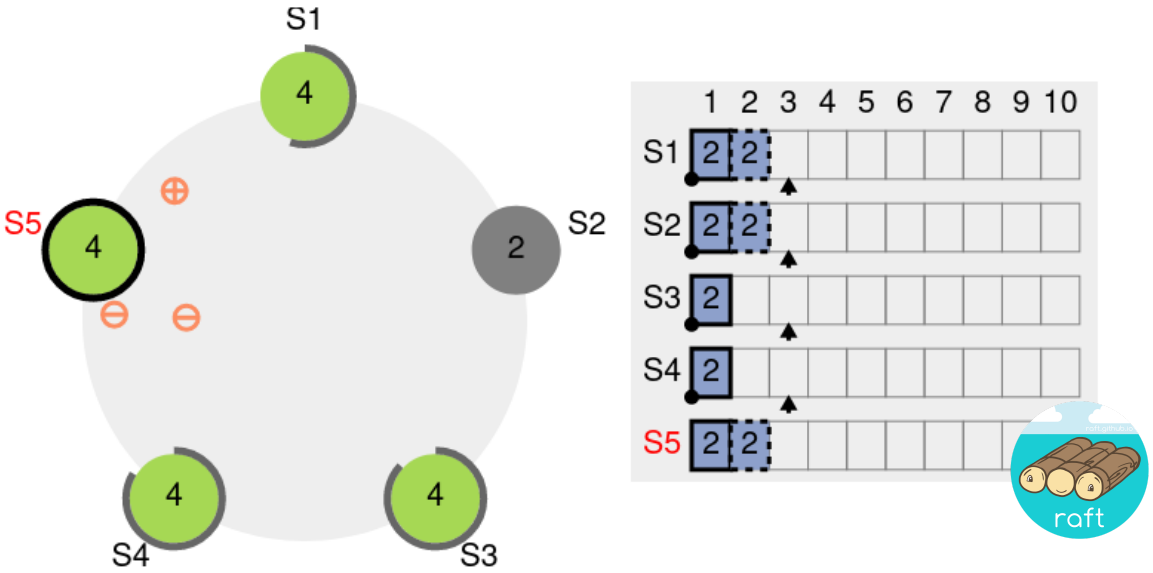
Práctica 4:
Raft 2ª parte



Sistemas Distribuidos

Universidad de Zaragoza, curso 2022/2023

Ayelen Nuño Gracia 799301

Loreto Matinero Angusto 796598

Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Descripción del problema:
En esta práctica se va a abordar el problema de la implementación del algoritmo de raft con tolerancia a fallos, y el uso de un servicio de almacenamiento representado mediante el tipo map de golang.

Diagrama de estados general:

