60009
Distributed Algorithms

Coursework

RAFT CONSENSUS

Submission Date: 21 Feb 2022

NAME

ALICIA LAW JIAYUN (AJL 115)

01105518

YE LIU (YL 10321)

02007614

1. Design and Implementation

The raft consensus algorithm is designed around the *server.ex* module, which handles all communications from servers, clients and databases. For all messages, pattern matching is used to distinguish between the different tasks requested. *Server.ex* will then call upon functions in other relevant modules and execute these accordingly. As server states (eg. roles, election terms, logs etc.) are constantly changing, guards (i.e. "when" keywords) were implemented as a form of efficient checks, to ensure only up-to-date requests are processed.

1.1 Leader Election

The raft consensus can be split into two distinct operations, leader election and log replication. The implementation of leader election was done using 2 modules:

- ***Server.ex:*** Handles messages sent between servers. Calls upon vote.ex to action relevant requests. Table 1 maps the interaction between messages received in *server.ex* and functions called in *vote.ex*. It also summarises when role changes are triggered.

- ***Vote.ex:*** Actions all requests pertaining to leader election.

Table 1: Interaction between Server.ex and Vote.ex

	Message received by Server.ex	Function called in Vote.ex	State Change (if any)	
			Before	After
1.	:ELECTION_TIMEOUT	*receive_election_timeout:* ➔ Broadcasts **#2**	Follower	Candidate
2.	:VOTE_REQUEST	*receive_vote_request_from_candidate:* ➔ Sends **#3** if server voted.	-	-
3.	:VOTE_REPLY	*receive_vote_reply_from_follower*	-	-
		become_leader ➔ If reaches majority, broadcasts **#4.**	Candidate	Leader
4.	:LEADER_ELECTED	*receive_leader*	-	-
5.	Any message from server with a larger term	*stepdown*	Leader/ Candidate	Follower

Figure 1 illustrates how the above messages in Table 1 flow between servers during an election period, with the table on the right showing the corresponding messages sent at each stage in sequence, along with a brief description of the action executed.

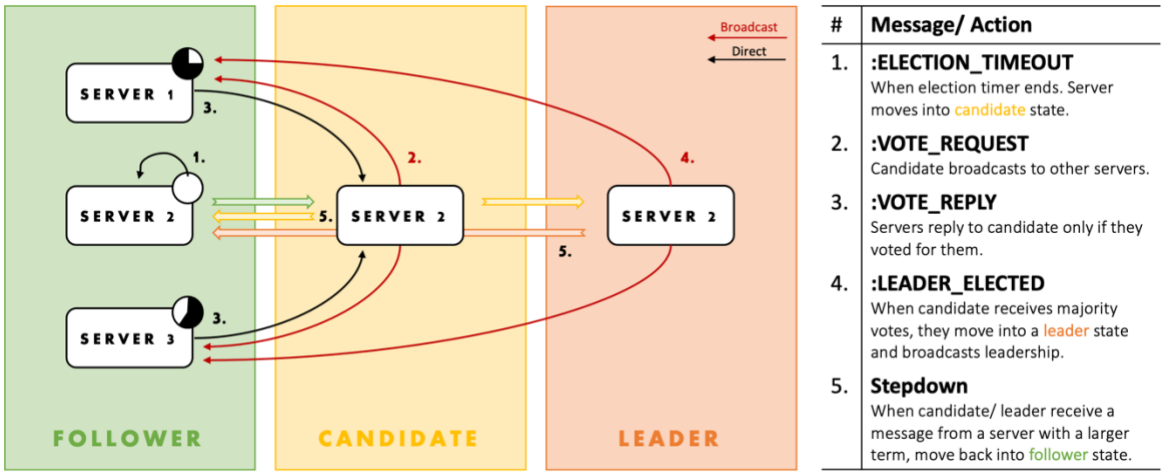


Figure 1: Normal flow of messages during Leader Election.

