Entrega Práctica 3 - Sistemas distribuidos

Raft 1ª parte

Autores:

Fernando Pastor Peralta (897113)

Javier Murillo Jimenez (897154)

Fecha de entrega: 30/10/2025

Grupo: Pareja:1-6

Índice

Introducción: 3
Diseño de nuestra solución: 3
Arquitectura de las estructuras principales: 4
Estructura de Datos principal 4
Estados del Nodo Raft 5
Estructuras de Entradas RPC 5
Implementación del Algoritmo de Elección de Líder 6
Máquina de estados principal 6
Estado Follower 7
Estado Candidate 7
Estado Leader 9
Implementación de RPCs 9
PedirVoto RPC 9
AppendEntries RPC 10
Gestión de Timeouts 12
Configuración de Timeouts 12
API Pública 12
Inicialización de un nodo 12
Sumisión de Operaciones 13
Obtener estado 16
Parar un nodo 16
Características de la implementación 17
Pruebas 17
Diseño del algoritmo: 18
Diagrama de secuencia de Elección de Líder: 18
Diagrama de secuencia de Replicación de operaciones: 19
Diagrama de secuencia de latidos periódicos y detección de fallos: 20
Diagrama de secuencia de Elecciones simultáneas: 21

Introducción:

El problema a resolver en esta práctica consiste en la primera parte de la implementación del algoritmo de consenso de Raft, enfocándose en el mecanismo de elección de líder y la funcionalidad básica de replicación de logs sin considerar fallos.

Los objetivos a realizar en esta práctica se presentan a continuación:

- Implementar el algoritmo de elección de líder de Raft con manejo básico de fallos.
- Desarrollar la llamada RPC AppendEntries para operaciones sin fallos.
- Garantizar la elección correcta de líder mediante votación mayoritaria (mayoría más uno).
- Establecer la infraestructura de comunicación entre nodos Raft.

Diseño de nuestra solución:

Al arrancar cada nodo (NuevoNodo()), este se inicia como Follower con log vacío índice desde 1, currentTerm=0, votedFor=-1, commitIndex=0 y lanza automaticamente la goroutine principal actualizarEstados().

El nodo espera un timeout aleatorio (150-300 ms) para convertirse en Candidate (solicita votación porque no le llega el latido a tiempo): incrementa currentTerm, se auto-vota y envía RequestVote (enviarPeticionVoto()) a todos los nodos con {Term, CandidateId=Yo, LastLogIndex=len(log), LastLogTerm=log[N-1].Term}.

Un Follower vota (PedirVoto()) si termCandidato >= currentTerm y logEsActualizado() (mayor Term o mismo Term e Indice >=). Con mayoría absoluta, el Candidate pasa a Leader: actualiza IdLider=Yo y lanza gorutina bucleHeartbeats(), que envía AppendEntries RPCs vacíos a todas las réplicas.

Los Followers procesan AppendEntries: rechazan si term < currentTerm o log previo inconsistente, por el contrario, añaden su nueva entrada y actualizan commitIndex.

Para ejecutar el servidor raft, mantenemos el main.go proporcionado por el profesorado. Por otro lado, para el diseño del cliente, creamos un paquete con instrucciones para comunicarse con el servidor raft (cltraft.go) que utilizan las funciones RPC de la API para comunicarse de forma efectiva entre cliente y servidor.

Por último hemos utilizado el programa con test que está proporcionado con el código original para depurar y hemos diseñado el test 4 de estabilidad de tres operaciones comprometidas en nuestro sistema. Más información sobre las pruebas en el punto de “Pruebas”.

Arquitectura de las estructuras principales:

