DISTRIBUTED SYSTEMS

RAFT Algorithm Part II

14/12/2023

GUILLERMO BAJO LABORDA, 842748@unizar.es

ÁLVARO DE FRANCISCO NIEVAS, 838819@unizar.es

Universidad Zaragoza

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Index

1. Introduction

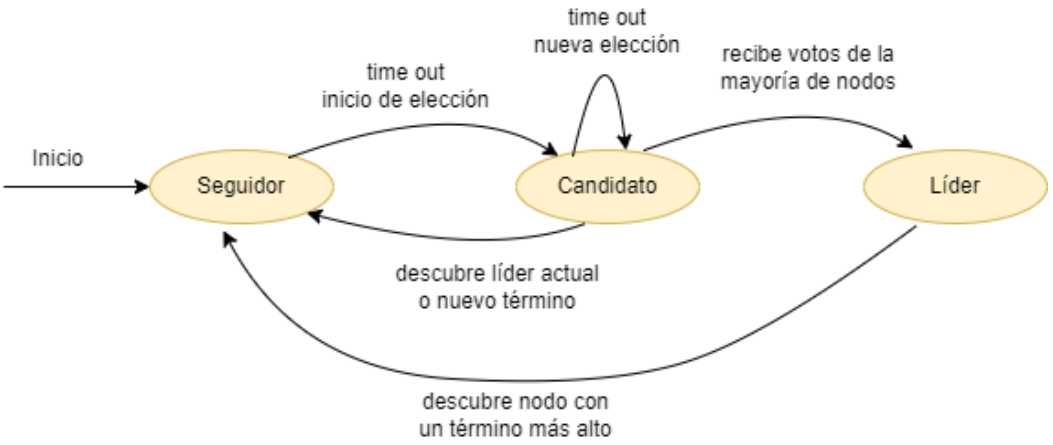
n this fourth practice for the Distributed Systems course, we embark on building an **in-memory key/value storage service**, with a particular focus on fault tolerance through **distributed replication based on Raft.**

Raft, as a consensus algorithm, establishes a framework for the replicated state machine, where the function raft.*SometerOperacion*, invoked by the leader, starts the propagation of operations to the replicas through RPC calls *AppendEntries*.

For a deeper understanding, we rely on the relevant theoretical material and the attached document, which provides crucial details about the specification of replica states and the reference RPC calls. These elements form the basis of our analysis and development in this practice.

2. 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 any given time. The role of the node in the distributed system is described by the following automaton.



We observe that initially, all nodes are Followers, switching to the candidate state once a timer expires. If the candidate wins the election, they will become the new Leader. The various operations corresponding to the states are described below:

For all roles, if we find that the highest committed index is greater than the last applied index, we will increment it by one. From here, the behavior changes depending on the role.

2.1.1. Follower

The follower node listens to two distinct channels:

1. **Heartbeat:** Receives *true* if the leader makes an RPC call to *AppendEntries* with no content in the *Entry* field, as long as the term of the node making the RPC call is greater than its own.
2. **Timeout:** Generates a random timeout indicating the maximum time the node will wait before starting a new election and presenting itself as a candidate.

Thus, if the timeout expires before a new heartbeat arrives, the role changes to “Candidate,” and the leader identifier is set 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.

3. Full Leader Election Algorithm

The implementation of the **complete leader election algorithm** has been designed robustly by considering both the last index and the last term of the log when sending the vote request. This strategy is crucial to ensure that the newly elected leader is **fully updated** and aware of all the transactions consolidated in the cluster up to that point.

By sending the vote request with information about the **last index and term of the log**, the candidate is providing the receiving nodes with a complete view of the current state of its log. This allows each node to accurately assess whether the candidate has stayed up to date with all previous transactions and whether it has a log that fully reflects the consensus achieved in the cluster. That is, each node will receive the arguments in the candidate's request, and if they determine that the candidate is updated and meets the necessary requirements to be the leader, they will vote for it, continuing until the candidate receives a majority of votes and can be proclaimed leader.

Considering the last index of the log ensures that the candidate has received and processed all entries prior to that index. This approach is crucial to **avoid electing a potential leader without knowledge of certain important transactions that have already been consolidated in the cluster**. Additionally, by including the last term of the log, it ensures that the candidate is aware of the latest version of each entry in its log, avoiding possible desynchronization between nodes.