1.2 Log Replication

Following leader election, the next operation is log replication. The implementation for this section involves 5 modules, the main module being *appendentries.ex*:

- ***Server.ex:*** Used to handle messages sent to servers. Calls upon the *clientreq*, *appendentries*, *database* and *client* modules. To see the interaction between the messages received in *server.ex* with these modules, see Table 2.
- ***Clientreq.ex:*** Used only by the leader server to service requests from the client.
- ***Appendentries.ex:*** Actions all requests pertaining to log replication, such as sending, receiving, storing and committing requests, as well as sending append entries replies.
- ***Database.ex:*** Used by database to communicate with their respective servers. Replicates logs into the database and responds to server regarding status of replication.
- ***Client.ex:*** Used by clients to communicate requests with the leader server.

Table 2: Interaction between Server.ex with Clientreq.ex, Appendentries.ex, Database.ex and Client.ex

	Message received by Server.ex	Triggers	
		Function	Module
1.	:CLIENT_REQUEST	*receive_request_from_client*	*Clientreq*
		send_entries_to_followers → Broadcasts #3.	*Appendentries*
3.	:APPEND_ENTRIES_REQUEST	*storeEntries* & *receive_append_entries_request* → Sends #5.	*Appendentries*
5.	:APPEND_ENTRIES_REPLY	*receive_append_entries_reply_from_follower* → Send #6. *send_entries_to_followers* (if repairing log) → Send #3.	*Appendentries*
6.	:DB_REQUEST	*send_reply_to_server* → Sends #7	*Database*
7.	:DB_REPLY	*receive_reply_from_db* (only for leader) → Broadcasts #8 → Sends #11	*Clientreq*
8.	:COMMIT_ENTRIES_REQUEST	→ Sends #6.	-
11	:CLIENT_REPLY	*receive_reply_from_leader*	*Client*

**Note: numbering corresponds to labelling in Figure 2.*

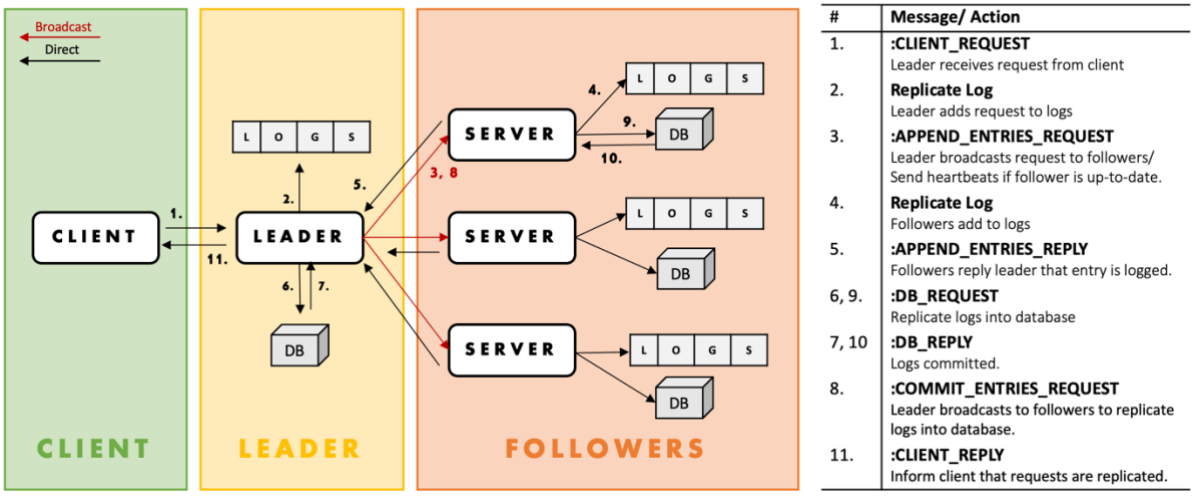


Figure 2: Normal flow of messages during a typical client request.

Figure 2 illustrates the normal flow of messages during a typical client request, with the table on the right showing the corresponding messages sent at each stage in sequence, along with a brief description of the action executed.

