DISTRIBUTED SYSTEMS

RAFT Algorithm

09/11/2023

ÁLVARO DE FRANCISCO NIEVAS, 838819@unizar.es

GUILLERMO BAJO LABORDA, 842748@unizar.es

Index

1. Introduction

In this practice, we have implemented the RAFT algorithm, a consensus algorithm for distributed systems. RAFT is widely used to ensure that a single node assumes the role of leader, which is essential for coordinating operations in a distributed system.

Through a leader election process based on terms and votes, RAFT ensures that leadership is unique and consistent over time. This key feature is crucial for maintaining consistency and coherence in decisions and operations within the system.

2. Implementation of the RAFT Algorithm

Our implementation of the RAFT system is fundamentally divided into the automaton that performs the relevant operations for each node's logic and the functions that offer various RPC calls used to communicate the nodes. Therefore, each node has its own set of attributes and operations exposed to the rest of the system.

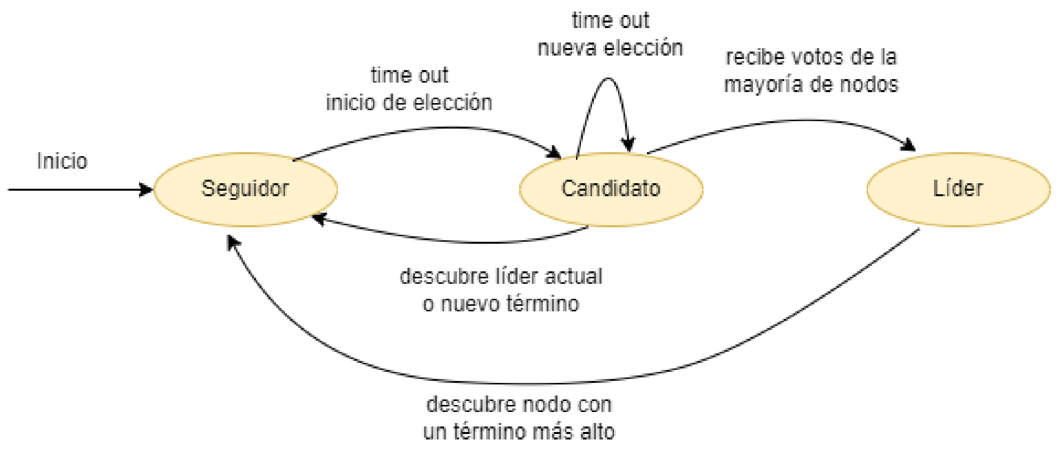
The attributes that make up a node's state are:

- List of **nodes** that are part of the system.
- **Identifier** of the current leader, which can be -1 if there is no leader.
- **Number of votes** received, in case the node is a candidate.
- **Communication channels** to notify role changes in the system, such as FollowerChan, LeaderChan, or Heartbeat.
- Node's **role**, which can be "Follower", "Candidate", or "Leader".
- The **last known term** of the node.
- The **index** of the node that was voted for during the term.
- The **highest known committed log index** and the highest applied index in the state machine.
- The **list** of the **next log** entry to send to each node. This is used when in the “Leader” state to send the log to other nodes.
- The **list** of the **last log** entry applied to the state machine for each node.

In the next part of the practice, we include the Log, a vector of operations forming the state of the system. The purpose of the RAFT algorithm is to maintain a consistent log across the nodes in the distributed system.

2.1. RAFT Coordinator

The coordinator runs as a goroutine and is responsible for managing the actions of the node. These actions vary depending on the role assigned at the time. The role of a node in the distributed system is described by the following automaton.



At the beginning, all nodes are *Followers*, transitioning to the Candidate state once a timer expires. If the candidate wins the election, they become the new Leader. The operations corresponding to each state are described below:

For all roles, if we see that the highest committed index is greater than the last applied index, we increase it by one. From here, the behavior changes for each role

.

