60009 Distributed Algorithms - Raft Consensus Report

Bruno Wu

February 22, 2022

1 Design & Implementation

The design and implementation follow the standard Raft procedure as seen in Figure 1 with the structural diagram showing the main components' interaction. Figures 2 and 3 have also been added to show an optimistic flow of the Raft procedure where there are no failures or extended delays.

An interesting aspect of the implementation is the handling of repeated client requests. Client requests are added to the leader's log and then replicated on the followers. Once there is a majority of servers with the request added to their logs, the leader first commits it to its database and only then do the followers commit their corresponding request to their local database. When the leader commits the request, it only then replies that replication has been successful to the client. However, if there are connectivity issues between the nodes, it can take a long time for the client to receive an acknowledgement from the leader and potentially resend the same request to the leader. To avoid repetitions, an extra check has been added before every request is appended to the leader's log, ensuring that the request is not processed twice (see `handle_client_request` in ClientReq.ex).

Database commits are handled synchronously, so when the server tries to commit data, it will wait for the database to reply successfully before continuing with tasks. This places an assumption that the database will always work correctly. This can be easily fixed by retrying in case of failure. However, it was not implemented, being out-of-scope.

The `last_applied` variable has been omitted because, essentially, it is the same as the latest index in the log. The latest index of the log is retrieved on-demand. This reduces the complexity of tracking an extra variable. There is no performance harm either since the function `map_size` runs in constant time in Elixir.

The algorithm to determine the latest entry common in all followers is quite ingenious. Instead of checking, one by one, and seeing which is the latest entry that is replicated on the majority of the nodes, this algorithm sorts the `match_index` of all the peers. It then returns the element in the array positioned in the majority index. For example, if the system has five nodes, a majority would be three and the third item in the sorted `match_index` list would be the latest majority entry index. This can be a very efficient way of computing the value (see `leader_commit_log_entries` in appendentries.ex)

2 Testing & Debugging

Testing an debugging is mainly done with a the help of the `debug`, `monitor` and `configuration` modules. In the `debug` module, the main functions used were `message` and `info`. For all the send and receive instances in the code a `message` function has been associated with it with the appropriate label in `DEBUG_OPTIONS` from the `Makefile` so that it can be customised to show

