Distributed Algorithms CW

Dhruv Jimulia (dj321) and Adi Prasad (arp21)

1 Architecture

The architecture for our Raft implementation is given in Figure 1 below.

2 Design and Implementation

The following were the unique aspects of our design:

- **Server per-file debugging.** In `configuration.ex` , we have a configurable parameter `per_server_file_logging`, which, when true, will output the debugging information for each server to a separate log file, for better debugging
- **Additional configuration for failing servers.** The following parameters were added to `configuration.ex` to easily configure crashing servers:
  - `crash_servers`: a map from `server_num` to the number of milliseconds after which the server crashes. Implemented via `:KILL_SERVER` message that all relevant servers send to themselves
  - `crash_leaders_after`: number of milliseconds after which the leader crashes. Implemented via the `:KILL_LEADER` message that the leader sends to itself
  - `repeated_crashing_leaders`: a Boolean that if true, will crash leaders repeatedly after every `crash_leaders_after` ms
- **Debugging functions.** In order to have comprehensive but customizable error messages, we use functions like `Debug.become_leader` and `Debug.received_append_entries_request`, that build on top of `Debug.message`. This allowed us to create custom messages when required, while at the same time use `DEBUG_OPTIONS` and `DEBUG_LEVEL` for configurable debugging.
- **Added database update throughput to monitor statistics.** The `monitor.ex` file was modified to print out the throughput of database updates, measured in number of updates across all servers per millisecond. This will be used as a metric for the performance of the algorithm.

3 Evaluation

The following experiments were conducted on a machine with 11th Gen Intel(R) Core(TM) i5-1155G7 @ 2.50GHz quad-core processor, x86_64 architecture, 8GB RAM, with the Ubuntu 22.04.3 LTS operating system.

