Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Practica 4

Raft 1ª parte

30221 - Sistemas Distribuidos

Aarón Ibáñez Espés, 779088
Ángel Espinosa Gonzalo 775750

Índice

1. Resumen **2**

2. Descripción del comportamiento e implementación **2**
2.1. Diagrama de máquina de estados 2
2.2. Diagrama de secuencias 3

3. Validación Experimental **4**

4. Conclusiones **4**

5. Anexo 1 **4**
5.1. Resultados de los test 4

1. Resumen

En esta practica se ha implementado la primera parte del algoritmo de consenso distribuido *Raft*. Para ello se supone el funcionamiento del sistema sin fallos, lo que facilita su implementación. La tolerancia a fallos se realiza en la siguiente parte.

2. Descripción del comportamiento e implementación

2.1. Diagrama de máquina de estados

En la Figura 1 se muestra el diagrama de maquina de estados de Raft, en el cual se pueden diferenciar sus tres estados posibles para un nodo, así como los eventos que desencadenan un cambio de estado.

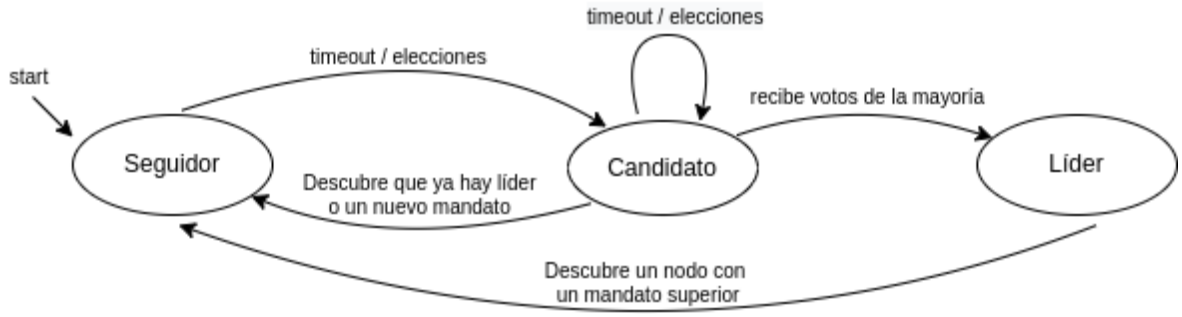

Figura 1: Diagrama de máquina de estados

Las funciones encargadas de simular el comportamiento de la maquina de estados en la implementación presentada son las siguientes:

- **gestionDeLider():** Esta rutina corresponde con el estado de nodo *Seguidor*, en ella se establece un timeout máximo en el que un nodo puede estar sin recibir latidos del Líder, en caso de que en ese tiempo no se haya recibido ningún latido el nodo asume que el Líder se ha desconectado y se comienzan elecciones. El timeout comprende un tiempo de entre 150 y 300 ms generado de forma aleatoria para evitar que varios nodos entren en elecciones al mismo tiempo.
- **elecciones():** Al comienzo de las mismas, el nodo cambia su estado de *Seguidor* a *Candidato*, incrementa el mandato actual en uno y se vota a si mismo, tras esto, envía concurrentemente la solicitud de voto al resto de los nodos. Si se obtiene mayoría de votos, el nodo se convierte en el líder del nuevo mandato y en caso contrario, se reinician las elecciones hasta encontrar un nuevo líder.
- **becomeLeader():** Esta rutina corresponde con el estado de *Lider*. Se incia en el momento que un nodo ha ganado las elecciones y en ella se cambia el estado a *Lider* y se comienza con el envió de latidos al resto de los seguidores. Dichos latidos son llamadas *AppendEntries()* con el campo entries vacío.
- **becomeFollower():** Esta rutina se utiliza para realizar la transicion a estado *Seguidor*. Se encarga de modificar el estado del nodo a *Seguidor*, establecer su voto a nulo y modificar el mandato al recibido por parametro, además de iniciar la gorutina *gestionDeLider()*

2.2. Diagrama de secuencias

A continuación se muestra un ejemplo de ejecución del algoritmo en un escenario con tres nodos.