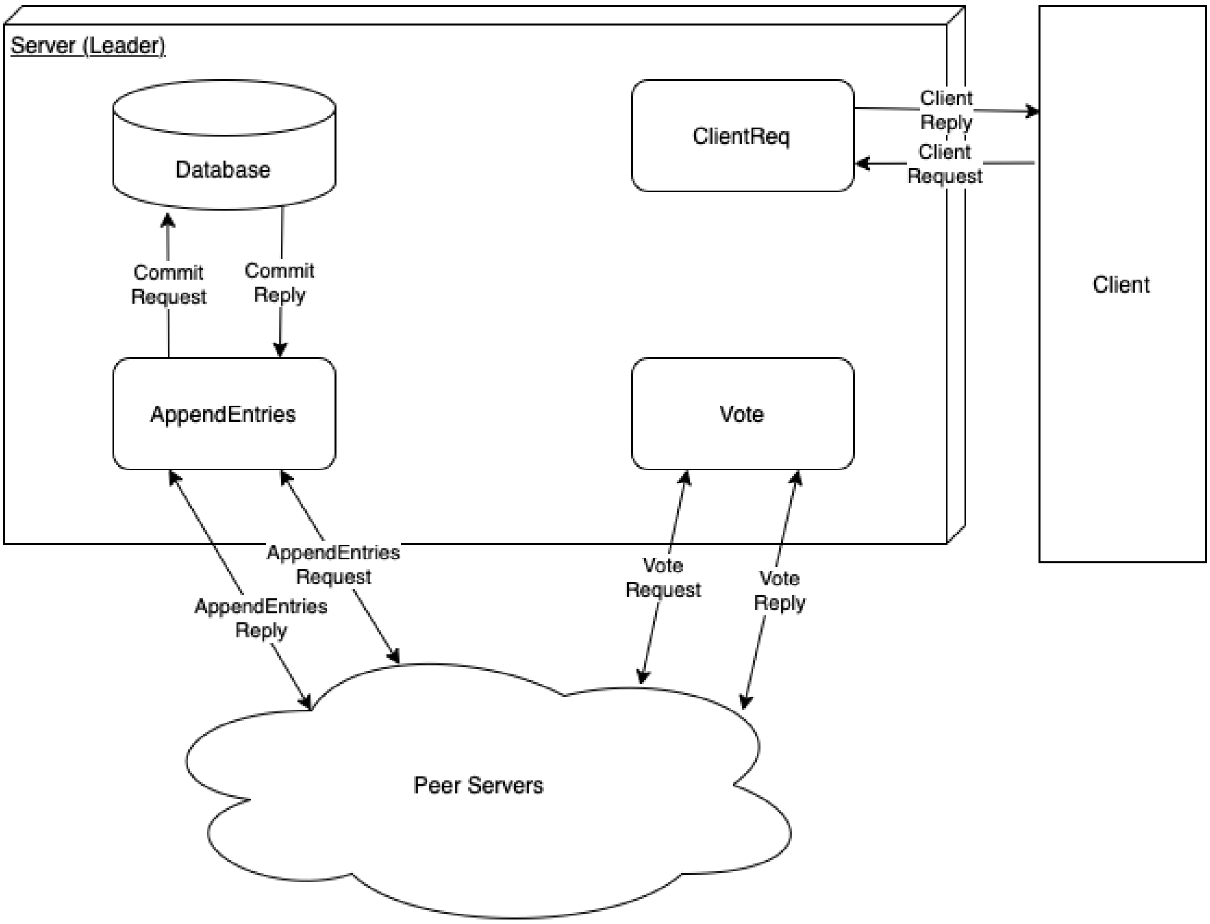

Figure 1: Structural diagram with all the interactions between different components

