Práctica 5

Memoria

Juan Antonio Rodríguez Gracia (NIP: 805001)
Miguel Beltran Pardos (NIP: 800616)

Sistemas distribuidos
Grado en Ingeniería Informática

Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza
Curso 2021/2022

1. Introducción

Hemos procedido a diseñar e implementar el algoritmo de consenso raft. El cual, por motivos de orden y limpieza, se ha separado en varios modulos dentro un msimo paquete, los cuales se citan a continuación.

1. general.go
2. raft.go
3. votation.go
4. appendEntries.go
5. stateMachine.go

2. Comportamiento del sistema

2.1. Comunicación

En cuanto a la comunicación usada entre los distintos nodos, se ha utilizado las llamadas RPC, concretamente se han usado las funciones proporcionadas en el fichero rpctimeout.go.

```
func (hp HostPort) CallTimeout(serviceMethod string, args interface{},
                    reply interface{}, timeout time.Duration) error {

    client, err := rpc.Dial("tcp", string(hp))
    if err != nil {
        // fmt.printf("Error dialing endpoint: %v ", err)
        return err  // Devuelve error de conexion TCP
    }

    defer client.Close() // AL FINAL, cerrar la conexion remota  tcp

    done := client.Go(serviceMethod, args, reply, make(chan *rpc.Call, 1)).Done

    select {
        case call := <-done:
            return call.Error
        case <-time.After(timeout):
            return fmt.Errorf(
                    "Timeout in CallTimeout with method: %s, args: %v\n",
                    serviceMethod, args)
    }
}
```

2.2. Timeouts

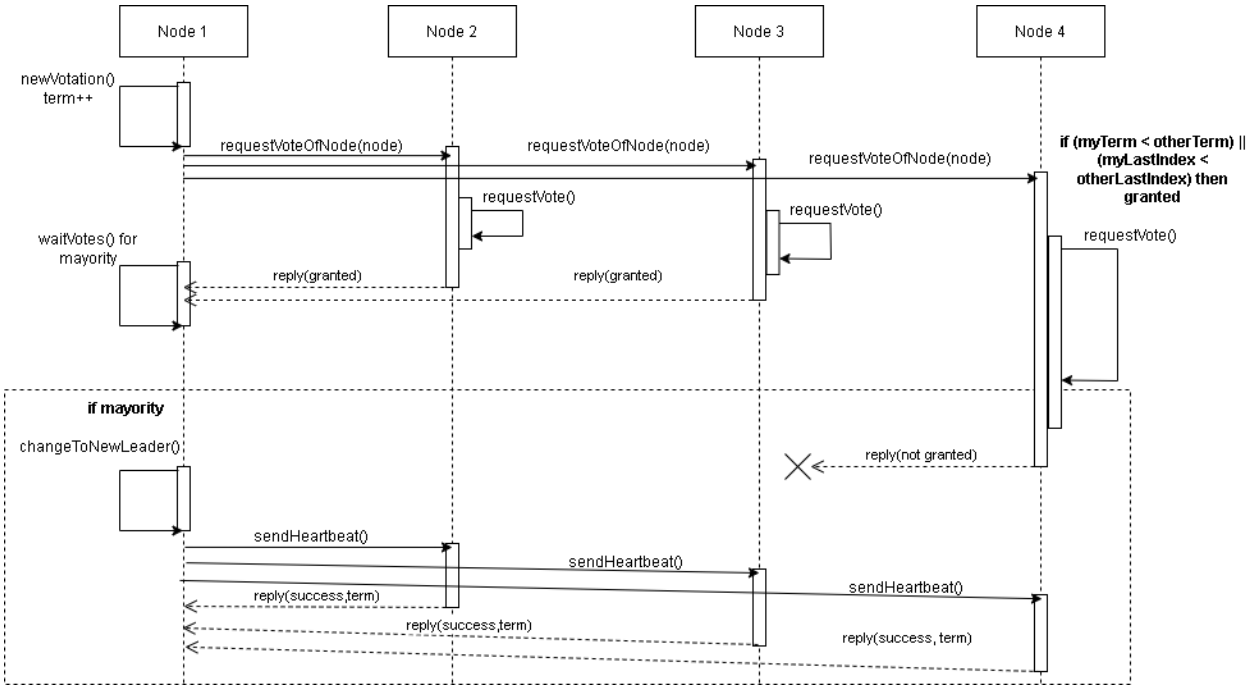
En cuanto a timeouts, se han considerado los siguientes a lo largo de la práctica, los cuales se comentan a continuación.

1. El primer timer a tener en cuenta, estará presente en los nodos que son seguidores de un lider, este se activará en un rango de entre 5000 a 8500 milisegundos si no ha recibido correctamente el latido del lider. Lo que llevaría al nodo a estado candidato y comenzaría una nueva votación. Hay que tener en cuenta que estos tiempos son elevados, se han propuesto asi debido a posibles errores que pudieran surgir debido a problemas de red en el entorno de trabajo.
2. Otro timer estara ubicado en el lider, el cual se activará cada 500 milisegundos para enviar el correspondiente latido a los nodos, si el lider no tiene ninguna nueva operación que enviar.

