Sistemas Distribuidos - Práctica 3
Algoritmo Raft - 1ª Parte

Selivanov Dobrisan, Cristian Andrei - 816456@unizar.es

Wozniak, Dorian Boleslaw - 817570@unizar.es

Índice

1 Introducción

En esta práctica, en una serie de tres prácticas relacionadas, se ha implementado parte del algoritmo Raft en el lenguaje de programación Go. El objetivo final de esta serie de prácticas es crear un sistema distribuido que ofrezca un sistema de almacenamiento clave-valor en memoria RAM, tal que el sistema sea consistente y tolerante a fallos y caídas de nodos.

En esta práctica, se ha implementado el sistema de elección de líder y de replicación de estado entre una serie de nodos. Los nodos, al iniciar, determinan entre ellos cuál será el líder, con el cual el cliente se comunicará. Este se encargará de recibir las solicitudes para realizar operaciones sobre el sistema, replicar la operación a sus seguidores, y comprometer la operación una vez la mayoría de nodos tengan este cambio aplicado.

Además, se ha diseñado una batería de tests de integración que verifiquen el correcto funcionamiento del sistema con las limitaciones dadas. El sistema debe garantizar la elección de un único líder cuando no haya uno tras un tiempo, y que no haya inconsistencias entre operaciones comprometidas asumiendo un líder estable. No comprueba, sin embargo, cuestiones como la elección de un líder adecuado según su réplica, ni gestiona recuperaciones de nodos caídos, ni otros tipos de fallos que puedan acabar con pérdida de datos al cambiar de líder. Tampoco se implementa el propio sistema de almacenamiento, por lo que las operaciones sólo se replican y comprometen, sin ejecutarse.

2 Diseño del algoritmo

2.1 Funcionamiento del algoritmo

El algoritmo Raft se compone de dos procesos que trabajan conjuntamente para obtener el consenso: elección de líder y replicación de estado. Todos los nodos pueden estar en uno de los siguientes estados: seguidor, candidato o líder.

El líder es el nodo que gestiona la comunicación con los clientes. Cada vez que recibe una solicitud para aplicar una operación, primero envía llamadas remotas a procedimiento (RPC) AppendEntries() a los seguidores. Los seguidores tratan, al recibir el RPC, de actualizar su réplica del estado del sistema. Cuando una mayoría de nodos ha añadido la operación, el líder la considera comprometida y ejecuta la acción solicitada. Un líder se vuelve seguidor si recibe un mensaje con mandato mayor que el suyo.

Si no hay una petición del cliente en un tiempo, el líder envía un AppendEntries() sin entradas nuevas, un latido, para avisar a los seguidores de que no se ha caído. Si un seguidor no ha recibido un latido

en un tiempo en un rango aleatorio, llamado *election timeout*, se vuelve candidato. Los candidatos se votan a sí mismos y envían RPC PedirVoto() al resto de los nodos. El *timeout* es aleatorio para evitar que las candidaturas se inicien simultáneamente, evitando bloqueos por empates. Si el candidato recibe la mayoría de votos, se vuelve líder y envía un latido al resto. Si hay un nuevo líder o recibe un voto con mayor mandato, se vuelve seguidor. Si expira el *election timeout*, reinicia las elecciones hasta que surja efecto.

