Prácticas 3 y 4:
SISTEMAS DISTRIBUIDOS

DISEÑO E IMPLEMENTACIÓN DEL ALGORITMO DE RAFT

Jorge Hernández Aznar (872838) y Paula Blasco Díaz (874320)

ÍNDICE

1. Introducción

El objetivo de las prácticas 3 y 4 es el diseño e implementación del algoritmo de RAFT. Este es un algoritmo con gran importancia en los sistemas distribuidos ya que permite lidiar con la tolerancia a fallos en servidores con estado. En esta memoria se recoge el desarrollo llevado a cabo durante las dos prácticas correspondientes, asimismo se proporcionan las pruebas específicas para verificar la corrección del modelo realizado.

2. Diseño e Implementación

2.1 Arquitectura general

El algoritmo de RAFT se basa en un servidor con un conjunto de nodos. Entre ellos hay un nodo que se erige como ***leader*** y es el nodo que interactuara con los clientes. El resto de nodos asumen el rol de ***followers*** y estos reciben un heartbeat por parte del líder para confirmar su correcto funcionamiento y sincronizar sus estados. En caso de no detectar este heartbeat por un tiempo tomarán el rol de ***candidate*** y uno de ellos se alzará como el nuevo líder. Cada vez que se convierte en candidato aumenta el mandato, en cada mandato solo puede haber un líder y los nodos con mayores mandatos serán los más actualizados

El heartbeat puede contener las entradas recibidas por el líder, en nuestro caso representan las operaciones escritura/lectura sobre un almacén clave/valor solicitadas por el cliente. Estas van siendo replicadas en los seguidores y cuando el líder puede confirmar que hay una mayoría simple replicada, confirma la operación.

El diagrama de estados que describe el comportamiento de los nodos del algoritmo de raft se puede observar la siguiente Figura 1.