This careful and complete implementation of the leader election algorithm not only **improves the system's reliability** by ensuring that the elected leader is fully informed but also **contributes to the consistency and integrity of the data in the cluster** by preventing the election of outdated or incomplete leaders.

4. AppendEntries RPC Call with Failures

Furthermore, the operational flow for the **SometerOperacion** RPC call by system clients has been implemented, which will trigger AppendEntries calls from the leader to the followers, advancing the index of committed log entries. Both have been developed under various failure scenarios. In addition, the operations are applied to a simple state machine with read and write operations on an in-memory data store using a Golang map data type.

Here's the process followed:

1. The **client** sends an operation to the Raft leader node.
2. The **leader adds** the new **operation** to its log in *someterOperacion*.
3. The **leader sends *AppendEntries*** RPC calls to each of the followers with this new entry added to its log.
4. The **followers add** this new entry to their respective **logs**.
5. Once the **majority** of followers add the entry and respond to the leader, the leader consolidates the entry and updates the *CommitIndex*.
6. Upon **increasing** the ***CommitIndex***, the operation is sent to the state machine via the aplicarOperacion channel.
7. The **result** is **returned** to the **client**.
8. The **leader confirms** to the followers that the entry has been consolidated, and they also consolidate it.

However, a change in leadership or failures in servers can generate **inconsistencies**. These inconsistencies can manifest as missing entries from the leader in a follower, entries present in the leader but not in the follower, or both. To address this, each *AppendEntries*() RPC call always verifies that the log contents of the leader and the follower receiving the call match.

To achieve consistency across the logs of all servers, the leader always tries to replicate its log on the other nodes, avoiding overwriting or deleting entries in its own log. The leader maintains, for each node, the index of the next log entry to be sent (*NextIndex*) and the highest replicated entry index (*MatchIndex*). If the candidate does not have this entry in its log, new entries are rejected by returning a false response. In this case, the leader must find the last matching entry in the follower's log and remove any entries from that point onward. Then, it sends all the remaining entries from its log. That is, if the leader has new entries to send to a server, it sends *AppendEntries*() with the entry present at

NextIndex. If successful, it updates *NextIndex* and *MatchIndex* for that server. Otherwise, it decrements *NextIndex* for that server and retries until the inconsistency is resolved.

5. Testing

Finally, extensive tests have been designed and executed to **validate the robustness and reliability of the system implementation**, addressing various operational scenarios and possible failures. Additionally, it has been confirmed that the new Raft algorithm implementation passes the tests specified in practice 3.

The following specific test cases were designed and executed:

1. **Achieving agreements on multiple log entries despite a replica disconnecting from the group.** This scenario simulates the disconnection of a follower during system operation. Initially, the distributed processes are launched, and a leader is identified among the three Raft nodes. Then, a write operation is performed, and a node is disconnected. After the disconnection, two more write operations are performed. The test verifies the system's ability to achieve consensus despite the follower's disconnection and its later reconnection. Once the node wakes up, the operations will have been replicated to it and added to the log without issues. Finally, a couple more operations are submitted, and the distributed processes are stopped.

2. **Failure to achieve agreement on multiple entries when two out of three Raft nodes disconnect.** This test focuses on fault handling when the majority of followers are disconnected. Again, it starts by identifying a leader among the three nodes and performing write operations. Next, two followers are disconnected, and operations are submitted with these disconnected replicas. The test evaluates the system's response to the lack of agreement due to the disconnections and how it recovers once the nodes are reconnected. It checks that operations with the disconnected replicas are not consolidated since the leader cannot commit entries if both replicas are disconnected. Finally, the two replicas are reconnected, and new operations are submitted to ensure they are correctly processed, as well as the operations from the previously disconnected nodes, which will now consolidate properly.