3. Por ultimo, en todos los envios RPC, se ha predefinido un tiempo de espera maxima de respuesta de 300 milisegundos

3. Diagramas

3.1. Votación

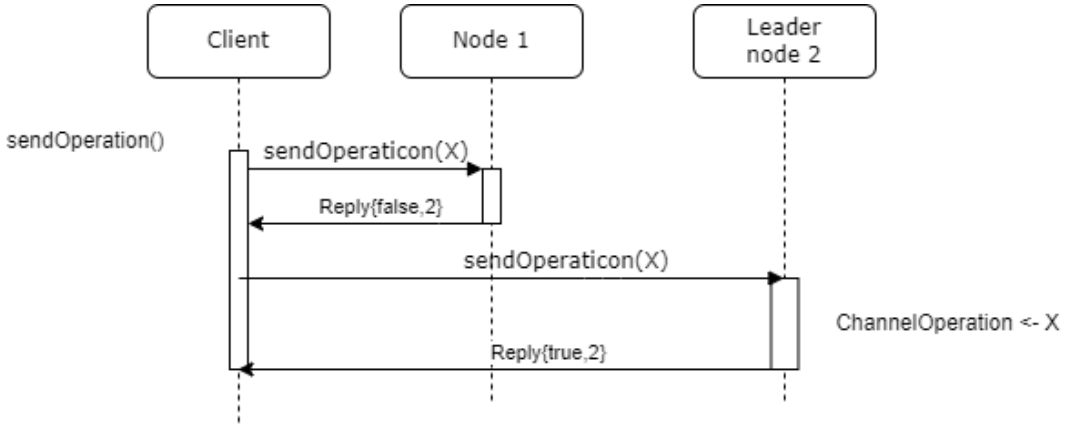


Como podemos observar en el diagrama cuando un nodo no recibe un hearbeat antes de que salte el timeout de espera. Este inicia una nueva votación ya que entiende que el líder ha caído. En ese momento cambia el estado a candidato y pide a el resto de nodos su voto enviándoles su termino actual más uno.

Una vez enviados el nodo tiene que esperar a la mayoría de los votos para poder ser el líder y empezar a enviar el heartbeats.

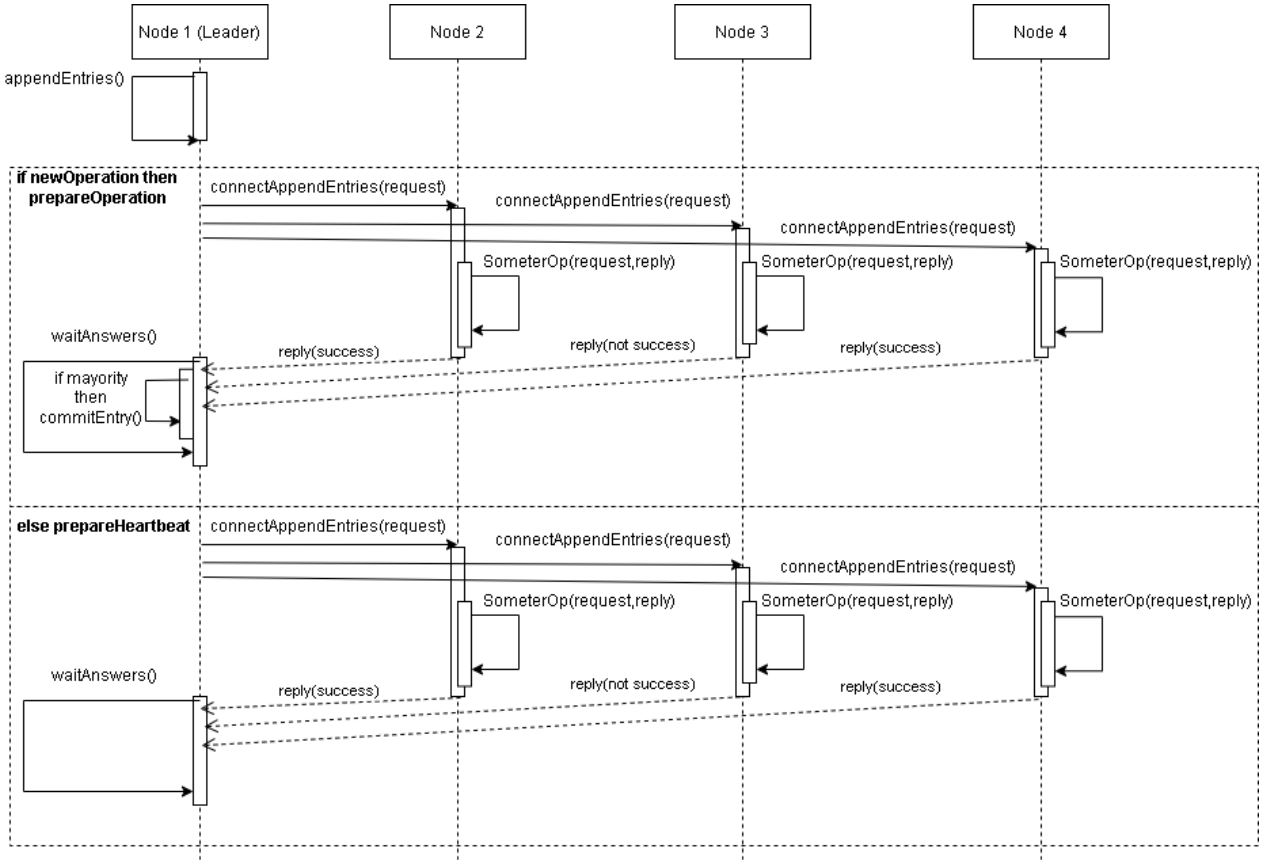
Si en medio de este proceso no se llega a la mayoría se reintentaría más adelante a no ser que recibiera un heartbeat de otro nodo el cual es el líder. Si durante la votación otro nodo envía un hearbeat el nodo actual comprobara los términos para ver si se trata de un nodo actual o viejo y en base al termino se colocara como líder o seguirá con la votación.