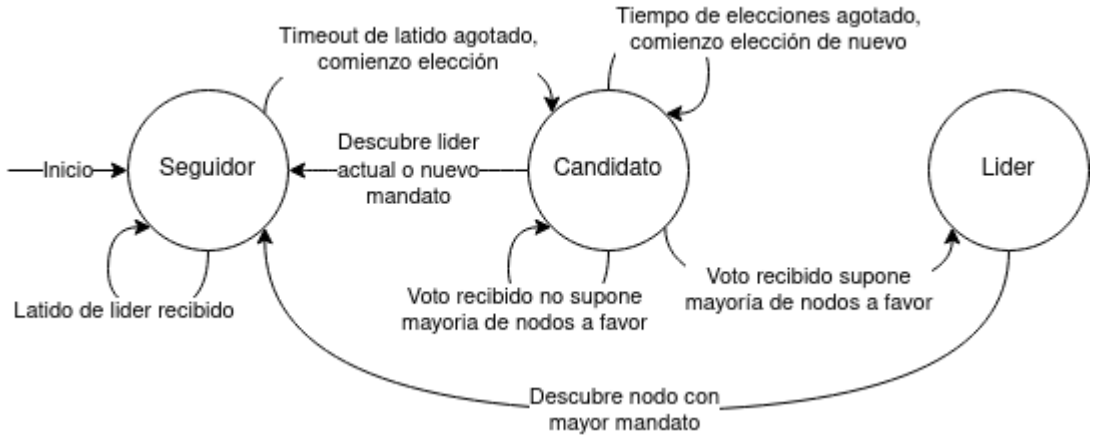

Fig. 1: Máquina de estados general para un nodo Raft

2.2 Elección de líder (RPC PedirVoto())

Cuando un nodo se vuelve en candidato, envía al resto de los nodos un RPC PedirVoto() para recibir los votos necesarios para convertirse en líder. Cada RPC tiene como argumentos el mandato e índice del candidato. El candidato recibe como respuesta el mandato mayor entre el enviado y el del receptor, y si le ha otorgado el voto.

Al recibir un RPC PedirVoto(), el nodo determina primero si su mandato es correcto. Todos los nodos, si reciben un mandato estrictamente mayor que el suyo sea en una RPC o una respuesta a esta, deben actualizar su mandato y convertirse en seguidores. Para PedirVoto(), una vez actualizado en caso de ser necesario, determina si le votará. Si el mandato recibido es mayor o igual al suyo original, y ademas no ha votado aún a nadie o ya ha votado al candidato, le otorga el voto.

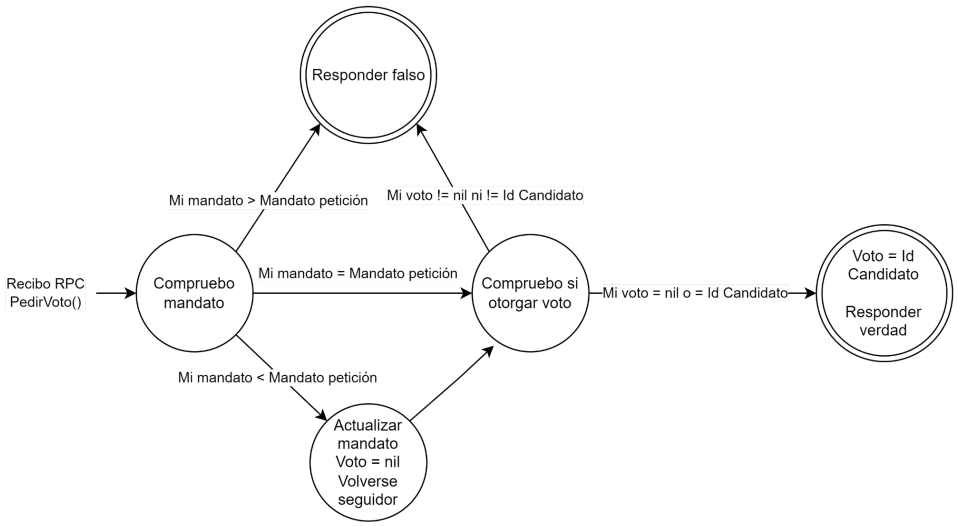

Fig. 2: Máquina de estados para una recepción de una RPC PedirVoto()

Al recibir una respuesta, el comportamiento depende del estado del nodo. Si el mandato está desactualizado, se vuelve seguidor. Si es mayor o igual si no es candidato, o mayor si lo es, o si no ha sido votado, continúa normalmente. Si su mandato es igual al del receptor y ha sido votado, se suma el voto y si tiene mayoría absoluta (la mitad de los nodos más él mismo), se convierte en líder.

