Practical Assignment 3 - Distributed Systems

Raft Part 1

Authors:

Fernando Pastor Peralta (897113)
Javier Murillo Jimenez (897154)

Delivery date: 30/10/2025

Group: Pair: 1-6

Index

Introduction:

The problem to be solved in this exercise consists of the first part of the implementation of the Raft consensus algorithm, focusing on the leader election mechanism and the basic functionality of log replication without considering failures.

The objectives to be achieved in this practice are presented below:

- Implement Raft's leader selection algorithm with basic fault handling.
- Develop the RPC AppendEntries call for fail-safe operations.
- Ensure the correct election of leader through majority vote (majority plus one).
- Establish the communication infrastructure between Raft nodes.

Design of our solution:

When each node starts (NewNode()), it starts as a Follower with an empty log index from 1, currentTerm=0, votedFor=-1, commitIndex=0 and automatically launches the main goroutineupdateStates().

The node waits for a random timeout (150-300 ms) to become a Candidate (requests a vote because it does not receive the heartbeat on time): increments currentTerm, votes for itself and sends RequestVote (sendRequestVote()) to all nodes with {Term, CandidateId=Yo, LastLogIndex=len(log), LastLogTerm=log[N-1].Term}.

A Follower votes (RequestVote()) if termCandidate >= currentTerm and logIsUpdated() (greater Term or same Term and Index >=). With an absolute majority, the Candidate becomes Leader: updates LeaderId=Me and launches the goroutine loopHeartbeats(), which sends empty AppendEntries RPCs to all replicas.

Followers process AppendEntries: they reject if term < currentTerm or previous log is inconsistent; otherwise, they add their new entry and update commitIndex.

To run the raft server, we use the main.go file provided by the instructors. For the client design, we create a package with instructions for communicating with the raft server (cltraft.go), which uses the API's RPC functions to ensure effective communication between the client and server.

Finally, we used the test program provided with the original code for debugging and designed test 4, a stability test for three committed operations in our system. More information about the tests can be found in the "Testing" section.

Architecture of the main structures:

Main Data Structure

```Go
// Go data type that represents a single raft node (replica)
type NodoRaft struct {
	Mux     sync.Mutex            // Mutex to protect access to shared
state.
	Nodes   []rpctimeout.HostPort // Host:Port of all Raft nodes
(replicas), in the same order.
	They      int                   // Index of this node in the array
field "nodes".
	IdLider int                   // Leader's Index.
	Logger  *log.Logger           // Optional use of this logger for
debugging; each Raft node has its own trace log.

	// ----  Persistent state on allservers: ----
	currentTerm int       // Current mandate.
	votedFor    int       // Who was voted for in currentTerm (-1 if
nobody).
	log         []Prohibited // Entry log, each entry contains the command
and the operation.
	// ----- Volatile state on all servers: -----
	commitIndex int // Highest committed index
	lastApplied int // Highest index applied to the state machine
	// ------- Volatile state on leaders: -------
	nextIndex  []int // For each server, index of the next entry to send.
	matchIndex []int // For each server, index of the largest record known
as replicated.
	// ----------------------------------------
	state                State               // Indicates whether I am a
follower, candidate, or leader.
	heartbeats              chan bool            // Timer for leader's
heartbeats.
	votes received        int                  // Election vote counter.
	finElection           chan bool            // To signal the end of the
election.
	quit                  chan bool            // To stop routines.
	startElection        time.Time            // Timestamp of the start of
the election
	inElection            bool                 // Indicates if the node is
in the election process
	channelApplyOperation chan ApplyOperation // next practice }
```

Raft Node States

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

RPC Input Structures

```Go
type ArgsPeticionVoto struct {
       Term            int
       CandidateID     int
       LastLogIndex int
       LastLogTerm   int
}
type ResponseRequestVote struct {
       Term         int
       Grant Vote bool
}
type ArgAppendEntries struct {
       Term            int
       IdLider         int
       PrevioLogIndice int
       PrevioLogTerm   int
       Tickets        []Prohibited
       LeaderCommit    int
}

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

Implementation of the Leader Election Algorithm

Main state machine

