Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Practica 5

Raft 2ª parte

30221 - Sistemas Distribuidos

Aarón Ibáñez Espés, 779088
Ángel Espinosa Gonzalo 775750

Índice

1. Resumen

En esta practica se ha finalizado la implementación de raft, implementando los últimos detalles del algoritmo para garantizar una elección de líder segura y la consistencia en los logs. Además, se ha implementado un servicio de almacenamiento clave valor en RAM.

2. Cambios realizados respecto a la practica 4

En esta practica se han implementado las medidas de seguridad para garantizar la correcta elección del líder y replicación de las entradas.

- Para prevenir que un candidato gana las elecciones cuando su log no contiene todas las entradas comprometidas, se ha añadido la restricción de que para darle el voto, su log debe estar como mínimo actualizado con todas las entradas comprometidas. Para determinar que log se esta más actualizado se comparan los índices de mandato. El log más actualizado sera el que tenga como ultima entrada un índice de mandato superior, y en caso de ser del mismo mandato, el más actualizado será el de mayor longitud.

- Debido a la concurrencia a la hora de comprometer operaciones, se ha implementado una rutina, a la que llama *AppendEntries*, para determinar el índice a partir del cual se debe escribir. Para ello se busca el punto en el que el mandato no coincida entre el log del nodo, partiendo de $prevLogIndex + 1$ con las entradas que se quieran comprometer.

- El líder mantiene un registro del siguiente índice a indexar en cada nodo en la variable *NextIndex*. Dicha variable solo se actualiza en el caso de haber recibido una respuesta de éxito por parte de la llamada *AppendEntries*, en caso contrario, se decrementa el valor de *NextIndex* en uno y se reintenta la petición. Esto garantida que el log de los seguidores es idéntico al del líder, ya que se replica a partir de la ultima entrada que coincide en ambos.

- Debido a que el entorno de la practica 5 ya no es un entorno sin fallos, se ha implementado una comprobación adicional en la función *checkReply()*, en la que se comprueba que la entrada que se ha solicitado comprometer se ha replicado en la mayoría de las replicas y por ello es estable y segura y se puede almacenar en el almacén. Para comprobar que la entrada se ha replicado en la mayoría de las replicas, se utiliza el campo del líder *MatchIndex*, el cual almacena el índice de la ultima entrada almacenada en el cada nodo seguidor.