Estructura de Datos principal

```Go
// Tipo de dato Go que representa un solo nodo (réplica) de raft
type NodoRaft struct {
	Mux     sync.Mutex            // Mutex para proteger acceso a estado compartido.
	Nodos   []rpctimeout.HostPort // Host:Port de todos los nodos (réplicas) Raft, en mismo orden.
	Yo      int                   // Índice de este nodo en campo array "nodos".
	IdLider int                   // Índice del lider.
	Logger  *log.Logger           // Utilización opcional de este logger para depuración, cada nodo Raft tiene su propio registro de trazas (logs).

	// ----  Persistent state on allservers: ----
	currentTerm int       // Mandato actual.
	votedFor    int       // A quién se votó en currentTerm (-1 si nadie).
	log         []entrada // Registro de entradas, cada entrada contiene el mandato y la operación.
	// ----- Volatile state on all servers: -----
	commitIndex int // Mayor índice comprometido
	lastApplied int // Mayor índice aplicado a la máquina de estados
	// ------- Volatile state on leaders: -------
	nextIndex  []int // Para cada servidor, índice de la siguiente entrada a enviar.
	matchIndex []int // Para cada servidor, índice del mayor registro conocido como replicado.
	// ----------------------------------------
	estado                Estado               // Indica si soy seguidor, candidato o lider.
	latidos               chan bool            // Temporizador para latidos del lider.
	votosRecibidos        int                  // Contador de votos en elección.
	finEleccion           chan bool            // Para señalar fin de elección.
	quit                  chan bool            // Para parar gorutinas.
	inicioEleccion        time.Time            // Marca de tiempo del inicio de la elección
	enEleccion            bool                 // Indica si el nodo está en proceso de elección
	canalAplicarOperacion chan AplicaOperacion // siguiente práctica }
```

Estados del Nodo Raft

```Go
type Estado string
const (
    Follower  Estado = "follower"
    Candidate Estado = "candidate"
    Leader    Estado = "leader"
)
```

Estructuras de Entradas RPC

```Go
type ArgsPeticionVoto struct {
       Term            int
       IdCandidato     int
       UltimoLogIndice int
       UltimoLogTerm   int
}
type RespuestaPeticionVoto struct {
       Term         int
       ConcederVoto bool
}
type ArgAppendEntries struct {
       Term            int
       IdLider         int
       PrevioLogIndice int
       PrevioLogTerm   int
       Entradas        []entrada
       LeaderCommit    int
}

type Results struct {
       Term  int
       Exito bool
}
```

Implementación del Algoritmo de Elección de Líder

Máquina de estados principal

```Go
func (nr *NodoRaft) actualizarEstados() {
	for {
		select {
		case <-nr.quit:
			return
		default:
		}
		nr.Mux.Lock()
		estado := nr.estado
//...
		// Watchdog: Si no somos líder y estamos en proceso de elección, miramos cuánto llevamos buscando líder.
		if estado != Leader && nr.enEleccion {
			// Calculamos cuánto ha pasado desde inicioEleccion.
			tiempo_eleccion := time.Since(nr.inicioEleccion)
			if tiempo_eleccion > 2500*time.Millisecond {
				nr.Logger.Printf("Error: no se ha podido elegir líder en %v. Abortando.\n", tiempo_eleccion)
				nr.Mux.Unlock()
				os.Exit(1)
			}
		}

		// Si somos líder, significa que ya hay líder y podemos poner a false enEleccion.
		if estado == Leader {
			nr.enEleccion = false
		}
//...
		nr.Mux.Unlock()
		// Ejecutamos el bucle correspondiente al estado actual.
		switch estado {
		case Follower:
			nr.bucleFollower()
		case Candidate:
			nr.bucleCandidate()
		case Leader:
			nr.bucleLeader()
		}	}
}
```

Estado Follower

```Go
func (nr *NodoRaft) bucleFollower() {
	timeout := time.Duration(150+rand.Intn(150)) * time.Millisecond
	timer := time.NewTimer(timeout)
	defer timer.Stop()

	for {
		select {
		case <-nr.quit: // Parada del nodo Raft.
			return
		case <-nr.latidos: // Latido recibido de un líder.
			if !timer.Stop() {
				<-timer.C
			}
			timer.Reset(time.Duration(150+rand.Intn(150)) *
time.Millisecond)
		case <-timer.C: // Timeout sin latidos, iniciar elección.
			nr.Mux.Lock()
			nr.estado = Candidate
			nr.enEleccion = true
		    nr.inicioEleccion = time.Now()
			nr.Mux.Unlock()
			return
		}
	}
}
```

