Práctica 4
Algoritmo de Raft Básico

Sistemas Distribuidos 2021-2022

Jorge Lisa 774248
David Zandundo 780500

Índice

Introducción

En esta práctica se va ha desarrollar una parte importante del algoritmo Raft. El algoritmo raft es un protocolo de consenso que trabaja eligiendo un líder central sobre el que se hacen las peticiones y coordina al resto de nodos (seguidores) para implementarlas.

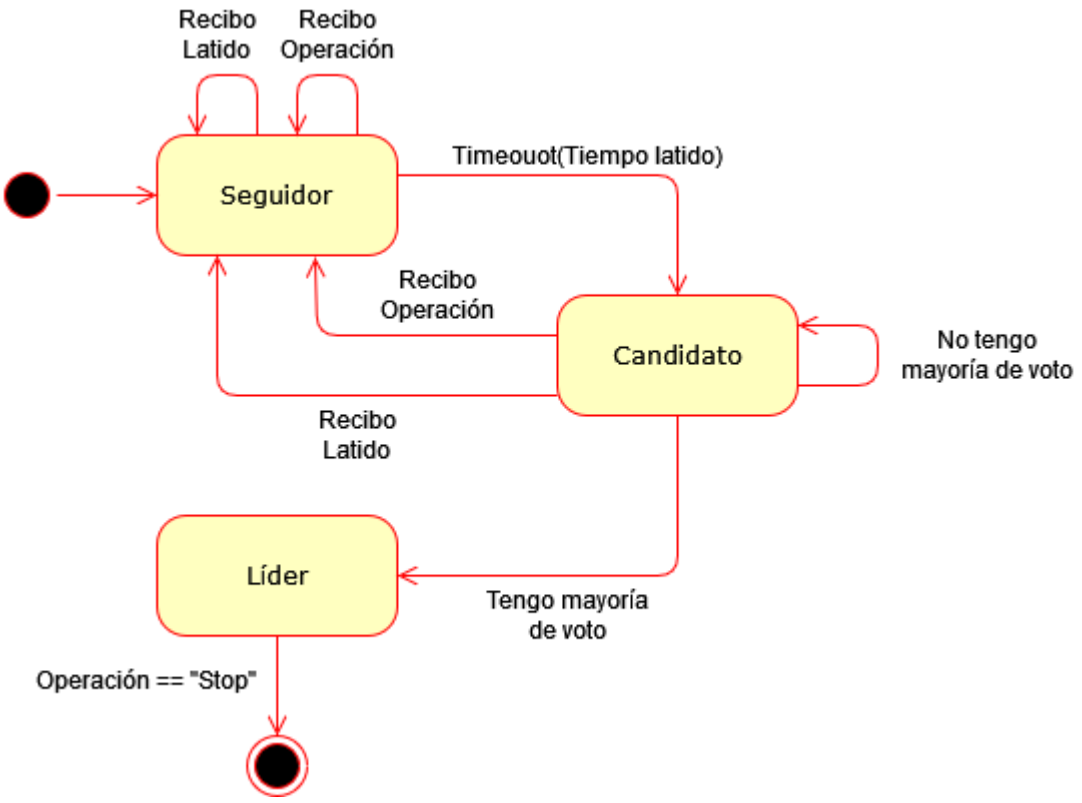
Los nodos participantes pueden estar en tres estados diferentes: Lider (Todos los cambios que se realicen en el cluster pasan por él primero), Candidato (Nodo que no ha encontrado líder y solicita su elección como tal) y Seguidor (Nodo cuya responsabilidad es actuar frente a las peticiones del nodo líder).

