Raft: Understandable Distributed Consensus

Dep. of Electrical and Computers Eng., Faculty of Engineering, University of Porto, Portugal

A. Aragão, A. Matos and M. Marques

I. Algorithm's overview

In Raft, each server can be, at a given time, in one of three states: leader, candidate or follower. Raft works by having one distinguished leader. The leader sends continuous messages - heartbeats, to the other nodes. If a follower does not receive any heartbeat after a certain time , which results in election timeout, an election is started. When the leader receives a command from the client, it adds a new entry with that command into its log. The leader decides when is it safe to apply a specific command to the state machines, i.e. to commit a specific entry, replicating its log to the other nodes.

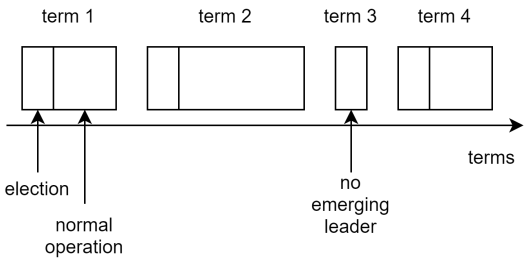

Fig. 1: Visual representation of the concept term

An important aspect is the concept of term. A term is an interval of time that starts with an election. If a leader is elected, then that server will be the leader for that term. If no server wins the election, i.e. no leader is elected, then that term will finish with the starting of a new election.

Each server saves its own current term, CurrentTerm, increasing its value monotonically and communicating it in each message it sends. If a server realizes that there's a server with a current term greater than his, it updates its own to the greatest current term found. By doing that, the server, whether a leader or a candidate, immediately becomes a follower.

A. Leader Election

Raft is based on two remote procedure calls, referred from now on as RPCs. In one hand there is AppendEntries RPC which is initiated by the leader periodically. Moreover, it replicates the long entries, if there are the need for it, and it provides a form of heartbeat. On the other hand, there is RequestVote RPC which is initiated by a follower to start an election.

If a follower does not receive any heartbeat, after a certain time, known as election timeout, it starts an election. To do so, it increments its own term and becomes a candidate. It votes for itself and request a vote from the other servers. Each server can only vote for one candidate, at most, in a term. There are three outputs from an election.

Firstly, the candidate receives votes for the majority of servers during the term, wins the election and becomes the leader. In the first heartbeat it sends, it will let other servers know its role. Secondly, while waiting for answers, the candidate receives an AppendEntries RPC from another server. If the other server's current term is greater, then the candidate returns to follower. Otherwise, the candidate rejects the RPC. Lastly, there is no majority in regards to the votes resulting in no candidate winning the election.Candidates will time out and start a new election with a bigger term.

B. Log Replication

As stated, the leader receives the commands from the client. When it receives a command, it adds a new entry with that command into its log. In the next AppendEntries RPCs, it replicates it.

Each entry contains the respective command, the term in which it was received, and an index that identifies its position in the log. These are used to make a consistency check: when sending an AppendEntries RPC, the leader sends the index and the term of the entry in its log that precedes the new entries that it is sending. The follower will try to find on its own log the respective entry, and if it does, it means the log of the follower is updated. Otherwise, it refuses the new entries. When the logs are inconsistent, the leader will start the process of replicate its own log until both logs are consistent. Furthermore, this topic it will be explained further.

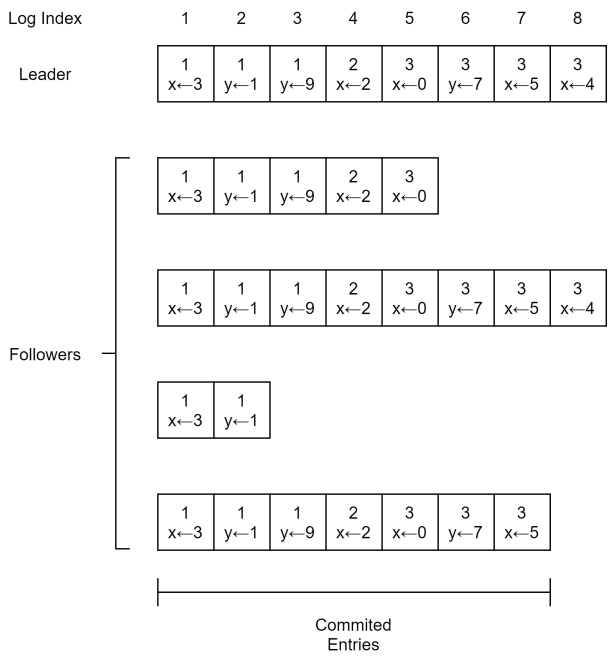

Fig. 2: Log replication example

C. State Machine

The leader decides when is it safe to apply a specific command to the state machines, i.e. to commit a specific entry. It commits a certain entry after replicating it on the majority of the servers. When doing this, it also commits all preceding entries in its log. The AppendEntries RPC also contains the commitIndex, which corresponds to the last commited index. The servers will acknowledge this, and apply the command to their own state machines.