Estado Candidate

```Go
func (nr *NodoRaft) bucleCandidate() {
	nr.Mux.Lock()
	nr.currentTerm++
	nr.votedFor = nr.Yo
	nr.votosRecibidos = 1
	nr.enEleccion = true
       nr.inicioEleccion = time.Now()
	termActual := nr.currentTerm
	nr.Mux.Unlock()

	// Limpiar cualquier notificación vieja de mayoría
	select {
	case <-nr.finEleccion:
	default:
	}

	// Enviar peticiones de voto.
	go nr.enviarPeticionesVoto(termActual)
```

```
	timeout := time.Duration(150+rand.Intn(150)) * time.Millisecond
	timer := time.NewTimer(timeout)
	defer timer.Stop()

	for {
		select {
		 case <-nr.quit: // Parada del nodo Raft.
			return
		 case <-nr.latidos: // Latido recibido de un líder.
			nr.Mux.Lock()
			// Se acabaron las elecciones, ya hay lider.
			nr.estado = Follower
			nr.enEleccion = false
			nr.Mux.Unlock()
			return
		 case <-nr.finEleccion: // Mayoría de votos recibida.
			nr.Mux.Lock()
			// Si tenemos mayoría de votes, nos gacemos líderes.
			if nr.votosRecibidos > len(nr.Nodos)/2 {
				//Iniciamos como lider
				nr.estado = Leader
				nr.enEleccion = false
				ultimoIndice := len(nr.log)
				for i := range nr.nextIndex {
					nr.nextIndex[i] = ultimoIndice + 1
					nr.matchIndex[i] = 0
				}
				nr.matchIndex[nr.Yo] = ultimoIndice
				nr.nextIndex[nr.Yo] = ultimoIndice + 1
				nr.IdLider = nr.Yo
				nr.Mux.Unlock()
				return
			}
			// Si no tenemos mayoría, volvemos a ser seguidores.
			nr.estado = Follower
			nr.enEleccion = false
			nr.Logger.Printf("[Nodo %d] CANDIDATE -> FOLLOWER (no mayoría)
term=%d votos=%d\n", nr.Yo, nr.currentTerm, nr.votosRecibidos)
			nr.Mux.Unlock()
			return
		 case <-timer.C: // Timeout de elección.
			nr.Mux.Lock()
			nr.estado = Follower
			nr.enEleccion = false
			nr.Mux.Unlock()
			return // Reiniciará el bucle de actualizarEstados y entrará
en Follower sin elección. Para reiniciar elección.
		 }
	}
}
```

Estado Leader

```Go
func (nr *NodoRaft) bucleLeader() {
	timer := time.NewTimer(50 * time.Millisecond)
	defer timer.Stop()

	for {
		select {
		case <-nr.quit:
			return
		case <-timer.C:
			nr.Mux.Lock()
			if nr.estado != Leader {
				nr.Mux.Unlock()
				return
			}
			nr.Mux.Unlock()
			nr.enviarLatidos()
			// Reiniciar el timer para el siguiente latido.
			timer.Reset(50 * time.Millisecond)
		}
	}
}
```

Implementación de RPCs

PedirVoto RPC

```Go
func (nr *NodoRaft) PedirVoto(peticion *ArgsPeticionVoto,
	reply *RespuestaPeticionVoto) error {
	nr.Mux.Lock()
	defer nr.Mux.Unlock()

	if peticion.Term < nr.currentTerm { // No se le concede el voto al reply si
term < currentTerm.
		reply.Term = nr.currentTerm
		reply.ConcederVoto = false
		return nil
	} else if peticion.Term > nr.currentTerm {
		nr.currentTerm = peticion.Term
		nr.estado = Follower
		nr.votedFor = -1
		nr.enEleccion = false
	}

	reply.Term = peticion.Term
```

```
        if (nr.votedFor == -1 || nr.votedFor == peticion.IdCandidato) &&
nr.logEsActualizado(peticion.UltimoLogIndice, peticion.UltimoLogTerm) {
                nr.votedFor = peticion.IdCandidato
                reply.ConcederVoto = true
                select {
                case nr.latidos <- true:
                default:
                }
        } else {
                reply.ConcederVoto = false
        }

        return nil
}
```