3.2. Operation client



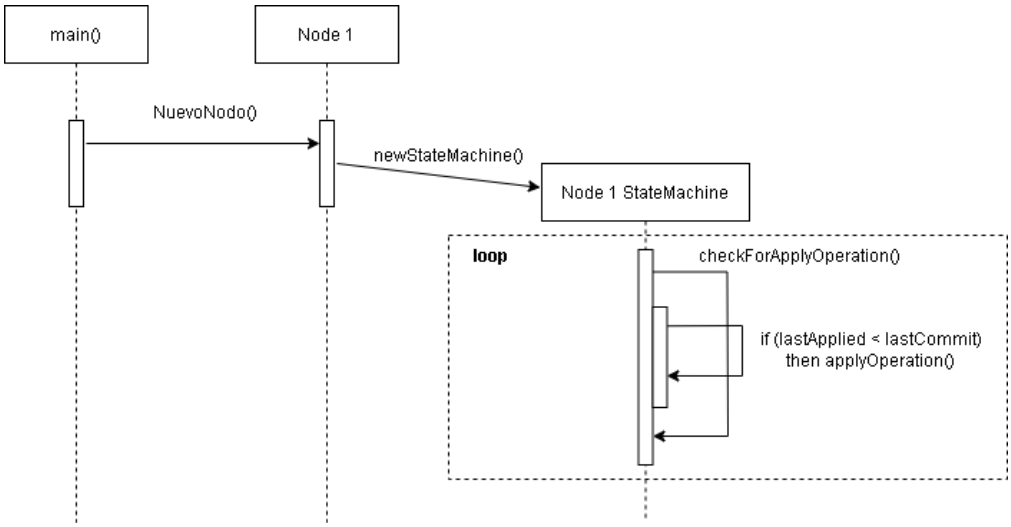
El cliente intenta enviar la petición a un nodo indicado por defecto si este no es el líder, el propio nodo le indicara al cliente cual es el nodo líder para que pueda reintentar la conexión.

3.3. Operation appendEntries



El líder tendrá dos opciones, enviar un heartbeat/latido o por el contrario enviar un appendEntries por cada una de las entradas correspondientes a las operaciones recibidas desde el cliente y almacenadas por el líder. Se encargara de enviar a todos nodos dicha información, donde si es un heartbeat simplemente enviara y recibirá respuesta (reply), con el paso de hearbeat comprobaremos a su vez que los nodos disponen de los indices correctos. Por el caso contrario, si envía una entrada, debera comprobar el numero de respuestas correctas por parte de los nodos en estado "seguidor", si obtiene que la mayoría de nodos le han respondido y la anterior entrada a la enviada está comprometida, pasará a comprometer la entrada enviada a dichos nodos.

3.4. State Machine



Una vez creado un nuevo nodo, se creará con el una máquina de estados, que mientras el nodo esté activo, comprobará si es necesario añadir una nueva entrada comprometida a la database.

4. Implementación

4.1. Votación

A la hora de realizar una nueva votación debido a que un nodo ha dejado de recibir el latido del lider, se realizarán los siguientes pasos.