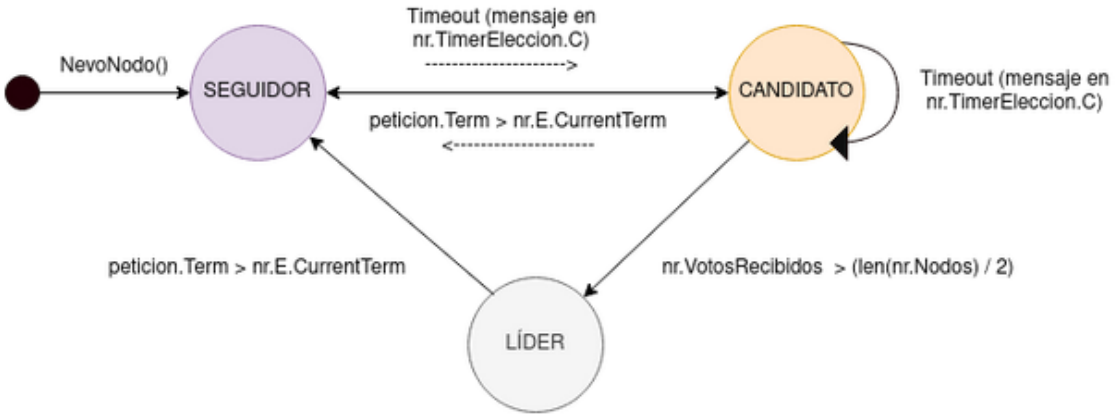
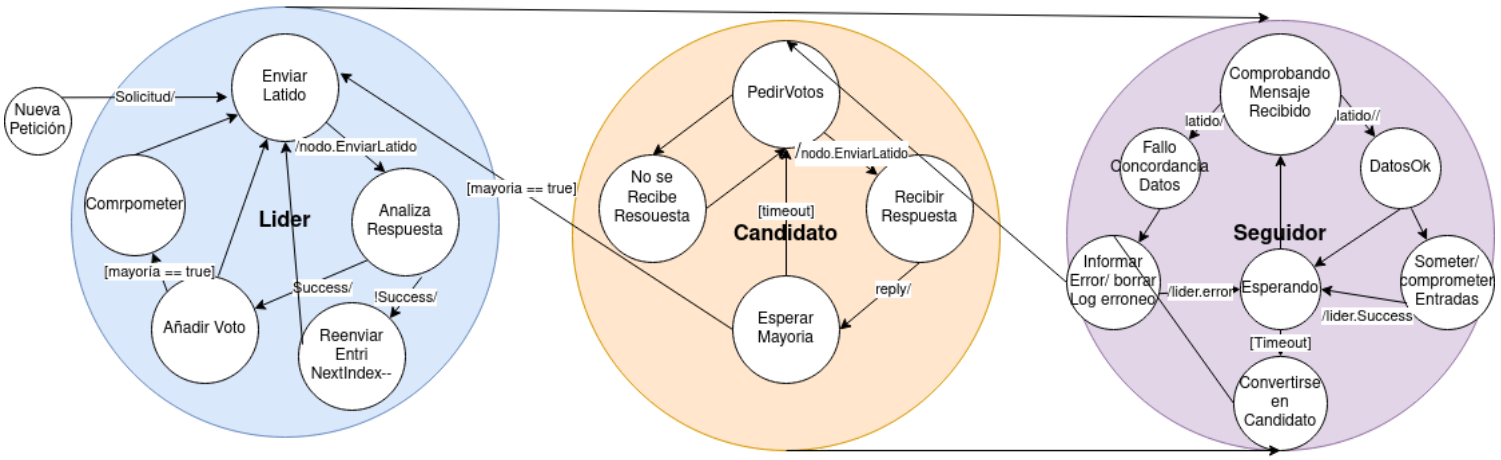