Objetivos

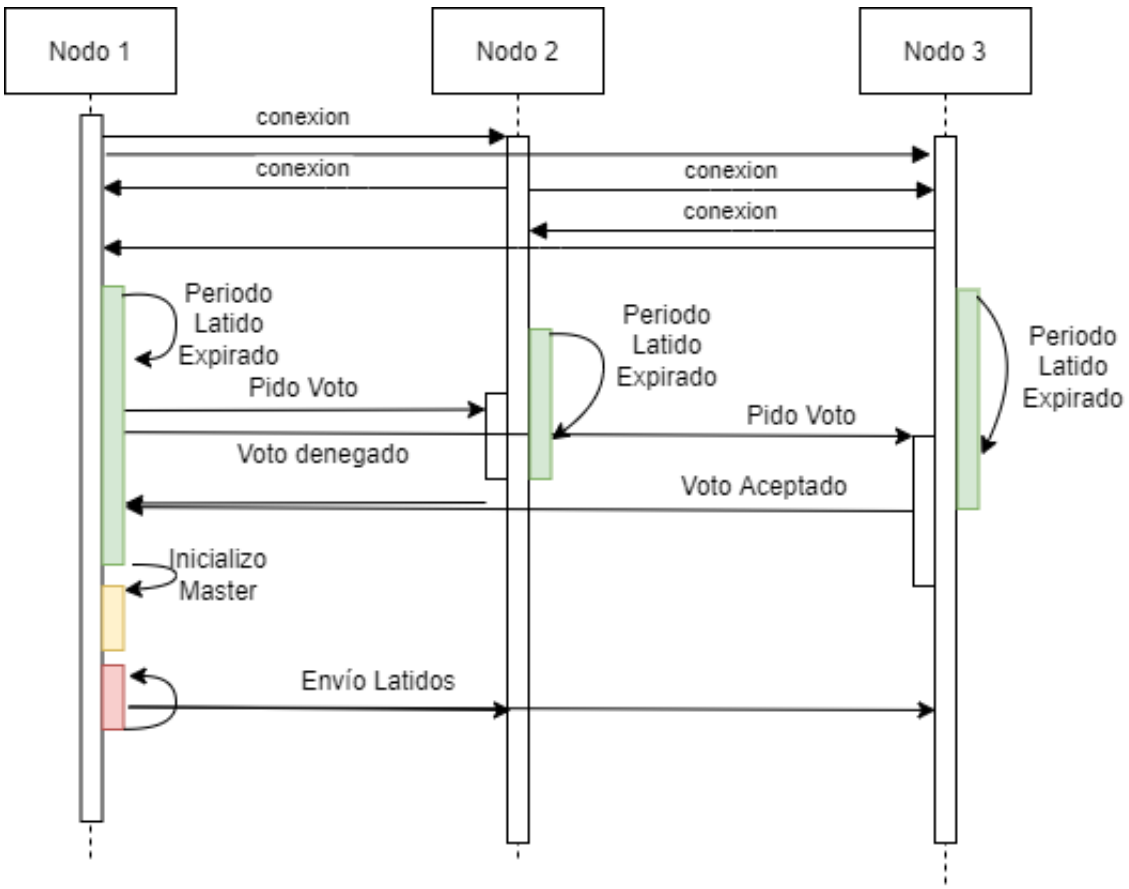
El objetivo de esta práctica es presentar una implementación del algoritmo de elección de líder de Raft que funcione en escenarios iniciales de fallos básicos, implementar el tratamiento de la llamada de RPC AppendEntry, suponiendo un funcionamiento sin fallos y comprobar que el sistema es capaz de comunicarse, elegir master y comprometer entradas de forma distribuida.

Metodología

Cada nodo puede estar en 3 posibles estados: seguidor, candidato o líder.
Al empezar el sistema, los nodos se inician en estado seguidor, a la espera de un latido de un posible master. Al arranque, no hay ninguno en ejecución, por lo tanto se expira el tiempo de latido y se pone en estado candidato.
En este estado, los nodos empiezan a pedir votos a todos los usuarios del sistema, independientemente del estado en el que se encuentren. Dependiendo de unos factores que posteriormente se explicarán, recibirán el voto o no. Si no reciben mayoría de voto empezaría otra candidatura.
Si reciben un latido, significa que alguien se ha convertido en máster y por lo tanto este nodo se convierte en seguidor de este máster. Por último, si recibe la mayoría de los votos, se convierte en master y lo avisa al resto de copias.



Una representación de la inicialización de este sistema ideal sería la siguiente:



Tras conectarse con todos los nodos se empieza el periodo de elección. Si ambos votos son denegados, se reiniciaría el periodo de candidatura (Los cubos de color verde). Si al menos un voto es aceptado, se inicia el periodo de inicialización del master (el de color amarillo) y este se pondría a enviar constantes latidos (apartado de color rojo).