1. Una vez que ha saltado el timer ya que no se ha recibido un latido en el tiempo correspondiente (vease en el apartado de timeouts), el nodo pasara a estado CANDIDATE, aumentará su término y se autovotará.

```
func (nrf *NodoRaft) newVotation() {
    // canal de respuesta de votos
    votes := make(chan definitions.ReplyRequestVote, len(nrf.Nodos))
    nrf.currentTerm = nrf.currentTerm + 1 // aumento mandato
    nrf.votedFor = nrf.Yo // indico que me autovoto
    for i, node := range nrf.Nodos { //Iniciamos unas conexion rpc para cada nodo
        if i != nrf.Yo {
            nrf.Logger.Println("send requestVote for:", node)
            go nrf.requestVoteOfNode(node, votes)
        }
    }
    nrf.waitForVotes(votes)
}
```

2. Tras esto, comenzará a enviar de forma paralela al resto de nodos, una petición para que comprobar si puede volverse lider o no. Esta petición contendrá el término e índice de la ultima entrada guardada. Una vez realizada la petición se esperará a recibirla respuesta de los nodos.

3. El nodo que recibe la petición de voto, tednrá que comprobar que es un candidato adecuado, esto lo hara comprobando el término e índice recibido en la request.

   a) Si la request presenta un término mayor que el nodo, el nodo actulizará su término actual y si no es FOLOWER pasará a ese estado.

   b) Si el nodo no ha votado aún y tiene un término menor o igual pero con menor índice en su ultima entrada, le concederá el voto al nodo CANDIDATE. En el resto de casos no se lo concederá.

4. Todas las replies de los nodos se enviaran por un canal donde el nodo CANDIDATE comenzará a procesarlas.

   a) Si nos han concedido el voto, aumentaremos el contador de votos (teniendo en cuenta que ya nos autovotamos previamente), si no nos lo conceden, comprobaremos si nuestro término es igual o mayor al recibido con la reply, si tenemos un término menor, pasaremos a estado FOLLOWER.

   b) Posteriormente comprobaremos que seguimos en estado CANDIDATE y no en FOLLOWER, si hemos cambiado a FOLLOWER cancelaremos la votación ya que no somos un candidato adecuado o ya se ha establecido otro lider.

   c) Si se recibe mayoria, pasaremos a estado LEADER y comenzaremos a realizar los cambios necesarios en el sistema para serlo.

   d) Una vez hecho esto, enviaremos latido al resto de nodos y terminara la votación.

   e) Si no se consigue mayoria, pasaremos a estado FOLLOWER y terminaremos la votación.

```
func (nrf *NodoRaft) waitForVotes(votes chan definitions.ReplyRequestVote) {
    numVotes := 1         //1 por que cuento mi propio voto
    loopWaitForVotes:
        for i := 0; i < len(nrf.Nodos) - 1; i++ { // espero la respuesta de los nodos
            select {
            case vote := <-votes:
                if vote.Granted { //compruebo si me han concedido el voto
                    nrf.Logger.Println("Voto concedido : ", vote)
                    numVotes++
                } else {
                    nrf.Logger.Println("Voto no concedido : ", vote)
                    nrf.checkCurrentTerm(vote.Term)
                }
        }
        if nrf.state == FOLLOWER {
            nrf.Logger.Println("Otro leader ha enviado heartbeat ")
            nrf.votedFor = -1
            return
        }
        if numVotes > len(nrf.Nodos)/2 { //Si obtengo mayoria soy el lider
            nrf.Logger.Println("Mayoria soy lider")
            nrf.state = LEADER
            nrf.changeConfigOfNewLeader()
            nrf.appendEntries()
            break loopWaitForVotes
        }
    }

    if numVotes <= len(nrf.Nodos)/2 { //si no tengo mayoria paso a seguidor
        fmt.Println("no mayoria")
        nrf.Logger.Println("No tengo mayoria vuelvo follower")
        nrf.state = FOLLOWER
        nrf.votedFor = -1
    }
}
```

4.2. Append Entries

Se han usado los siguientes tipos de datos para las requests y replies.

```
type RequestAP struct {
    Term int
    LeaderId int
```

```
    PrevLogTerm int
    PrevLogIndex int
    LeaderCommit int
    EntryElement Entry
}

type ReplyAP struct {
    Term int
    Success bool
    IDNode int
    MatchIndex bool
}

//El canal para enviar las reply será de este tipo de dato, pasa saber a que request se refiere la re
type PairRequestReplay struct {
    Request RequestAP
    Reply ReplyAP
}
```

En cuanto a la implementación, se ha realizado la siguiente lógica a la hora de realizar la función appendEntries.

1. Por cada nodo, el lider creará una request que modificara en base a si el nodo que la recibe tiene las ultimas entradas o no. Esto se comprobara mediante el slice nextIndex[] encargado de almacenar la siguiente entrada que debe de llegarle a cada nodo seguidor.