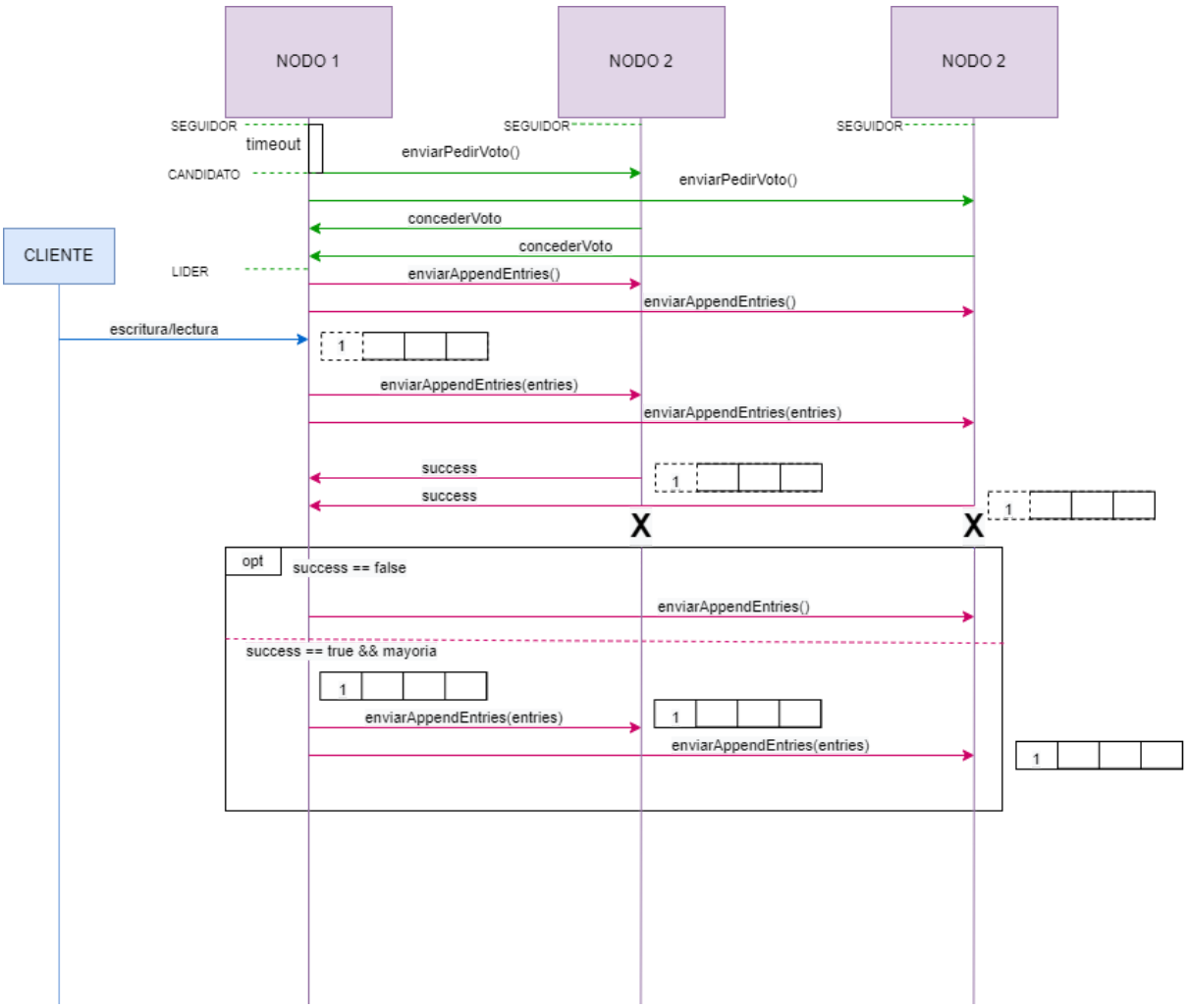