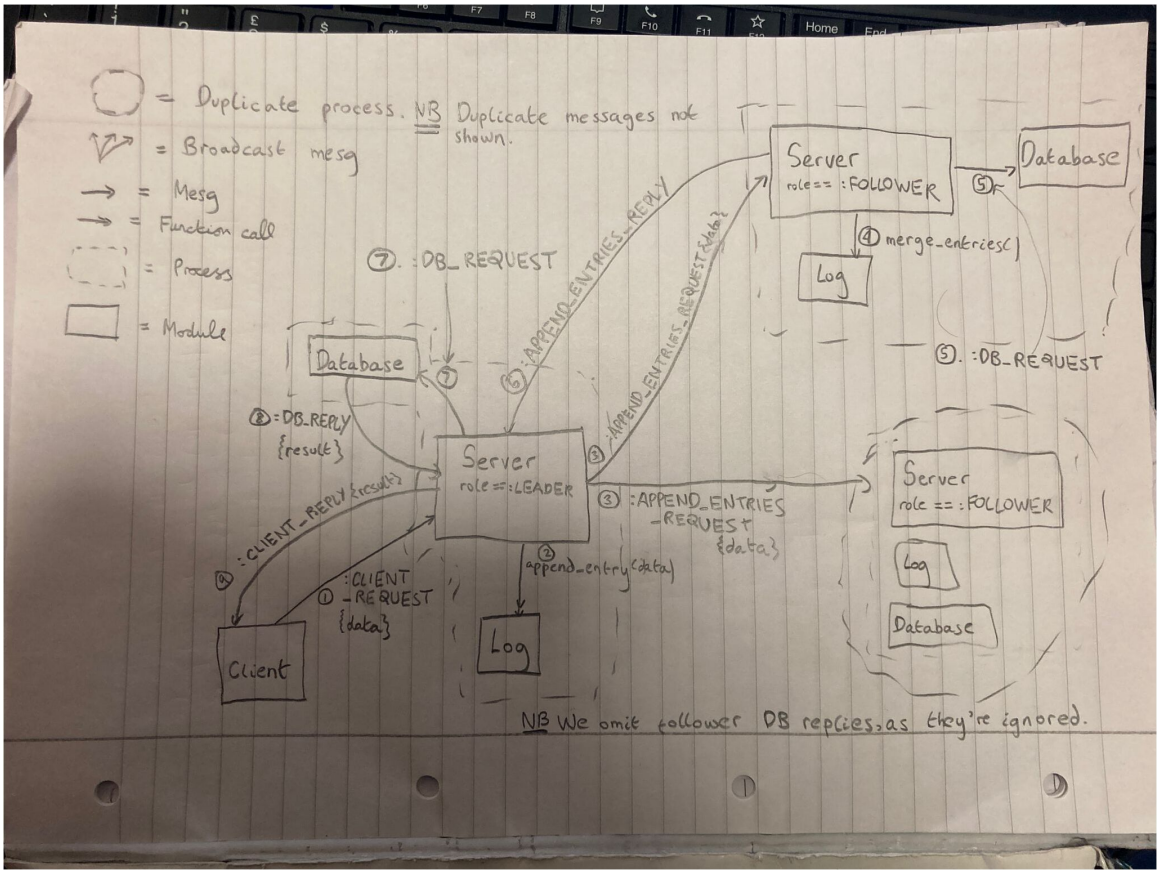

Figure 1: Architecture

3.1 Under normal situations

When the default configuration given as part of the skeleton is run for 15 and 60 secs, the evaluation output is as given as in `00_run_15_secs` and `01_run_60_secs` respectively. In this case, one server remains leader from the beginning, the leader receives slowly processes client requests, and all servers eventually apply the corresponding updates to their databases, as expected. In the 15-second experiment, the average db update throughput across all servers is 1.259 updates/ms, and in the 60-second experiment, this average throughput is 0.617 updates/ms.

The lower throughput when running the algorithm for a longer period of time can be explained by the large number of append entries messages that clog up over time that need to be handled and actual new database updates from occurring. This implies that the append entries timeout is too low compared to the optimal timeout.

3.2 Under failures

We run the algorithm for 15 seconds, and consider what happens on different types of server failures:

- **Follower server crashes.** In `02_server_one_crash`, server 1 (which happens to be a follower in the case), crashes after 5000 milliseconds. In this case, the existing leader continues to handle client requests. Compared to normal conditions outlined in section 3.1, the average throughput only slightly decreases to 1.209 updates/ms. This can be attributed to garbage collecting the presumably long log that the follower server may have had due to the append entries messages.

- **Leader crashes.** In `03_leader_crash`, the leader crashes after 5000 milliseconds. Here, a new server (server 3 in this case) becomes the leader and continues to handle the client requests. Overall throughput reduces to 1.215 updates/ms. This can be attributed to the overhead of elections and the fact that there are no database updates during elections.

- **Leader crashes repeatedly.** In `04_leader_crashing_repeatedly`, a leader crashes every 3000 milliseconds. Like the previous case, another server takes over as the leader when the leader crashes. However, when the majority of the servers crash (3 out of 5 in this case), none of the candidate servers are able to get a majority in the voter election, no leaders are elected, and no progress is made on the database updates. Throughput does decrease quite significantly to 0.95 updates/ms, which not only occurs due to the overhead of more elections, but also because after a point, no progress is made in the system.

3.3 Varying election timeout

As we vary election timeout ranges from 100-500, 500-1000 and 1000-5000, the average throughput decreases, as shown in Figure 3 and in the corresponding outputs `05`, `06`, and `07`. This makes sense: as the election timeouts increase, the time required for the initial election to occur increases, just slightly decreasing the throughput. However, we still do not have any leadership changes, and so we pay this overhead only once.

This explanation is also reinforced by looking at the output of `07_election_timeout_1000_5000`, where, in the first 1000 milliseconds, there are no client requests seen and no database updates done (since a leader is not elected).

We observed a bell curve relationship between the number of servers and system throughput, as shown in Figure 2.