2. Debugging and Testing

During development, we worked on the algorithm one feature at a time, each time performing "unit testing" before moving on. For instance, while implementing the leader election algorithm, we started by implementing vote requests, testing its correctness in isolation, before implementing vote replies and leader selection. To debug and test the program, we used the following strategies:

1. **Start small.**
   During development, we set the number of servers to 3, number of clients to 1, and the maximum number of client requests to 1. Only when the system proved to be handling the requests well, did we begin to increase the complexity and test edge cases (see #3).
2. **Debug with the IO module.**
   We used IO.puts and IO.inspects to output and compare the server's state before and after it receives a certain request to ensure the server states (such as logs during append entries, or roles during election) were updating correctly. During which, we print only essential messages to avoid information overload.
3. **Simulate unexpected events**
   To ensure we had a robust system that worked correctly in all scenarios, we rigged our system (eg. using timeouts, sleeps, node exits) to trigger certain behaviors. See Section 3 for more detail.

3. System Evaluation

As mentioned in Section 2, several experiments/ simulations were done to test the robustness and performance of our system. The below subsections explore these in greater detail.

3.1 Baseline

We define our baseline with the following parameters:
Number of servers: 3, Number of clients: 3, Max client requests: 1000, Timelimit: 15000ms

The output file *baseline.txt* shows that 695 client requests have been processed by servers. The same parameters are assumed for later experiments if not specified when evaluating a system's performance.

3.2 Leadership Change

Leadership change was simulated in 2 ways – (i) crashing and (ii) slowing down leaders.

3.2.1 Crashed Leaders

- Output file : crashed_leader.txt
- Number of servers : 4
- Aim : Trigger instant leadership change. A "stalemate" should also occur.
- How : Process.exit() to kill leader servers once elected.

As shown in the output files, whenever a leader process dies, a new election term starts. This is because leaders cannot send their followers heartbeats to prevent an election timeout. This process repeats (Server 3 is elected in Term 1, Server 1 in Term 2). However, once half the number of servers in the system is killed, the remaining servers will be stuck in a perpetual election cycle (see Server 2 and 4, where they constantly take turns being candidates). This is because the system will now never reach a majority vote to re-elect a new leader successfully (3 votes are needed but 2 servers remain).

3.2.2 Slow Leaders

- Output file : slow_leader.txt
- Aim : Observe leader stepdown.
- How : Process.sleep(100) applied before every append entries request.

While 3.2.1 shows the re-election implementation is correct, it does not show the leader stepdown process. By slowing a leader, this gives followers the opportunity to reach a timeout and trigger an election. In *slow_leader.txt*, Server 3 is seen to start a new election as its leader (Server 1) was too slow. As Server 3 now runs for candidacy and has a higher term, its old leader (Server 1) steps down upon receiving the vote request. This clearly shows that a slow leader will be replaced and revert to a follower state.

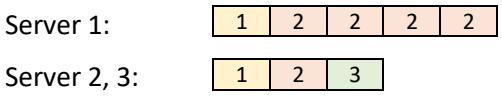
3.3 Split Votes

- Output file : split_votes.txt
- Aim : Re-elections should occur until a clear leader is elected.
- How : set a short election_timeout_range = 25..50ms

In *split_votes.txt*, we find that all servers run an election almost at the same time at the start and vote for themselves. Since each server is allowed to vote for at most one candidate in a given term, a split vote happens in Term 1 and 2. However, as raft uses randomised election timeouts to ensure that draws can be quickly resolved, split votes can be seen to end by Term 3, where Server 2 is elected leader. We also observed by increasing election timeout ranges, split votes rarely occur, illustrating the importance of setting reasonable election timeouts.

3.4 Inconsistent Logs

- Output file : log_repair.txt
- Aim : Observe log repair. Server 1 cannot be leader as part of raft's safety property.
- How : Initialize logs at the start for each server such that they are inconsistent.



Given the log initialisation, Server 1 should not be elected as the leader since it has a lower LastLogTerm. This is consistent with what was observed in log_repair.txt, where Server 2 is elected as leader.

When the leader (Server 2) received a client request, it first deleted Server 1's extraneous entries (index 3-5). It then merged the missing entries with the leader's remaining logs (index 3 and also new log at index 4). Server 1's log is now the same as the leader's.