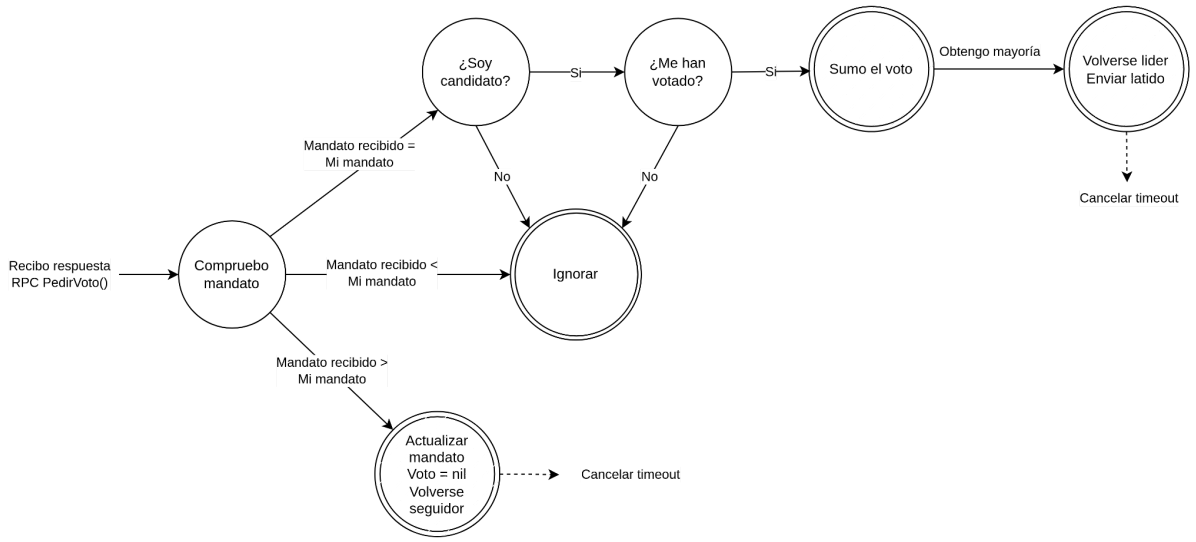

Fig. 3: Máquina de estados para una recepción de una respuesta a una RPC PedirVoto()