2.1.1. Follower

The node in the Follower state listens to two distinct channels:

1. **Heartbeat:** Receives true if the leader calls the RPC *AppendEntries* with no content in the Entry field, as long as the term of the node calling the RPC is greater than the current term.
2. **Timeout:** A random timeout is generated, indicating the maximum time the node will wait before initiating a new election and presenting itself as a candidate.

Therefore, if the timeout expires before a new heartbeat is received, the node changes its role to "Candidate" and sets the leader's identifier to false (-1).

2.1.2. Candidate

The candidate node performs a procedure in which it asks for votes from the other nodes to become the leader. First, it increments its term, votes for itself, and calls the RPC *RequestVote* to the system nodes with arguments: its own term and identifier, as well as the index and term of the last known log entry.

Once the request is made, it listens for three possible messages: remaining a Follower (either because of election results or receiving a Heartbeat), becoming the *Leader*, or having the election timeout expire without a decision, in which case it must start a new election.

As the candidate receives responses from the nodes, it checks that the term indicated in the response is less than or equal to its own. If this condition is met, it adds a vote, and when the number of votes is a majority, it sends a message via a channel to indicate that it has won the election. If the term is smaller, it sends a message using the *FollowerChan* channel, indicating that the node should change its role to *Follower*.

2.1.3. Leader

The leader node periodically sends a heartbeat to the other nodes while ensuring it is the "legitimate leader." For this, it uses the RPC AppendEntries call. If the response shows that the other node's term is greater than its own, it changes its term and sends a message using the FollowerChan channel, which the leader is listening to. Upon receiving the message, the node changes its role to Follower.

If the leader crashes, upon restarting, it will enter the Follower role.

2.2. Llamadas RPC

2.2.1. RequestVote

Nodes vote when a candidate invokes the *RequestVote* RPC. They follow a *first-come-first-served* principle, meaning they grant their vote to the first candidate that requests it. They will grant only one vote per election period, and only if the candidate's term is greater than their own. The response will contain the node's term and whether the vote was granted. If the voter's role is not *Follower*, it sends a message to the main RAFT process via a channel, indicating that it should change its role to *Follower*.

2.2.2. AppendEntries

The *AppendEntries* RPC distinguishes two cases: the heartbeat and the operation. When the node receives an operation, it stores it in its log (or displays it on the screen in this case).

If a heartbeat is received, the node always responds with its term. If it is equal to or smaller than the term of the invoking node, it acknowledges it as the leader, stores the term, and sends "True" to the *Heartbeat* channel. Additionally, if the term is smaller and the node is not a *Follower*, it will send a message via the *FollowerChan* channel to indicate that the node should change its role.

2.2.3. SubmitRaftOperation

SubmitRaftOperation is the service RAFT provides for clients to submit operations. The call is supported by all nodes but only takes effect if the receiving node is the *Leader*. If this is the case, it will store the index and the received term in its log, along with the operation itself.

Then, using the *AppendEntries* RPC with a specific timeout, it sends this same information concurrently to all nodes. If the majority of nodes respond favorably, it increases the *CommitIndex*, and the operation becomes part of the distributed system's state machine.

3. System description

This system consists of **three machines** in the cluster, each running a node with which the correct implementation of the RAFT consensus algorithm has been verified.

To set up this system, a pair of asymmetric **cryptographic keys** has been generated, consisting of a private key and a public key. The public key is shared between the machines. The main machine is considered the one from which the system is launched, while the other two machines must have the main machine's public key to allow for connections between them. These keys are generated using cryptographic algorithms.

A **distributed architecture** has been implemented where the three processes run on separate machines and communicate with each other over the network using the Remote Procedure Call (**RPC**) library. Each process has the ability to invoke RPC calls from other nodes, allowing for the transparent execution of functions and methods on remote machines. Communication between processes is done through specific ports. Each process uses a designated port to receive incoming RPC requests and send responses.

4. Testing

To validate the correct functioning of the system, a testing module, *testintegracionalraft1_test*, has been designed, which includes several tests to verify that the behavior is as expected and that there are no issues in the distributed execution.