2. Si dicho nodo esta al día, el lider procederá a enviar un latido, que consiste en enviar un tipo de dato Entry vacío. Si el nodo requiere una nueva entrada, se enviará dicha entrada. Y se enviará mediante el uso de RPC la request a los nodos de forma paralela.

```
for i, v := range nrf.Nodos {
    if i != nrf.Yo {
        nIndexOfNode := nrf.nextIndex[i] //listado de indices de insercion de los nodos

        // Si tenemos entries y el indice no es un caso corner sacamos termino y indice
            if len(nrf.logEntries) != 0 && nIndexOfNode > 0 {
            request.PrevLogTerm = nrf.logEntries[nIndexOfNode - 1].Term
            request.PrevLogIndex = nIndexOfNode - 1
            } else {
            //Si no indicamos que case corner con 0 y -1
            request.PrevLogTerm = 0
            request.PrevLogIndex = nIndexOfNode - 1
        }

        //Si hay Entry disponible para ese nodo le envio la entry
        //Si el indice es diferente a los entries actuales enviamos el entry
        if nIndexOfNode < (len(nrf.logEntries)) {
            request.EntryElement = nrf.logEntries[nIndexOfNode]
        } else{
        //si no heartbeat
        request.EntryElement = definitions.Entry{}
        }

        go nrf.connectRPCAppendEntries(chReply, v, request)
    }
}

nrf.Mux.Unlock()
```

3. En cada nodo, se ejecutara mediante RPC el método llamado SometerOperacion(), donde se enviará una request y se recibirá una reply

   a) Comienza creando una reply por defecto, donde se pasara nuestro término y se podrá que la operación no se ha completado satisfactoriamente. Posteriormente ejecutaremos en exclusión mutua el resto de la función.

   b) Primero se comprueba si el término de la request es mas antiguo que el del nodo, si lo es, termina la funcion devolviendo que no se ha completado correctamente la operación (success = false).

   c) Si el termino es igual o mayor, actualizaremos la id de nuestro lider a la recibida en la request, avisamos de que hemos recibido una petición de un nodo con termino igual o superior a nosotros y actualizamos nuestro índice de última comprometida.

   d) Si recibimos en la request una entrada no vacia y con valor - 1 de prevLogIndex, estaremos en el caso de recibir por primera vez una entrada. Por lo que añadiremos la entrada correctamente, diremos en la reply que se ha añadido correctamente y que no ha habido problemas de índices. Terminando la operación.

```
if (op.PrevLogIndex == -1 && op.EntryElement != (definitions.Entry{}))
    && len(nrf.logEntries) == 0 { // caso inicial
    reply.MatchIndex = true
    nrf.logEntries = append(nrf.logEntries, op.EntryElement)
    reply.Success = true
    return nil
}
```

   e) En caso contrario, tendremos los siguientes casos.

      1) Si recibo una entrada vacia y el valor recibido de prevLogIndex es -1, estaremos en el caso de recibir el primer latido, devolveremos que no hay problemas de indices y termina la operación.
      2) Si recibimos en la request un valor de 0 en prevLogIndex y no tenemos ninguna entrada, avisaremos de que los indices no están correctos y termina la operación.
      3) Si no tenemos menos datos de los que deberiamos, avisamos de que hay problemas con los indices y termina la operación.
      4) Si el prevlogIndex se puede indexar comparamos términos, si la anterior entrada que deberia de estar añadida encontramos que no coincide el término, eliminaremos desde esta hasta el final de la lista de entradas. En caso de ser iguales, añadiremos la nueva entrada a la lista y devolveremos que la operación ha terminado correctamente.

```
if len(nrf.logEntries) - 1 >= op.PrevLogIndex + 1  {
    reply.MatchIndex = true
    return nil
}

termOfPrevLogIndex := nrf.logEntries[op.PrevLogIndex].Term
//si el termino es diferente eliminamos hasta esa entrada
if termOfPrevLogIndex != op.PrevLogTerm {
    nrf.logEntries = nrf.logEntries[:op.PrevLogIndex]
    reply.MatchIndex = false
    return nil
} else{ //Si son iguales añadimos el nuevo registro
    if op.EntryElement != (definitions.Entry{}) {
            nrf.logEntries = append(nrf.logEntries, op.EntryElement)
        reply.Success = true
        reply.MatchIndex = true

        return nil
    } else { //si es hearbeat indicamos que el indice esta bien
        reply.MatchIndex = true
        return nil
```

```
            }
        }
```

4. Una vez terminada la llamada RPC, se recibirá una reply por cada nodo. Dicha reply se enviará a través de un canal, donde otra función de forma concurrente estará recibiendolas y procesandolas.

5. Por cada reply recibida, el nodo lider comprobará si sigue teniendo un término mayor o igual al recibido en la reply, si no es asi, pasará a estado FOLLOWER y termina el appendEntries().