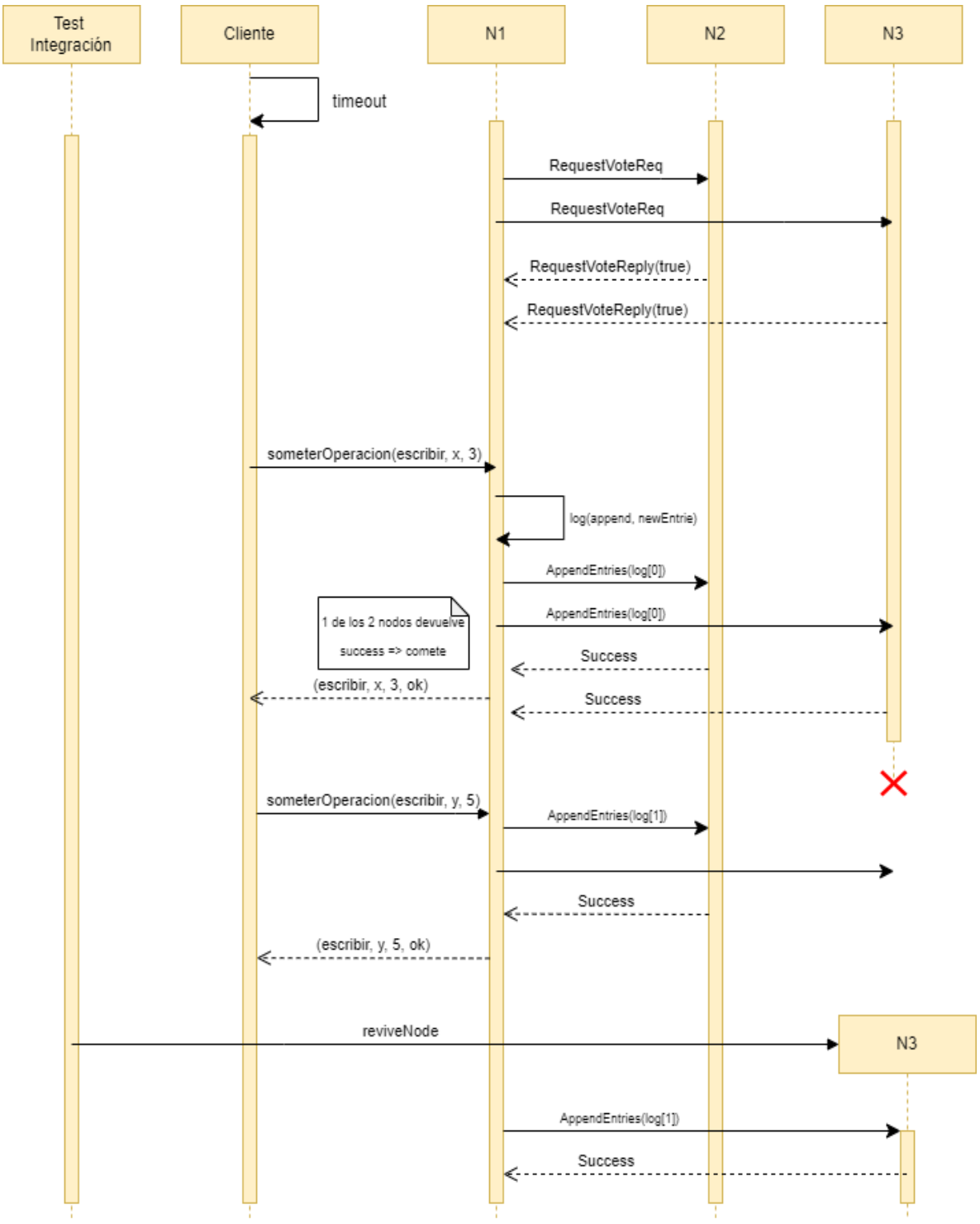
3. **Submitting 5 concurrent client operations and checking log index progress.** In this scenario, five operations are submitted concurrently to the system. After identifying a leader, simultaneous write and read operations are executed. The test evaluates the system's ability to

handle concurrent operations and maintain log consistency in the presence of multiple simultaneous requests. Finally, the *checkEstadoRegistro* function is called to verify the state of the log for a given index across all replicas.

