# SISTEMAS DISTRIBUIDOS

## PRÁCTICA 3

874055 - Ariana Porroche Llorén
871627 - Irene Pascual Albericio

# ÍNDICE



# Introducción

En esta práctica, construiremos un sistema de almacenamiento de datos que opere de manera distribuida y sea resistente a fallos. Para lograrlo, utilizaremos el algoritmo Raft, que nos garantiza que todos los nodos en la red estén siempre alineados respecto al estado de los datos, incluso si algunos de ellos fallan.

Raft organiza los nodos de tal manera que uno de ellos actúa como líder, quien toma decisiones y se asegura de que los demás sigan el mismo "plan". En esta primera parte de la práctica, implementaremos el proceso de elección de líder: cada nodo puede proponerse como líder y, si cuenta con el apoyo de la mayoría, asume el mando. Una vez elegido, el líder es responsable de mantener a todos sincronizados mediante el envío de latidos de confirmación (heartbeat) para que los demás nodos sepan que sigue activo.

Además comprobaremos que en caso de que caiga un líder, los demás nodos proponen una nueva elección y eligen a otro líder para continuar con la ejecución.

Esta práctica es el primer paso hacia un sistema robusto, capaz de operar incluso ante problemas. En la siguiente práctica, nos centraremos en someter las operaciones y en mejorar aún más la tolerancia a fallos.

# Diseño de raft

A continuación vamos a explicar cómo funciona un nodo en Raft, gracias a explicar los tipos de datos y funciones clave que hemos implementado. Estos datos y operaciones las llevan a cabo cada nodo para acordar un líder, comunicarse entre ellos, informar de desactualizaciones a otros nodos…

Tipos de Datos Principales

- **TipoOperacion**: este tipo define las operaciones básicas que el nodo puede realizar, como "leer" o "escribir". También incluye una clave y un valor asociados, dependiendo de si se trata de una lectura o escritura de datos.

- **AplicaOperacion**: indica una operación específica que debe llevarse a cabo en el sistema y registra su posición en el registro.

- **TipoEstadoNodo**: Este tipo determina el rol del nodo, es decir, si es líder, candidato o seguidor.

- **TipoEstadoVital**: este tipo determina si el nodo se encuentra Vivo o Caído en dicho momento.

- **TipoReg**: es el registro de cada entrada del nodo, que almacena el mandato y la operación.
- **NodoRaft**: es el componente nodo en sí, y almacena las siguientes variables:
    - un mutex Mux para garantizar el acceso a las variables en exclusión mutua
    - un vector Nodos con las direcciones ip:puerto de cada réplica del algoritmo raft
    - una componente Yo que guarda el identificador del propio nodo
    - un IdLider que almacena el identificador del líder en curso
    - el MandatoActual
    - el identificador del nodo al que ha votado en la elección VotadoA
    - el EstadoActual del nodo (si es Seguidor, Candidato o Líder)
    - el EstadoVital del nodo (si está Vivo o Caído)
    - el registro Reg que almacena las entradas con las operaciones y el mandato
    - un IndiceReg que indexa la última componente del registro Reg
    - finalmente, tenemos 4 canales para los distintos aspectos de sincronización y cambio de estados del nodo: CanalAplicarOperacion, CanalVotos, CanalCambioEstadoActual y CanalLatidos
- **ArgsPeticionVoto**: es la estructura que se manda a los nodos para solicitar su voto. Enviamos el mandato y el identificador del candidato que será el de nuestro nodo.
- **RespuestaPeticionVoto**: es la estructura que recibimos tras enviar la petición de voto. Recibimos si nos han concedido o no el voto, y el mandato del nodo al que hemos enviado la petición, para el caso en el que estuviésemos desactualizados.
- **ArgAppendEntries**: es la estructura que enviamos para mandar un latido o para comprometer una entrada del registro. Tiene las componentes mandato, identificador del líder y las entradas a someter. En el caso de enviar un latido, estas entradas estarán vacías. En la próxima práctica tendremos más componentes en este registro.
- **Results**: es la estructura recibida tras haber enviado ArgAppendEntries. Nos dice si ha tenido éxito la acción y el mandato del nodo al que ha sido enviada.