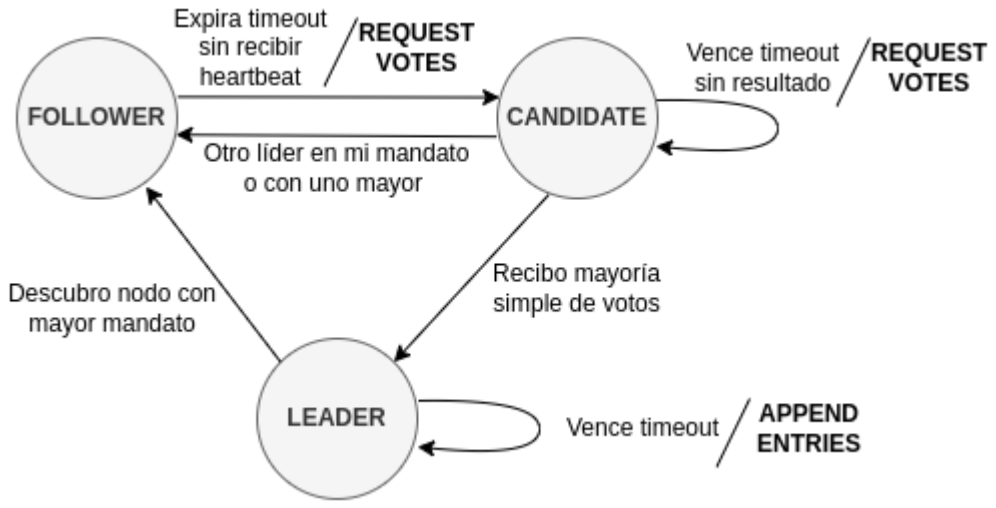

Figura 1: Diagrama de estados de los nodos del algoritmo de raft

Para llevar a cabo nuestra implementación hemos creado una función que realice estas transiciones mediante canales asíncronos y timers. El primer canal indica la conversión en follower, que se realiza cuando llega un heartbeat de un nodo con mayor mandato. El siguiente representa que se ha ganado la elección y se convierte el nodo el líder. El último canal representa que se ha recibido el heartbeat del líder y se puede reiniciar el timer. Los timers tienen distintas duraciones, el heartbeat del líder es enviado cada 50ms, los followers esperan entre 150 y 300ms para recibirlo de forma que hay tiempo más que suficiente. Finalmente en caso de no ganar la elección ni convertirse en follower, los candidatos esperan 2.5s para reiniciar una nueva solicitud de votos. La estructura que hemos seguido para el autómata es la siguiente función llamada **raftHandler**.

```
for {
	if nr.CommitIndex > nr.LastAplied {
		nr.LastAplied++
		op := AplicaOperacion{nr.LastAplied, nr.Logs[nr.LastAplied].Op}
		nr.ApplyOp <- op
		op = <-nr.ApplyOp
		nr.OpCommited <- op.Operacion.Valor
	}
	for nr.Rol == "FOLLOWER" {
		timerFollower := time.NewTimer(getRandomTimeout())
		select {
		case <-nr.Heartbeat: // Recibe el heartbeat
		// Sigo como follower
		case <-timerFollower.C: // Expira timeout
			nr.IdLider = -1
			nr.Rol = "CANDIDATE"
		}
	}
	for nr.Rol == "LEADER" {
		nr.IdLider = nr.Yo
		// Enviar heartbeats
		sendAppendEntries(nr)
		timerLeader := time.NewTimer(50 * time.Millisecond)
		select {
		case <-nr.Follower: // Descubre mandato mayor
			nr.Rol = "Follower"
		case <-timerLeader.C: // Expira el time out
			// Sigo como leader y compruebo si he actualizado el índice
			if nr.CommitIndex > nr.LastAplied {
				nr.LastAplied++
				op              :=              AplicaOperacion{nr.LastAplied,
nr.Logs[nr.LastAplied].Op}
				nr.ApplyOp <- op
				op = <-nr.ApplyOp
				nr.OpCommited <- op.Operacion.Valor
			}
		}
	}
	for nr.Rol == "CANDIDATE" {
		nr.CurrentTerm++
		nr.VotedFor = nr.Yo
		nr.VotesReceived = 1
		requestVotes(nr)
		// Tarda 2.5 segundos hasta iniciar una nueva elección
		timerCandidate := time.NewTimer(2500 * time.Millisecond)
		select {
		case <-nr.Leader: // Se convierte en líder
			nr.Rol = "LEADER"
		case <-nr.Follower: // Se convierte en follower
```

```
                        nr.Rol = "FOLLOWER"
                case <-timerCandidate.C: // Se acaba el timeout
                        nr.Rol = "CANDIDATE" // Se vuelve a presentar como candidato
                }
        }
}
```

2.2 Algoritmo de elección de líder

El aspecto más relevante para escoger un líder es si nos llega una solicitud de voto, si le otorgamos nuestro voto o no. Para ello seguimos los siguientes criterios.

- Si llega una solicitud con mandato menor o igual que el nuestro le negamos el voto y le informamos del mandato actual y de su líder.
- Si llega una solicitud con mandato mayor le concedemos el voto solamente si su logger que almacena las entradas está más actualizado. (Se considera más actualizado si el término de su última entrada es mayor que el de la nuestra o en caso de que sean iguales que tenga más entradas cometidas que nuestro logger)

Cabe destacar que si un nodo ya ha proporcionado su voto a un nodo, no puede proporcionar su voto a otro nodo en el mismo mandato. A continuación se muestra una parte de la función **PedirVoto** que representa en qué casos se otorga el voto.

```
if peticion.CandidateTerm > nr.CurrentTerm { // Si me llega un mandato mayor al mío le puedo dar el voto
        if len(nr.Logs) == 0 || isBetterLeader(nr, peticion) {
                reply.Term = peticion.CandidateTerm
                reply.VoteGranted = true
                nr.CurrentTerm = peticion.CandidateTerm
                nr.VotedFor = peticion.CandidateID
                if nr.Rol == "LEADER" || nr.Rol == "CANDIDATE" {
                        // Vuelvo a ser follower
                        nr.Follower <- true
                }
                // No le doy el voto
        } else {
                // nr.CurrentTerm = peticion.CandidateTerm no es necesario ??
                reply.Term = nr.CurrentTerm
                reply.VoteGranted = false
        }

} else { // Si llega un mandato igual o menor al mío no le doy el voto
        reply.Term = nr.CurrentTerm
        reply.VoteGranted = false
}
```