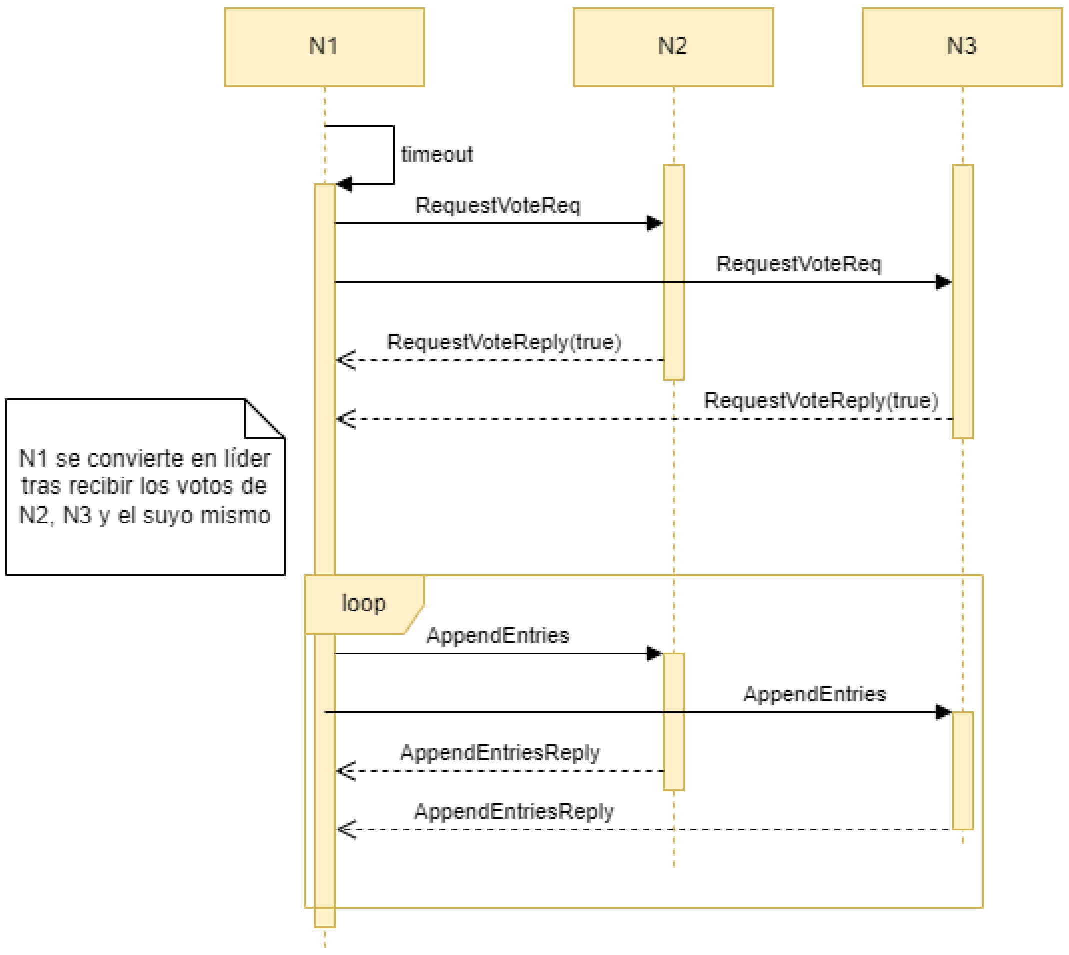
For this third practice of the course, a total of four tests must be passed:

1. **Node Startup and Shutdown**: In this first test, distributed processes are deployed or started on the RAFT nodes, and then the processes are stopped. This verifies that the nodes of the system can be correctly started and stopped.
2. **First Leader Election**: This test starts the distributed system, verifies that there is a unique leader, and then stops the system.
3. **New Leader Election after Previous Failure**: After starting the distributed system and verifying that there is a valid and unique leader, the leader is made to crash by calling the pararNodo function. The leader is then relaunched, and a 5-second wait is introduced to ensure it is started correctly. Finally, the system is checked again to ensure there is a unique leader and then stopped.
4. **Commitment of Three Stable Operations**: This last test aims to verify the correct operation of the operations submitted to the leader, as well as their commitment. The system is started, a unique leader is verified, and three operations are submitted using the checkOp helper function. Finally, the distributed processes are stopped.