Operaciones:

- **NuevoNodo**: crea y configura un nuevo nodo en raft, inicianizándolo y poniendo en marcha la máquina de estados de los nodos con una goroutine.
- **para**: sirve para desactivar un nodo raft (es decir, la simulación de la caída de un nodo).
- **obtenerEstado**: devuelve el estado actual de un nodo (índice del nodo, mandato actual, si es líder o no, índice del líder).
- **someterOperacion**: si el nodo no es el líder, devolverá un error ya que sólo el líder puede someter una operación. En caso de que sea el líder, añadirá la operación a su registro, actualizará el índice y devolverá los argumentos necesarios.
- **PedirVoto**: es la llamada RPC en la que le llega una solicitud de otro nodo para que le vote. Compara los mandatos y concede el voto o no según si el nodo solicitante está actualizado o si somos nosotros los que nos tenemos que actualizar.
- **AppendEntries**: esta llamada RPC, de momento, si recibe un latido del líder y está todo correcto, envía una señal a su canal de latidos para reiniciar el timeout. Si está desactualizado, se actualiza, y si el líder es el desactualizado, le manda actualizarse. En la siguiente práctica, gestionaremos el hecho de recibir una entrada a aplicar.
- **enviarPeticionVoto**: es la llamada RCP que envía PedirVoto al nodo indicado en los argumentos. En función de la respuesta, si le han concedido el voto, enviará una señal por el canal de votos. Si no le han concedido el voto y además está desactualizado, se actualiza y pasa a ser seguidor.
- **maquinaEstados**: es la máquina de estados de los nodos del algoritmo, muy similar a la vista en clase. Se encarga de comprobar el estado del nodo y redirigir a su máquina de estados correspondiente.
- **maquinaEstadosSeguidor**: el seguidor programa su timeout y se queda esperando a que le llegue un latido o se agote el tiempo. Si le llega un latido, resetea el timer. Pero si se agota el tiempo, quiere decir que no hay un líder y pasa a ser candidato.
- **maquinaEstadosCandidato**: el candidato programa el timer y comienza una elección. Si se agota el timer y no ha recibido respuesta o la mayoría de los votos, vuelve a iniciar otra elección, incrementando el mandato y reseteando los votos y el timer. Si recibe un latido, es que hay otro líder en curso, entonces pasaría a ser seguidor. Si recibe un voto, comprueba si tiene la mayoría, y en ese caso pasaría a ser líder. Si recibe que tiene que cambiar de estado, acabaría y volvería a la máquina de estados global para ser redirigido.
- **maquinaEstadosLider**: el líder envía cada cierto tiempo un latido a todos los nodos menos a él, para que el resto sepan que sigue vivo. Cuando se agota este tiempo, vuelve a enviar los latidos. Si le llega que debe cambiar de estado, acaba su ejecución y vuelve a la máquina de estados global para ser redirigido.

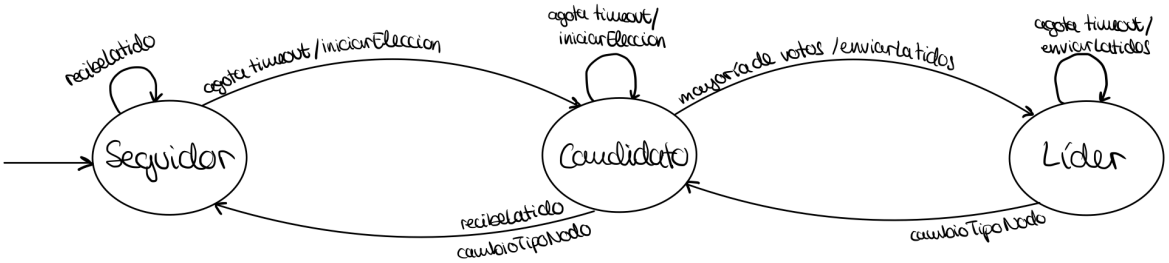
- **iniciarEleccion**: el candidato inicia una elección y envía la petición a todos los nodos vivos excepto a él mismo, lanzando una goroutine de enviarPeticionVoto.

