Distributed Algorithms Coursework Report

Yuchen Niu (yn621) and Zian Wang (zw4821)

February 2022

0.1 Structure and Connectivity

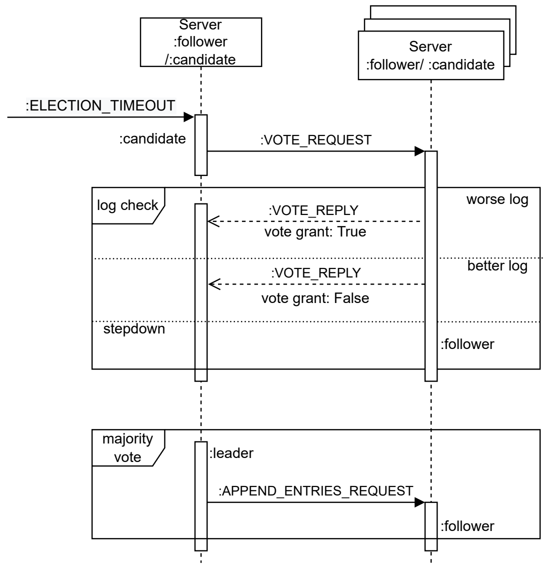

Figure 1: Vote procedure. Roles beside activation is role change.

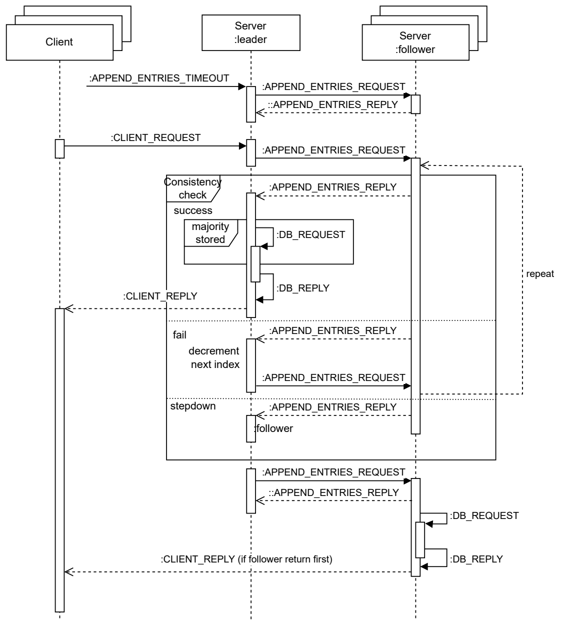

Figure 2: Append entries and client requests procedure.

0.2 Implementation

Our implementation is based on the skeleton provided. We handle voting, appending entries and client requests of the Raft algorithm. We do not write any replacements of existing code to avoid side effects on the correctness of the implementation. After servers are spawned with their replicated state machines respectively, every server has its own state as a keyword map. The state contains information about the server and its log, as listed in the Raft paper. Since all servers are initialised as followers and there is no leader, they wait for election signals before accepting client requests.

Vote The follower or candidate who is received a election timeout assumes the leader has crashed and starts a new election. The server increments the current term (election) and restarts the election timer. Then, the server changes its role to candidate and votes for itself because it does not aware of the log states of other servers. The vote request consists of the PID of the sender, current election, last term and last index of sender's log. Due to side effect of the code in the skeleton, current election is essentially the current term, which is used to drop old messages. To maintain the consistency with existing code, we use current election, instead of current term, in the vote implementation.

After receiving the vote request, the receiver checks whose log is likely to contain all committed entries by comparing the last term and last index. If the receiver has not voted for anyone and its log is less complete, the receiver votes for the sender and restarts election timer. While handling the vote request, the receiver steps down to follower if the sender has a higher current election than that of the receiver. In the Montresor's slide, the receiver simply steps down and does not vote. However, one candidate will never receive the majority votes in their implementation if other servers do not vote. Therefore, we send a vote reply back to the sender to increment the vote number and the voted-for state will be reset in the following heartbeat.

After receiving the vote reply, the vote is appended to voted-by if the current election is matched and the voted-for is the PID of receiver. Once a candidate received the majority of votes, it becomes leader. It then cancels its own election timer and re-initialises next indices and match indices. After that, it broadcast heartbeat messages to establish authority and prevent new elections.

Append Entries The leader broadcasts periodically append entries requests that carry no log entries as heartbeats to maintain their authority. The heartbeats also consist of the commit index, last index, and last term of the leader. Therefore, the commit indices of followers keep synchronised with that of the leader, as long as followers have the same logs as the leader's log.

Unlike the implementation in Montresor's slide, the leader sends all entries between the next index of the follower and the last index of the leader's log to the follower for efficiency. When a follower receives append entries requests from the leader, it checks for the consistencies, whether the follower's log is long enough and the previous terms of the leader and the follower are matched. If the consistency check is failed, the follower sends the response to the leader. The leader will decrement the next index of the follower and try to append entries again. If consistency check is successful, entries from the next one of the previous index to the end will be replaced by entries from the leader for simplicity, since the leader forces the followers' logs to duplicate their own in the raft.