6. Tras realizar esta comprobación, se revisa si se ha conseguido enviar y añadir correctamente la entrada al nodo seguidor, si se ha conseguido, aunmentaremos el contador de la siguiente entrada a enviar de dicho nodo y aumentamos los votos para comprometer, si no se ha conseguido, comenzaremos a reducir el indice de siguiente entrada a enviar.

```
nrf.Mux.Lock() //actualizamos los valores con mutex

//tenemos que estar en matchIndex y success
if pairRequestReplay.Reply.Success && pairRequestReplay.Reply.MatchIndex
    && (pairRequestReplay.Request.EntryElement != (definitions.Entry{})) {

    //si hemos tenido, aumentamos contador de mayoria
    nrf.nextIndex[pairRequestReplay.Reply.IDNode]++

    //aumentamos el voto en su indice previo + 1
    nrf.entriesVoted[pairRequestReplay.Request.PrevLogIndex + 1]++

} else if !pairRequestReplay.Reply.Success && !pairRequestReplay.Reply.MatchIndex {
    //si no tenemos success y no hay matchindex disminuimos el indice hasta encontrar match
    nrf.nextIndex[pairRequestReplay.Reply.IDNode]--
}

nrf.Mux.Unlock()
```

7. De forma paralela a la anterior comprobación, tendremos un proceso que se encargara de comporobar si se reciben mayoria de votos para comprometer la entrada enviada a los nodos, una vez recibida la mayoria, mediante una serie de canales, comprobará si la anterior entrada ha sido comprometida para poder comprometer la entrada actual. Si no esta comprometida la anterior, esperara en un canal hasta que dicha entrada se comprometa.

```
func (nrf *NodoRaft) commitOperation(lastIndex int) {
    if lastIndex != -1 { //Comrpuebo si hay que commitear
        if lastIndex > nrf.commitIndex {
            votos := 0

            //espera activa a tener mayoria
            for votos <= len(nrf.Nodos)/2 {
                nrf.Mux.Lock()
                votos = nrf.entriesVoted[lastIndex]
                nrf.Mux.Unlock()
            }

            //espero a la anterior
            <-nrf.previousCommit[lastIndex]

            nrf.Mux.Lock()
            //actualizo el indice y aviso a la siguiente a comprometer
            nrf.commitIndex = lastIndex
            nrf.previousCommit[lastIndex + 1] <- true
```

```
                nrf.Mux.Unlock()
            }
        }
    }
```

4.3. State Machine

La maquina de estados o State Machine constara de un tipo de dato mapa clave/valor de strings, donde almacenaremos las entradas comprometidas.

```
type stateMachine struct {
    database map[string]string
}
```

La cual se encargará de comprobar si se han comprometido nuevas entradas para ir almacenandolas, esta operación la realizará cada 500 milisegundos.

```
func (nrf *NodoRaft) checkForApplyOperation() {
    for  {
        time.Sleep(500 * time.Millisecond)
        if nrf.commitIndex > nrf.lastApplied {
            nrf.lastApplied++
            nrf.stateMachine.applyOperation(nrf.logEntries[nrf.lastApplied].Operation)
        }
    }
}
```

Se comprobara si la operación a realizar es de lectura o de escritura y dependiendo de esto se mostrará o se almaenará la nueva entrada.

```
func (sm *stateMachine) applyOperation(operation definitions.TipoOperacion) {
    if operation.Operacion == "leer" {
        fmt.Println("Leo")
        fmt.Println(sm.database[operation.Clave])
    } else if operation.Operacion == "escribir"{
        fmt.Println("escribo")
        sm.database[operation.Clave] = operation.Valor
    } else {
        fmt.Errorf("Operacion no definida en la maquina de estados")
    }
}
```

5. Validación y tests

Se han realizado e implementado los siguientes tests para comprobar el correcto funcionamiento de la arquitectura.