- **enviarLatidos**: manda a todos los nodos menos a él mismo un latido mediante la goroutine enviar1Latido (ArgAppendEntries con las entradas vacías).

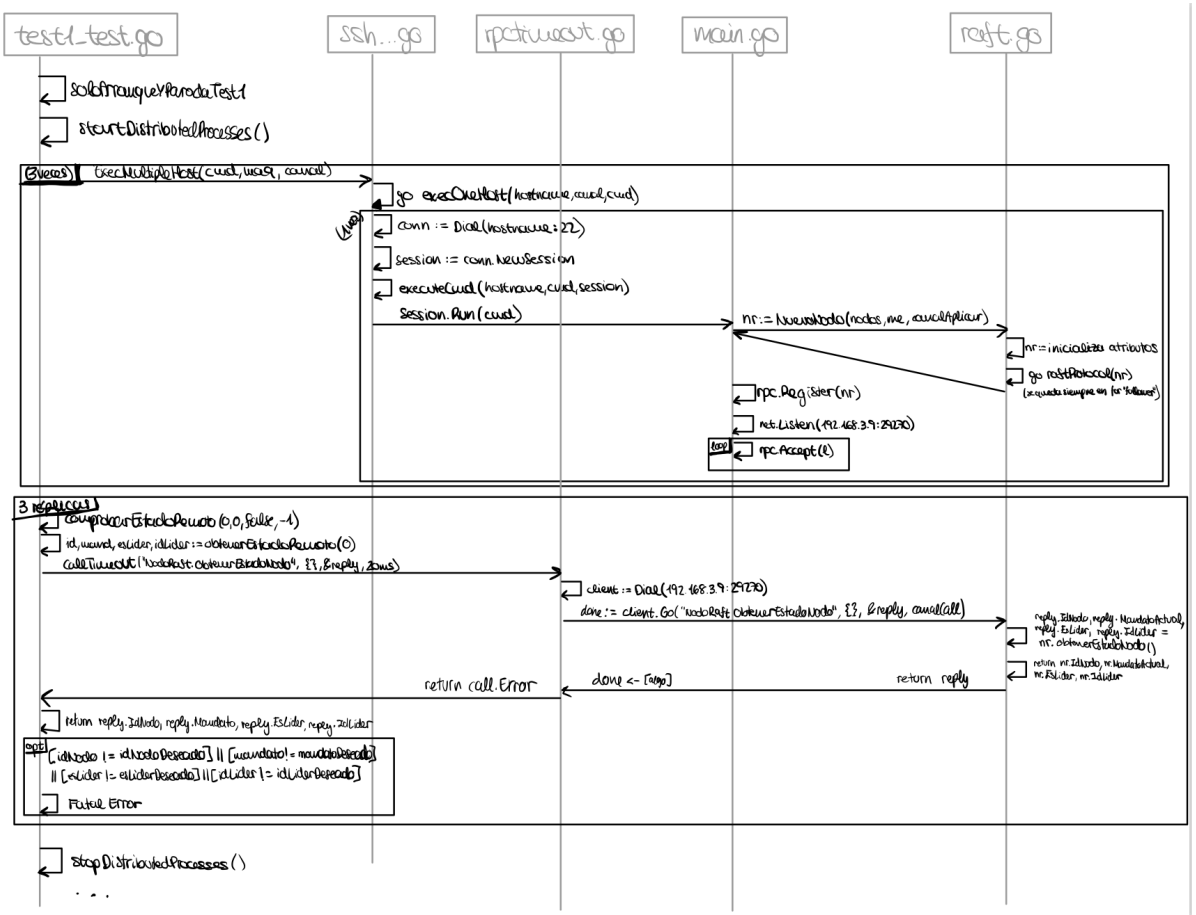
- **enviar1Latido**: envía un latido a un nodo en concreto mediante la llamada CallTimeout para que ejecute AppendEntries. En caso de que haya errores, o el mandato del líder esté desactualizado, el líder pasará a ser seguidor.