3.5 Slow Append Entries Reply from Follower

- Output file : slow_aereply_1.txt and slow_aereply_2.txt
- Aim : Observe system performance during slow follower replies.
- How : Use Process.sleep (80) before sending :APPEND_ENTRIES_REPLY.
- Append_entries_timeout : 50ms in slow_aereply_1.txt,
  100ms in slow_aereply_2.txt

According to the raft algorithm, leaders cannot replicate entries into their database until majority of followers have replied, indicating that they have committed the entry to their logs. Hence, slow replies severely impact system performance. This is shown to be true in *slow_aereply_1.txt*, where servers process only 5 requests in 15 seconds, a significant decrease compared to our baseline of 695 requests. This is because more requests are being delivered to the servers than what can be processed, causing a backlog in the server's mailbox.

Hence, by increasing the append_entries_timeout to 100ms, larger than the reply delay of 80ms, the system improves from before, processing 12 requests in 15 seconds – a two-fold improvement (see slow_aereply_2.txt).

3.6 Slow Reply from Leader to Client

- Output file : slow_creply.txt
- Aim : Observe how leaders handle repetitive client requests
- How : Use Process.sleep (30) before sending :CLIENT_REPLY.

To ensure that leaders do not end up replicating the same requests into their logs, the clientreq.ex module uses check_req_status (line 68) to check the status of each request and processes it accordingly. A request can be classified and dealt with in 3 ways:

- NEW_REQ : A new, unseen, request.
  Leader to append and send an append entries request to followers.
- COMMITED_REQ : A request committed to logs but not replicated onto the database.
  Leader to ignore the request.
- APPLIED_REQ : A request that has been replicated to the database.
  Leader to send a :CLIENT_REPLY only.

As shown in slow_creply.txt, the system behaves correctly. For instance, Server 3 (leader) received request {1,1} three times. The first time (Line 94), the leader recognizes it to be a :NEW_REQ and processes it. However subsequently, when it receives {1,1} again in Line 100 and 337, the leader recognizes that it is an APPLIED_REQ and "just replies" the client.

3.7 Stress Test

- Output file : heavy_load.txt and heavy_load_300s.txt
- Aim : Model system performance under increasing load.
- How : Increment the number of servers from 5 to 10, handling 5 clients and 5000 request

Figure 3 shows how our model performance degrades with increasing number of servers. By increasing the number of servers from 5 to 10, the number of client requests handled halves.

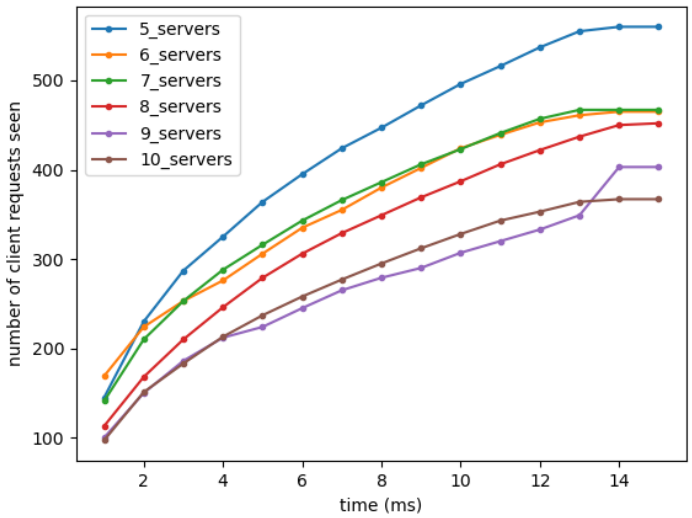


Figure 3: System performance with increasing number of servers. Values obtained from heavy_load.txt.

Finally, we ran the system for 300 seconds with 10 servers. The system performance is seen to decrease significantly over time, processing only 2.1 requests per second in the last 10s, compared to the 30.7 requests per second in the first 10s. Overall, *heavy_load_300s.txt* shows the system is able to process 1460 requests in 5 minutes without crashing.