AppendEntries RPC

Go

```
func (nr *NodoRaft) AppendEntries(args *ArgAppendEntries, results *Results) error {
          nr.Mux.Lock()
        defer nr.Mux.Unlock()

        // valor por defecto.
        results.Term = nr.currentTerm
        results.Exito = false

        // Rechazar si el Term del líder es viejo.
        if args.Term < nr.currentTerm {
                return nil
        }

        // Si el líder tiene término más nuevo, actualizamos.
        if args.Term > nr.currentTerm {
                nr.currentTerm = args.Term
                nr.estado = Follower
                nr.votedFor = -1
                nr.enEleccion = false
        }

        // Ya sabemos quién es el líder actual.
        nr.IdLider = args.IdLider
        nr.enEleccion = false

        select {
        case nr.latidos <- true:
```

```
	default:
	}

	// Comprobamos consistencia del log previo.
	if args.PrevioLogIndice > 0 {
		// No hay suficiente log.
		if len(nr.log) < args.PrevioLogIndice {
			return nil
		}
		// Tengo esa entrada, pero el Term no coincide, esto es conflictivo.
		if nr.log[args.PrevioLogIndice-1].Term != args.PrevioLogTerm {
			return nil
		}
	}

	// Insertar/sobreescribir entradas nuevas del líder.
	if len(args.Entradas) > 0 {
		// Cortar mi log hasta PrevioLogIndice
		nr.log = nr.log[:args.PrevioLogIndice]
		// Añadir las entradas nuevas
		for _, e := range args.Entradas {
			nr.log = append(nr.log, e)
		}
	}

	// Actualizar commitIndex siguiendo al líder.
	if args.LeaderCommit > nr.commitIndex {
		if args.LeaderCommit > len(nr.log) {
			nr.commitIndex = len(nr.log)
		} else {
			nr.commitIndex = args.LeaderCommit
		}
	}

	results.Term = nr.currentTerm
	results.Exito = true
	return nil

}
```

Gestión de Timeouts

Configuración de Timeouts

```Go
// Timeouts aleatorios para evitar elecciones simultáneas
timeout := time.Duration(150+rand.Intn(150)) * time.Millisecond

// Heartbeats cada 50ms
heartbeatTimer := time.NewTimer(50 * time.Millisecond)

// Timeout global de elección: 2.5 segundos
tiempo_eleccion := time.Since(nr.inicioEleccion)
if tiempo_eleccion > 2500*time.Millisecond {
	nr.Logger.Printf("Error: no se ha podido elegir líder en %v. Abortando.\n",
tiempo_eleccion)
	nr.Mux.Unlock()
	os.Exit(1)
}
```

API Pública

Inicialización de un nodo

```Go

// Devuelve el nodo Raft incializado y lanza la gouroutina de gestión de estados.
func NuevoNodo(nodos []rpctimeout.HostPort, yo int,
	canalAplicarOperacion chan AplicaOperacion) *NodoRaft {
	nr := &NodoRaft{}
	nr.Nodos = nodos
	nr.Yo = yo
	nr.IdLider = -1

	rand.Seed(time.Now().UnixNano())
	if kEnableDebugLogs {
		nombreNodo := nodos[yo].Host() + "_" + nodos[yo].Port()
		logPrefix := fmt.Sprintf("%s", nombreNodo)

		fmt.Println("LogPrefix: ", logPrefix)

		if kLogToStdout {
			nr.Logger = log.New(os.Stdout, nombreNodo+" -->> ",
				log.Lmicroseconds|log.Lshortfile)
		} else {
			err := os.MkdirAll(kLogOutputDir, os.ModePerm)
```

```
                        if err != nil {
                                panic(err.Error())
                        }
                        logOutputFile, err := os.OpenFile(fmt.Sprintf("%s/%s.txt",
                                kLogOutputDir, logPrefix),
os.O_RDWR|os.O_CREATE|os.O_TRUNC, 0755)
                        if err != nil {
                                panic(err.Error())
                        }
                        nr.Logger = log.New(logOutputFile,
                                logPrefix+" -> ", log.Lmicroseconds|log.Lshortfile)
                }
                nr.Logger.Println("logger initialized")
        } else {
                nr.Logger = log.New(ioutil.Discard, "", 0)
        }

        nr.currentTerm = 0
        nr.votedFor = -1
        nr.log = make([]entrada, 0, 64)
        nr.commitIndex = 0
        nr.lastApplied = 0
        nr.nextIndex = make([]int, len(nodos))
        nr.matchIndex = make([]int, len(nodos))
        nr.estado = Follower
        nr.latidos = make(chan bool, 1)
        nr.votosRecibidos = 0
        nr.finEleccion = make(chan bool, 1)
        nr.quit = make(chan bool, 1)
        nr.enEleccion = false
        nr.inicioEleccion = time.Now()
        nr.canalAplicarOperacion = canalAplicarOperacion

        // Inicializamos nextIndex y matchIndex para líderes.
        for i := range nr.nextIndex {
                nr.nextIndex[i] = 1
                nr.matchIndex[i] = 0
        }

        go nr.actualizarEstados()
        return nr
}
```