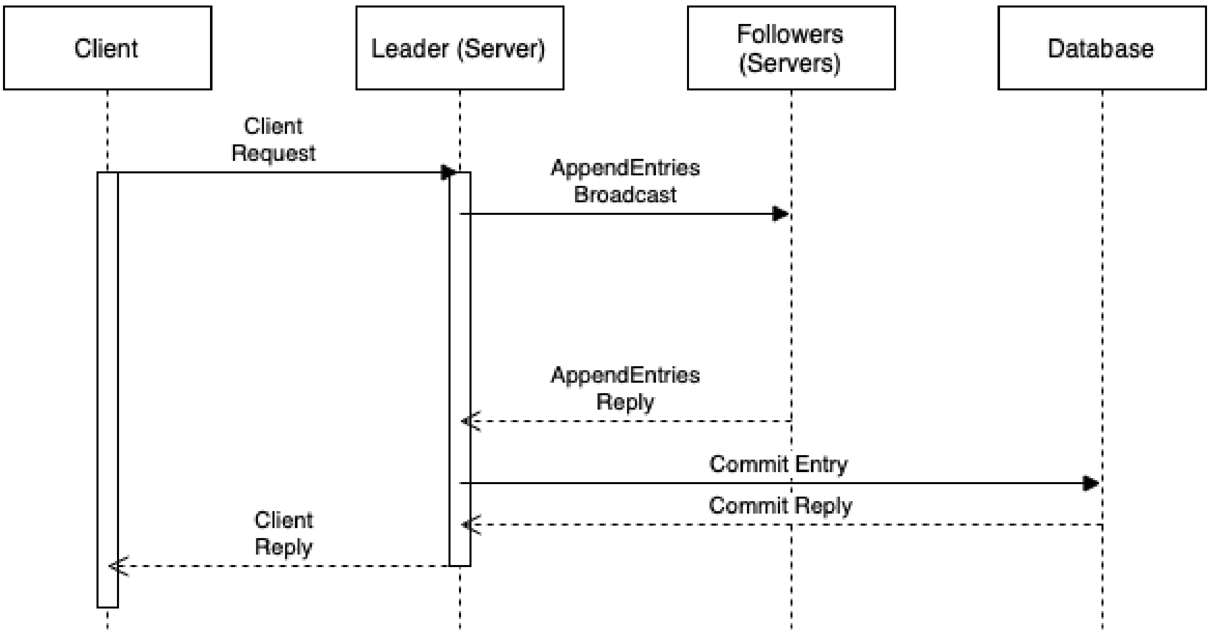

Figure 2: Sequence diagram of a successful client request procedure

specific kind of messages for debugging purposes. All the `AppendEntries`, `Vote`, `Election` and `Client` messages have been implemented which have been crucial for debugging. On the `monitor`, side the client reads and database commits loggers have been implemented to test the system. The `configuration` module has also been crucial to create test scenarios for the Raft program to run in and set up server crashes, sleeps and etc.

Additional testing mechanisms have been added to the system. For example a unreliable send has been added to the `configurations`, where the sending reliability can be specified. Then the `monitor` module tracks how many packets are lost due to unreliable send to log the metrics. The unreliable send affects all outward messages send from the server to other modules such as the database and the client. Additionally, abilities to trigger sleeping and crashing on specific servers or for a leader at specific times have been added. This helps to test the reliability of the system and can be easily activated in the `configurations`. All of these configurations are setup and the mode of testing can be specified easily in the `Makefile`

3 Results

The testing performed on the Raft implementation has been detailed in Table 1. The main interesting takeaway is the high reliability that the system has. Despite the protocol losing half of its packets, it can still perform really well but at a lower performance due to resending packets. Furthermore, the election process is thoroughly tested with leaders sleeping and crashing. To test the log replication, nodes have slept and recovered the lost logs very quickly, soon after waking up again. The system has been tested under high load, where clients would send messages with 1-second intervals, and the system remained highly performant.

The main implementation concern found is that when a server goes to sleep and comes back will trigger a new election because the election timeout becomes 0. Hence it unnecessarily triggers new elections to all nodes. In consequence, sometimes, other valid nodes will also overthrow the current leader. This may not be desirable but has no reliability concerns, only with performance where there is a need to host a new election.

