C347 Raft Consensus

Miki Ivanovic

March 2020

1 Design

My project is a fairly straightforward implementation of the raft algorithm described in the paper. I did not use any sub-processes, rather I created modules to delegate server logic. I created a total of five modules, three for each server type and two for the remote procedure calls (RPCs).

Each server starts in **server.ex** which checks its current role, then calls the appropriate module to deal with its logic (i.e. if it is a candidate we call the candidate module). The way a server runs is essentially on recursive calls to **Server.next** so in this function we also implement general server rules, namely updating its **commit index**.

In general each server module has two main functions, **init** and **next**. The former is called when a server is changing its role; it is given the previous state of the server, updates it to match the server's new role, and returns this new state. The latter is responsible for message handling, state manipulation and returns a call to **Server.next**.

1.1 Timeouts

An important feature of this consensus algorithm are timeouts; a server needs to have some notion of the passage of time in between receiving messages. It would need to know how long it should wait to receive RPCs, and how much of that time remained. This led me to use a map parameter which would contain **total** timeout, and **remaining** timeout which would be passed as an argument in every call to a **next** function.

Initially these values are the same, and in each **receive** block we would default after the remaining timeout. When we receive messages, we recursively call **next** with an updated remaining timeout (using Elixir's in-built **DateTime** module), or the timeout is reset.

1.2 RPCs

Since request logic is shared between each role, is seemed appropriate to create a modules to deal with each RPC. The overall architectures is the same for both

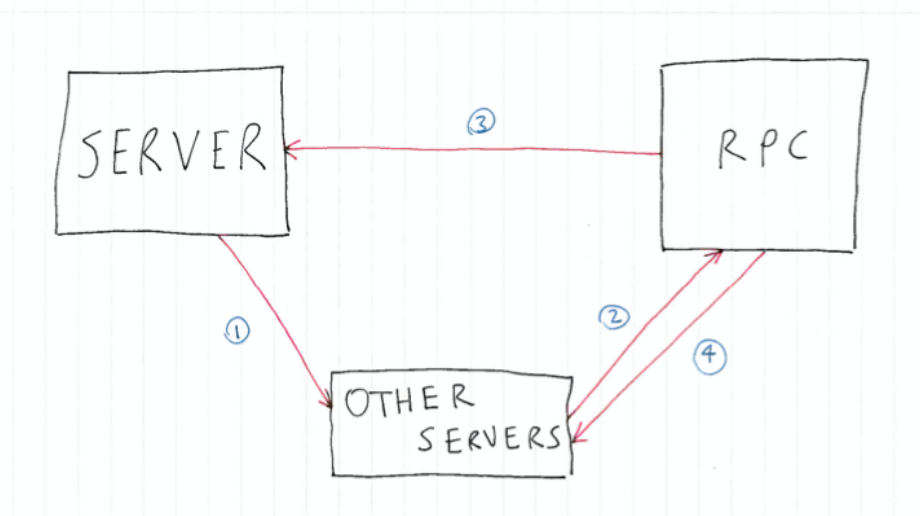

Figure 1: RPC requests

vote and append entries RPCs, and there are two types of messages in each one: requests, and responses.

1.2.1 RPC requests (Figure 1)

A server will send a RPC_ REQ to all the other servers (1), which will in turn each call the **request** function in the RPC module (2). The RPC will then process the message information by comparing the callee's state to the message data, and send a RPC_ REP response to the original server. It will finally return an updated state and timeout to its caller.

1.2.2 RPC responses (Figure 2)

When a server receives a RPC_ REP (1), it forwards the message data and its state to the **reply** function in the RPC module (2). The data is processed and an updated state and timeout are returned to the caller (3), which then calls **Server.next** (4).

1.3 Client requests (Figure 3)

When a client sends a CLIENT_ REQUEST to the leader (1), the leader will append the command to its log and send APPEND_ ENTRIES_ REQ to the other servers (2). The servers reply with an APPEND_ ENTRIES_ REP to the leader (3). After the majority of servers have committed the clients request, the leader sends a response (4) back to the client.

The challenge of client requests is knowing when to send a response back to the client, since the criterion is that the command has been applied to the state

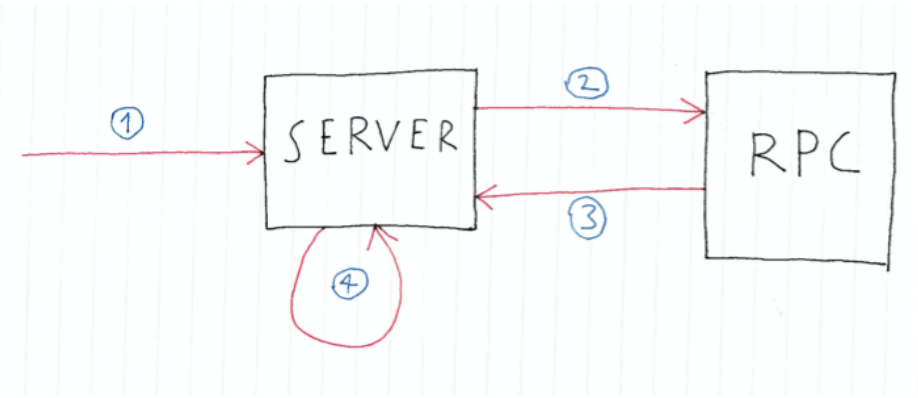

Figure 2: RPC replies

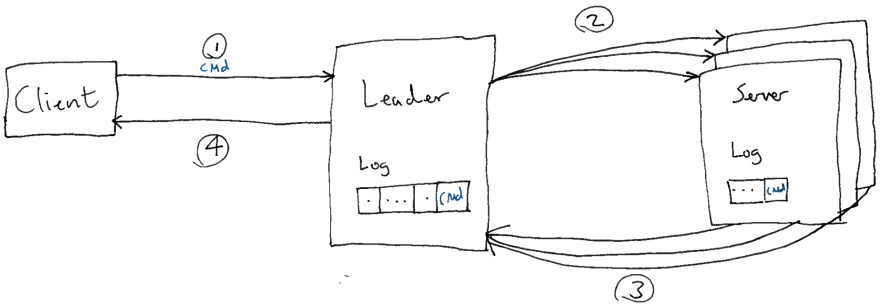

Figure 3: Client request.

machine. This condition is satisfied when we know that the leader's commit index is at least as big as the command's index within the state log, so my idea was to keep track of requests within the leader's state with a new queue **client_requests**.

Each item in this queue is a tuple containing the index of the request in the log and the client's id. Upon each call to **Leader.next**, we keep popping elements from the queue and sending responses to clients as long as the above condition is satisfied. From the nature of this abstraction we are guaranteed that the queue is ordered, and so this method is safe.

2 Methodology

I wanted to make it easy to identify problems with the code, so my main "strategy" was to divide and conquer - logic would be practically delegated to helper functions. This would help reduce clutter within functions, and allow logic to

be used elsewhere. The idea was that this would help with troubleshooting as well since bugs could be isolated quickly to individual behaviours.

My actual debugging was unfortunately very inefficient, it boiled down to halting the system and printing server states at fixed time intervals, and outputting server states. This was done with the in-built **Process.send_after/4** function, and with **Monitor.state/3**.

3 Analysis

3.1 Normal operation

In an ideal system, there are no server problems and leader election will run smoothly. Here every client request seen by the leader should be applied to all state machines.

This means that over the programs lifetime, even if there are multiple leaders, the sum of the total of the client requests seen by each leader should be, over time, equal to each of the db updates done by each server. An example of this can be seen in **output/5_servers_5_clients**.

This should still work irrespective of the number of servers and clients, so I also tested the the two cases where these values are different, specifically for 5 servers and 9 clients, and 9 servers with 5 clients.

3.2 Varying election timeout

The Raft paper explains that for leader election to work, **broadcast time** (the average time to send messages to each server in parallel) should be an order of magnitude smaller than election timeout. It also says that election timeout should be at least an order of magnitude smaller than the **MTBF** (average time between server failures), but since we can directly control **MTBF** in this practical there's no need to worry about the upper bound for election timeout.

Elixir's internal wall clock time of the **send/1** primitive is a few nanoseconds, therefore an election timeout of a couple of milliseconds should certainly satisfy the given criterion. This means that leader election works, and certainly in all of my experiments, a leader was elected eventually. In the actual output which measures client requests seen, there may be periods where no requests are seen which is due to the client randomly guessing who the leader is.

3.3 Crashes

During runtime, if a server crashes (I arbitrarily have my crashed servers recover after one second) then it will not receive log instructions from the leader during this time period (I call it the "crash window"). This means we don't expect to see any db updates from that server in that time, but as soon as it recovers then it will immediately apply all the db updates it missed during the crash window. This is because on the next append entries RPC our server receives from the leader, it will update its log to match that of the leader, and apply

of all of these db updates on its next iteration. We see this with the examples **output/1_crash**, and **output/2_crashes**.

An interesting case is when a majority of servers crash in the same time frame. In this case the leader can not guarantee that its log entries have been committed by its followers, and therefore won't update its own commit index. This means that for this window, the leader will see requests coming from the client, but can't apply them to its state machine since these requests aren't applied to the majority of servers. Under this test, my program outputs **output/majority_crashes**.

3.4 Varying the append entries timeout

In my code, the leader does not actually send append entries RPCs to its followers in parallel, rather it does so sequentially and waits for each follower to reply before sending the next one. To prevent its followers from timing out, the leader can only wait for a fixed period of time to receive a reply - it must keep sending RPCs to preserve its role. This timeout is the **append_entries_timeout**.

Under normal circumstances each server will reply almost instantly so the leader doesn't really have to wait, however when one of its followers crashes, it will always wait for this timeout to expire. For example, if append entries is much larger than election timeout, a normal run shows expected results: **output/large_append_timeout**.

However when one server crashes, the leader will wait for the append entries timeout to expire before sending RPCs to its other followers. If append entries timeout is bigger than the timeouts of its followers, then we will have a new election. The same thing keeps happening to the new leaders, and essentially loops until the crash window expires. An example of this scenarios is **output/large_append_timeout_crash** in which is set the append entries timeout to twice that of the election timeout. There are no db updates in this window.

Similarly, if the append entries is equal to the election timeout and there are two crashes, then get stuck in a leader election loop until the servers recover. This is shown in **output/same_timeout_2_crashes_overlap**. This only happens when the crashes overlap since **output/same_timeout_2_crashes** shows expected crash output.

This leads me to think that there is some kind of inverse relationship between the two timeouts, correlated with the number of servers in the network **N**, and that to avoid this issue we apply a constraint on the append entries timeout:

$$\textbf{broadcast\_time} << \textbf{append\_entries\_timeout} < \frac{2}{N}\textbf{election\_timeout}$$

Note that the two factor appears since the maximum timeout a follower can have is twice that of the election timeout.