When the leader receives a response with a match index from a follower and the consistency check succeeds, the leader updates the next index and the match index of the follower. Once the leader knows that an entry has been stored on the majority of servers by comparing the match indices and leader's commit index, the leader will increment its commit index and inform the followers in the following append entries requests. If a leader crashes and recovers later, it will receive append entries requests from the new leader with a larger term. The crashed leader reverts to follower, updates its term, and processes requests normally.

Client Requests Clients try to send commands with a unique command ID to the leader after they spawn. In the beginning, the receiver drops the client requests silently due to no leader. Clients will retry the requests until a leader is elected. If a follower receives a client request, it informs the client about the leader. Henceforward, the client only sends requests to the leader unless the leader crashes.

If the leader receives a client request and the entry does not include in the leader's log, the leader appends the new entry to its log and broadcast append entries requests to followers. Otherwise, the client's request is dropped silently. For duplication check, we enumerate through the entries of the log and stop once an entry matches with the new entry. This might not be as efficient as MapSet. But we do not want to change the data structure of the log since it will involve plenty of work.

When a server knows its commit index is larger than its last applied index, it will increment the last applied index and send the next log to its replicated state machine. Instead of waiting for the state machine to reply, the server can handle other messages. As a result, we must include some client information, like command ID and client PID, in the database reply. However, we observe slight performance improvement. Clients only communicate with the leader in the Raft paper. In our implementation, the clients only send requests to the leader and every server can reply the command result to the client once its state machine finishes the task. This reduces the complexity of communication between the leader and followers and does not affect the outcomes in this particular case.

0.3 Debug

Methodology The test part mainly depends on the debug module given in the source code. In addition, we introduced some flags to control the debug process. In general, we strictly follow the coursework specification: organize the debugging information into different options based on various circumstances, as well as divide debugging information into different levels. It is because we did it throughout the coding process that made our coding process relatively easy. However, during the debugging, due to each process having its unique state at each time, we found it really intricate to debug a distributed system. Although we tried some debugging tools including Erlyberly, the majority of the time, we still debug the system in a very traditional and inefficient way - print the debug information module by module.

Debug Implementation In terms of debugging implementation, one of the most important flags is the DEBUG_MODE defined in Makefile. It directly controls which branch the code will execute in the start function - Raft.start. The case statement is adopted which makes sure the future user can easily scale the test case - just add a unique keyword in Makefile and create a corresponding test block in the case statement. Another useful flag is named SETUP, which is used to specify which customized configuration will be used. The customized settings include a maximum number of requests that each client can send and specify the crashed server... With both of the two flag, as well as the fully organized debug module, user can easily customize the debug execution and configuration and print the useful information as they want.

0.4 Test

After we debug module by module, we use dozen of test case to make sure the system correctly assemble as a whole. In this step, we focus more on various configurations. The test result you can see the corresponding output file in the output folder. (Note: blank label means same to the default)

	client timelimit	max client requests	client request interval	client reply timeout	election timeout range	append entries timeout	crash servers
Default	60000	5000	5	50	100∼200	10	
long_append_entries						1000	
low_election_timeout					10∼10		
low_vote_timeout					30∼70	50	
high_requests		50000	2				
server1_3_crash							1 =>1500 3 =>5000

0.5 Evaluation

We evaluate the performance of our implementation under interesting scenarios. First, we stress the system with 5 servers and 5 clients to testify the maximum requests the system can handle. Each client sends 5000 requests in total with an interval of 5 milliseconds. On the test platform with Intel I7-11800H CPU, the servers receive and handle approximately 4000 requests, respectively.

To verify the leader election procedure works, we let two servers, server1 and server5, crash after 5 seconds. The rest of the parameters are not changed. Since the leader election, in the beginning, is not deterministic, crashing 2 servers can increase the probability of leader crash. From the result, we observe that server1 (follower) and server5 (leader) crashes after 5 seconds and server3 becomes the new leader. Since the system works as usual before 5 seconds, server3 simply has a smaller election timeout rather than a more complete log. After servers crash, we observe that the remaining servers keep handling the client requests until the maximum time is reached. However, the alive servers only handle about 3000 requests since the leadership shift take some time. If the number of crashing servers increases to more than half of the total servers, our system fails. Because the majority is a constant in our implementation, the new leader will be never elected if a candidate does not receive votes from the majority of servers.

To further explore the system performance, we conduct experiments by adjusting append entries timeout and election timeout range. Without changing other parameters and server crashes, we set the append entries timeout to 1 second. As expected, every time a new leader is elected, it cannot maintain authority due to no heartbeats. In our implementation, the heartbeat is append entries request and relies on append entries timeout. Therefore, the leadership keeps shifting until a client sends a client request to the leader speculatively. Then, the leader broadcasts its log and followers inform clients about the leader. After that, the system works as normal. Surprisingly, even the system wastes 8 terms, the servers still handle around 3500 requests. So, the election procedure is not time costly. We also adjust entries timeout range to 10 milliseconds, which is as same as the append entries timeout. Under this setting, the leadership changes through the entire workflow. The final term is 734. However, servers still handle approximately 1500 requests since the clients send requests in a round-robin way.