5.1. Test 1

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

```
// Se consigue acuerdo a pesar de desconexiones de seguidor -- 3 NODOS RAFT
func(cfg *configDespliegue) AcuerdoApesarDeSeguidor(t *testing.T) {

    fmt.Println(t.Name(), ".....................")
```

```
cfg.startDistributedProcesses()

fmt.Printf("Lider inicial\n")
time.Sleep(time.Second *8)
cfg.pruebaUnLider(3)

// Comprometer una entrada
cfg.enviarOperacionCLiente()
time.Sleep(time.Second * 4)
cfg.stopNumberOfDistributedProcesses(1)
time.Sleep(time.Second * 6)
//  Obtener un lider y, a continuación desconectar una de los nodos Raft

_, e1Term,_,_ := cfg.obtenerEstadoRemotoWithNodesOff(0)
_, e2Term,_,_ := cfg.obtenerEstadoRemotoWithNodesOff(1)
_, e3Term,_,_ := cfg.obtenerEstadoRemotoWithNodesOff(2)
fmt.Println(cfg.obtenerEstadoRemotoWithNodesOff(0))
fmt.Println(cfg.obtenerEstadoRemotoWithNodesOff(1))
fmt.Println(cfg.obtenerEstadoRemotoWithNodesOff(2))

count := 0
if e1Term != -1 {
    count++
}
if e2Term != -1 {
    count++
    }
if e3Term != -1 {
    count++
}
if count > 2 {
    t.Errorf("Hay mas de 2 nodos arrancados")
}

// Comprobar varios acuerdos con una réplica desconectada
cfg.enviarOperacionCLiente()
fmt.Println(cfg.obtenerEntriesNodoWithNodeOut(0))
fmt.Println(cfg.obtenerEntriesNodoWithNodeOut(1))
fmt.Println(cfg.obtenerEntriesNodoWithNodeOut(2))
cfg.enviarOperacionCLiente()
fmt.Println(cfg.obtenerEntriesNodoWithNodeOut(0))
fmt.Println(cfg.obtenerEntriesNodoWithNodeOut(1))
fmt.Println(cfg.obtenerEntriesNodoWithNodeOut(2))
cfg.enviarOperacionCLiente()
fmt.Println(cfg.obtenerEntriesNodoWithNodeOut(0))
fmt.Println(cfg.obtenerEntriesNodoWithNodeOut(1))
fmt.Println(cfg.obtenerEntriesNodoWithNodeOut(2))

// reconectar nodo Raft previamente desconectado y comprobar varios acuerdos
cfg.startDistributedProcesses()//Lanzo todos los nodos de nuevo
time.Sleep(time.Second * 10)

fmt.Println(cfg.obtenerEntriesNodo(0))
fmt.Println(cfg.obtenerEntriesNodo(1))
fmt.Println(cfg.obtenerEntriesNodo(2))
```

```
    entrysNodo1, commitI1 := cfg.obtenerEntriesNodo(0)
    entrysNodo2, commitI2 := cfg.obtenerEntriesNodo(1)
    entrysNodo3, commitI3 := cfg.obtenerEntriesNodo(2)

    if commitI1 != commitI2 || commitI1 != commitI3 || commitI2 != commitI3 {
        t.Errorf("Indice de comiteadas diferente")
    }
    if len(entrysNodo1) != len(entrysNodo2) || len(entrysNodo1) != len(entrysNodo3)
        || len(entrysNodo2) != len(entrysNodo3){
        t.Errorf("Replycacion de nodos diferente")
    }else {
        for i,element := range entrysNodo1 {
            if element != entrysNodo2[i] || element != entrysNodo3[i]
                || entrysNodo2[i] != entrysNodo3[i]  {
                t.Errorf("Error Logs diferentes ")
            }
        }
    }

    cfg.stopDistributedProcesses()
}
```

5.2. Test 2

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

```
func(cfg *configDespliegue) SinAcuerdoPorFallos(t *testing.T) {
    //t.Skip("SKIPPED SinAcuerdoPorFallos")
    fmt.Println(t.Name(), ".....................")

    cfg.startDistributedProcesses()

    fmt.Printf("Lider inicial\n")
    time.Sleep(time.Second *8)
    cfg.pruebaUnLider(3)
    e1,_,e1Leader,_ := cfg.obtenerEstadoRemoto(0)
    e2,_,e2Leader,_ := cfg.obtenerEstadoRemoto(1)
    e3,_,e3Leader,_ := cfg.obtenerEstadoRemoto(2)

    if !e1Leader && !e2Leader && !e3Leader {
        t.Errorf("NO hay lider")
    } else {
        fmt.Println("Estado actual")
        fmt.Println("nodo ", e1, " isLeader: ", e1Leader)
        fmt.Println("nodo ", e2, " isLeader: ", e2Leader)
        fmt.Println("nodo ", e3, " isLeader: ", e3Leader)
    }
    // Comprometer una entrada
    cfg.enviarOperacionCLiente()
    time.Sleep(time.Second * 2)
    //  Obtener un lider y, a continuación desconectar 2 de los nodos Raft
    fmt.Println(cfg.obtenerEntriesNodo(0))
    fmt.Println(cfg.obtenerEntriesNodo(1))
    fmt.Println(cfg.obtenerEntriesNodo(2))
```

```
cfg.stopNumberOfDistributedProcessesNoleader()
time.Sleep(time.Second * 2)
var e1Term, e2Term, e3Term  int
e1, e1Term,e1Leader,_ = cfg.obtenerEstadoRemotoWithNodesOff(0)
e2, e2Term,e2Leader,_ = cfg.obtenerEstadoRemotoWithNodesOff(1)
e3, e3Term,e3Leader,_ = cfg.obtenerEstadoRemotoWithNodesOff(2)
fmt.Println(cfg.obtenerEstadoRemotoWithNodesOff(0))
fmt.Println(cfg.obtenerEstadoRemotoWithNodesOff(1))
fmt.Println(cfg.obtenerEstadoRemotoWithNodesOff(2))

count := 0
leaderID := -1
if e1Term != -1 {
    count++
    leaderID = e1
}
if e2Term != -1 {
    count++
    leaderID = e2
}
if e3Term != -1 {
    count++
    leaderID = e3
}
fmt.Println(count)
if count > 1 {
    t.Errorf("Hay mas de un nodo arrancado")
}
// Comprobar varios acuerdos con 2 réplicas desconectada
// Comprometer una entrada
cfg.enviarOperacionCLiente()
fmt.Println(cfg.obtenerEntriesNodo(leaderID))
// Comprometer una entrada
cfg.enviarOperacionCLiente()
fmt.Println(cfg.obtenerEntriesNodo(leaderID))
// Comprometer una entrada
cfg.enviarOperacionCLiente()
fmt.Println(cfg.obtenerEntriesNodo(leaderID))
time.Sleep(time.Second * 2)
fmt.Println(cfg.obtenerEntriesNodo(leaderID))
// reconectar lo2 nodos Raft  desconectados y probar varios acuerdos
cfg.startDistributedProcesses()
time.Sleep(time.Second * 10)

fmt.Println(cfg.obtenerEntriesNodo(0))
fmt.Println(cfg.obtenerEntriesNodo(1))
fmt.Println(cfg.obtenerEntriesNodo(2))

entrysNodo1, commitI1 := cfg.obtenerEntriesNodo(0)
entrysNodo2, commitI2 := cfg.obtenerEntriesNodo(1)
entrysNodo3, commitI3 := cfg.obtenerEntriesNodo(2)

if commitI1 != commitI2 || commitI1 != commitI3 || commitI2 != commitI3 {
    t.Errorf("Indice de comiteadas diferente")
}
if len(entrysNodo1) != len(entrysNodo2) || len(entrysNodo1) != len(entrysNodo3)
```

```
        || len(entrysNodo2) != len(entrysNodo3){
        t.Errorf("Replycacion de nodos diferente")
    } else {
        for i,element := range entrysNodo1 {
            if element != entrysNodo2[i] || element != entrysNodo3[i]
                || entrysNodo2[i] != entrysNodo3[i]  {
                t.Errorf("Error Logs diferentes ")
            }
        }
    }
    cfg.stopDistributedProcesses()
}
```