Diagrama de estados específico:

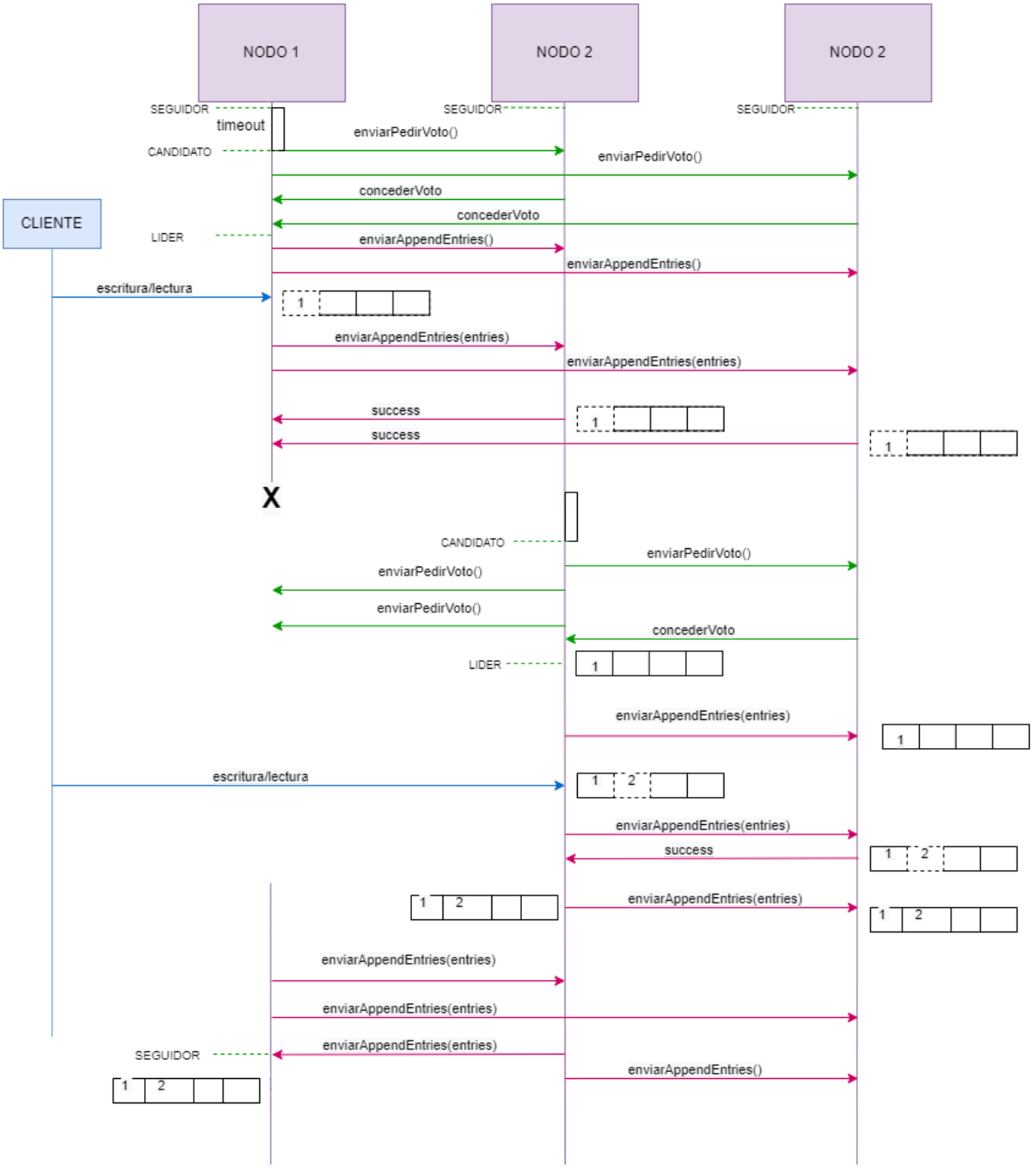


Diagramas de secuencia.

- **Caen seguidores antes de que el líder comprometa una entrada:**



- **Líder cae antes de comprometer una entrada y vuelve a conectarse:**



Cambios respecto a la práctica anterior.

El primer cambio realizado respecto de la práctica anterior es en la funcion `ConvertirseEnCandidato()`. Cuando en un sistema formado por una serie de nodos el líder se cae, o simplemente aún no ha sido asignado ninguno de los nodos como líder, cuando a uno de esos nodos se le acabe su timeout entrará en estado de candidato y tendrá que enviar peticiones de voto a los demás nodos para así una vez obtenida la mayoría se convierta en líder.

En la práctica anterior no se abarcaba el caso en el que alguno de esos nodos no estuviera activo, por lo que cuando un nodo se convertía en candidato mandaba una petición de voto a los demás nodos y si alguno de estos no estaba activo no se volvia a mandar la petición a ese nodo, en esta práctica se tiene que comprobar la respuesta que se recibe al realizar la llamada de la función enviarPeticionVoto, si los nodos no nos han respondido ni voto concedido ni voto rechazado, se tiene que volver a enviar la petición de voto.
Para realizar esto se ha añadido la función `mandarSpamVoto()`, donde se comprueba la respuesta de la función `enviarPeticionVoto()` y si es el caso se vuelve a enviar la petición de voto al nodo correspondiente.

```
func (nr *NodoRaft) mandarSpamVoto(nodo int, args
*ArgsPeticionVoto,
  reply *RespuestaPeticionVoto) {
  recepcioConfirmada := false
  for !recepcioConfirmada && nr.Roll == CANDIDATO {
     recepcioConfirmada = nr.enviarPeticionVoto(nodo, args, reply)
  }
}
```

Los siguientes cambios que se han realizado son principalmente en las funciones `AppendEntries()` y `enviarAppendEntries()`, para así controlar el correcto envío de entradas que envía un líder a sus seguidores y que se comprometan las operaciones correspondientes solicitadas por clientes a pesar de que alguno de los nodos seguidores se haya caído o haya dejado de estar accesible por el líder en algún instante de la ejecución.

En primer lugar se ha modificado la función `AppendEntries()` añadiendo una condición con la que los seguidores puedan comprobar si su registro `Log` no contiene una entrada en `prevLogIndex` cuyo término coincida con `prevLogTerm`, esto significa que el registro de ese seguidor no está bien actualizado y por ello tendrá que avisar a el líder, el cual volverá a enviarle las `Entries` correspondientes.
Para poder añadir las nuevas entradas al registro del servidor en primer lugar se tiene que truncar el registro desde el índice `prevLogIndex` para evitar entradas de registro conflictivas, y poder añadir las nuevas entradas enviadas por el líder.

Después se ha modificado la función `enviarAppendEntries()`. Como hemos comentado sobre la función `AppendEntries` se ha añadido que el servidor informe al líder cuando el término que se encuentra en el índice `prevLogIndex` no coindice con el del líder en ese mismo índice, para esto se ha añadido un caso en la funcion `enviarAppendEntries` mediante el cual el líder además de decrementar el `NextIndex` del nodo que le ha respondido, vuelve a enviarle las entradas para ver si ahora el seguidor ya puede añadirlas a su registro o se tiene que seguir decrementado su `NextIndex`.
Otro aspecto que se ha modificado dentro de esta función es cuando el líder puede comprometer una entrada.
Con todos los cambios realizados, antes de comprometer una entrada se tiene que comprobar que una mayoría de servidores tienen su registro Log correcto y que además el líder actual el cual quiere comprometer las entradas tiene un mandato igual al actual.

