U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Raft Leader Election Implementation

Pedro Miguel Marinho Almeida[1]

January 18, 2019

***Abstract*— This paper describes a partial Raft implementation in Java. After an introduction to this consensus algorithm, the functioning of this implementation is explained with bigger detail, more specifically, the Leader Election part, and finally a characterization of the application. This implementation was heavily based on the original raft paper "In Search of an Understandable Consensus Algorithm" [1].**

I. INTRODUCTION

Achieving consensus is one of the well-known and often-discussed problems in distributed systems. It is important, but it is also surprisingly difficult to solve. In the context of distributed computing systems, where several processes subject to failures execute simultaneously, often on physically distant computers, and that are also subject to delays and losses, raised the need to implement algorithms which guarantee the reliability of the system. Algorithms of consensus dictate the behavior of the various processes involved in the system so that it is possible to ensure agreement on the value of a variable common to processes, managed by a leader in a computer network agreement in order to carry out a task that must be executed by all processes, among others. In a large part of consensus problems, solutions are implemented by implementing Paxos or variations of this algorithm. The Raft algorithm was born in order to create a simpler and easier algorithm for design than Paxos, while maintaining the effectiveness of the algorithm and producing equivalent results.

II. RAFT

As a basis for its operation Raft has it that at any given time there is at most one server of the distributed computing network - also called cluster - considered leader. In order to maintain this leadership, the leader periodically sends messages called heartbeats signaling that the leader remains in operation. At the absence of this message the remaining servers will start a process of leader election that will be described more specifically in the remaining of this article. This algorithm is also based on a principle that divides time into terms that arbitrarily vary in duration. Each server records what the current term may be and it update itself if it receives any message from any another server with a higher current term. The term is increased whenever an election is held.

Guidance provided by Faculdade de Engenharia da UP

[1]Pedro Almeida is with the Faculty of Engineering from the University of Porto ee12052@fe.up.pt

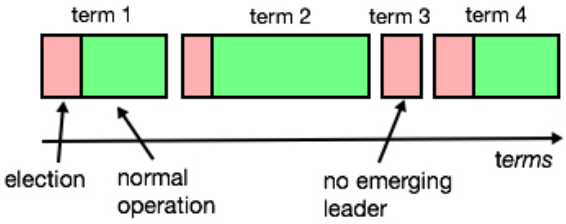

Fig. 1. Time in raft is divided into terms. It's like a logical clock.

Thus, Raft implements an algorithm of replication of state machines on multiple servers. These servers should keep a message log that should be identical on all servers. These messages are introduced by a client directly in the cluster leader, who replicates this message on the remaining servers. This is the second part of the algorithm that was not implemented and therefore will not be addressed on this paper.

A. Leader Election

As previously mentioned, the Raft algorithm is based on architecture with at most one leader. This leader sends periodical messages to its followers - called heartbeats. That message is to indicate to the followers that the leader is still in operation. When the leader ceases to function followers detect the absence of heartbeats and begin a new election. When running, a server could be in one of three states - depicted in Figure 2.

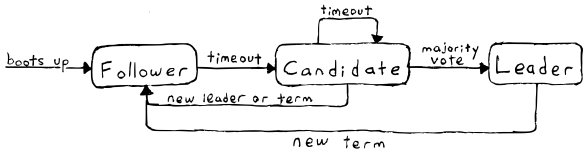

Fig. 2. State machine implemented in the servers for election of leader.

Each server stores a random time value between 150 and 350 milliseconds, which it uses as a timeout. Within that period of time, if a server does not receive a message of heartbeat or a message of voting request, it becomes candidate to leader. To start the application, the server increments the value of the term, sends a message asking to vote to all of the other servers in the cluster and votes for itself. All

servers can only vote one time per term. Then, as shown in Figure 2 , this server becomes the leader if it receives a majority of the members votes of the cluster, it will be a follower if another server declares itself as leader, or remains as a candidate until the election timeout expires again and starts a new election. The process of electing the leader and the guarantee of the existence of a leader are important in the context of Replicated State Machines that use Raft.

III. IMPLEMENTATION

The Raft algorithm was implemented in Java where a pre-defined number of threads execute in parallel, each one emulating the operation of a server. The scheduling of the tasks is exclusively handled by the OS which is not too relevant in this case. The processes communicate according to the UDP protocol. The fact that the threads do not have priorities above each other it's not to relevant in this implementation because UDP is unreliable and the loss of packets is expected. Note that the servers only communicate through messages, there is no shared memory whatsoever. The operation of this Raft implementation is approximately the same as that described in Chapter II and will be detailed how this implementation has been achieved and mention instances of the code that I consider more relevant. For a more intuitive and graphic interpretation of the results produced in each thread a graphical interface was created, like the one shown in Figure 2 indicating values referring to this server as: server ID, actual state, current term and an indication if the server is in operating mode and it is possible to interrupt via the Stop button- This feature is used to manually force an election or "kill" a server to check the behavior of the system as a whole if a certain server fail stops.