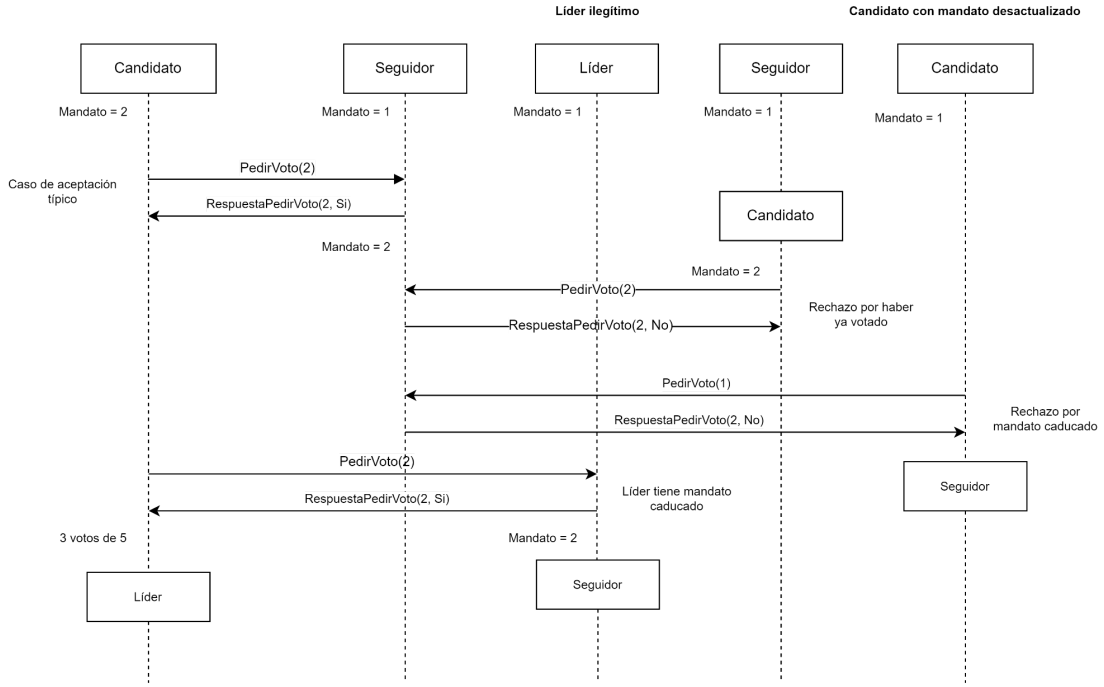

Fig. 4: Diagrama de secuencia con los diferentes casos que se pueden dar al enviar un RPC PedirVoto()

2.3 Replicación de estado (RPC AppendEntries())

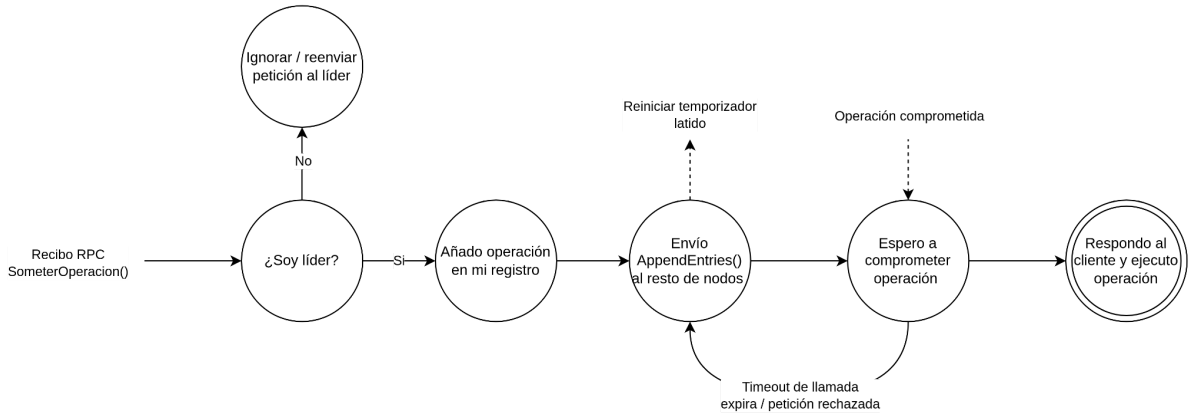

Fig.5: Máquina de estados para la recepción de una RPC SometerOperacion()

Los RPC AppendEntries() pueden ser enviados como consecuencia de un latido, o al solicitar un cliente una operación al cliente. Junto a su mandato e índice, el líder también envía la última entrada añadida (índice y mandato al insertar), las entradas a añadir a continuación de la anterior, y el índice de la última entrada comprometida por el líder. Los seguidores responden con el mayor de entre los dos mandatos y si han aceptado la operación.

En caso de ser solicitadas por un cliente, el líder primero añade la entrada en la posición solicitada de su registro. Si un cliente hubiera recibido la entrada accidentalmente, podría reeenviar la solicitud al líder o ignorarla, si existe un mecanismo en la interfaz del cliente que asegura que la solicitud la acabará recibiendo el líder. Una vez añadida la operación, el líder tratará de enviar la parte del registro de estado faltante a los seguidores dada la última entrada añadida conocida para cada nodo. El sistema, en caso de fallo, reintenta el envío reduciendo el último nodo añadido conocido en uno para los nodos que fallen. Una vez que la mayoría de los nodos confirmen al líder de que han implementado los cambios a su réplica, envía una respuesta al cliente y marca la operación como comprometida y la ejecutará. El líder seguirá intentando actualizar mientras tanto la réplica de los nodos que aún falten.