II. Implementation

The implementation was based on the suggestions of [1]. Variables were named according to the paper for a matter of consistency.

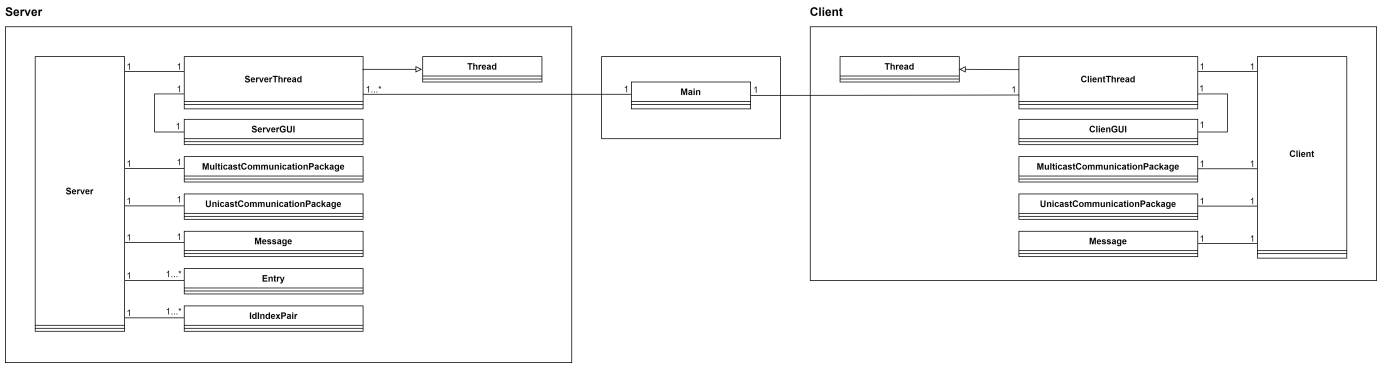

Fig. 3: Class diagram

The crash of nodes was not a concern for this simulation but the network's loss of packets was. To do so, a random number generator was used.

Each server makes use of a MulticastCommunicationPackage and a UnicastCommunicationPackage that implement communication services . As a consequence, all servers support multicast and unicast communication.

All communications occur over UDP and server nodes are thread-based. Even though all server nodes are running on the same machine, all communication happens through messages, with no use of shared memory. Due to the lack of reliability of UDP, the application might have to deal with loss of packets, delays and different orders of arrival.

In the beginning, all nodes, including the client, exchange their IDs, which correspond to their ports. The multicast port is known a priori, which is passed as an argument when a server is created.

The multicast communication is only used for the communication between the client and the leader. It is important to emphasize that the reason why multicast needs to be supported by all server nodes is not only a result of the dynamic server role enforced by Raft but also of our own implementation. The client does not know which server is the leader, so it sends a multicast message to the multicast group. All servers, except for the leader, must ignore that message. The response from the leader to the client is a unicast message.

All the other communications are based on unicast. The AppendEntries RPC and the RequestVote RPC are sent in parallel to all other servers by sending separate messages to each server individually, thus using unicast and not multicast communication. The function in use is called broadcastMessage(), but the name is just suggestive, as it is based on unicast. One of the reasons to this implementation is the simulation of messaging lost in the network.

As mentioned before, Raft has to be fault tolerant. This means that the state machines should converge in the presence of crash of nodes, delays of messages or the different arrival order of messages. The crash of nodes was not a concern for this project. The delay and consequent different order of arrival of messages was achieved through the use of a random function that outputs a percentual value. Depending on its value, a message is sent, or not.

There is also one class for the Message, associated to each server. It allows to build messages of different types. These types are enumerated and agreed between all servers. Every server sends the messages along with their respective type identifier. The id 0 corresponds to a message from a client. Moreover the id 1 is related to an AppendEntries RPC from the leader. The id 2 and 3 correspond to acknowledge for AppendEntries RPC with new Entries and wihtout any Entry, respectively. Finally, the id 4 is linked to a RequestVote RPC and the id 5 to the acknowledge for this RequestVote RPC.