Sumisión de Operaciones

```
Go
func (nr *NodoRaft) SometerOperacionRaft(operacion TipoOperacion, reply
*ResultadoRemoto) error

func (nr *NodoRaft) someterOperacion(operacion TipoOperacion) (int, int,
        bool, int, string) {
```

```
	nr.Mux.Lock()
	defer nr.Mux.Unlock()

	//Comprobamos si somos el líder.
	if nr.estado != Leader {
		return -1, nr.currentTerm, false, nr.IdLider, ""
	}
	// Añadir entrada al log.
	nuevaEntrada := entrada{
		Term:      nr.currentTerm,
		Indice:    len(nr.log) + 1,
		Operacion: operacion,
	}
	nr.log = append(nr.log, nuevaEntrada)

	// Actualizamos nuestros índices internos de replicación para "yo".
	nr.matchIndex[nr.Yo] = len(nr.log)
	nr.nextIndex[nr.Yo] = len(nr.log) + 1

	// Replicar esta entrada en TODOS los seguidores.
	acks := 1 // yo mismo
	mayoria := len(nr.Nodos)/2 + 1

	for i := range nr.Nodos {
		if i == nr.Yo {
			continue
		}
		if nr.replicaSeguidor(i, nuevaEntrada.Indice) {
			acks++
		}
	}

	// Si hay mayoría, subimos commitIndex hasta esa entrada.
	if acks >= mayoria {
		if nuevaEntrada.Indice > nr.commitIndex {
			nr.commitIndex = nuevaEntrada.Indice
		}
	}

	return nuevaEntrada.Indice, nr.currentTerm, true, nr.Yo, ""
}

/*
 * Auxiliar de someterOperacion:
 * Intenta que "nodo" tenga al menos la entrada "hastaIndice".
 * Devuelve true si el seguidor alcanza esa entrada.
 */
func (nr *NodoRaft) replicaSeguidor(nodo int, hastaIndice int) bool {
	for {
		// Devuelve true si ya está replicado
		if nr.matchIndex[nodo] >= hastaIndice {
			return true
		}
```

```
                // Calcular PrevioLogIndice, PrevioLogTerm y las entradas pendientes.
                IndicePrevio := nr.nextIndex[nodo] - 1
                TermPrevio := 0
                if IndicePrevio > 0 && IndicePrevio <= len(nr.log) {
                        TermPrevio = nr.log[IndicePrevio-1].Term
                }

                entradas := append([]entrada(nil), nr.log[nr.nextIndex[nodo]-1:]...)

                args := ArgAppendEntries{
                        Term:            nr.currentTerm,
                        IdLider:         nr.Yo,
                        PrevioLogIndice: IndicePrevio,
                        PrevioLogTerm:   TermPrevio,
                        Entradas:        entradas,
                        LeaderCommit:    nr.commitIndex,
                }

                reply := Results{}
                ok := nr.enviarLatido(nodo, &args, &reply)

                if !ok {
                        return false
                }

                // Si el follower nos devuelve un término más grande, hemos dejado de
ser líder.
                if reply.Term > nr.currentTerm || nr.estado != Leader {
                        nr.currentTerm = reply.Term
                        nr.estado = Follower
                        nr.votedFor = -1
                        return false
                }

                if reply.Exito {
                        // Actualizamos nextIndex y el matchIndex de ese follower.
                        nr.matchIndex[nodo] = args.PrevioLogIndice + len(args.Entradas)
                        nr.nextIndex[nodo] = nr.matchIndex[nodo] + 1

                        done := nr.matchIndex[nodo] >= hastaIndice
                        if done {
                                return true
                        }

                } else {
                        // Fallo de consistencia: retroceder nextIndex y reintentar.
                        if nr.nextIndex[nodo] > 1 {
                                nr.nextIndex[nodo]--
                        }
                        // loop y reintenta.
                }
        }
}
```

Obtener estado

```Go
func (nr *NodoRaft) ObtenerEstadoNodo(args Vacio, reply *EstadoRemoto) error;
func (nr *NodoRaft) obtenerEstado() (int, int, bool, int) {
        nr.Mux.Lock()
        defer nr.Mux.Unlock()
        var yo int = nr.Yo
        var mandato int = nr.currentTerm
        var esLider bool = (nr.estado == Leader)
        var idLider int =nr.IdLider
        return yo, mandato, esLider, idLider
}
```

Parar un nodo

```Go
func (nr * NodoRaft) ParaNodo(args Vacio, reply *Vacio) error;
func (nr *NodoRaft) para() {
        nr.quit <- true
        go func() {time.Sleep(5 * time.Millisecond); os.Exit(0) } ()
}
```

Características de la implementación

Con nuestra implementación podemos asegurar que se cumplen las siguientes características:

- Timeouts aleatorios para evitar elecciones de líder simultáneas.
- Heartbeats periódicos cada 50 ms.
- Control del timeout global de 2,5 segundos por cada elección.
- Verificación de logs para garantizar consistencia.
- Comunicación RPC robusta con timeouts adecuados.