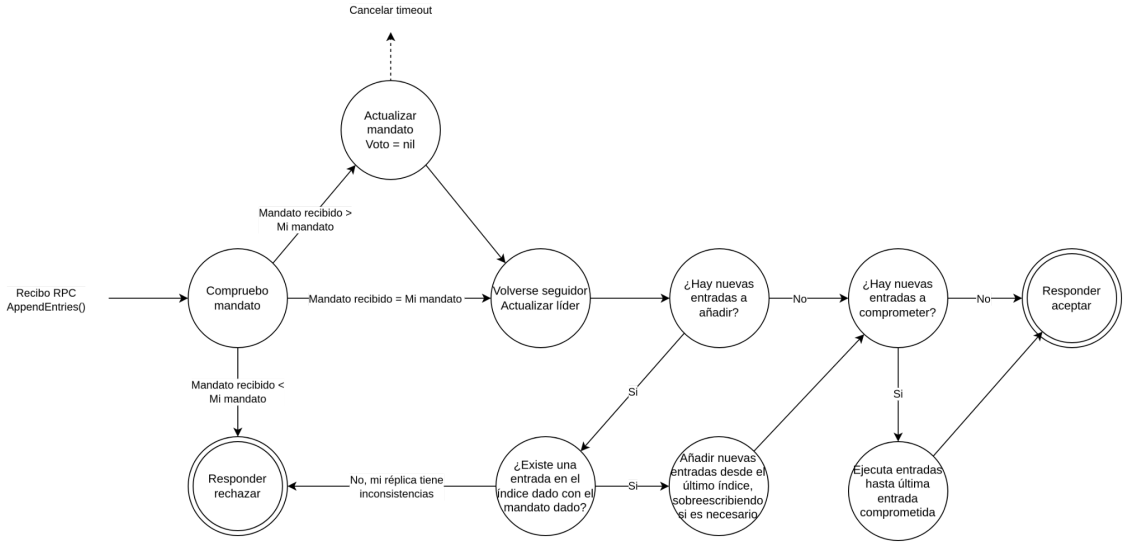

Fig. 6: Máquina de estados para la recepción de una RPC AppendEntries()

Cada vez que un nodo reciba un AppendEntries(), asumiendo que es seguidor y su mandato no se encuentra desactualizado, si tiene entradas a añadir, comprobará que la entrada en el índice dado por el líder no solo existe sino que la operación fue durante el mismo mandato. Si no se diese caso, no se garantiza que sea la misma operación, y debe rechazar la inserción, teniendo que reintentar el líder

con un índice menor. Una vez añadidas las entradas comenzando desde el último índice, revisa si el líder ha actualizado el índice de la última operación comprometida. Si es así, realizará las operaciones desde la última entrada comprometida hasta ahora y la nueva entrada.

Al recibir una confirmación de AppendEntries(), siendo el nodo un líder activo, comrpueba si la entrada ha sido añadida correctamente. Si es así, añade una confirmación y, si encuentra una mayoría, actualiza el índice de la operación comprometida. Si no se es líder o la respuesta no es de este mandato, la ignora. Si, por el contrario, el nodo se encuentra desactualizado, revierte a seguidor.