2.3 Heartbeat y replicación del logger

La replicación de las entradas por parte del líder hacia los nodos followers es una parte esencial del algoritmo. En nuestra implementación, el líder debe escoger que debe mandar a cada nodo, un heartbeat simple con una entrada vacía o la entrada correspondiente. Esto se logra con el siguiente extracto de código de la función **sendAppendEntries**.

```
        for i := 0; i < len(nr.Nodos); i++ {
                // Evita enviar a sí mismo
                if i == nr.Yo {
                        continue
                }
```

```
                // Determina el índice de la siguiente entrada a enviar
                nextIndex := nr.NextIndex[i]
                var entry Entry
                // Si nextIndex está dentro del rango del log, selecciona las entradas
                if nextIndex < len(nr.Logs) {
                        entry = nr.Logs[nextIndex]
                } else {
                        entry = Entry{} // Entrada vacía para indicar heartbeat
                }
                // Determina el índice y el término del log previo
                prevLogIndex := nextIndex - 1
                prevLogTerm := 0
                if prevLogIndex >= 0 && prevLogIndex < len(nr.Logs) {
                        prevLogTerm = nr.Logs[prevLogIndex].Term
                }
                ...
        }
```

A continuación cada nodo comunica su respuesta sobre el heartbeat, en caso de ser afirmativa se aumentan la siguiente entrada a enviar y los AKS's recibidos para poder comprometer la entrada en caso de haber recibido una mayoría. En caso contrario si no tiene un logger actualizado se intentará con la entrada previa. El siguiente extracto de código muestra el comportamiento de la función **sendAppendEntry** tras recibir una respuesta.

```
if results.Success { // Si se recibe confirmación del logger
        nr.MatchIndex[nodo] = nr.NextIndex[nodo]
        nr.NextIndex[nodo]++
        nr.Mux.Lock()

        if nr.MatchIndex[nodo] > nr.CommitIndex {
                nr.ACKsCommit++
                if nr.ACKsCommit >= len(nr.Nodos)/2 {
                        nr.CommitIndex++
                        nr.ACKsCommit = 0
                }
        }
        nr.Mux.Unlock()
// El logger del nodo solicitado es inconsistente se intenta con la Entry previa
} else {
        if args.Entries != (Entry{}) {
                nr.NextIndex[nodo]--
        }
}
```

Finalmente un nodo comunica su respuesta afirmativa en caso de recibir una entrada no vacía y tener o su logger vacío o actualizado con el líder. En esos casos afirmativos añade la entrada a su logger y actualiza el CommitIndex en caso de que sea necesario. El siguiente fragmento muestra el flujo de acciones que sigue un nodo follower al recibir la llamada RPC a la función **AppendEntries**.

```
// Heartbeat simple
if args.Entries == (Entry{}) {
        results.Success = false
// Recibido entry
} else {
        // Logger vacío
        if len(nr.Logs) == 0 {
                nr.Logs = append(nr.Logs, args.Entries)
```

```
                results.Success = true
        // Si no tengo mi logger vacío
        } else {
                results.Success = isLoggerUpdated(nr, args)
                // Mi logger coincide para PrevLogIndex en mandato
                if results.Success {
                        // Me quedo con los logs desde el inicio hasta el indice actualizado
                        nr.Logs = nr.Logs[0 : args.PrevLogIndex+1]
                        nr.Logs = append(nr.Logs, args.Entries)
                }
        }
}
if args.LeaderCommit > nr.CommitIndex {
        nr.CommitIndex = min(args.LeaderCommit, len(nr.Logs)-1)
}
```

3. Pruebas y verificación

3.1 Escenario de las pruebas

Las pruebas para verificar el funcionamiento de nuestra implementación del algoritmo de raft van a tener lugar con 3 nodos en las máquinas 192.168.3.6-8. Para ello desde la máquina 192.168.3.5 generamos y distribuimos una clave asimétrica público-privada del tipo RSA. De este modo podremos conectarnos vía ssh para levantar los procesos de raft. Asimismo trabajamos en el directorio compartido entre todas las máquinas, /misc/alumnos/sd/sd2425/aNIP, así es más sencillo compilar los ejecutables y los test para todas las máquinas. Para levantar los procesos se invoca a un ejecutable previamente compilado con go build -o main main.go en el directorio interno practica3/cmd/srvraft/. Este main invoca a la función **NuevoNodo** del módulo raft y además crea el sistema de almacenamiento clave/valor con el se comunicará el nodo raft mediante un canal llamado *canalAplicarOperacion*.