2.1. Diagrama de secuencias inicialización de Raft

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

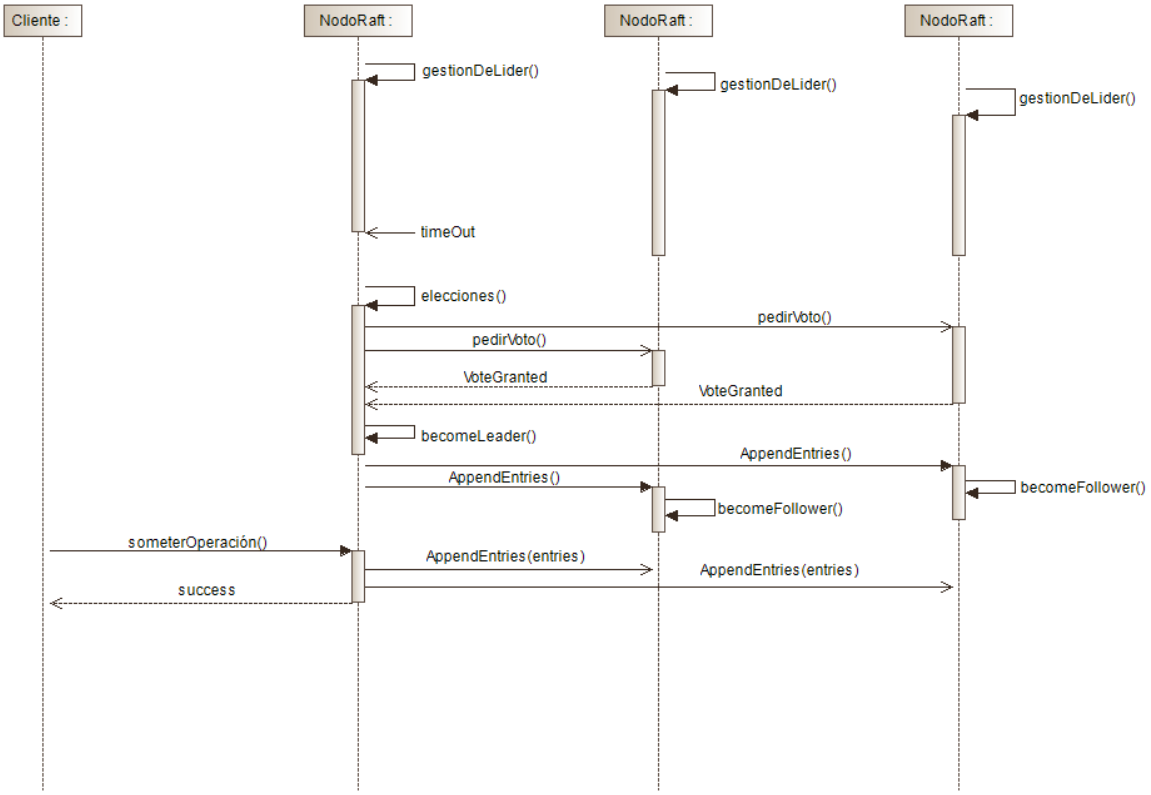

Figura 1: Diagrama de intercambio de mensajes

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. Almacén Clave/Valor

Para la implementación del almacén clave/valor se ha creado un nuevo paquete llamado *almacenamiento*, el cual dispone de varias operaciones para inicializar el almacén, leer, escribir y mostrar los datos almacenados.

Para inicializar el almacén, en el proceso main se ha implementado una *gorutina* en la que se crea el almacén y se queda escuchado el canal *canalAplicar*, por el cual le van a llegar los datos de tipo *AplicaOperacion* con la información necesaria para almacenar los datos en el almacén clave/valor.

El proceso de almacenamiento se ilustra con los dos siguientes diagramas de secuencias. En el primero se muestra el proceso para almacenar una entrada si el nodo es *Líder* y en el segundo se muestra el mismo procedimiento en caso de nodo *Seguidor*.