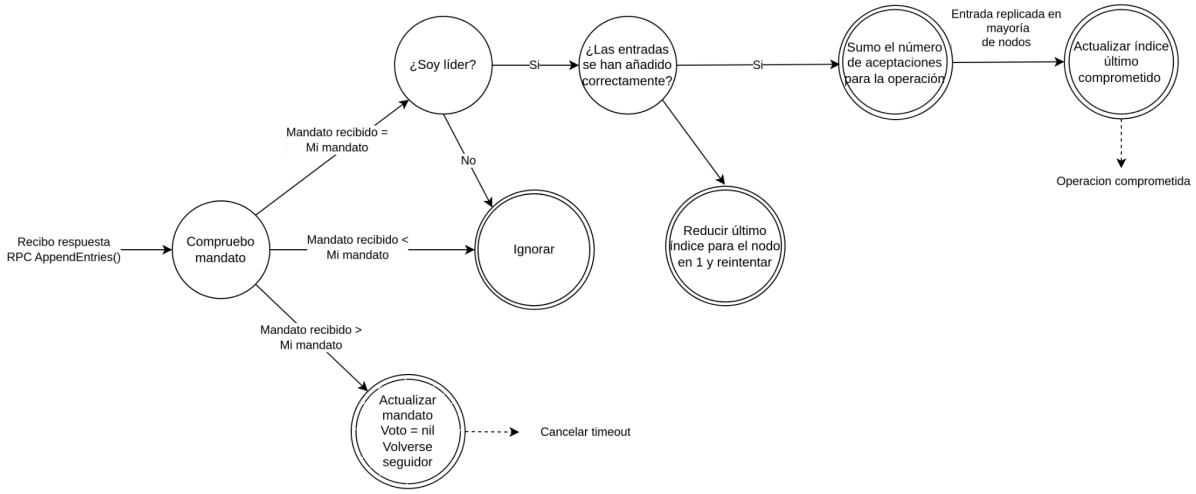

Fig. 7: Máquina de estados para la recepción de una respuesta a una RPC AppendEntries()