Se ha de destacar que se ha usado un script llamado *stop_raft_nodes.sh* que sirve para detener los nodos raft residuales, en caso de que haya fallado alguna prueba. Asimismo se ha utilizado los logs creados por cada nodo raft en los que hay mensajes de depuración. Además se ha creado el script *manage_logs.sh* que crea dos archivos *join.txt* y *sorted.txt*. En el primero se almacenan las 300 primeras líneas del log de cada nodo y en el segundo se almacenan ordenadas temporalmente estas 900 líneas. Gracias a esto la depuración ha sido más sencilla.

3.2 Listado de pruebas realizadas

El primer test comprueba el arranque y parada de los nodos remotos, para ello se inicializan los nodos y se ajusta un delay en el autómata de **raftHandler** para que los nodos estén sin inicializar. De este modo el mandato de los nodos es el 0 y el id del líder es -1 ya que ninguno lo es.

El segundo test realizado simplemente comprueba que al cabo de cierto tiempo hay un nodo que ha sido elegido como líder y que el resto de nodos son conscientes. Para ello la función **pruebaUnLider** verifica que solo haya un líder en el mismo mandato.

El tercer test es similar al anterior ya que después de verificar que haya un solo líder desconecta el nodo líder, por ende los 2 nodos restantes tienen que ser capaces de escoger un líder. Después se relanza el nodo detenido y se comprueba que se ha alcanzado un acuerdo para conseguir un líder en los siguientes mandatos.
El cuarto test se centra en la replicación de las entradas, para ello dado un líder procedemos a someter 3 operaciones secuenciales. Con el método **checkSometerOperation** indicamos los índices del logger y el valor devuelto esperado, siendo para las escrituras "ESCRITO CORRECTAMENTE" y para las lecturas el valor del almacén con la clave proporcionada. En este test observamos que realiza 2 escrituras y 1 lectura de forma satisfactoria.

El quinto test se diseñó para comprobar que el sistema es capaz de someter operaciones a pesar de que haya un nodo que esté desconectado gracias a la tolerancia a fallos. Para ello se compromete inicialmente una operación, a continuación se desconecta uno de los seguidores, se comprometen 3 operaciones de forma correcta, se vuelve a reconectar el nodo. Una vez estén todos reconectados se comprometen otras 3 operaciones y se comprueba el estado de los loggers de los 3 nodos siendo equivalente para los 3 con la función **checkLoggers**.

El penúltimo test está pensado para comprobar que cuando más de la mitad de los nodos están desconectados el líder es incapaz de comprometer las entradas. Para ello de forma similar al test anterior se compromete una operación inicial, se proceden a desconectar todos los seguidores y con la función **checkSometerFail** se comprueba que salta el timeout al no poder someter 3 operaciones. A continuación se relanzan todos los nodos y se comprometen otras 3 operaciones. Finalmente podemos comprobar que todas las operaciones han sido comprometidas comparando con la función checkLoggers ya usada en el test previo.

Por último, el séptimo test se encarga de validar el funcionamiento del algoritmo cuando se someten varias entradas de forma concurrente. Inicialmente se compromete una sola entrada y seguidamente se ejecutan 5 operaciones de forma concurrente utilizando las gorutines. Finalmente se comprueba que los logs son iguales para los 3 nodos. Cabe destacar que las operaciones realizadas de forma concurrente son 3 escrituras y 2 lecturas. Puede suceder que si hay una lectura sobre una clave que aún no haya sido escrita debido a no ser un código secuencial desemboque en un error. No obstante sustituyendo las 2 lecturas por escrituras este problema se solucionaría.

4. Diagramas de secuencia de interacción

El siguiente diagrama muestra como es el comportamiento de los nodos RAFT cuando a dos de ellos se les vence el timeout del heartbeat de forma simultanea y solicitan erigirse como líderes.

