RAFT Consensus Algorithm Report

Alexandru Moraru (acm19)
Casey Williams (cjw19)

February 22, 2022

1 Implementation

As highlighted in the figures below, the `server.ex` module is the gateway to all the functionality that our solution provides. It handles the receiving of messages from either the client or the database itself and passes it onto 3 different modules that each have independent responsibilities.

1.1 `vote.ex` module

The `server.ex` module handles the `:ELECTION_TIMEOUT`, `:VOTE_REPLY`, and `:VOTE_REQUEST` messages by passing the state of the process and contents to the respective function in the `vote.ex` module. This module handles the change of a server from a `CANDIDATE` state to a `LEADER` if received enough votes, or handles the change of state to a `FOLLOWER` when the term of a request is greater.

1.2 `clientreq.ex` module

When the `server.ex` module receives a `:CLIENT_REQUEST` message, it passes it to the `clientreq.ex` module to check whether the request has already been committed or logged. In this module, we handled the ability of not adding an additional load onto the server if the client's request has already been made. If a client's entry has been `COMMITTED`, we then send the client a reply message to confirm that is has succeeded. If the client's entry has been `LOGGED`, we ignore their request in order to avoid logging duplicate entries. This is done in the hope that when the client requests for the entry again, it would or will be committed.

1.3 `appendentries.ex` module

This modules handles all the `:APPEND_ENTRIES_TIMEOUT`, `:APPEND_ENTRIES_REQUEST` and `:APPEND_ENTRIES_REPLY` messages that have been passed on from the `server.ex` module. Regarding the entries that is sent by the leader during an `:APPEND_ENTRY_REQUEST` message, we decided that the leader should send all the entries it thinks the followers is lacking instead of send it one entry at a time. This decision was taken with efficiency in our minds, in order to serve as many clients possible in as little time possible.

1.4 Summary of Typical Interaction

A typical interaction that occurs in our system is described as followed and highlighted by figure 1.

1. Client sends the server a `:CLIENT_REQUEST`
2. Leader broadcasts the `:APPEND_ENTRY_REQUEST` to all of its followers
3. Followers then add the entries sent to their respective logs
4. Leader then keeps check of commit indices of followers by sending empty `:APPEND_ENTRY_REQUEST` messages.
5. Leader already knows what entries to commit, hence sent their database a `:DB_REQUEST`
6. Followers then know what entries to commit, hence send their database a `:DB_REQUEST`.

2 Design Choices

We decided to implement a solution that would be simple to reason about as well as easy to debug. Our approach was to separate the main functionalities in the RAFT consensus algorithm: the leader election process, appending entries to their respective log and handling requests from the client. As a result, we created the modules `vote.ex`, `appendentries.ex` and `clientreq.ex` respectively to handle each functionality relatively independently.

3 Debugging and Testing Methodology

When starting on the project, we noticed that the `Debug.ex` module had level options, where our `Debug` messages would print according to the level configured in the `Makefile`. As a result, we made a decision to implement a system where each level of the debug option would represent a different case.

0. No debug output
1. Essential output such as core messages and state updates
2. Useful information such as messages sent and received
3. Extra information about actions a process is taking
4. Used for outputting calculation steps

This structure has made it easier for us to simply increase the debug level settings when encountering serious issues, in order to have a more in depth view of the situation.

At the beginning, we introduced artificial messages in order to trigger an `:ELECTION_TIMEOUT` event to test our implementation of leader election. This was simply done by restarting the election timer of a specific server of our choosing in order to force the respective server to be leader.

In addition, in order to test scenarios of how our system recovers if certain servers crashed, we edited the configuration file to add the number of the respective server we wanted to crash and the time it should crash at. Our `Raft.ex` module sends itself a reminder using the `send_after` function with a PID, which triggers the module to use the `Helper.halt` function in order to create an 'unexpected' crash' in order to test whether a new leader would be elected.

An issue that we struggled a lot with is the use of `Enum` functions for manipulating the server's state log instead of using the functions provided to us in the `Log.ex` module from the beginning. We realised that once a map reaches a certain capacity (around 32 items), it would not sort the items in order when converting it into a list for the `Enum` functions. This resulted in us unnecessarily modifying certain entries in the state's log due to a confusion of indices, and as a result this would be one the reasons cause our solution to crash handle high load.

4 Evaluating our System

4.1 5 Servers & 5 Clients - Normal

Under normal circumstances (no crashing and manageable load) our implementation performs as expected. The initial election quickly results in a sole leader being chosen at which point it begins to serve client requests from 5 clients.

In the log file `servers5_clients5_normal.log` we can see the log replication happening as the followers continue to receive append requests and update their own logs. We can see that the final state ends with each of the databases having done the same number of moves and also without any errors occurring elsewhere as the database receives the commands in the order and uniqueness expected.

For an example interaction see line 3108 where the leader sends an append entries request and it is immediately received by the follower on the next line. We can see on the subsequent line the follower's log before and after updating and then see when it replies to the leader indicating its updated index on line 3117.

4.2 5 Servers & 5 Clients - Leader Crash

We wanted to evaluate how our leader election functionality works when a leader would crash, hence resulting in followers lacking a server to handle the client requests. As you can see in the `servers5_clients5_leader_crash.log` file, server 3 crashes at around line 280892. Our system then responds to this event with server 4 receiving an `:ELECTION_TIMEOUT` that triggers the server to send `:VOTE_REQUEST` messages to other servers, as it has triggered a new election process by increasing the term to 3 as well.

