new leader after crash



(b) Reference graph from Consensus Paper

3.4 Performance under Failures

3.4.1 Leader Crashes

Servers: 5, Clients: 3

The configuration is set up to crash the leader after 1500ms for a duration of 5000ms. At time=11000, after the crashed process has come back online, all client requests will stop in order to show the eventual convergence of each server to the correct state.

02_leader_crash.txt: When server 2 crashes, we see that it's database updates lag behind the other servers and a new leader (server 3) is elected. Once server 2 comes back online at time=6500, it becomes a follower and updates its log with the entries that it missed. By time=9000 server 2 has successfully caught up with its database updates and the system continues as expected, eventually converging on 5475 database updates for every server after all client requests have been received. The delay between server 2 coming back online and being up to date with the rest of the servers is in line with the 400-500 updates/second average of the entire system during this run.

Note: There is a bug where an erroneous election occurs and server 3 and server 4 become leaders in quick succession after server 2 comes back online. However this does not change the outcome or correctness of the database updates, it is just an inefficiency in the system.

3.4.2 Follower Crashes

Servers: 5, Clients: 3

The configuration is set up to crash server 3 after 1500ms for a duration of 2000ms and server 4 after 2000ms and to not recover. At time=11000, after the crashed process (server 3) has come back online, all client requests will stop in order to show the eventual convergence of each correct process to the correct state.

03_follower_crash.txt: When server 3 and 4 crash, we see that their database updates lag behind. When server 3 comes back online it quickly becomes up to date with the rest of the majority of servers. Once all client requests have been received at t=12000, every correct processes database update's agree on 4640 and server 4 (which is still crashed) has fallen behind with only 1494 updates.

References

[1] Diego Ongaro and John Ousterhout. "In search of an understandable consensus algorithm." In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USENIX ATC'14. Philadelphia, PA: USENIX Association, 2014, pp. 305–320. ISBN: 9781931971102.