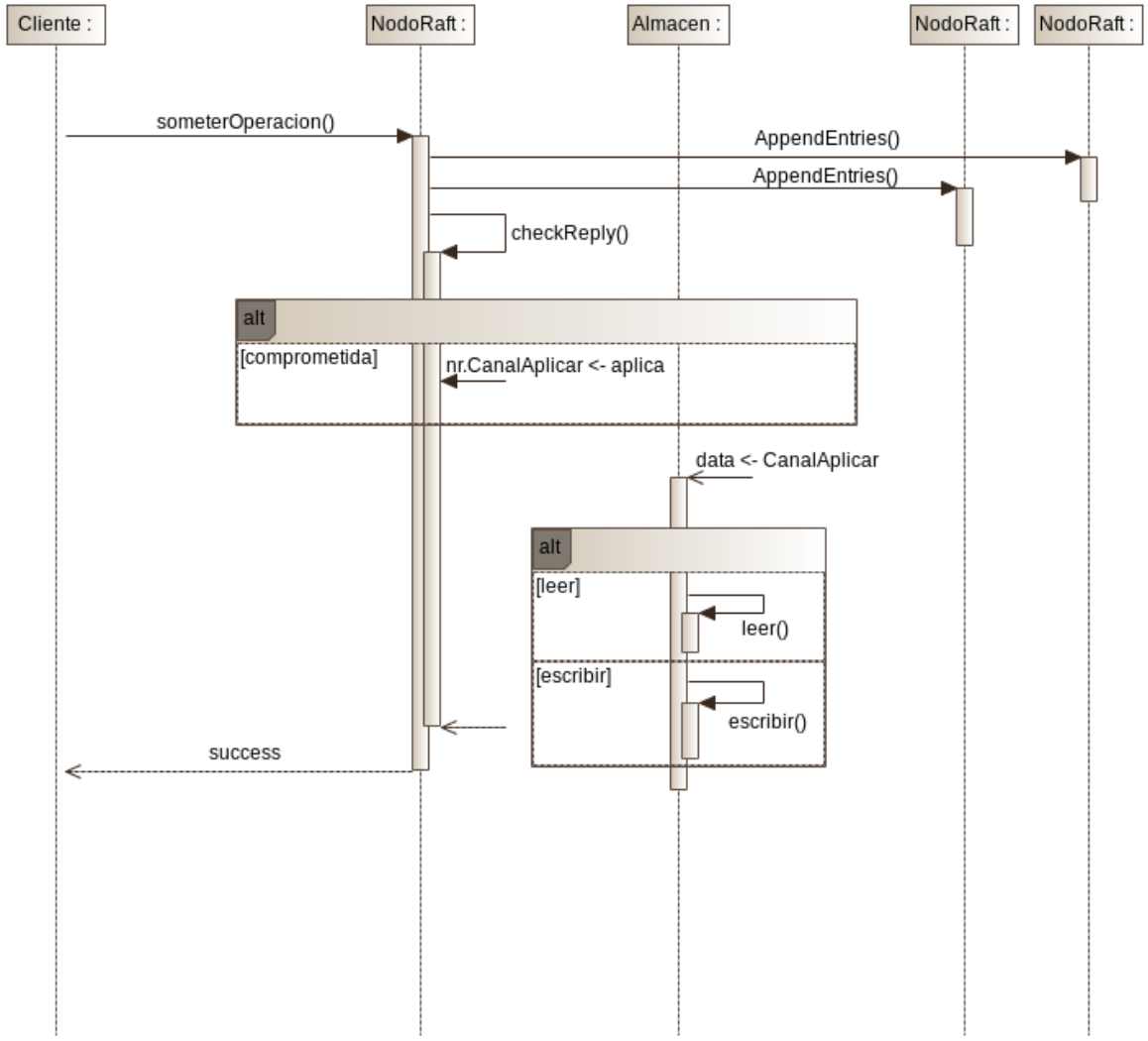

Figura 2: Diagrama de de secuencias para almacenar una operación en el líder

En el momento en el que un cliente hace una petición para someter una operación, el líder añade esa entrada a su *log* y hace una llamada *AppendEntries* con la entrada recibida para replicarla en el resto de seguidores y poder garantizar su persistencia. Después comprueba si se ha replicado correctamente en la mayoría de las replicas, y si es así, se envía la entrada por el *canalAplicar* para añadirla al almacén.

