Práctica 5
Algoritmo de Raft Avanzado

Sistemas Distribuidos 2021-2022

Jorge Lisa 774248
David Zandundo 780500

Introducción

En esta práctica se va ha terminar el algoritmo de consenso de Raft empezado en la práctica anterior. La funcionalidad que se va a incluir es la de tolerancia a fallos. Esta versión deberá ser capaz de que se comprometan entradas y que exista la posibilidad de que un nodo no introduzca una entrada que el máster acaba de recibir.

Objetivos

Cambiar el funcionamiento del código para poder adaptarse a un sistema con fallos y asegurarse que funcione correctamente.
Las pruebas que debe pasar son las siguientes:

1) Asegurarse de que con 1 máster y 1 seguidor (en un sistema de 3 nodos) se pueden comprometer entradas.
2) No permitir que se comprometan entradas en un sistema Raft con sólo 1 master y ningún seguidor.
3) Introducir 5 entradas y observar el puntero de compromiso de las mismas.

Metodología

Como se parte de la práctica anterior, todo lo descrito en ella, como la descripción básica del nodo, se obviará.

Cambios a Nodo Raft

Primero, se han hecho cambios a la información del nodo, principalmente al nodo que es master. Las variables introducidas están relacionadas a la información que tiene el master respecto a las entradas de cada seguidor. Esta información se obtine cuando se envía un latido y el nodo que recibe dicho latido, le responde con el número de entradas que tiene.

Para tratar el problema de que las entradas de los seguidores pueden no estar en el mismo momento que el master las recibe, se ha creado un canal que guarda cual es la siguiente entrada "**`Entrada`**" que un seguidor "**`Quien`**", debe introducir, usando un nuevo tipo struct llamado **`entradaIntroducir`**.

```
type entradaIntroducir struct {
      Entrada      string
      Quien        int
}

canalAplicar              chan entradaIntroducir
indiceUltimaEntradaNodo   []int
comprometiendo            []bool
```

Nuevo Someter Operación

La función de la versión anterior asume que todos los seguidores van a recibir la entrada y guardarla en su registro, por lo tanto podíamos enviar la entrada desde la misma función sin tener en cuenta otrs posibilidades.
En esta versión, puede que un seguidor tenga que guardar todavía la entrada anterior o muchas más.
Para asegurarnos de que las entradas llegan en orden y sin errores, no la podemos envíar instantáneamente, por lo que sólo se va a guardar en el registro del máster y otra función se encargará de realizar está acción, ya que se deben hacer varias comprobaciones antes.

Cambio al envío de latidos

El envío de las entradas va a ser realizado por los latidos. Cada vez que se envíe un latido, se recibirá el número de entradas que tiene cada seguidor y se guardará en la posición del nodo del vector **`indiceUltimaEntradaNodo`**.

```
err := nr.nodos[id].Call("NodoRaft.RecibirLatido", args, &respuesta)
if err == nil {
	nr.indiceUltimaEntradaNodo[id] = respuesta
}
```

Si el número de entradas recibido no es igual a las que tiene el máster, este enviará por el canal la siguiente entrada que tiene que guardar dicho seguidor, usando el tipo de datos nuevo.

```
indice := nr.indiceUltimaEntradaNodo[id] + 1
if !nr.comprometiendo[id] {
	entrada := entradaIntroducir{nr.entradas[indice], id}
	nr.canalAplicar <- entrada
}
```

Antes de enviar la entrada al canal y después de recibir la respuesta del seguidor, se comprueba el compromiso de la siguiente entrada a comprometer. Esto se hace llamando a la función **`comprobarCompromiso()`**, la cual lee los valores de todas las posiciones del vector anteriormente mostrado.

También, no debemos enviar 2 veces la misma entrada, ya que podría causar problemas y si además se hace continuamente, supondría un exceso de carga del sistema.

Comprobación del Compromiso

Esta función es totalmente nueva, ya que ahora no sabemos si las entradas se van a comprometer nada más ser enviadas.

Para comprometer una entrada, debe haber por lo menos un total de nºUsuarios/2 seguidores con esa entrada en su registro más el máster. Para conseguir esto, se recorre el vector **`indiceUltimaEntradaNodo`** y se guarda el número de veces que el número de entradas de cada seguidor supera o iguala al número de entradas comprometidas más uno. Si ese valor es mayor al mínimo para compromoter una entrada, se asume que la siguiente entrada se ha comprometido correctamente.

El número de entradas comprometidas, será envíada por el máster a los seguidores a través de un latido (cuando sea el momento) para que si éste cayese, el siguiente que salga en el próximo mandato pueda continuar a comprometer desde ahí.

```
nextEntry := nr.ultimaEntradaComprometida + 1

graterOrEqual := 1

for _, val := range nr.indiceUltimaEntradaNodo {
	if val >= nextEntry {
		graterOrEqual++
		if graterOrEqual > constants.USERS/2 {

			nr.ultimaEntradaComprometida++
			return
		}
	}
}
```

Envío de entradas

Tal y como se ha mostrado en las anteriores funciones, ahora se usa un canal capaz de almacenar que entrada debe el seguidor “Quien” almacenar en su siguiente posición.
Esto se hace justo después de comprobar el compromiso de la última entrada en la función que envía latidos.
Esta función debe ser ejecutada como una gorutina, ya que está recibiendo del canal indefinidamente.

```
for {
	entradaIntroducir := <-nr.canalAplicar
	var correcto bool

	nr.comprometiendo[entradaIntroducir.Quien] = true

	operacion := AplicaOperacion{
		nr.indiceUltimaEntradaNodo[entradaIntroducir.Quien] + 1,
		entradaIntroducir.Entrada}

	rpctimeout.CallTimeout(nr.nodos[entradaIntroducir.Quien],
"NodoRaft.AppendEntries", operacion, &correcto, time.Second)

	nr.comprometiendo[entradaIntroducir.Quien] = false
}
```

Primero se dice que se está intentando comprometer una entrada en este nodo, se crea la opreación a introducir y se envía al nodo “**Quien**”.

Esta función no comprueba si ha habido errores o no, solo intenta introducir una entrada en la máquina de estados de un seguidor. El índice que guarda cuál entrada es la última de cada nodo se modifica al recibir un latido.

Esta función sólo será ejecutada por el máster tras ganar una candidatura, de esta manera ahorramos recursos en los nodos seguidores.