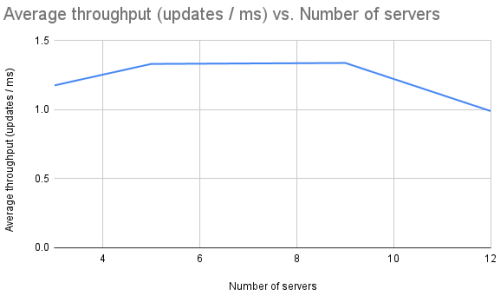

Figure 2: Average db update throughput against number of servers

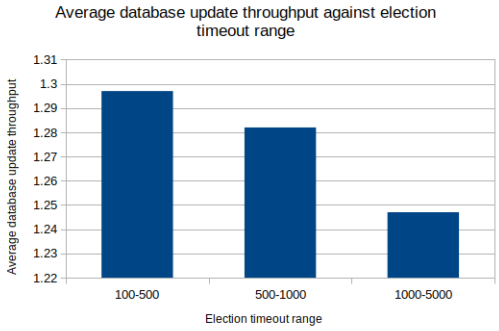

Figure 3: Average db update throughput against election timeout range

The initial increase in throughput was unexpected to us. We hypothesise it is because of the following. In order to update a database, the leader must have received replies from a majority of servers. When the number of servers is very low, if even a small number of followers are temporarily "down" (e.g. because their process was descheduled from the OS), the leader will not be able to observe a majority and will have to wait to update - hence decreasing throughput. As we increase the number of followers, our follower cluster becomes more reliable - it is less likely that a majority of servers will be descheduled or have something else happen to them, simply because there are more servers so the impact of one or two going down does not matter. Hence, the leader is able to observe a majority faster and make updates faster.

After this, throughput decreases. Past a point, increasing the number of followers does not help as it is already highly unlikely a majority of servers will be temporarily "down". Meanwhile, we need to wait longer and longer to commit a request as more servers must confirm they have received a request before we can commit. There are also other smaller factors: the message queue of the leader will be clogged with more append entries timeouts and append entries replies, so we will spend more time handling them and less time handling client requests. Additionally, since these experiments are all happening on one machine, we may be bottlenecking on the servers' accesses to shared memory or the number of threads the CPU has available. The plateau in the middle may simply be both the aforementioned effects "cancelling each other out".

3.4 Under high load (varying the number of clients): effect on throughput

Up to 4 clients, throughput increases. This is not particularly interesting; it is simply because the system is simply not processing many requests when the number of clients is so low, so the throughput appears low not because the system is bottlenecked but because there is not much load on the system. Throughput is then constant. As load increases, throughput remains statistically constant. Although we are receiving more client requests per millisecond, we are now "at capacity" and the speed at which these are processed is not changed, because we still execute the same steps to process a client request.

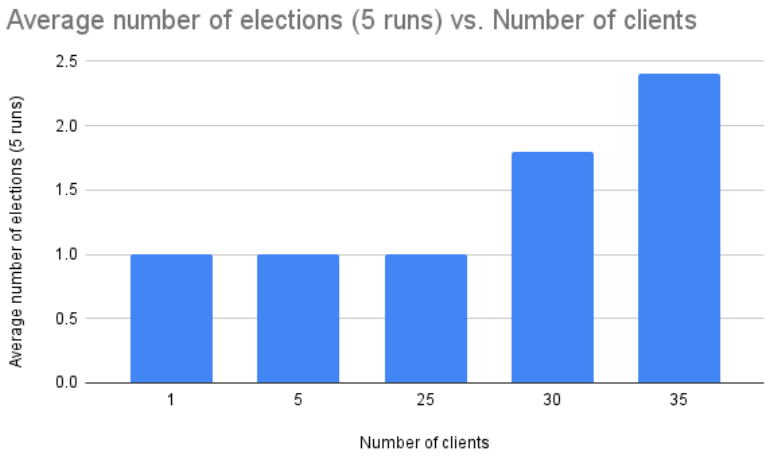

Figure 4: Average number of elections (5 runs) vs number of clients

3.5 Under high load (varying the number of clients): effect on election frequency

It is possible that were we to keep increasing the number of clients, we would discover a falloff in throughput due to eg memory running out, but we could not do this. Past a certain number of clients (we started noticing an effect past 30 clients), leader changes become so common that it becomes impossible to execute an experiment without a leader change and so experiments cannot be compared to the lower-load ones with only a single experiment. The leader changes become more common as load increases because the leader's message queue becomes increasingly clogged with client requests. This means 1) it takes longer to get to append entries timeout messages from itself, and 2) the OS process it is within spends proportionally more of its time servicing the client request. This means it will send append entries messages to its followers at longer and longer intervals, and hence followers are more likely to assume it crashed and start an election. As discussed in "Under failures", the overhead of the election significantly decreases throughput. One other interesting thing to note is the non-linear relationship here: even as load increases linearly, the average number of elections stays at 1. This is because the decision to start a new election itself is non-linear: we can wait however long for an append entries message, and it is only once they start taking longer than our election timeout value that we stand for election.