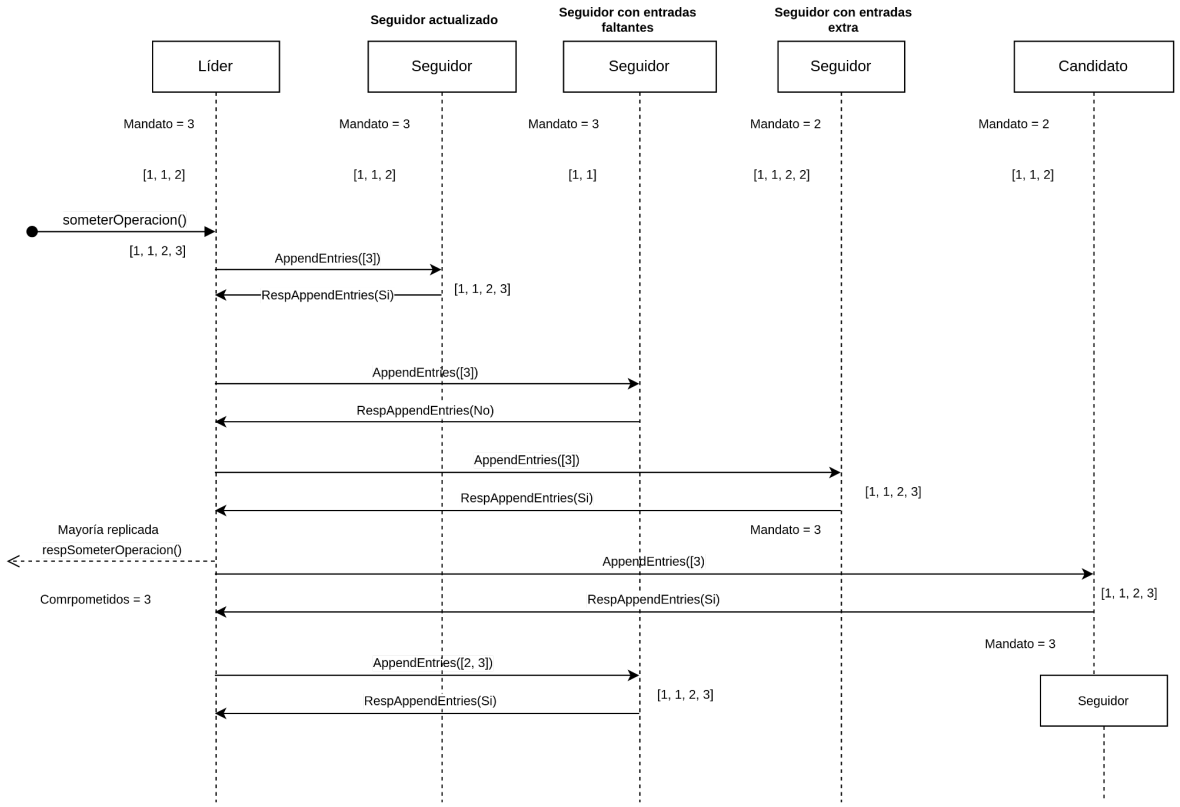

Fig. 8: Diagrama de secuencia para un envío de AppendEntries()

3 Implementación

La implementación principal del algoritmo Raft se ha realizado en internal/raft/raft.go.

El algoritmo ofrece una interfaz para iniciar un nuevo nodo (NuevoNodo()), añadir una nueva operación a comprometer (SometerOperacion()), consultar el estado actual del nodo (ObtenerEstado()) y detener el nodo y terminar el programa (Para()).

Internamente, el estado del nodo se representa mediante un *struct*, que es inicializado al invocar NuevoNodo(). Esta función ademas lanza dos gorutinas, una para registrar las RPC y comenzar la escucha en el puerto dado, y la segunda inicia el algoritmo de consenso. La segunda rutina gestiona el

comportamiento del nodo según su estado como los *election timeout* o el envío rutinario de latidos, mientras que el cambio de estado y actualización de mandato y logs se implementan en los tratamientos de RPC y sus respuestas.

Para la comunicación mediante RPC, se utiliza el módulo net/rpc de Go, que permite registrar una serie de RPC. Una RPC queda registrada si es hay una función pública del *struct* cuyos argumentos son *structs* argumento y puntero a respuesta, y devuelve un tipo error. Para

Junto a PedirVoto() y AppendEntries() para la comunicación entre nodos, se han implementado RPC para llamar a las funciones de la interfaz mediante un cliente (salvo NuevoNodo()), ademas de tipos de datos para los argumentos y la respuesta para cada RPC. En pkg/cltraft/cltraft.go se implementan los métodos para que un cliente pueda comunicarse con los nodos.

4 Validación

Se han realizado una serie de cuatro tests para verificar que el sistema funciona correctamente asumiendo las condiciones de funcionamiento sin fallos. Los tests quedan implementados en internal/testintegracionraft1/testintegracionraft1_test.go

El primer test comprueba que el sistema se puede arrancar y parar. El test se considera un éxito si todos los nodos definidos para el test arrancan y devuelven, al solicitarlo el programa de pruebas, un estado. Para esta prueba solo interesa que estén en funcionamiento en un tiempo dado. Además, el sistema tratará de parar los nodos.

El segundo test comprueba que se establece un líder estable al arrancar inicialmente los nodos. Para pasar el test, los nodos deben, en un tiempo dado, obtener un consenso sobre quién es el líder sin que no haya dos líderes simultáneamente. Si al pasar un tiempo solo hay un nodo en cuyo estado se considere líder, y el resto hayan obtenido correctamente en su estado quién es el líder, el test es pasado.

El tercer test funciona de forma similar al segundo, pero en este caso se detiene el nodo líder actual para forzar unas elecciones. Si se obtiene un resultado igual (obviando la ausencia de uno de los nodos), el test se considera superado.

El último test trata de comprometer, en una situación con un líder estable, tres entradas nuevas en el sistema. Si el líder contesta a las tres RPC de forma positiva en un tiempo razonable, el sistema quedaría completamente probado.