5.3. Test 3

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

```
// Se somete 5 operaciones de forma concurrente -- 3 NODOS RAFT
func(cfg *configDespliegue) SometerConcurrentementeOperaciones(t *testing.T) {
// t.Skip("SKIPPED SometerConcurrentementeOperaciones")

    fmt.Println(t.Name(), ".....................")

    cfg.startDistributedProcesses()

    fmt.Printf("Esperamos Lider inicial\n")
    time.Sleep(time.Second * 8)
    cfg.pruebaUnLider(3)
    e1,_,e1Leader,_ := cfg.obtenerEstadoRemoto(0)
    e2,_,e2Leader,_ := cfg.obtenerEstadoRemoto(1)
    e3,_,e3Leader,_ := cfg.obtenerEstadoRemoto(2)

    if !e1Leader && !e2Leader && !e3Leader {
        t.Errorf("NO hay lider")
    }else {
        fmt.Println("Estado actual")
        fmt.Println("nodo ", e1, " isLeader: ", e1Leader)
        fmt.Println("nodo ", e2, " isLeader: ", e2Leader)
        fmt.Println("nodo ", e3, " isLeader: ", e3Leader)
    }
    // Obtener un lider y, a continuación someter una operacion
    fmt.Println("EStado previo del log")
    fmt.Println(cfg.obtenerEntriesNodo(0))
    fmt.Println(cfg.obtenerEntriesNodo(1))
    fmt.Println(cfg.obtenerEntriesNodo(2))
    cfg.SometerConcurrentementeOperacionesCliente()
    time.Sleep(time.Second * 4)
    // Comprobar estados de nodos Raft, sobre todo
    // el avance del mandato en curso e indice de registro de cada uno
    // que debe ser identico entre ellos
    fmt.Println(cfg.obtenerEntriesNodo(0))
    fmt.Println(cfg.obtenerEntriesNodo(1))
    fmt.Println(cfg.obtenerEntriesNodo(2))
    entrysNodo1, commitI1 := cfg.obtenerEntriesNodo(0)
    entrysNodo2, commitI2 := cfg.obtenerEntriesNodo(1)
    entrysNodo3, commitI3 := cfg.obtenerEntriesNodo(2)
```

```
    if  commitI1 != commitI2 || commitI1 != commitI3 || commitI2 != commitI3 {
        t.Errorf("Indice de comiteadas diferente")
    }
    if  len(entrysNodo1) != len(entrysNodo2) || len(entrysNodo1) != len(entrysNodo3)
        || len(entrysNodo2) != len(entrysNodo3){
        t.Errorf("Replycacion de nodos diferente")
    } else {
        for i,element := range entrysNodo1 {
            if  element != entrysNodo2[i] || element != entrysNodo3[i]
                || entrysNodo2[i] != entrysNodo3[i]  {
                t.Errorf("Error Logs diferentes ")
            }
        }
    }
}
```