6. Appendix

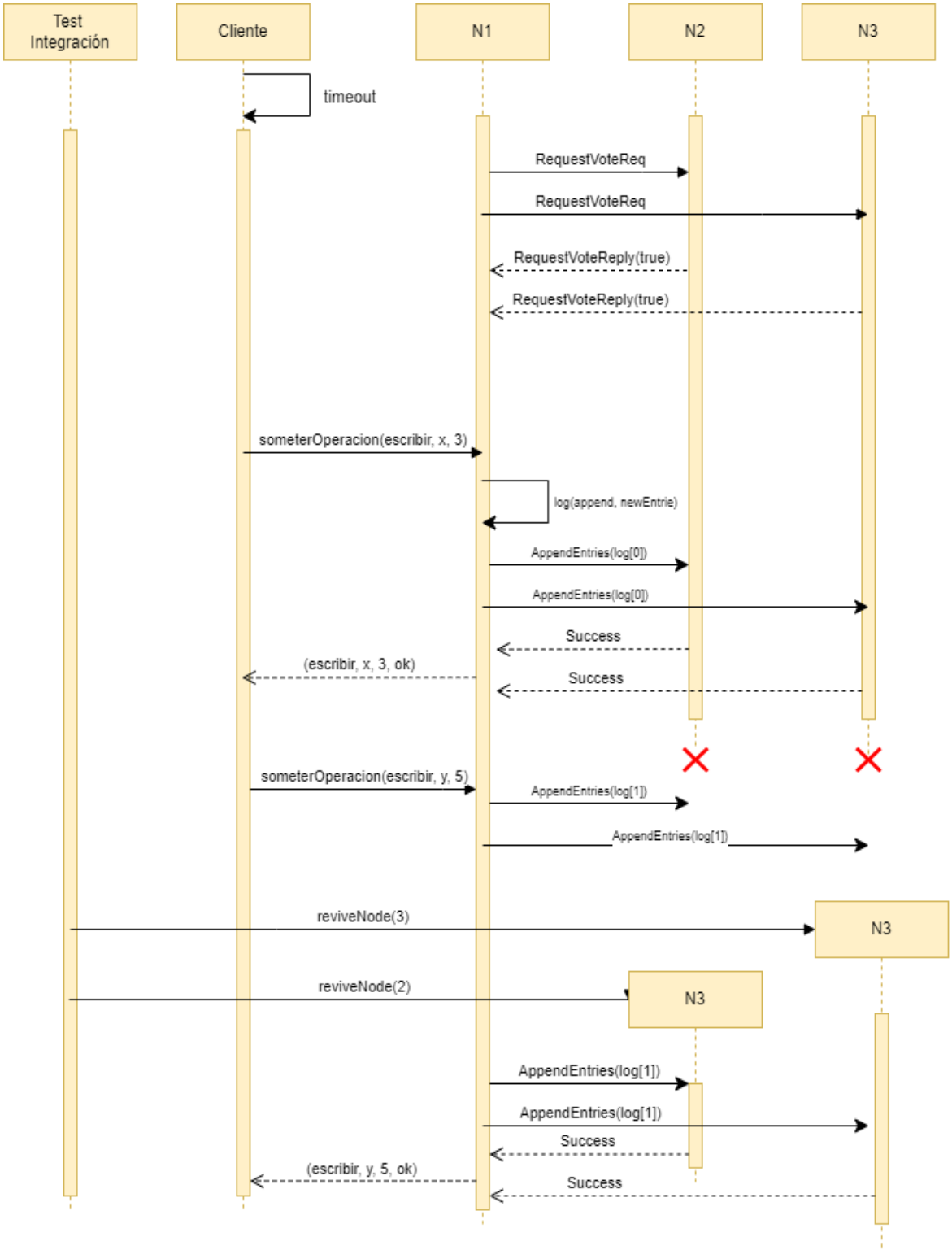
1. **Sequence Diagram for Submitting Operations with a Fallen Node**

   The sequence diagram illustrates how an entry is consolidated with two live nodes, then node 3 falls, and finally,