The other correct servers (1 & 2) then received these `:VOTE_REQUEST` messages and as a result reply with a `:VOTE_REPLY` message back to server 4 by indicating that it is voting for them (server 4), as well as keeping their term value up to date by setting it to 3 as well.

As a result, this process of our evaluation highlights that our system can handle a key property that the RAFT consensus algorithm provide, that is a system should always managed to find a new leader provided that there are a sufficient number of correct servers available The system also manages to reach an end state that is correct as all the correct servers updated their respective databases an equal amount of times without causing any errors.

4.3 5 Servers & 5 Clients - Slow Append Reply

In this experiment we use the standard setup of 5 servers and 5 clients, however we make the followers wait 11ms before sending replies to append requests. This is because the leader's append requests timeout is 10ms and this effectively models slow replies.

You can see in the `servers5_clients5_slow_append_reply.log` that our implementation once again behaves as expected and the logs are successfully replicated. You are able to see the timeouts occurring more often for the leader but it still receives the replies as on line 9702, where it sent the initial request on line 2861.

We can see that the final state ends with each of the databases having done the same number of moves and also without any errors occurring elsewhere as the database receives the commands in the order and uniqueness expected.

4.4 10 Servers & 5 Clients - High load

Our implementation has some errors in this experiment due to leaders being overloaded and then some servers being behind on terms and then overwriting their database incorrectly. This is visible on line 365265 where server 4 incorrectly overwrites values in its database.

There are multiple leaders at one point in time during the experiment but they have different terms. We think this is because previous leaders are busily processing old requests whilst new leaders have taken over. One thing we are unsure of is how these old leaders continue to commit entries whilst other servers are replying to the new leader and committing its values.

In addition, as mentioned earlier in the debugging section of this report, we originally thought that the use of `Enum` functions instead of `Map` functions was the main error, but this was not the case.

We also tried altering the `Appendentries.ex` module to send followers entries to update their logs one-by-one, which would then cause our solution to pass this exact same test. However, even though the database would not throw any errors, this is was also due to the fact we would serve client requests at a much slower pace. As a result, it was not really testing the boundaries of our system - hence making it unfair to evaluate our system as good enough to handle a high load.

5 Diagrams

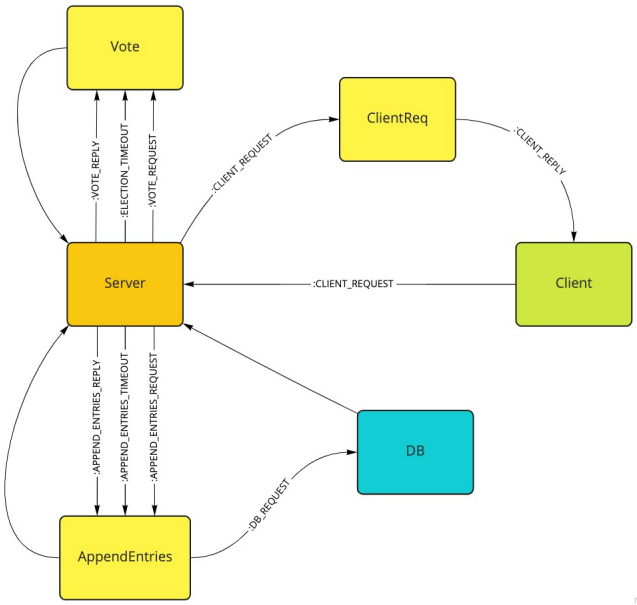

Figure 1: Overview of our modules interacting with each other

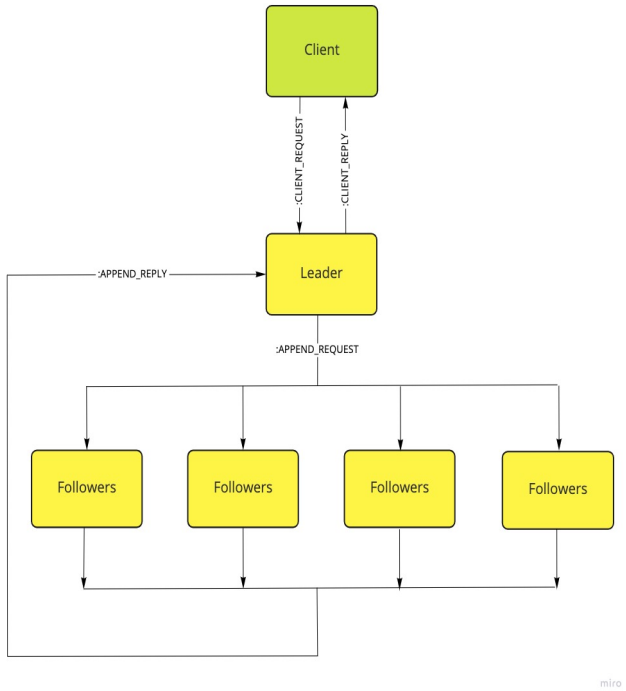

Figure 2: Overview of how states and client interact