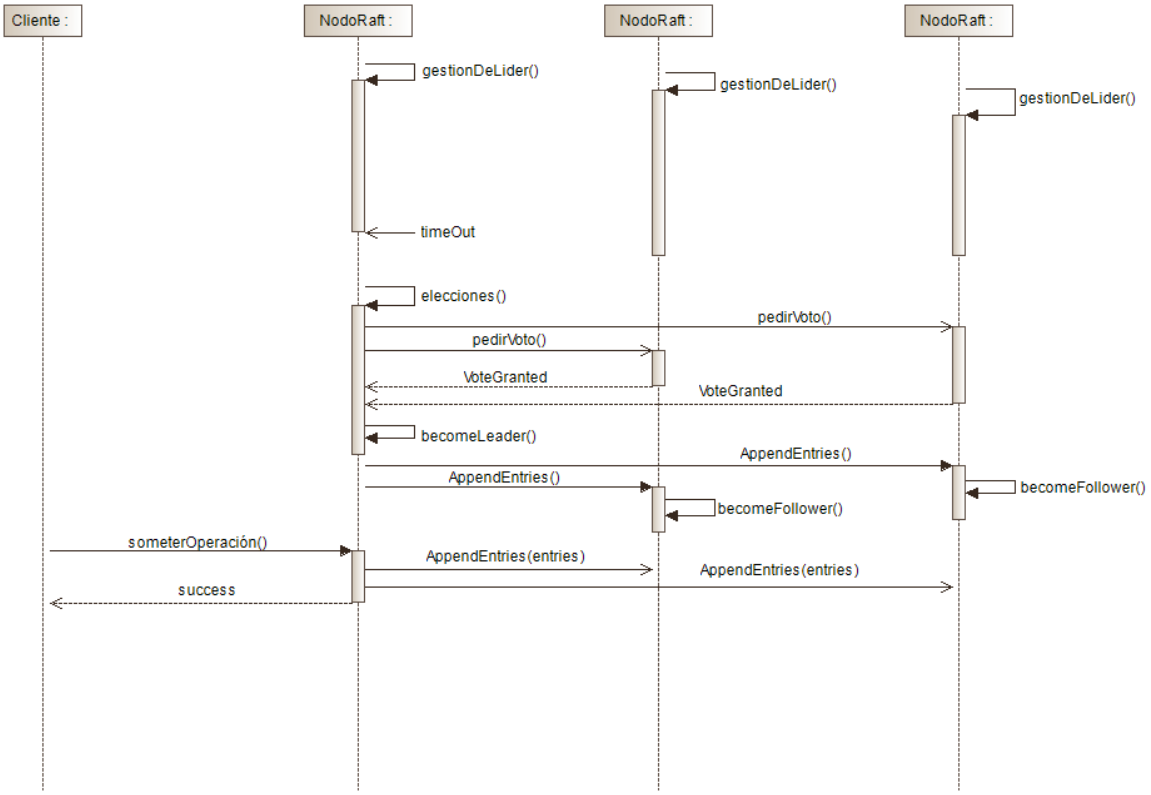

Figura 2: Diagrama de intercambio de mensajes

Incialmente todos los nodos se inician con estado *Seguidor*, por lo que ejecutan la rutina *gestionDeLider()*. Al no haber líder, el nodo que tenga un timeout menor, en este caso el nodo1, comienza elecciones. Para ello envía una petición de voto al resto de los servidores y espera su respuesta, una vez se ha obtenido la mayoría, el nodo ganador se convierte en líder del primer mandato y comienza a enviar latidos al resto de servidores para comunicarles que sigue vivo.

En el momento en el que un cliente hace una petición para someter una operación, en caso de haber sido realizada al líder se envía success y sino denied. Posteriormente, se replica la entrada en el resto de servidores para poder garantizar su persistencia.

3. Validación Experimental

Para la validación se ha utilizado el modulo de testing de golang.
Para comprobar la correcta implementación del algoritmo se han realizado los siguientes test.

- **soloArranqueYparadaTest1:** En este test se comprueba que se pueden lanzar y parar nodos de forma correcta, sin errores.
- **ElegirPrimerLiderTest2:** En este test se comprueba que tras lanzar los nodos se ha iniciado un proceso de elección quedando como máximo un lider por mandato.
- **FalloAnteriorElegirNuevoLiderTest3:** Una vez el test anterior se ha superado, en este se prueba que tras caerse un lider, se elige un nuevo lider para el siguiente mandato.
- **tresOperacionesComprometidasEstable:** Tras comprobar que el despliegue y elección de líder funcionan de forma correcta, finalmente, se comprueba que se pueden comprometer 3 entradas.

Para la ejecución de los test se cuenta con dos funciones disponibles, *startLocalProcesses* y *startDistributedProcesses*, las cuales permiten la realización de los test en un entorno local y distribuido respectivamente.

En el Anexo 1 se dispone de una imagen con el resultado de los test y un breve comentario sobre el mismo.

4. Conclusiones

En esta practica hemos podido comprobar que pese a ser Raft un algoritmo pensado para ser mas fácil de comprender e implementar, este tiene sus dificultades. Sin embargo, una vez se ha conseguido entender el completo funcionamiento del mismo se ha podido comprobar su utilidad como algoritmo de consenso y replicación distribuida, ya que permite mantener la consistencia de las entradas en caso de fallos como particiones de red.

5. Anexo 1

5.1. Resultados de los test

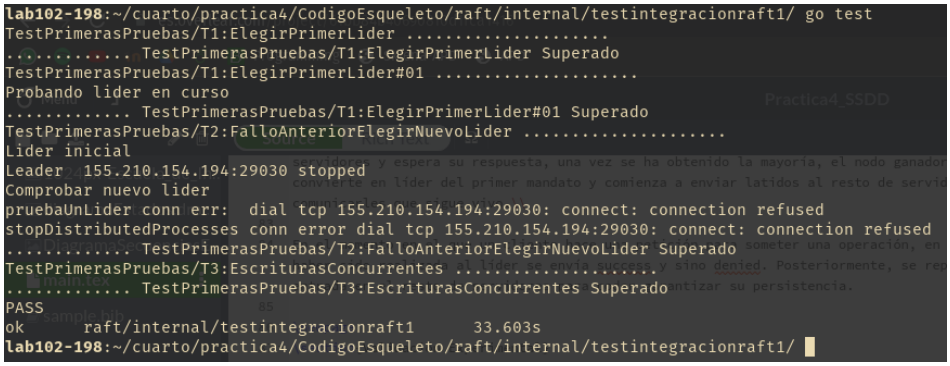

Figura 3: Resultados de los test de validación

Como se puede ver en la imagen superior el algoritmo implementado es capaz de superar todos los test de forma satisfactoria. En el *FalloAnteriorElegirNuevoLider* se puede ver como aparece un error de conexión con el nodo 155.210.154.194, pero esto es debido a que previamente el nodo ha sido detenido para comprobar que tras caer un líder, se elegía otro nuevo.