# Diagramas de secuencia:

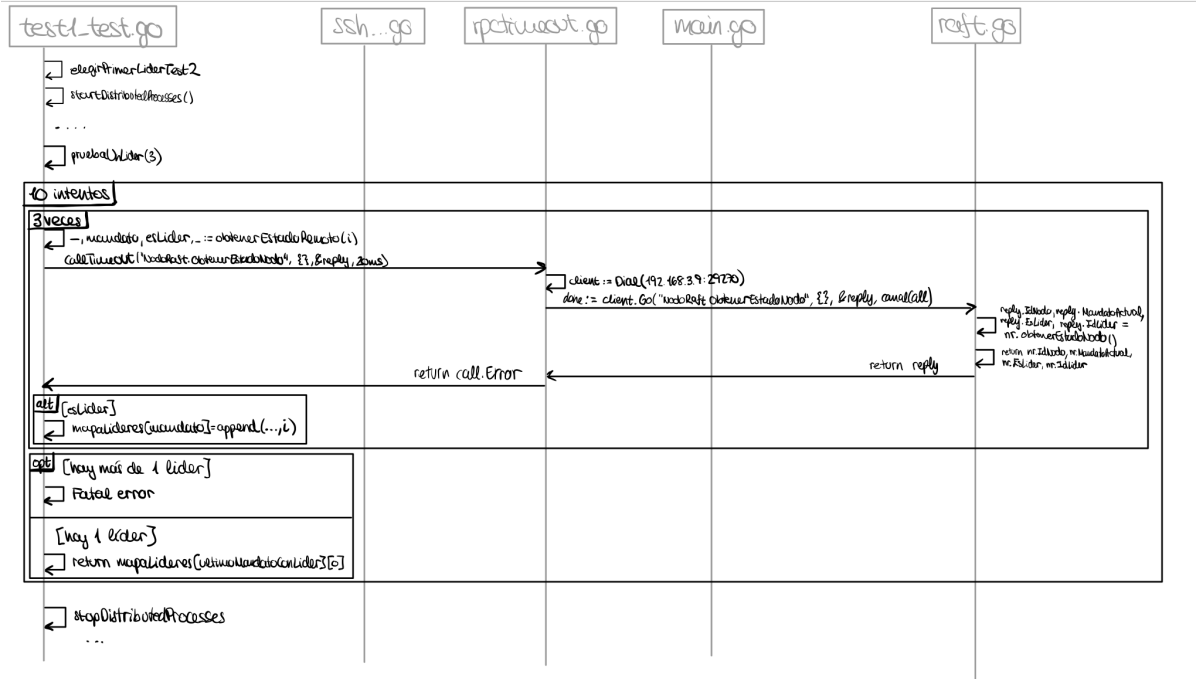
Primeramente, queremos presentar la **máquina de estados** seguida en nuestro programa. Más concretamente, las acciones realizadas por los nodos en función de su estado actual, las cuales están explicadas anteriormente.




A continuación, tenemos el diagrama de secuencia que sigue el **Test1**, el cual se encarga de comprobar que los nodos han sido creados correctamente. Primero, inicia los procesos distribuidos en las 3 réplicas, para lo cual crea la sesión ssh y crea el nodo tal y como ha sido especificado anteriormente. Una vez creados los 3 nodos y puestos en marcha con su máquina de estados, comprueba que el estado sea el correcto, es decir, que aún no haya ningún líder y todos tengan el mandato 0. Finalmente, para los procesos distribuidos.



El **Test2** comienza realizando un protocolo similar al primero, iniciando los procesos distribuidos. Ahora prueba durante varios intentos, si se ha elegido un líder correctamente, comprobando cada cierto tiempo el estado de los nodos. Si se ha elegido únicamente un líder, finalizará correctamente la ejecución, pero si no se ha elegido ninguno o más de uno, devolverá un error.




Por último, el **Test3** también realiza un proceso similar al Test2. Inicia los procesos distribuidos, prueba que se haya elegido un líder correctamente y en este caso, paraliza la ejecución del líder, dejando el nodo como caído. Entonces, vuelve a comprobar que se elija un nuevo líder correctamente entre las 2 réplicas restantes.