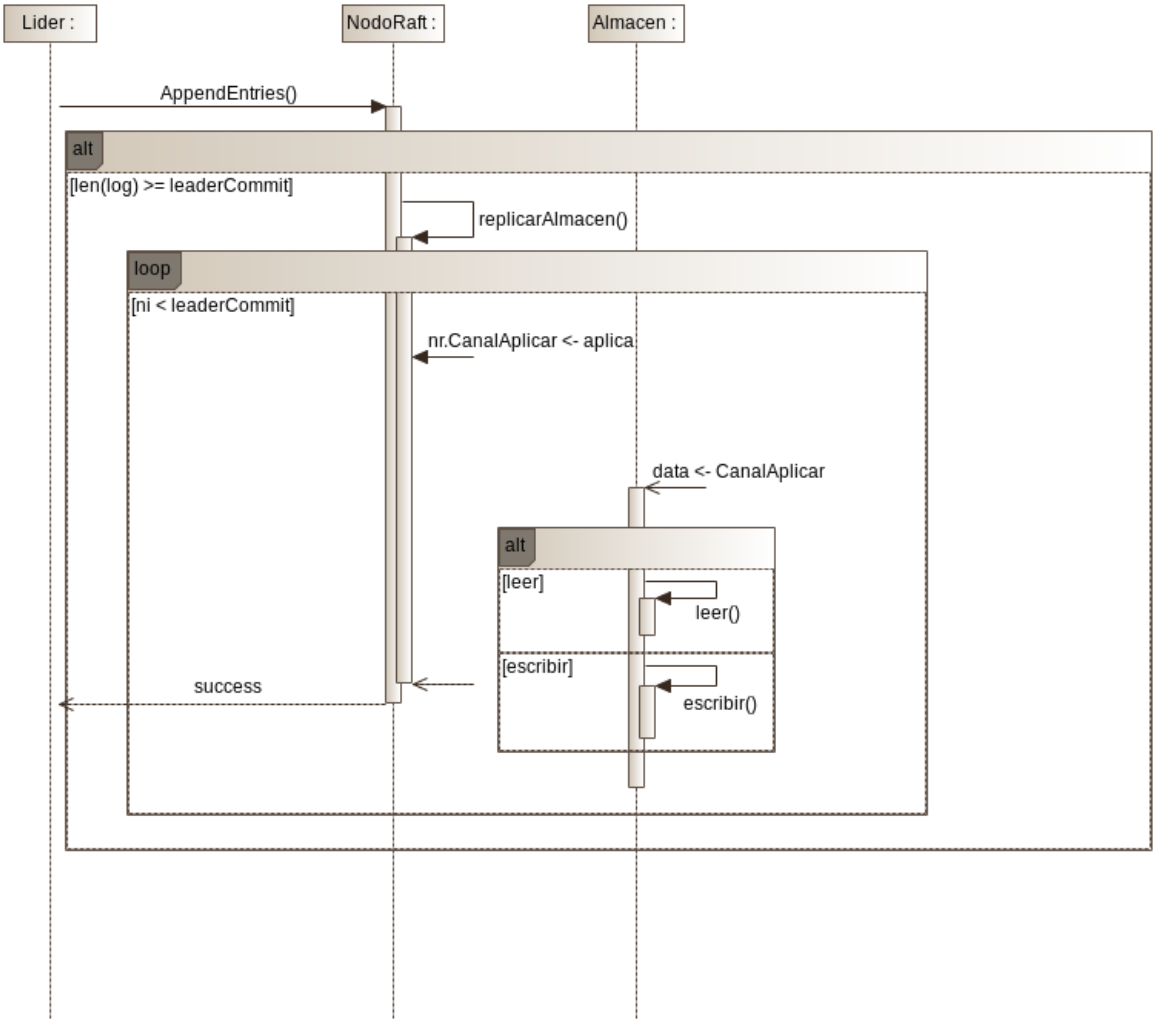

Figura 3: Diagrama de de secuencias para almacenar una operación en un seguidor

En el caso de los nodos *Seguidor*, cada vez que se recibe una petición de tipo *AppendEntries*, se comprueba si el argumento recibido por parámetro *leaderCommit*, que indica el índice de la ultima entrada comprometida del líder, es superior a su *commitIndex*, que almacena la ultima entrada comprometida del nodo. En caso de que eso se cumpla, se lanza la rutina *replicarAlmacen*, la cual se encarga de replicar el almacén del líder desde el *commitIndex* hasta el *leaderCommit*.

4. Validación Experimental

Para comprobar el correcto funcionamiento del algoritmo, se han pasado los test de la practica anterior, y además, se han implementado 3 nuevos test utilizando el modulo de testing de golang.

4.1. AcuerdoAPesarDeDesconexionesDeSeguidor

En este test se comprueba que se consigue acuerdos de varias entradas de registro a pesar de que un replica se desconecta del grupo.

```
TestAcuerdosConFallos/T5:AcuerdoAPesarDeDesconexionesDeSeguidor_ .....................
Desconecto a: 0
He encontrado al lider:  1
BIEN: 1
BIEN: 2
He encontrado al lider:  1
BIEN: 0
BIEN: 1
BIEN: 2
.............. TestAcuerdosConFallos/T5:AcuerdoAPesarDeDesconexionesDeSeguidor_ Superado
```

Figura 4: Resultado del test

Como se puede ver en la Figura 3, la ejecución del test sigue los siguientes pasos

1. Se levantan las tres replicas de nodo Raft.
2. Se desconecta uno de los nodos seguidores, en este caso en nodo con id 0.
3. Se someten 3 operaciones con el nodo 0 desconectado y se comprueba que se han sometido correctamente.
4. Se reconecta el nodo 0 y se someten 3 operaciones más.
5. Finalmente se comprueba que las entradas se han comprometido las entradas y el log de todos los nodos esta correctamente replicado.

4.2. SinAcuerdoPorFallos

En este test se comprueba que NO se consigue acuerdo de varias entradas al desconectarse 2 de los 3 nodos de Raft.

```
TestAcuerdosConFallos/T5:SinAcuerdoPorFallos_ .....................
Desconecto a: 0
Desconecto a: 2
He encontrado al lider:  1
El nodo  1  no tiene todas las entradas comprometidas
He encontrado al lider:  1
BIEN: 0
BIEN: 1
BIEN: 2
.............. TestAcuerdosConFallos/T5:SinAcuerdoPorFallos_ Superado
```

Figura 5: Resultado del test

Como se puede ver en la Figura 4, la ejecución del test sigue los siguientes pasos

1. Se levantan las tres replicas de nodo Raft.
2. Se desconectan los dos nodos seguidores, los nodos 0 y 2.
3. Se someten 3 operaciones con los nodos desconectados y se comprueba que no se ha podido comprometer la entrada.

4. Se reconectan los nodos desconectados y se someten 3 operaciones más.

5. Finalmente se comprueba que tanto las entradas pendientes como las nuevas han sido correctamente replicadas y comprometidas.

4.3. SometerConcurrentementeOperaciones

En este test se comprueba que se consiguen someter 5 operaciones cliente de forma concurrente y comprobar avance de índice.

```
TestAcuerdosConFallos/T5:SometerConcurrentementeOperaciones_ ..................
He encontrado al lider:  1
He encontrado al lider:  1
He encontrado al lider:  1
He encontrado al lider:  1
He encontrado al lider:  1
BIEN: 0
BIEN: 1
BIEN: 2
............. TestAcuerdosConFallos/T5:SometerConcurrentementeOperaciones_ Superado
PASS
```

Figura 6: Resultado del test

Como se puede ver en la Figura 5, la ejecución del test sigue los siguientes pasos

1. Se levantan las tres replicas de nodo Raft.

2. Se someten 5 operaciones de forma concurrente.

3. Finalmente se comprueba que todas las entradas han sido correctamente replicadas y comprometidas.

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.

5. Conclusiones

En esta practica hemos podido comprobar como Raft hace el trabajo de replicación y es un algoritmo útil para implementar servicios de persistencia gracias a su metodología de elección de líder y seguridad de las entradas comprometidas. Además, se pone en valor dicha función debido a los problemas que se han tenido para poder garantizar la consistencia de todos los nodos en el momento en el que ha aparecido la concurrencia.