Para el desarrollo de este algoritmo lo primero de todo ha sido tener claro las funciones y las diferentes representaciones de los estados del algoritmo. Para ello se ha estudiado la documentación dada y empezado por la creación de las estructuras de datos.

El tipo de datos más importante, y el que más se va a desarrollar y modificar es NodoRaft:

```
type NodoRaft struct {
	// Variables de control y depuración
	mux         sync.Mutex    // Mutex para proteger acceso a estado compartido
	nodos       []*rpc.Client // Conexiones RPC a todos los nodos (réplicas)
	logger      *log.Logger   // Logger para depuración
	canalLatido chan bool
	endChan     chan bool

	// Variables que deberían ser iguales entre todos los nodos (sin tener
	// en cuenta posibles errores)
	candidaturaAnterior int
	candidaturaActual   int
	masterActual        int

	// Varaibles de cada candidatura
	soyCandidato           bool
	heVotadoA              int
	votosCandidaturaActual int

	// Variables propias de cada Nodo
	yo                        int
	entradas                  []AplicaOperacion
	ultimaEntrada             int
	ultimaEntradaComprometida int
	periodoLatido             time.Duration
	periodoCandidatura        time.Duration

	// Variables únicas del Master actual (si el nodo actual no es master, se
	// deben ignorar sus valores)
	totalCompromisosUltima int
}
```

Este struct se encuentra en cada nodo y contiene información del estado actual de ese nodo, así como la información actual de todo el sistema.
Guardamos la información propia de cada nodo como su id, sus entradas, cual es la última entrada que tiene y cual es la última comprometida, cuanto dura su espera del latido y cuanto la de su candidatura.

Posteriormente, tenemos las variables de cada candidatura como si es candidato de la candidatura actual, cuántos votos tiene y a quíen a votado. Estas varaibles se reinician en cada candidatura.

También se guarda la información general del sistema, como en qué candidatura nos encontramos y de cual venimos y quién es el master actual.

Por último, si el nodo actual es master, también guarda cuantos nodos han comprometido su última entrada.

Para arrancar el sistema lo primero que debemos hacer es ejecutar la función NuevoNodo(), esta función lo que hace es inicializar todas las variables del nodo, así como calcular de forma aleatoria los tiempo de espera para recibir un latido así como el periodo para recibir los votos cuando si se presenta a candidato.

Posterior a la Inicialización de cada nodo, tenemos la parte principal del sistema, la recepción de latidos e inicio de candidaturas.

Escucha de Latidos

```
for {
	if nr.masterActual == nr.yo { // Yo soy el master
		nr.comunicarLatidos()
		time.Sleep(pulseDelay)
	} else {
		select {
		case <-nr.canalLatido: // El master ha respondido a tiempo
			break
		case <-time.After(nr.periodoLatido): // El master ha tardado mucho
			nr.prepararCandidatura()
			break
		}
	}
}
```

Este bucle se ejecuta infinitamente. Si el nodo actual es el máster, envía latidos a todos los seguidores, y espera un pequeño tiempo.

Si no es máster, se queda a la espera de un latido de corazón. Si este llega a tiempo, vuelve a esperar otro, y así indefinidamente. Si no llega a tiempo (se agota el **time.After**), significa que debemos empezar una candidatura, ya que el máster se ha caído.

```
for {
	select {
		case <-nr.canalLatido: // Alguien se ha convertido en master
			return
		case <-time.After(nr.periodoCandidatura):
			// Operaciones Votación
	}
}
```

Envío de votos

La votación está basada en esta estructura. Si tras no recibir un latido (de la función anterior), recibo uno (un master ha entrado en ejecución antes de que yo haya emepzado a pedir votos), vuelvo a la función anterior, sino empiezo la votación. Este bucle se ejecuta hasta que alguno de los nodos gana la candidatura.