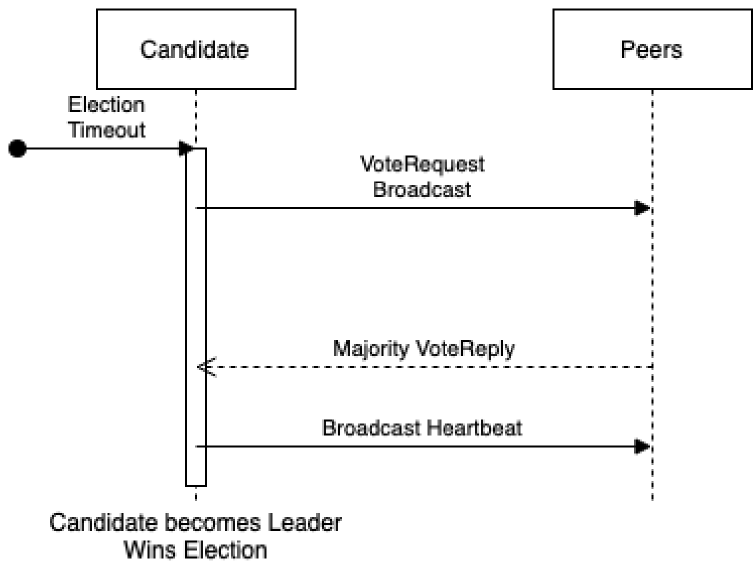

Figure 3: Sequence diagram of a successful election

Test	Configuration	Rationale	Result	Observations	Log file
Default	5 servers, 5 clients, 1000 messages per client (mpc)	Normal procedure to ensure Raft works	Success: All 5000 request committed & replicated correctly (CRC)	Working under expectation	default.log
Single Crash	Default — server 1 crashing at 3s	Checks if it can continue with 1 follower crash	Success: All requests CRC, except the crashed server	Once the server crashed all others have been able to continue smoothly	single_crash.log
Maximum Crash	Default — server1 & server2 crashing at 3s	Checks if it can continue working with only majority remaining (ie 3/5)	Success: All requests CRC, except the crashed servers	With just majority of servers running it seems that performance is slower. Potentially due to higher requirement of replications, since now the 3/3 servers must all replicate and agree to commit	max_crash.log
Leader Crash	Default — the leader at 3s crashes	Checks if a new leader can be elected whilst maintaining reliability.	Success: All requests CRC, except the crashed leader. New leader successfully elected and taken over	As soon as the leader crashed a new one was successfully elected. No requests were lost in the process	leader_crash.log
Multiple Leader Crash	Default — the leader at 3s and 5s crashes	Checks if multiple leaders can crash and maintain reliability. Forcing multiple leaderships	Success: Same as Leader Crash	Same as Leader Crash. Some repeated messages have been replied to the Client. However, the database committed messages are still correct. As long as the Client is able to sort out repeated replies it is fine	multi_leader_crash. log
Single Sleep	Default — server1 sleeping for 1s at 3s	Checks if log replication is working when a follower become unavailable and returns	Success: All requests CRC	When a server wakes up from sleep it triggers a new election, where it is rejected and forces other processes to begin elections too. However, reliability is still maintained and all requests are CRC successfully	single_sleep.log
Majority Sleep	Default — server1 & server2 sleeping for 1s at 3s and 3.5s respectively	Checks if reliability can be maintained and logs replicated when followers go to sleep	Success: All requests CRC	Same election issue as Single Sleep. When a sleeping process wakes up it triggers new elections (sleep election issue)	majority_sleep.log

Leader Sleep	Default — leader at 3s will sleep for 1s	Checks if leader can go to sleep and recover appropriately and new leader can takeover	Success: All requests CRC	Sleep election issue rarely	leader_sleep.log
Multiple Leader Sleep	Default — leaders at 3s, 5s, 7s will sleep for 1s	Checks if multiple leader sleeping can be handled reliably	Success: All requests CRC	Sleep election issue rarely	multi_leader_sleep.log
High Load	Default — client_request_interval 1ms and 3000 mpc	Checks if Raft works with a lot of messages incoming	Success: All requests CRC	No issues and high performance maintained	high_load.log
High Load & Leader Sleep	High Load — leader sleeps for 1s at 5s	Checks if log replication and election work under high load	Success: All requests CRC	No issues and high performance maintained, high reliability in log replication and election	high_load_leader_sleep.log
Unreliable Send	Default — send reliability of 90%. All sent messages to the peers, database and client are compromised	Checks if the reliability of Raft can still be maintained with unreliable send	Success: All requests CRC	The leader seems to have the highest number of messages lost, reliability is still maintained despite all the losses. No messages lost. Really impressive levels of reliability where on average 1000 packets have been lost by followers and 5000 lost by leaders.	unreliable_send.log
Unreliable Send & Multi Leader Sleep	Unreliable Send + Multi Leader Sleep	Test if leader election works under unreliable environments	Success: All requests CRC	Leader election works.	unreliable_send_multi_leader_sleep.log
Very Unreliable Send	Default — send reliability of 50%, 100MPC	Test if Raft is able to work under very unreliable environments	Success: All requests CRC	Surprisingly reliability is maintained despite losing half of the packets. Followers loose 1.5 sends for every database commit, whilst leaders loose 12 sends for every commit. Performance highly degraded, but still insanely reliable. (Check logs for details)	very_unreliable_send.log
Ultimate Test	Unreliable Send & Multi Leader Sleep with client request interval of 1	Test if Raft works with leader sleeping, unreliable send and high loads	Success: All requests CRC	Overall the system works well, however, performance is significantly slower due to leader sleeping and unreliable send	high_load_unreliable_send_multi_leader_sleep.log

Table 1: Test Results