Por otro lado se ha implementado una función la cual va estar escuchando de forma constante el canal `CanalAplicaOp`, y dependiendo del tipo de operación almacenará información mediante la escritura, o leerá el almacén mediante la operación de lectura.

```
func (nr *NodoRaft) gestionALmacenDatos() {
  for {
      Dato := <-nr.CanalAplicaOp
      if Dato.Operacion.Operacion == "escritura" {
                            nr.AlmacenDeDatos[Dato.Operacion.Clave]=
Dato.Operacion.Valor

      }
  }
}
```

Las operaciones que se añaden a este almacén (propio de cada nodo e implementado como un tipo map) llegarán por el canal `CanalAplicaOp` cada vez que un líder o un seguidor quiera comprometer una o varias entradas.

Pruebas realizadas.

1. Se consigue acuerdos de varias entradas de registro a pesar de que un réplica (de un grupo Raft de 3) se desconecta del grupo

La prueba consiste en tirar uno de los nodos de raft, de esta forma sigue existiendo mayoría, por tanto al solicitar el cliente una entrada esta es comprometida por los otros dos nodos.

```
TestPrimerasPruebas/T5:AcuerdoAPesarDeDesconexionesDeSeguidor_ .....................
Desconectamos un nodo
Me pienso si lo mato  true 0
Me pienso si lo mato  false 1
Me muero localhost:29002
Me pienso si lo mato  false 2
El  nodo  localhost:29002  esta muerto
- TestPrimerasPruebas/T5:AcuerdoAPesarDeDesconexionesDeSeguidor_ Se han comprometido las entradas en los nodos vivos
- TestPrimerasPruebas/T5:AcuerdoAPesarDeDesconexionesDeSeguidor_ Reconectamos el nodo
Conectamos localhost:29002
- TestPrimerasPruebas/T5:AcuerdoAPesarDeDesconexionesDeSeguidor_ Se han comprometido las entradas en todos los nodos
.............. TestPrimerasPruebas/T5:AcuerdoAPesarDeDesconexionesDeSeguidor_ Superado
```

2. NO se consigue acuerdo de varias entradas al desconectarse 2 nodos Raft de 3.

Dado que el número de nodos es 3 para poder comprometer una entrada se necesita que al menos dos de los nodos respondan y guarden esa entrada. Al desconectarse dos de las tres entradas es imposible que la solicitud sea comprometida, por lo que quedará sometida en el Log del líder, pero sin llegar a comprometerse en ningún instante.

```
TestPrimerasPruebas/T6:SinAcuerdoPorFallos_ .....................
Me pienso si lo mato  true 0
Me pienso si lo mato  false 1
Me muero localhost:29002
Me pienso si lo mato  false 2
Me pienso si lo mato  true 0
Me pienso si lo mato  false 1
Me pienso si lo mato  false 2
Me muero localhost:29003
El  nodo  localhost:29002  esta muerto
El  nodo  localhost:29003  esta muerto
Conectamos localhost:29002
Conectamos localhost:29003
.............. TestPrimerasPruebas/T6:SinAcuerdoPorFallos_ Superado
```

3. Someter 5 operaciones cliente de forma concurrente y comprobar avance de indice del registro.

Esta última prueba es sencilla consiste en solicitar cinco operaciones a los nodos, funcionando correctamente todos ellos, de esta forma las cinco entradas son comprometidas en la máquina de estados de cada uno de los nodos.

```
=== RUN   TestPrimerasPruebas/T7:SometerConcurrentementeOperaciones_
TestPrimerasPruebas/T7:SometerConcurrentementeOperaciones_ .....................
.............. TestPrimerasPruebas/T7:SometerConcurrentementeOperaciones_ Superado
```

Resumen general de la ejecución:

```
--- PASS: TestPrimerasPruebas/T2:ElegirPrimerLider (20.17s)
--- PASS: TestPrimerasPruebas/T3:FalloAnteriorElegirNuevoLider (22.34s)
--- PASS: TestPrimerasPruebas/T4:tresOperacionesComprometidasEstable (22.04s)
--- PASS: TestPrimerasPruebas/T5:AcuerdoAPesarDeDesconexionesDeSeguidor_ (26.05s)
--- PASS: TestPrimerasPruebas/T6:SinAcuerdoPorFallos_ (26.07s)
--- PASS: TestPrimerasPruebas/T7:SometerConcurrentementeOperaciones_ (22.06s)
```