Pruebas

Nuestras pruebas se basan en dos pilares:

- Sistema de logs: Ya estaba en gran medida implementado, nosotros hemos puesto varios mensajes de log en distintas zonas del código, para ayudarnos a verificar el correcto funcionamiento del programa.
- Programa de test: El proporcionado originalmente contenía 3 test ya creados:
    1. Test que únicamente arranca y para un nodo.
    2. Test en el que se verifica que se elige al primer líder.
    3. Test en el que se tras un fallo en el sistema, se elige un nuevo líder.

    También hemos implementado un cuarto test:

    4. Test de estabilidad de tres operaciones comprometidas en nuestro sistema.

    El diseño de este último es básicamente una identificación del líder, la creación de un cliente mediante el paquete de cltraft.go, el sometimiento de tres operaciones y la detección de errores al comprometer cada una de ellas.

    Los 4 test dan resultado positivo. Hay que tener en cuenta que para verificar el primer test, debido a la naturaleza de nuestra solución en la que se inicializa la votación de forma automática en cuanto se crea el nodo, hemos creado una constante AutoEleccionesAlArrancar, que debe estar a false en todos los servidores raft para probar este test. Para los otros 3 (test 2-4), la constante tiene que estar a true.

Diseño del algoritmo:

Diagrama de secuencia de Elección de Líder:

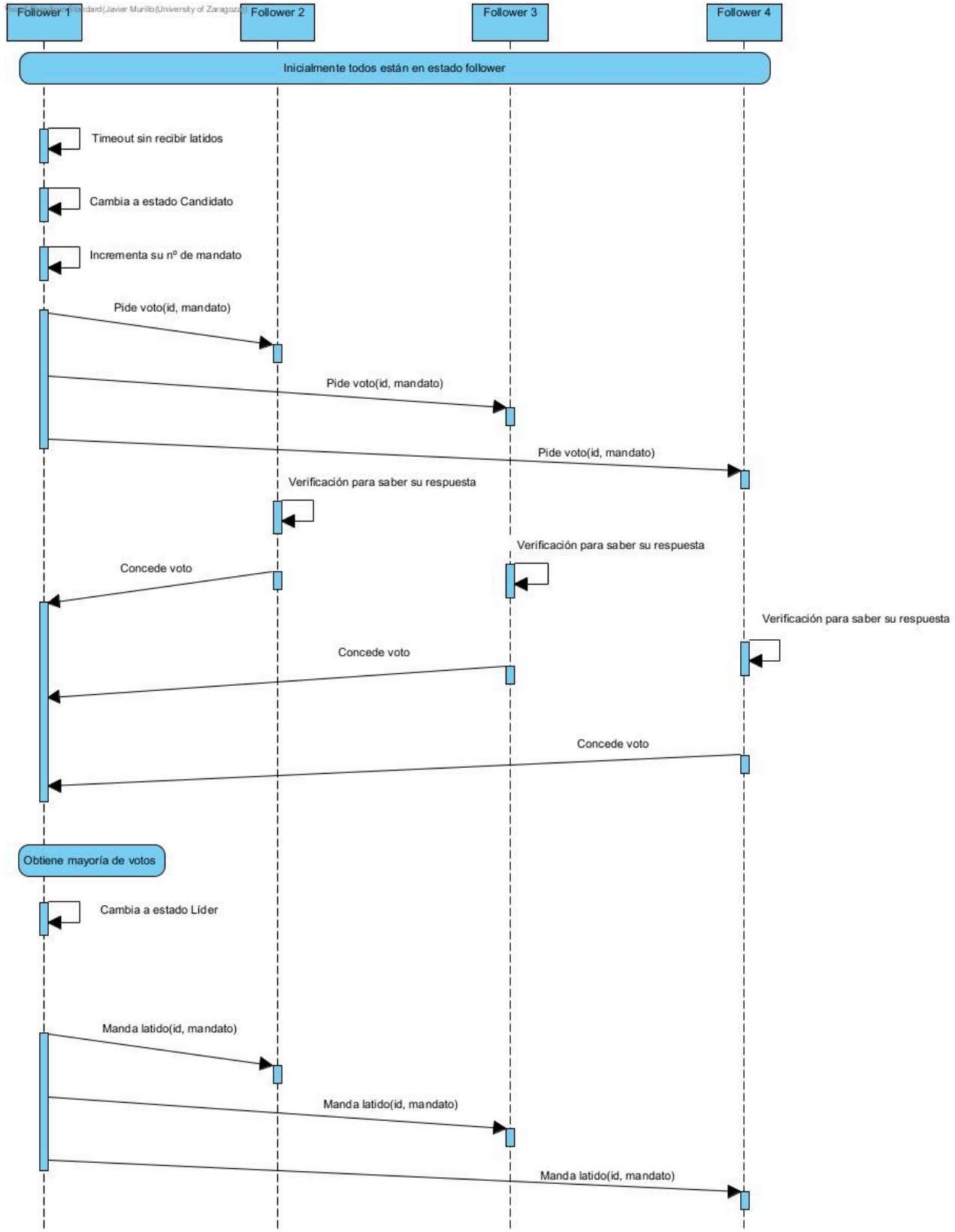

Diagrama de secuencia de Replicación de operaciones:

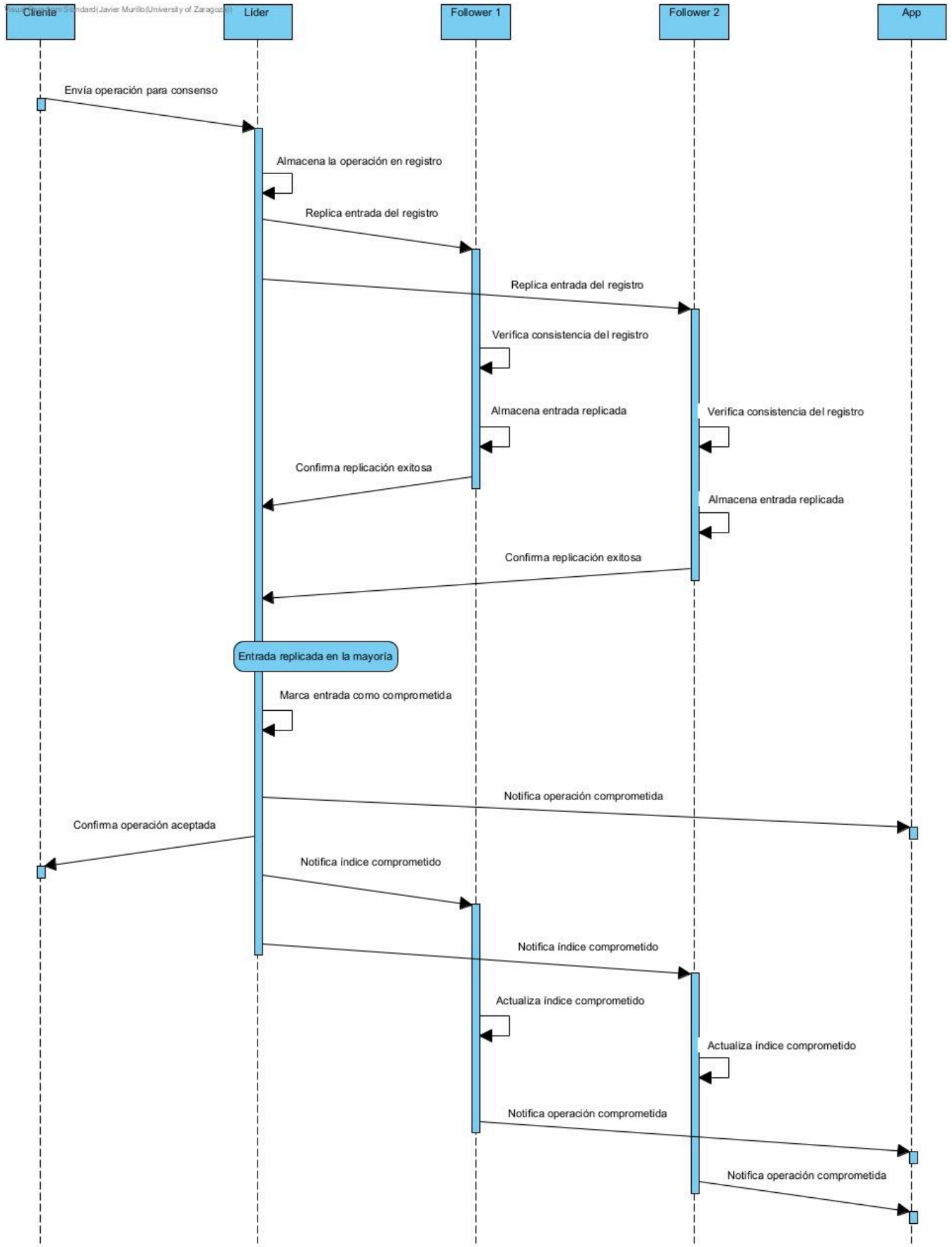

Diagrama de secuencia de latidos periódicos y detección de fallos:

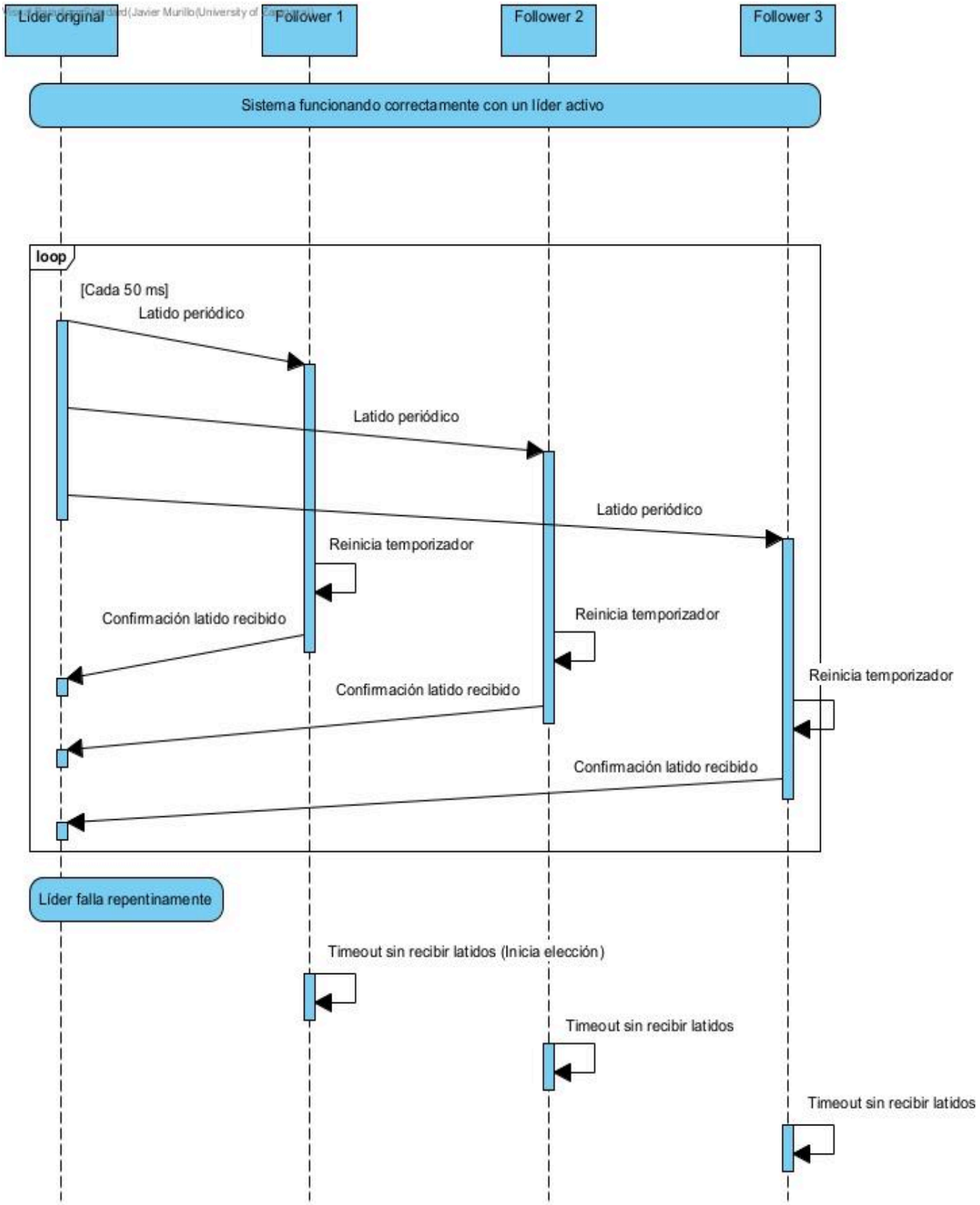

Diagrama de secuencia de Elecciones simultáneas: