SISTEMAS DISTRIBUIDOS

Práctica 4

Algoritmo RAFT Parte II

Viernes B Mañana Pareja 2

14/12/2023

GUILLERMO BAJO LABORDA, 842748@unizar.es

Universidad Zaragoza
1542

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Índice

1. Introducción

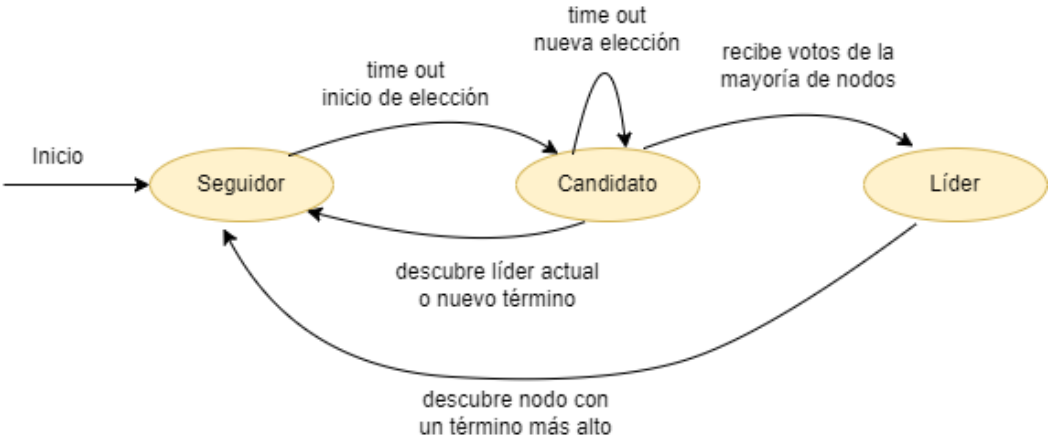
En esta cuarta práctica de la asignatura de Sistemas Distribuidos, nos embarcamos en la construcción de un **servicio de almacenamiento clave/valor** en memoria RAM, con un enfoque particular en la tolerancia a fallos mediante la **replicación distribuida** basada en Raft.

Raft, como algoritmo de consenso, establece un marco para la máquina de estados replicada, donde la función *raft.SometerOperacion* invocada en el líder inicia la propagación de operaciones hacia las réplicas a través de llamadas RPC AppendEntries.

Para un entendimiento más profundo, nos apoyamos en el material teórico relevante y en el documento adjunto al guión, que proporcionan detalles cruciales sobre la especificación del estado de las réplicas y las llamadas RPC de referencia. Estos elementos forman la base de nuestro análisis y desarrollo en esta práctica

2. Coordinador del RAFT

El coordinador se ejecuta como una gorutina y se encarga de gestionar las acciones que realiza el nodo. Estas son distintas en función del rol que tenga asignado en ese momento. El rol que tiene el nodo en el sistema distribuido está descrito por el siguiente autómata.



Observamos como en un comienzo todos los nodos son *Followers*, cambiando a estado candidato una vez venza un temporizador. Si este gana la elección será el nuevo *Leader*. Las distintas operaciones correspondientes a los estados se describen a continuación:

Para todos los roles, si vemos que el índice más alto comprometido es mayor que el último índice aplicado, lo aumentaremos en una unidad. A partir de aquí, el comportamiento cambia para cada rol.

2.1.1. Follower

El nodo en estado follower escucha dos canales distintos:

1. **Heartbeat:** recibe *true* si el líder hace una llamada a RPC *AppendEntries* sin contenido en el campo *Entry*, siempre y cuando el término del nodo que hace la llamada RPC se mayor que el propio.
2. **Timeout:** generamos un timeout aleatorio que indica el tiempo máximo que el nodo esperará antes de comenzar una nueva elección y presentarse como candidato.

Por tanto, si el timeout vence antes de que llegue un nuevo heartbeat, cambiamos de rol a "*Candidate"* y pondremos el identificador del líder como falso (-1).

2.1.2. Candidate

El nodo candidato realizará un procedimiento en el que pide el voto de los otros nodos para convertirse en líder. Primero aumenta su término, se vota a sí mismo e invoca a la RPC *PedirVoto* de los nodos del sistema con argumentos: su propio término e identificador así como el índice y término del último registro que conoce.

Una vez hecha la petición, escucha tres posibles mensajes: que siga siendo *Follower*, ya sea por resultado de la elección o por recibir un Heartbeat; que se ha convertido en *Leader*; o que ha vencido el timeout de la elección sin una decisión, por lo que tendrá que comenzar una nueva elección.

A medida que el candidato recibe respuestas de los nodos comprueba que el término indicado en la respuesta es menor o igual que el propio. Si se cumple esta condición, suma un voto y cuando el nº de votos sea mayoría enviará por un canal un mensaje que indica al proceso principal que ha ganado la elección. En dicho caso, se proclamará líder y configurará NextIndex y MatchIndex para la replicación de logs. En caso de que el término sea menor, enviará un mensaje usando el canal *FollowerChan* en el que se le indicará que cambie su rol a *Follower*.

2.1.3. Leader

El nodo líder envía periódicamente un heartbeat al resto de nodos al tiempo que comprueba que es el *"líder legítimo"*. Para ello utilizará la llamada RPC *AppendEntries*. Si en la respuesta ve que el término del otro nodo es mayor que el suyo, cambiará su término y enviará un mensaje utilizando el canal *FollowerChan,* por el que el líder estará escuchando. Cuando reciba el mensaje, el nodo cambiará su rol a *Follower.*

Si el líder se cayera, al volver a comenzar la ejecución entraría con rol *Follower.*

3. Algoritmo de elección de líder completo

La implementación del **algoritmo de elección de líder completo** ha sido diseñada de manera robusta al considerar tanto el último índice como el último término del log al enviar la petición de voto. Esta estrategia es fundamental para garantizar que el líder recién elegido esté **completamente actualizado** y tenga conocimiento de todas las transacciones consolidadas en el clúster hasta el momento.

Al enviar la solicitud de voto con información sobre el **último índice y término del log**, el candidato a líder está proporcionando a los nodos receptores una visión completa del estado actual de su registro. Esto se traduce en que cada nodo pueda evaluar de manera precisa si el candidato ha logrado mantenerse al día con todas las transacciones previas y si posee un registro que refleje completamente el consenso alcanzado en el clúster. Es decir, cada uno de los nodos recibirán los argumentos en la solicitud del candidato, y si consideran que este está actualizado y cumple los requisitos necesarios para ser líder, lo votarán, así hasta que este reciba una mayoría de votos y pueda ser proclamado líder.

Al considerar el último índice del log, se asegura de que el candidato haya recibido y procesado todas las entradas anteriores a ese índice. Este enfoque es crucial para **evitar que un líder potencial sea elegido sin tener conocimiento de ciertas transacciones** importantes que ya han sido consolidadas en el clúster. Además, al incluir el último término del log, se garantiza que el candidato esté al tanto de la última versión de cada entrada en su registro, evitando posibles desincronizaciones entre nodos.

Esta implementación cuidadosa y completa del algoritmo de elección de líder no solo **mejora la confiabilidad del sistema** al garantizar que el líder elegido esté plenamente informado, sino que también contribuye a la **coherencia y la integridad de los datos en el clúster** al evitar la elección de líderes desactualizados o con registros incompletos.

4. Llamada RPC AppendEntries con fallos

Por otra parte, se ha implementado la operativa de llamada RPC **SometerOperaicon** por parte de los clientes del sistema replicado que provocará llamadas AppendEntries desde el líder a los seguidores con avance del índice de entradas de registro comprometidas. Ambos se han desarrollado en diferentes escenarios de fallos. Además, las operaciones se aplican sobre
una máquina de estados simple, con operaciones de lectura y escritura sobre un almacén
de datos en RAM mediante un tipo de datos map de Golang.

A continuación, se va a describir el procedimiento seguido:

1. El **cliente** envía una operación al nodo Raft líder.
2. El líder añade esa nueva operación a su **log** en ***someterOperacion***.
3. El líder envía llamadas RPC **AppendEntries** a cada uno de los seguidores con esta nueva entrada añadida a su log.
4. Los seguidores añaden esta nueva entrada a sus respectivos **logs**.
5. Cuando la mayoría de los seguidores añaden la entrada y respondan al líder, el líder consolida la entrada y actualiza el ***CommitIndex***.
6. Al aumentar el CommitIndex, en la máquina de estados se envía la operación por el canal **aplicarOperacion**.
7. El resultado es devuelto al **cliente**.
8. El líder confirma a los seguidores que la **entrada** ha sido **consolidada**. Estos la consolidan también.

Sin embargo, un cambio en el liderazgo o fallas en los servidores pueden generar **inconsistencias**. Estas inconsistencias pueden manifestarse en la falta de entradas del líder en un seguidor, la presencia de entradas no presentes en el líder o ambas situaciones. Para abordar esto, cada llamada RPCAppendEntries() verifica siempre que el contenido del registro del líder y el del seguidor que recibe la llamada coincidan.

Con el objetivo de lograr la consistencia en los registros de todos los servidores, el líder siempre intenta replicar su registro en el resto de los nodos, evitando sobrescribir o eliminar entradas de su propio registro. El líder mantiene, para cada nodo, el índice de la siguiente entrada del registro que debe enviar ,*NextIndex*, y el índice de la entrada replicada más alta, *MatchIndex*. Si el candidato no tiene esta entrada en su registro, se rechazan nuevas entradas devolviendo un false. En este caso, el líder debe encontrar la última entrada coincidente en el seguidor con una de su registro, eliminar cualquier entrada del seguidor a partir de ese punto y enviar todas las entradas restantes del registro del líder. Es decir, si el líder tiene nuevas entradas que enviar a un servidor, envía AppendEntries()

con la entrada del registro presente en NextIndex. Si el resultado es exitoso, se actualizan NextIndex y MatchIndex para ese servidor. En caso contrario, se decrementa NextIndex para ese servidor y se reintenta hasta que se resuelva la inconsistencia.

5. Test de pruebas

Finalmente, se han diseñado y ejecutado pruebas exhaustivas para **validar la solidez y la confiabilidad de la implementación del sistema**, abordando diversos escenarios de funcionamiento y posibles fallos. A su vez, se ha comprobado que la nueva implementación del algoritmo de Raft diseñada pase los test especificados en la práctica 3.

A continuación, se detallan los casos de prueba específicos que se han diseñado y ejecutado:

1. **Conseguir acuerdos de varias entradas de registro a pesar de que una réplica se desconecta del grupo.** Este escenario simula la desconexión de un seguidor durante la operación del sistema. Inicialmente, se lanzan los procesos distribuidos y se identifica un líder entre los tres nodos Raft. Luego, se realiza una operación de escritura y se desconecta un nodo. Tras haber desconectado este nodo, se realizan dos operaciones más de escrituras. La prueba verifica la capacidad del sistema para lograr acuerdos a pesar de la desconexión de un seguidor y la posterior reconexión del mismo. Una vez despertado el nodo, las operaciones se habrán replicado en dicho nodo y se habrán añadido al log sin problemas. Finalmente se someterán un par de operaciones más y se detendrán los procesos distribuidos.

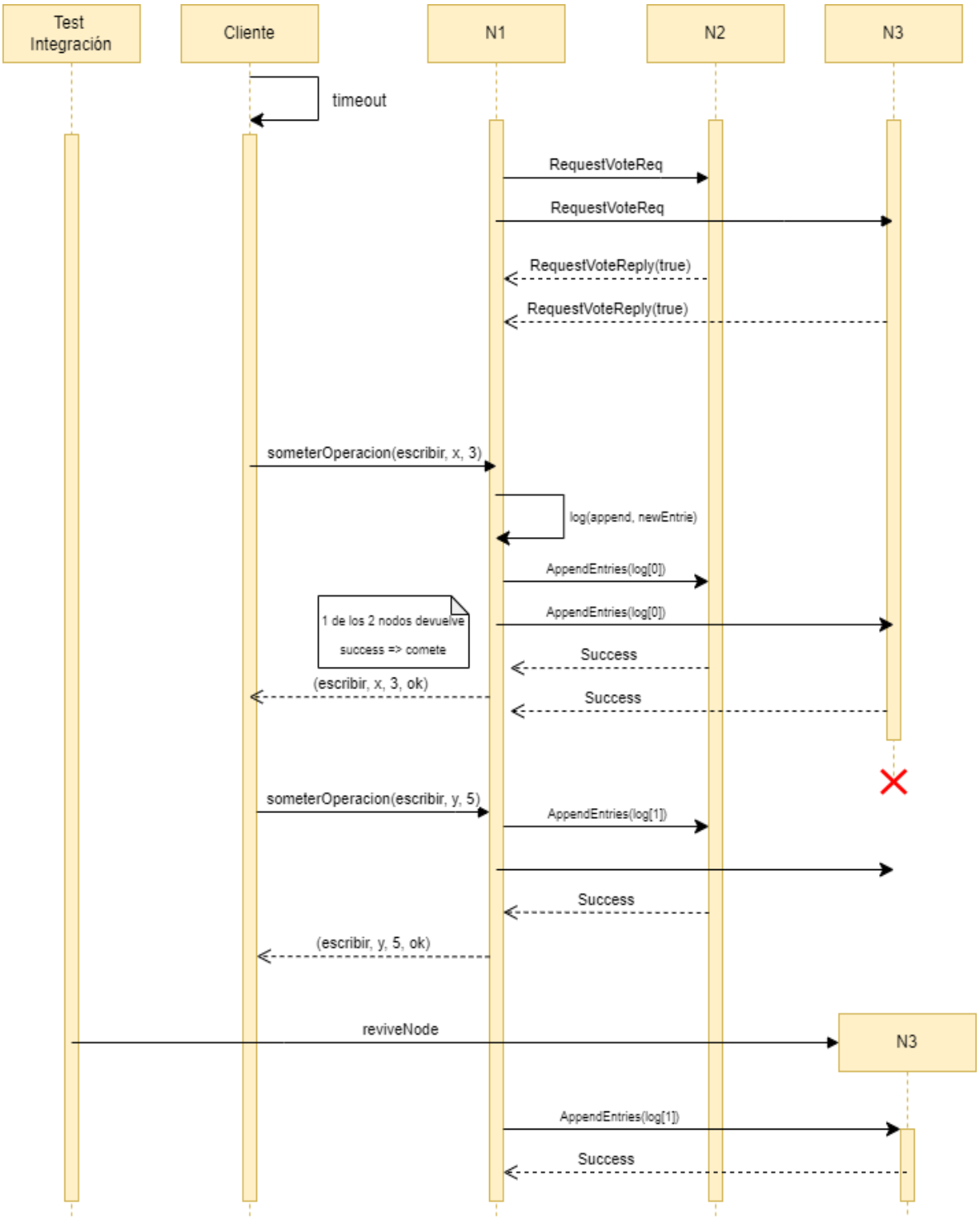
2. **No conseguir acuerdo de varias entradas al desconectarse dos de los 3 nodos Raft.** Esta prueba se centra en el manejo de fallos al desconectar la mayoría de los seguidores. De nuevo inicia identificando un líder entre los tres nodos y realizando operaciones de escritura. A continuación, se desconectan dos seguidores, y se someten operaciones con estas réplicas desconectadas. La prueba evalúa la respuesta del sistema ante la falta de acuerdo debido a las desconexiones y cómo se recupera una vez que los nodos son reconectados. Se comprueba que las operaciones realizadas con las réplicas caídas no se consoliden ya que el líder no puede consolidar las entradas si ambas réplicas están desconectadas.Finalmente, se vuelven a conectar ambas réplicas y se someten nuevas operaciones, para comprobar que se someten correctamente, así como aquellas realizadas con los nodos caídos, que ahora que están despiertos deberán consolidarse correctamente.

3. **Someter 5 operaciones cliente de forma concurrente y comprobar avance de índice del registro.** En este escenario, se someten cinco operaciones de manera concurrente al sistema. Después de identificar un líder, se ejecutan operaciones de escritura y lectura simultáneamente. La prueba evalúa la capacidad del sistema para manejar operaciones concurrentes y mantener la coherencia del registro en presencia de múltiples solicitudes simultáneas. Finalmente, se llama a la función checkEstadoRegistro, que verifica el estado del registro para un índice de registro dado en todas las réplicas

6. Anexo

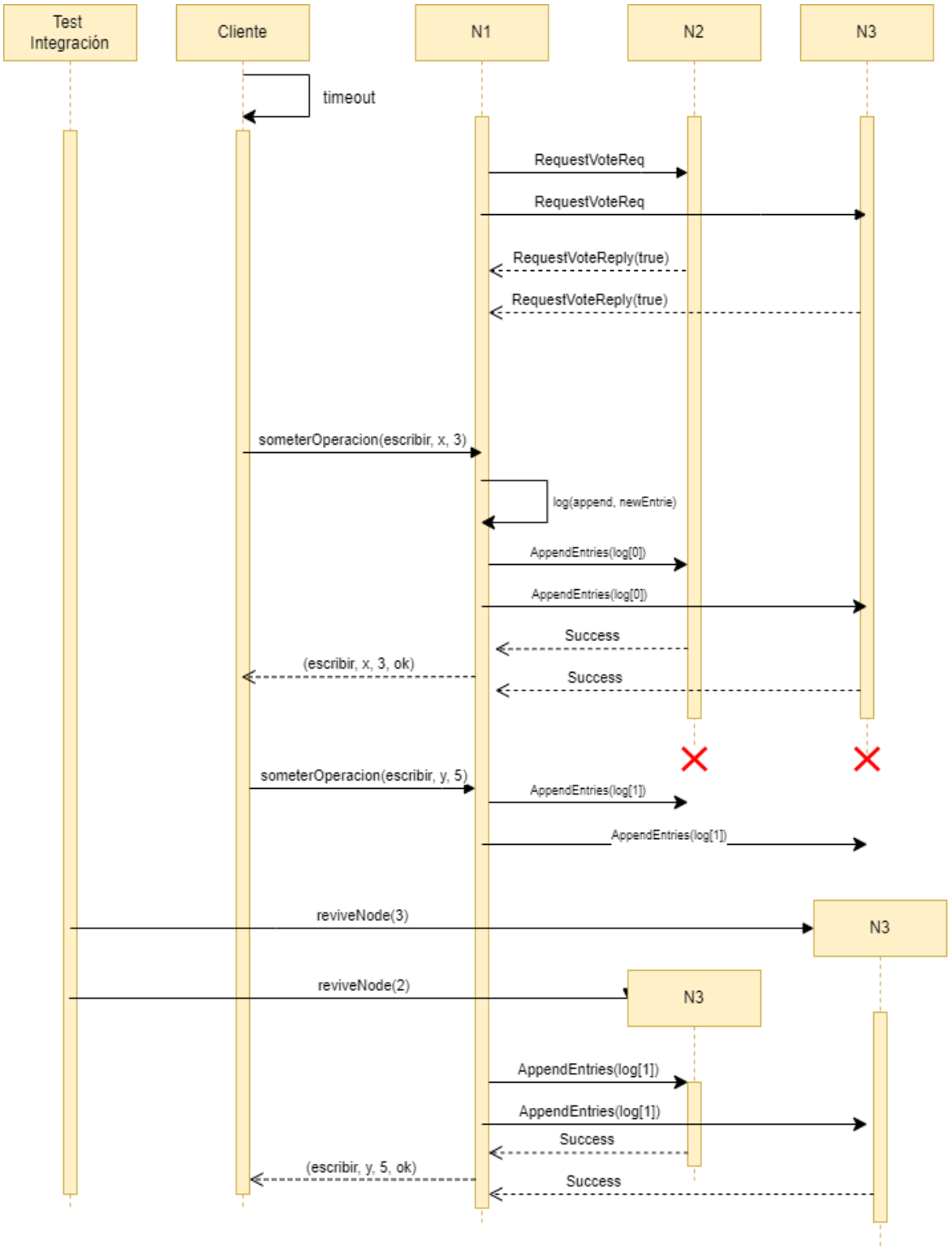
1. **Diagrama de secuencia de someter operaciones con un nodo caído**

   En el diagrama de secuencia se ve como se consolida una entrada con dos nodos vivos, luego cae el nodo 3, y finalmente al revivir, este consolida la segunda entrada que le ha quedado pendiente de consolidar con respecto a los otros nodos que han permanecido vivos.