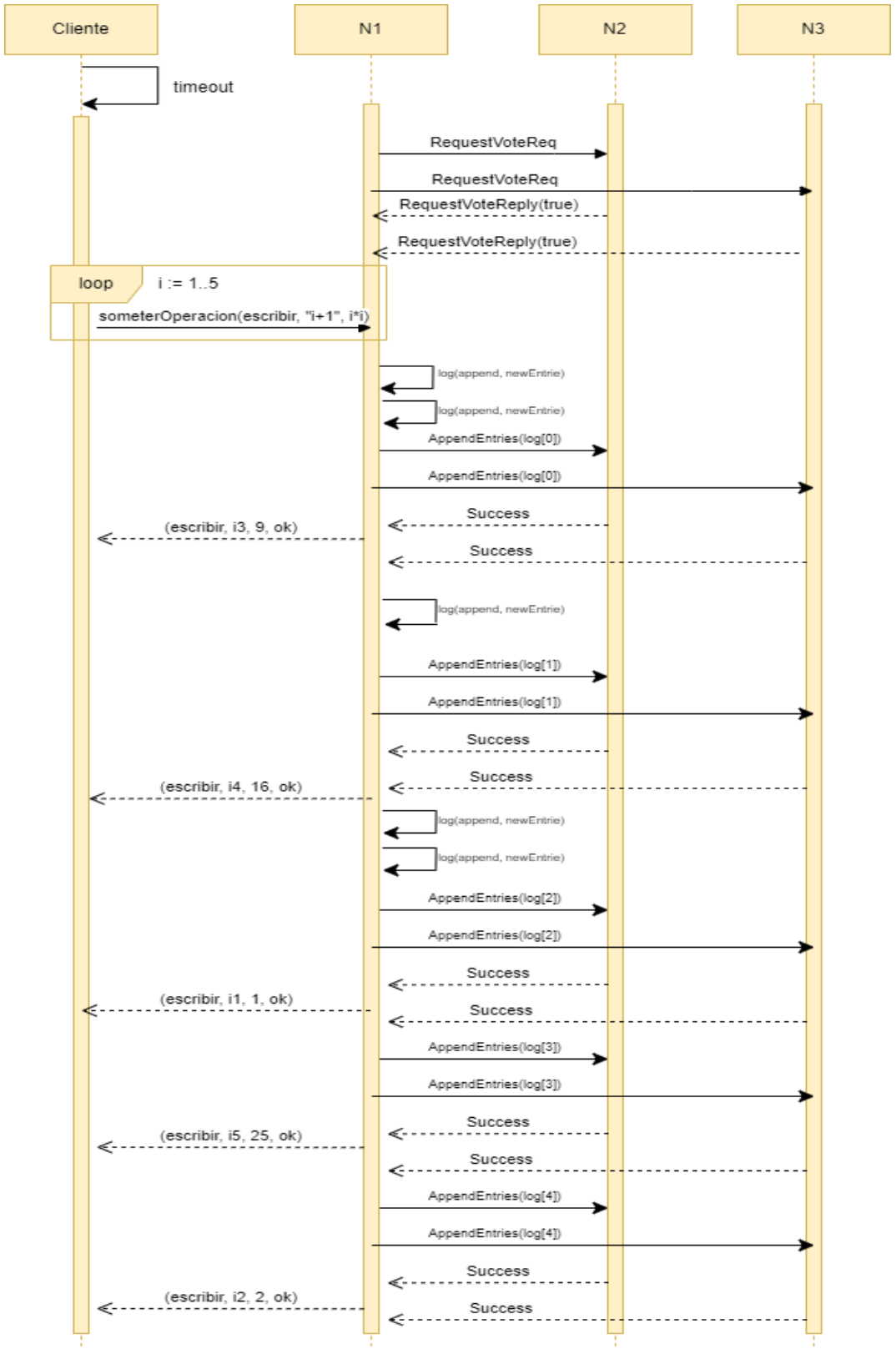
2. Sequence Diagram of Submitting Operations with Two Nodes Down

As can be seen, the diagram is quite similar. The difference is that the second operation is not consolidated until at least one of the two nodes returns a Success to the leader.



3. Sequence Diagram of Submitting 5 Concurrent Operations

This diagram shows the last test case. Obviously, since these are concurrent operations, there are various possible execution histories. The diagram shown illustrates an example of one possible execution story.



7. Bibliography

RaftDoc:

https://raft.github.io/raft.pdf

Algorithm simulation:

https://raft.github.io/

https://thesecretlivesofdata.com/raft/