Primero incializo las variables con mi voto y diciendo que soy candidato para estas elecciones.

```
args := ArgsPeticionVoto{
	nr.candidaturaActual, nr.yo, nr.ultimaEntrada, nr.candidaturaAnterior}
nr.soyCandidato = true
```

A continuación contacto con todos los nodos envíandoles el voto (posteriormente se observará cómo decide cada nodo si le da el voto o no)

```
for id := range nr.nodos {
	var respuesta RespuestaPeticionVoto
	ok := nr.enviarPeticionVoto(id, args, &respuesta)

	if ok { // El nodo no está caído
		if !respuesta.VotoGrantizado { // No me da el voto
			if nr.candidaturaActual < respuesta.Candidatura {
				nr.candidaturaActual = respuesta.Candidatura
			}
		} else { // Me da el voto
			nr.votosCandidaturaActual++
			if nr.votosCandidaturaActual >= constants.USERS/2+1 {
				nr.inicializarMaster()
				return
			}
		}
	}
}
```

Si el nodo no me da el voto, se observa que tenga correctamente actualizado el mandato. Si me da el voto, sumo 1 en el número de votos recibidos y compruebo si es mayor o igual a la mayoría (Número de nodos / 2 + 1). Si es así, inicializo el master y terminamos las votaciones.

Recepción de votos

Para comprobar si votamos a un nodo que nos lo ha pedido, primero comprobamos que yo no estoy también preparado para ser candidato o que ya he votado a alguien, ya que en este caso denegaríamos su acceso.

Si no soy candidato y todavía no ha habido master en este sistema, le aceptamos el voto. Si ya ha habido algun máster, tenemos que comprobar que el nodo que pide el voto tiene una longtud de entradas mayor o igual a la del nodo recpetor y que además tenga que el mandato anterior es mayor o igual al anterior del nodo receptor.

```
// Si soy candidato o ya he votado, no doy el voto
if nr.soyCandidato || nr.heVotadoA != -1 {
	*reply = RespuestaPeticionVoto{nr.candidaturaActual, acceso}
	return nil
}

// Aún no ha habido un master
if nr.candidaturaAnterior == -1 {
	nr.mux.Lock()
	nr.heVotadoA = args.Candidato
	nr.mux.Unlock()
	acceso = true

// El candidato tiene por lo menos con las entradas de este nodo
} else if args.UltimaCandidatura >= nr.candidaturaAnterior &&
	args.UltimaEntrada >= nr.ultimaEntrada {

	nr.mux.Lock()
	nr.heVotadoA = args.Candidato
	nr.mux.Unlock()
	acceso = true
}


*reply = RespuestaPeticionVoto{nr.candidaturaActual, acceso}
```

Por último, tenemos la introducción de entradas por parte de un master a un nodo seguidor.

```
if nr.yo != nr.masterActual {
	nr.canalLatido <- true // El master sigue vivo
	nr.ultimaEntrada++
	nr.entradas = append(nr.entradas,
				AplicaOperacion{nr.ultimaEntrada, operacion})
	*correct = true
} else {
	*correct = false
}
```

Cuando un seguidor recibe una entrada del master, sabemos que sigue vivo, por lo tanto metemos en el canal un valor para que sea leído en la primera función mostrada, habiendo comprobado antes que un seguidor no nos está intentando introducir una entrada.
Como no sabemos si esta entrada está comprometida o no, solo modificamos el índice que marca cuantas entradas tenemos.

Compromiso de entradas

Para comprometer entradas, lo hacemos con los latidos. Tras envíar los latidos, el maestro recibe de los seguidores cuál es la última entrada de ese mismo nodo. Si la operación con ese índice concuerda con la del máster, aumentamos en 1 el número de entradas comprometidas y comprobamos si son la mayoría de ellas.

```
args := ArgsLatido{nr.yo, nr.candidaturaActual, nr.ultimaEntradaComprometida}

for id := range nr.nodos {
	var ultimaEntrada AplicaOperacion
	nr.nodos[id].Call("NodoRaft.RecibirLatido", args, &ultimaEntrada)
	if nr.ultimaEntrada != -1 { // Hay alguna entrada en mi nodo
		if ultimaEntrada.Indice != -1 { // El nodo con quien contacto tiene
alguna entrada
			if nr.entradas[ultimaEntrada.Indice].Operacion ==
ultimaEntrada.Operacion {
				nr.totalCompromisosUltima++
				if nr.totalCompromisosUltima > (constants.USERS)/2 {
					nr.totalCompromisosUltima = 0
					nr.ultimaEntradaComprometida++
				}
			}
		}
	}
}
```

Hay que tener en cuenta que hacemos pocas comprobaciones ya que estamos en un momento en el que no puede haber fallos de consenso, solo caídas del máster.