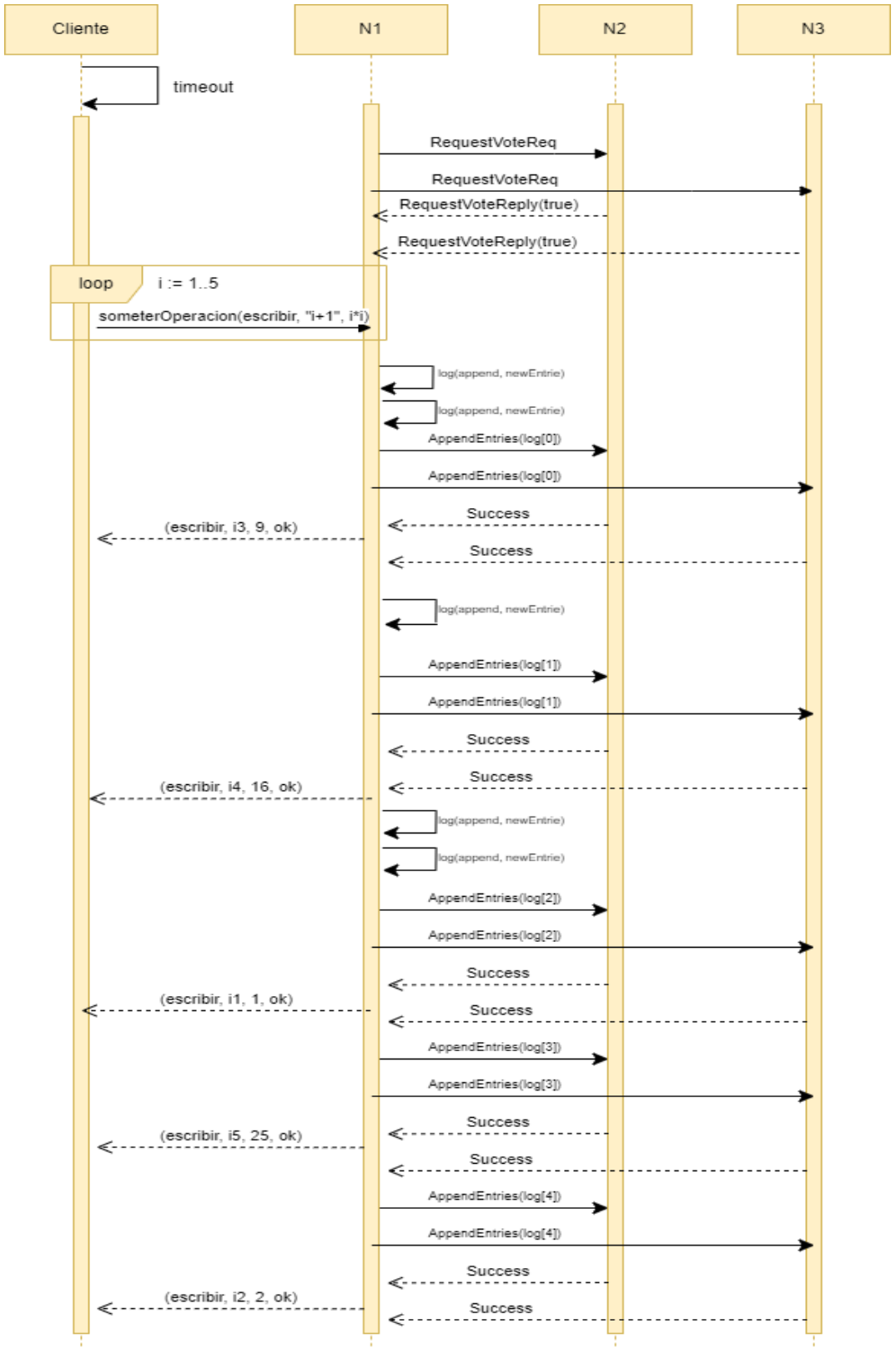
2. Diagrama de secuencia de someter operaciones con dos nodos caído

Como se puede observar, el diagrama es bastante similar, la diferencia es que la segunda operación no se consolida hasta que al menos uno de los dos nodos le devuelve *Success* al líder.



3. Diagrama de secuencia de someter 5 operaciones concurrentes

Este diagrama muestra el último caso de los test. Obviamente, al ser operaciones concurrentes, existen diversas historias de ejecución posibles. En el diagrama mostrado se ha mostrado un ejemplo de una posible historia de ejecución.



7. Bibliografía

RaftDoc:

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

Simulación del algoritmo:

https://raft.github.io/

https://thesecretlivesofdata.com/raft/