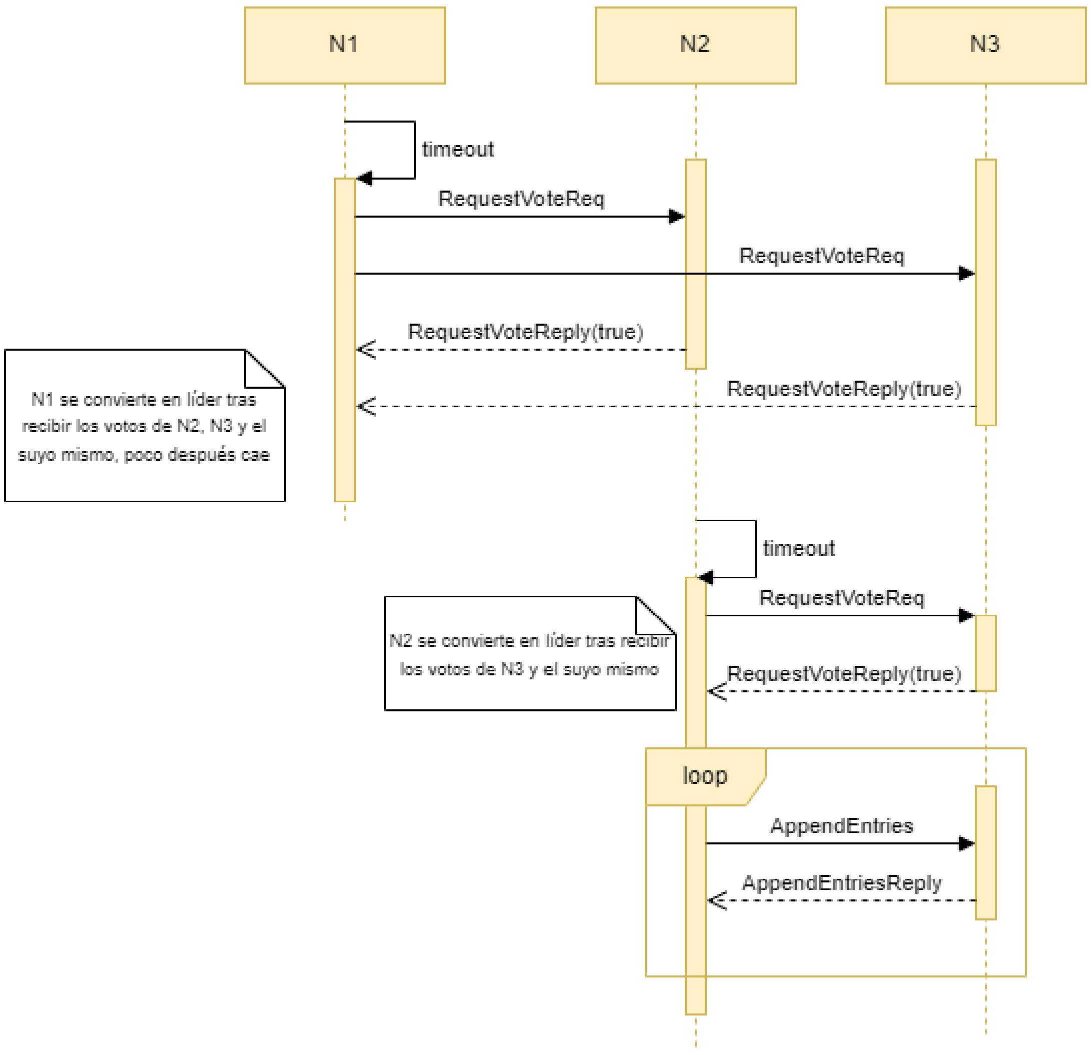
5. Sequence Diagrams

To represent the behavior of the RAFT consensus algorithm more simply, various sequence diagrams have been designed to show the most representative cases:

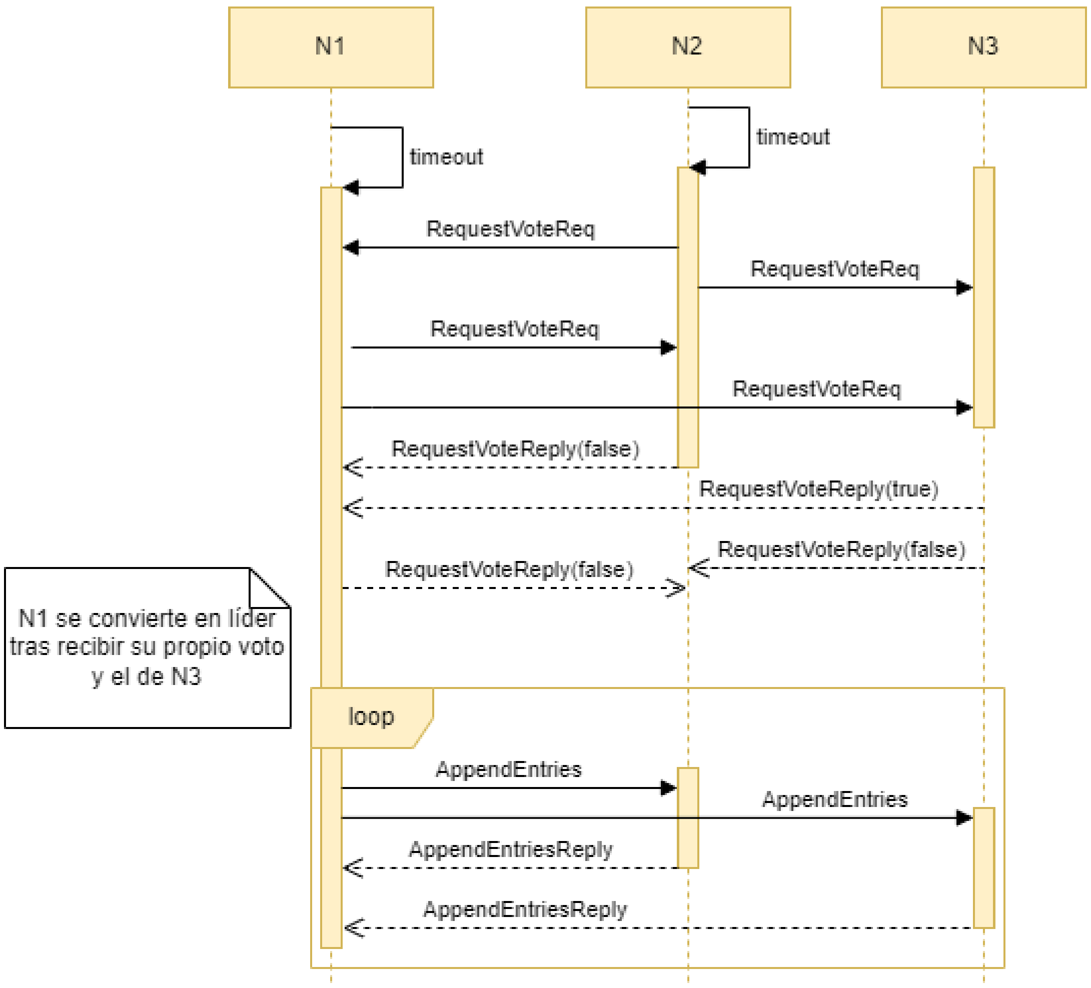
Standard RAFT Operation



Leader Crash and Timeout Expiry for New Election



Election Process with Two Candidates


N1
N2
N3
timeout
timeout
RequestVoteReq
RequestVoteReq
RequestVoteReq
RequestVoteReq
RequestVoteReply(false)
RequestVoteReply(true)
RequestVoteReply(false)
RequestVoteReply(false)
N1 se convierte en líder tras recibir su propio voto y el de N3
loop
AppendEntries
AppendEntries
AppendEntriesReply
AppendEntriesReply