The structure of the system

The first part of the system is leader election. Each server and the bound database are spawned processes which interact with each other within the system by sending and receiving messages. Each server has a unique election timeout which is randomly assigned to them between a chosen amount of time and its double. The server in which its election timeout elapses first becomes a candidate and broadcasts Vote\_request to the other servers. The other servers then answer with Vote\_reply which tells the candidate whether it’s approved to be a leader or not. There is at most one leader per term and the voting servers decides to vote for the candidate by considering whether the requested server has a more complete log, or a higher term value.

The servers can be in one of the three states: Follower, Candidate, Leader. Every server is by default a follower in the beginning, and when a server’s election timeout elapses it becomes a candidate and broadcasts requests for vote. When a follower becomes a candidate, it’s term value is incremented indicating that the new term has started. When the other servers receive vote\_requests, they vote either yes or no to the candidate and the candidate becomes a leader if it has gotten the majority of the votes. If there were multiple candidates at a given term, the votes might be split between them and the majority vote might not be achieved by any of the candidates. In this case, a new term is started when one of the server’s timeout elapses and a new election takes place. Split votes are effectively avoided by randomly assigning election timeout values to the servers.

The next part is executing received commands from clients and sending back the result. The leader chosen from leader election communicates with clients. When it receives commands from the clients, it appends them in its log. It then broadcasts Append\_Entries\_Request asking the followers to append the new command. When the leader knows that the new command is replicated by the majority of the servers, it commits the command and the command can be executed by its state machine. The result from this process is then sent back to the client. One thing to note is that the leader regularly broadcasts empty append\_entries request which is called heartbeat to inform the followers that it is alive.

# Design and implementation choices

1. Vote Request, voted\_for and curr\_term fields within Server state

The format of vote request broadcasted to other servers during leader elections was designed as a list: [pid of server, last term of its log, last index of its log, its current term].

This designed was chosen to pass all the required information compactly. The pid of the server indicates where the voting servers should answer back. When the voting servers decide whether to vote yes or no, they use the given current term, last term, and index value of the requested server. If the requested term has a higher term value, the voting servers have to update their term values to match it and, in this case, they approve the requested server to be the leader of the term. If the voting servers have the matching term value as the requested server, they have to check if the requested server’s log is more up to date compared to theirs. Which log is more up to date can be determined by checking which log has a higher last term value and if they have the same last term value, the last index value is checked for determining which log contains more information from the last term.

A server has multiple fields attached to it and “voted\_for” and “curr\_term” are one of the essential fields in the process of leader election. “voted\_for” field indicates whether the server has voted during the term and who it gave its vote if it has voted. This field was essential as it prevents servers from voting multiple times in the same term. “curr\_term” field was compared with the passed current term value of the requested server to see which server is up to date.

1. Key value “success” in append entries request and reply

Every append entries request and reply is a map that contains key values of “success”, “leader” or “follower” indicating where this message comes from, “old\_entry” and “new\_entry” giving information regarding the leader’s log for its replication, and “last\_index” telling the index of the successfully replicated log within a follower. Particularly, “success” key plays an important roll which is telling the recipient what this request or reply is about. One value it can contain is “heartbeat”, which is regularly sent by the leader to its followers to inform them that the leader is operating. When the followers receive the heartbeat, their append entries timer is restarted. When one of the followers’ append entries timers elapses as the leader has not sent any request to them for a while, they assume that the leader process has crashed and starts a new leader election. Another value “success” can contain is “append” which tells the follower that the request also contains the leader’s log information so the followers can use them to replicate the leader’s log. The method implemented for replicating log uses an induction step. The leader initially only sends the new received command from the client and the index and term values of the previous entry in its log. The index and term values here are compared with the follower’s last entry in its log. If they match, it shows that the follower has had an up to date log and the new command can be successfully appended to its log. However, if the given index and term values do not match, it means that the logs are inconsistent. In this case, the follower sends a reply to the leader with the value of “success” key being “inconsistent”. Then the leader sends a longer “old\_entry” value which also contains one more previous entry of the leader’s log. This step is repeated until the follower’s log becomes consistent with the leader’s log and then the new entries stored in “new\_entry” are appended to the follower’s log.

1. “match\_index” in the leader’s state

As the leader sends out its log for the follower’s to replicate it, the leader receives the last index of each follower’s log when the follower succeeds in replicating the log. These indices are stored in a field called “match\_index” of the leader, which can be used to determine whether a command inside its log can be commited, meaning that it’s safe to execute in its state machine and send the result back to the client that gave it the command.

# Debugging and testing methodology

For debugging and testing the implemented functions, a set of print statements using “IO.puts” and the “assert” function from “Exunit.Assertions” were mainly used. Especially, the field “server\_num” of each server was used effectively to ease the debugging process by printing out the server’s number directly instead of its pid which may not be the best value to indicate each server during the debugging process. An example of the print statements is shown as below:

Server2: election started

Server2: current\_term: 1

Server2: current\_role: :CANDIDATE

Server2: number of votes: 1

Server2: sending vote request with msg: [#PID<0.147.0>, 2, 0, 0, 1] to #PID<18308.147.0>

Server2: sending vote request with msg: [#PID<0.147.0>, 2, 0, 0, 1] to #PID<18309.147.0>

Server3: vote request received from Server2

Server1: vote request received from Server2

Each line starts with the operating server’s name and then a useful information which shows what each process is doing. This particular example shows a part of a leader election which Server2 started and broadcasted a vote request to the other servers: Server1 and Server3. It can be observed that Server1 and Server3 have received the vote request and they continue by determining whether to approve Server2 to be the leader for the current term or not.

“Assert” function was another useful tool to make sure that data is handled and transformed as expected which can sometimes be very difficult to be spotted. An example of this would be:

s = if msg.success === "consistent" do # assumes that the follower successfully replicated leader's log

assert msg.last\_index === map\_size(s.log)

The if statement in the beginning indicates that the message delivered to the leader by a follower shows that the leader’s log is successfully replicated. If this is true, the last index of the follower’s log delivered should be the same as the the last index of the leader’s log obtained by computing the leader’s log size. This is checked by the assert function and if this assert fails, an error message would be shown.

# Evidence that my system works

My system currently does not allow operation with spawned clients from the make file given as the function that handles committing entries based on whether it is successfully replicated by the majority of the servers in the system is not implemented. However, the functionality of the leader knowing the last index of the successfully replicated logs of the followers is implemented within the “match\_index” field of the leader’s state. Therefore, this implemented functionality can be used to implement the commit functionality.

The leader election and log replication were tested thoroughly and the results of these tests can be found inside the directory with the name “output\_leaderElectionUpdated” and “output\_replicatedLog”. The other output files were used for debugging purposes as it was easier to visualize what was happening during each step by printing them out.