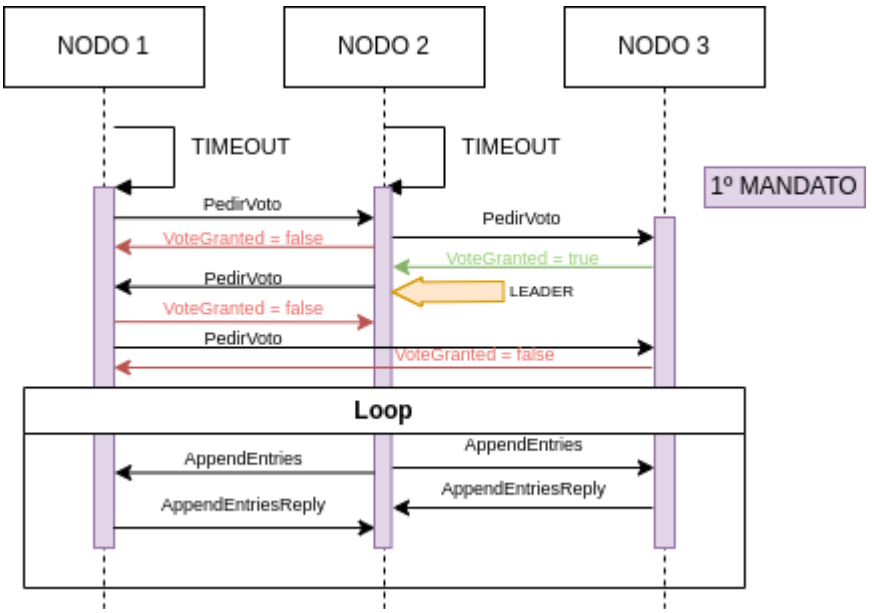

Figura 2: Diagrama de secuencias para escoger un líder

A continuación se muestra un diagrama que representa el flujo del quinto test en el que se comprometen entradas pese a la desconexión de un nodo.