In the subsequent paragraphs, a more detailed overview on the implementation is going to be given.

A. Client

The commands from the client have the form of addition or subtraction of the current value for an integer. The commands are generated in a random way. The client runs in an infinite loop. It send a new command either after receiving the response to the previous one or after 14 seconds.

B. Servers' State Machine

Each server runs in an infinite loop that has a switch condition according to the state of the server. This takes place on the class ServerThread, which implements the state machine.

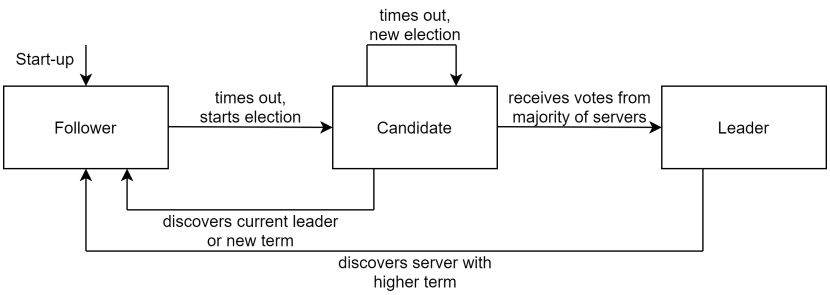

Fig. 4: Servers' State Machine

1) Leader: As mentioned before, the leader checks if any command is received from the client and if so, the leader adds it as a new entry to its log. The socket has a timeout of 1 ms, as it is not supposed to be halted expecting a command. If 50ms have already passed, which is the heartbeat period, then it must send another AppendEntries RPC. This period was implemented using the System.nanoTime() for precision purposes. This includes the entries from the oldest to the newest, which the leader knows to have not been received by at least one follower. If there is none, then it will be just a heartbeat. Afterwards, it waits for unicast messages, which will be one of the many types stated in the class Message. Again, it uses a timeout of 1 ms just so it is not halted whilst waiting for commands. The type of this messages can be an acknowledge to AppendEntries RPC, a RequesVote RPC from a candidate or an AppendEntries RPC from another leader.

The leader also checks every instance of the loop if it has some entry to commit, according to the responses received to the AppendEntries RPCs, as explained in the last section. This is done by comparing its own commitIndex to the matchIndex of the followers. The matchIndex of each follower is saved in the leader, and it corresponds to the last entry that the leader knows has been commited to that server. If there's a majority of matchIndex to an entry not yet commited by the leader itself, then it will commit it. It will send, using unicast, the response to the client. The leader will updade its commitIndex so it is sent in the AppendEntries RPCs, and other followers find out.

2) Candidate: The period for sending the RequestVote RPC is implemented the same way as the AppendEntries RPC in the leader. The timeout for waiting for an answer is defined in a random way when creating the server. This has a value between 150ms and 300ms, as suggested in the paper, and it is fixed for each server during the whole execution. In the same way as the leader, the candidate waits for unicast messages. These messages can be an acknowledge to a RequestVote RPC. Additionally, their nature can be a RequestVote RPC from another candidate or an AppendEntries RPC from another leader.

3) Follower: The follower has a passive role. It simply reads the unicast messages it receives. Simply put, they can be either AppendEntries RPC from a leader or RequestVote RPC from a candidate.

If it does not receive an AppendEntries RPC after a certain predetermined timeout, it turns into Candidate. As alluded to above, when the follower receives an AppendEntries RPC it must perform a consistency check in order to see if its log is updated or not. The logs might be inconsistent. There might be missing entries, extra entries or both.

To implement the log replication algorithm presented before, the leader must maintain a nextIndex for each follower, which corresponds to the next entry to be sent to that follower. These are initialised when the server becomes leader to the index after the last entry in its log. This means that the leader assumes that the followers are updated. If this turns out to be false, then the AppendEntries RPC consistency check will fail. The leader will decrement the nextIndex for that follower, and retry the AppendEntries RPC. Eventually the answer will be true, meaning, they reached a point in which they are consistent.

Note that if a follower receives an AppendEntries RPC request that includes log entries already present in its log, it ignores those entries in the new request by answering true. This last AppendEntries RPC will eliminate any subsequent entries from the follower's log and append the new ones.

C. Interface

The Graphical User Interface, GUI, was designed having in mind simplicity and ease of use since it was only developed with the objective of making it easier to evaluate the behaviour of the implemented algorithm in an empiric way. Consequently, for each server the corresponding GUI shows the server id, the election timeout, the current role. Moreover, it has information about the current term and the actual committed value.

Besides that, each server's GUI also has a button to check its log. With regards the client's GUI, these only displays the client id, its local value and the latest received value from the leader. The received value is displayed to easily check the consistency between the server's and the client value. Having the same goal in mind, it also has a button to check the command list.

III. Results and Performance Analysis

The values were taken from various amounts of servers (3, 5 and 7) and for 6 different values of percentages of error, where error means percentage of lost messages.

A. Latency

The latency measures the time interval between the leader receiving a command from the client and the instant in which the client sends the response to that command.

When evaluating the results, it is clear that the latency increases with the percentage of error and number of servers.

When we focus on the simulation that has 7 servers and 50% of error, it is noticeable that after every glitch there is a linear decrease. This linear decrease is due to the leader sending the response to other previous requests that had not been committed yet, whilst sending the response to the client. Multiple requests can be pending since the client sends a new command after receiving the previous answer or after 15 seconds without receiving anything.

B. Election Time

The election time measures the time interval between a certain follower becoming candidate and the time instant in which it stops being a candidate. This is done either by turning into a leader or into a follower again, as they both mean another leader is recognised.

As expected, the election time grows with both the number of servers and the percentage of error. The main factor for this behaviour is that more followers will turn into candidates with the increase of the error which results in many parallel RequestVote RPCs sent in parallel. Adding the loss of messages that create more candidates and the loss of messages that keep them from getting the majority and winning an election, the time increases considerably.

C. Convergence Time

The convergence time is the time needed for the state machines of the servers to be consistent. It starts when the current leader commits a certain entry (with a specific index), and it ends when the last server commits that same entry.

The delay to consensus does not change much when comparing a different number of servers. Thus, the graphic available is only for the case where there are 3 servers. However, for different amounts of error, the results are distinct.

An observation made is that the number of glitches increases with the number of servers. This happens when the leader receives a majority of positive acknowledges (for instance, 4 in the case of 7 servers) and it commits that entry together with those followers. However, the other 2 followers will need log replication, which will lead to a clear delay on reaching the consensus.

The network traffic measures the number of messages exchanged during nodes since the moment in which the leader receives the command from the client until all servers have applied that command to their state machine.

It is relevant to note that, for low error percentages, the whole graphic shifts to a higher number of messages when the servers increase. In the example with 3 servers and no error, the leader has to receive two acknowledge messages and each server receives one AppendEntries RPC. This adds up to a total of 4 exchanged messages, as we can see in the graphic. By applying the same reasoning to the others, we reach the same conclusion. When the error increases so do the exchange messages. This happens due to many reasons. Firstly, every time it does not get a majority of acknowledges, it has to send again the messages to every server, even though many have already received it. Secondly, the loss of messages makes it harder to win an election, and consequently more elections will occur. Thirdly and lastly, more followers will not have received the AppendEntries and they will turn into candidates. This creates conflicts between VoteRequest RPCs which increases the network traffic.

IV. General Review

The tests performed were only in the context of message loss since server crashes were not taken into consideration.

It can be acknowledged that the safety of Raft does not depend on timing: sooner or later all server nodes will converge to the same state. This happened even when testing with 70% of message loss, which shows the robustness of the algorithm.

The results made it obvious, however, that the availability is strongly related to the message loss. Availability is described ”the ability of a system to respond to clients in a timely manner”, as stated in Raft paper, which was measured through the latency testing. It can be concluded that the higher the message loss the lower the availability of the system.

Nerverthless, the latency is not its biggest problem. The normal course of events contemplates the rejection of messages from the client when the server node is a candidate or a follower. A problem arises when a server node changes from leader to follower: when this happens, at the same time, a candidate is changing to leader, and a message loss might happen due change ocurring before the client's command is read.

Understandability is one of the strongest points offered by Raft, however, it is hard to make any strong conclusions when comparing Raft with Paxos. This is due to the fact that Paxos was not studied as thorough as Raft.

When comparing Raft to Paxos, it is worth noting that the implementation of Raft is more straight forward according to existing literature. The reduced number of message types and the simplicity of the state machine on which it relies are the most relevant characteristics responsible for that.

It can be understood that altough simpler, Raft still ensure safety by limiting the ways in which the logs might be inconsistent.

V. Further Work

Crash tests should be taken into consideration when evaluating Raft performance. Besides, using different languages, and programming methods, should provide a stronger understanding of Raft performance.

References

[1] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. *Proceedings of the 2014 USENIX Annual Technical Conference, USENIX ATC 2014*, pages 305–319, 2019.