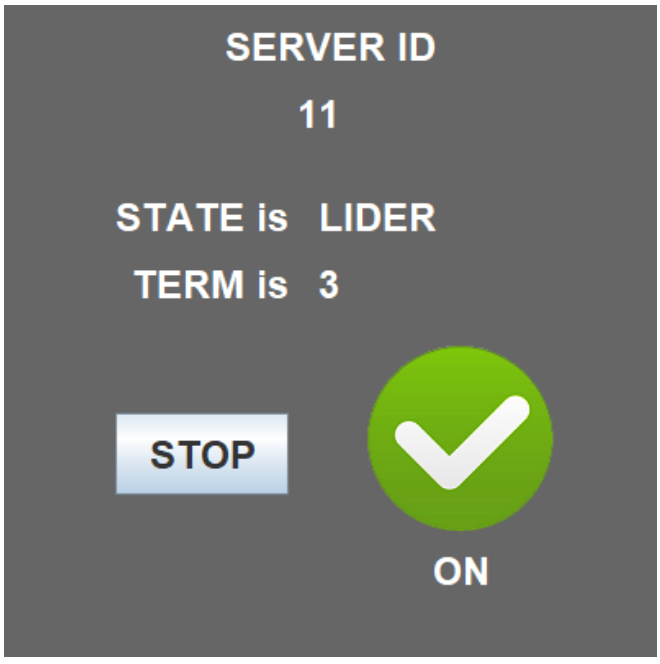

Fig. 3. Graphical interface corresponding to a server.

The implementation of leader election is achieved almost entirely through the Server class.

This class executes on a thread and allows the server to run as a candidate, follower or leader. In case of being a follower it correctly responds to heartbeat messages while also having implemented a timeout on the UDP socket that triggers the follower to become a candidate at the proper time. When it runs as a candidate it uses a random timeout to ensure that the time you wait until you start a new election is random. When it receives a message that indicates the existence of a legitimate leader it returns to the follower state and if it receives a majority of votes from the remaining members of the cluster it correctly transitions to the state of leader. When operating in the leader state it sends heartbeats to the remaining servers and goes to the follower state when it verifies the existence of a higher term within a new received message.

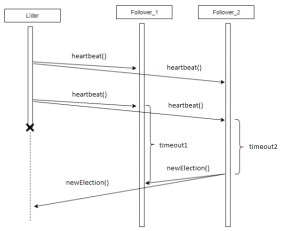

Fig. 4. A leader fail and how a RPC works in this particular case.

Figure 4 shows the behavior of the system in case of leader failure. As can be seen the leader periodically sends heartbeats until it stops getting responses. From the moment they received the last heartbeat, followers start a random election timeout, between 150 and 350 ms. As soon as the timeout expires, this follower start the application by calling the newElection method. In order to introduce faults in the leader you can kill the leader with the Stop button which will do with that messages of hearbeats are not sent, which will cause for the election timeout to expire and trigger the start of a new process of electing a new leader.

IV. RESULTS

The final result was very satisfactory. Although a raft implementation can be tested in very different ways there are a few that are crucial. The tests vary with the available hardware and with the language the program was written. The connection of the network is also relevant. In order to see if the implementation lives up to the expectations we can try a few things with this implementation such as:

- Keep on killing the leader and bring it back.
- Keep on killing the follower and bring it back.
- Keep on killing random number of nodes and bring them back.
- Set the heartbeat and election time very small to help find race.
- Make partitions in the cluster and merge them back again.

The only one that I was not able to test or reproduce is the last one, all of the other can be easily tested in this implementation.

V. CONCLUSION

With this work it was possible to understand more about this consensus algorithm that aims to achieve the same results as multi-Paxos. I have found that the algorithm whose main objective was to be comprehensible was reflected in a robust and fast algorithm and which offers the possibility of expansion by others. Unfortunately it was not possible to compare the results with other possible algorithms and also because it doesn't exist many implementations available. Also the lack of support of a group or other human elements didn't made it possible to go further in the implementation and do a whole raft implementation instead of a partial one. However, the results are all as expected and can be used as a reference in future work. I enjoyed very much working on this project alone though because it allowed me to face problems I encountered on my own and it allowed me to understand better the basis and the complexity of a distributed system.

REFERENCES

[1] D. Ongaro and J. Ousterhout, In search of an understandable consensus algorithm, USENIX Annual Technical Conference, 2014.

[2] "Raft Visualization", http://thesecretlivesofdata.com/raft/

[3] "Students Guide to Raft", https://thesquareplanet.com/blog/students-guide-to-raft/