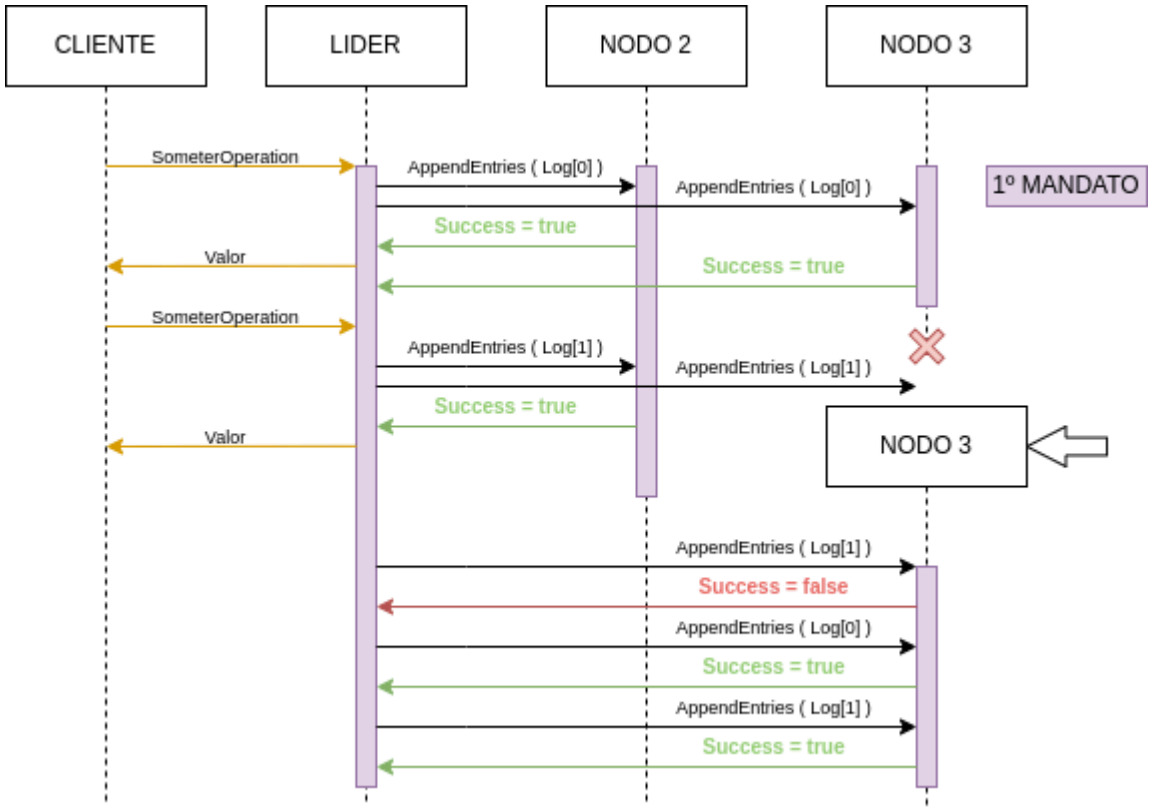

Figura 3: Diagrama de secuencias con dos entradas sometidas y un nodo desconectado

Por último, el tercer diagrama de secuencias representa como se someten dos entradas, durante la primera están los dos followers desconectados y en la segunda ya se encuentran dos conectados. El flujo de mensajes intercambiados es muy similar a la tercera y cuarta operación sometidas durante el sexto test.

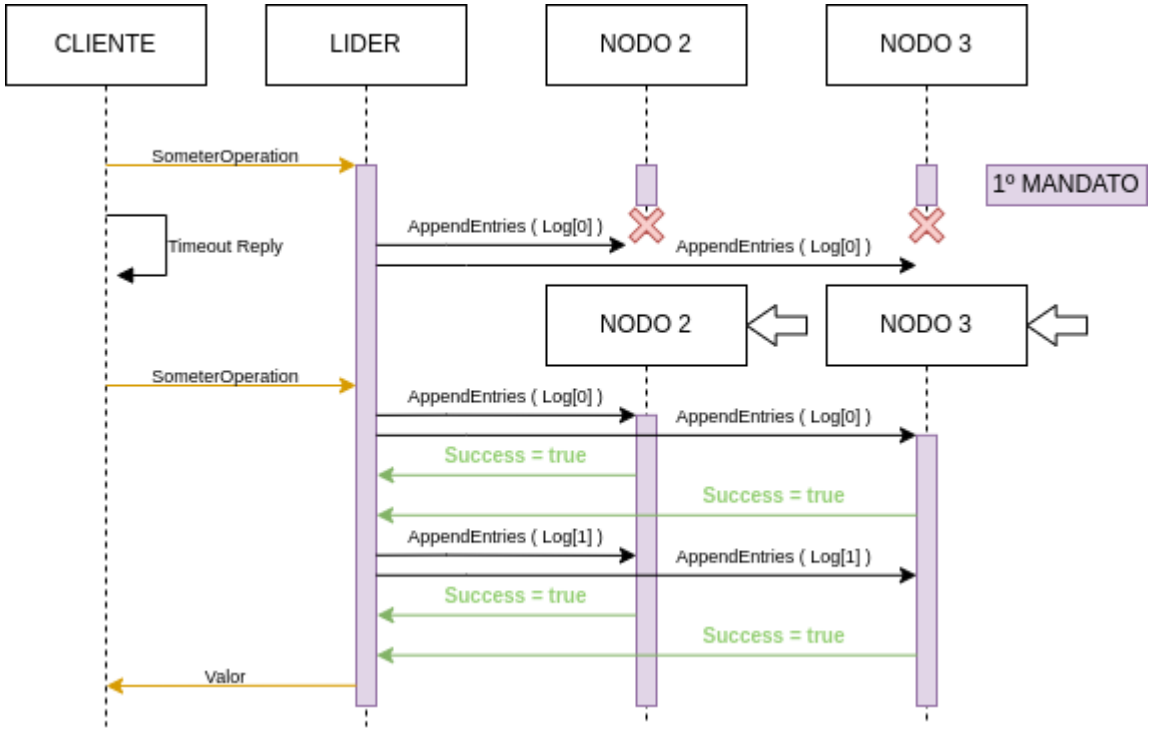

Figura 4: Diagrama de secuencias con dos entradas sometidas y dos nodo desconectado

5. Referencias

Documentación del algoritmo de RAFT:
Search of an Understandable Consensus Algorithm (Extended Version) - Diego Ongaro and John Ousterhout, Stanford University

Herramienta de Simulación del algoritmo de RAFT:
https://raft.github.io

ChatGPT OpenAI:
https://chat.openai.com