```Go
func (no. *NodoRaft) updateStates() {
	for {
		select {
		case <-nr.quit:
			return
		default:
		}
		nr.Mux.Lock()
		state := nr.state
//...
		// Watchdog: If we are not a leader and are in the process of being elected, we look at how long we have been looking for a leader.
		if state != Leader && nr.enEleccion {
			// We calculate how much time has passed since the start of the election.
			election_time := time.Since(nr.inicioEleccion)
			if election_time > 2500*time.Millisecond {
				no.Logger.Printf("Error: Could not select a leader in %v. Aborting.\n", election_time)
				nr.Mux.Unlock()
				os.Exit(1)
			}
		}

		// If we are the leader, it means there is already a leader and we can put false in Election.
		if state == Leader {
			nr.enEleccion = false
		}
//...
		nr.Mux.Unlock()
		// We execute the loop corresponding to the current state.
		switch state {
		case Follower:
			nr.bucleFollower()
		case Candidate:
			nr.bucleCandidate()
		case Leader:
			no.loopsLeader()
	}	}
}
```

Follower Status

```Go
func (no. *NodoRaft) bucleFollower() {
	timeout := time.Duration(150+rand.Intn(150)) * time.Millisecond
	timer := time.NewTimer(timeout)
	defer timer.Stop()

	for {
		select {
		case <-nr.quit: // Raft node stop.
			return
		case <-nr.beats:// Heartbeat received from a leader.
			if !timer.Stop() {
				<-timer.C
			}
			timer.Reset(time.Duration(150+rand.Intn(150)) *
time.Millisecond)
		case <-timer.C:// Timeout with no heartbeat, start election.
			nr.Mux.Lock()
			nr.state = Candidate
			nr.inElection = true
		     nr.startElection = time.Now()
			nr.Mux.Unlock()
			return
		}
	}
}
```

Candidate State

```Go
func (no. *NodoRaft) bucleCandidate() {
	nr.Mux.Lock()
	nr.currentTerm++
	no.votedFor = nr.Yo
	nr.votosRecibidos = 1
	nr.enEleccion = true
	nr.startElection = time.Now()
	termActual := nr.currentTerm
	nr.Mux.Unlock()

	// Clear any old notifications from the majority
	select {
	case <-nr.finElection:
	default:
	}

	// Send voting requests.
	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: // Raft node stop.
                        return
                 case <-nr.beats: // Heartbeat received from a leader.
                        nr.Mux.Lock()
                        // The elections are over, there is a leader.
                        nr.state = Follower
                        nr.enEleccion = false
                        nr.Mux.Unlock()
                        return
                 case <-nr.finElection: // Majority of votes received.
                        nr.Mux.Lock()
                        // If we get a majority of votes, we become leaders.
                        if nr.votosRecibidos > only(No. Nodos)/2 {
                                We started as leaders
                                nr.state = Leader
                                nr.enEleccion = false
                                last Index := only(log no.)
                                for i := range no.nextIndex {
                                        nr.nextIndex[i] = last Index + 1
                                        no.matchIndex[i] = 0
                                }
                                nr.matchIndex[nr.Yo] = last Index
                                nr.nextIndex[nr.Yo] = last Index + 1
                                no.IdLider = nr.Yo
                                nr.Mux.Unlock()
                                return
                        }
                        // If we don't have a majority, we go back to being followers.
                        nr.state = Follower
                        nr.enEleccion = false
                        no.Logger.Printf("[Node %d] CANDIDATE -> FOLLOWER (not
majority) term=%d votes=%d\n", nr.Yo, nr.currentTerm, nr.votosRecibidos)
                        nr.Mux.Unlock()
                        return
                case <-timer.C: // Election timeout.
                        nr.Mux.Lock()
                        nr.state = Follower
                        nr.enEleccion = false
                        nr.Mux.Unlock()
                        return // This will restart the updateStates loop and enter
Follower mode with no choice. To restart the choice.
                }
        }
}
```

Leader State

```Go
func (no. *NodoRaft) loopsLeader() {
	timer := time.NewTimer(50 * time.Millisecond)
	defer timer.Stop()

	for {
		select {
		case <-nr.quit:
			return
		case <-timer.C:
			nr.Mux.Lock()
			if nr.state != Leader {
				nr.Mux.Unlock()
				return
			}
			nr.Mux.Unlock()
			nr.enviarLatidos()
			// Reset the timer for the next heartbeat.
			timer.Reset(50 * time.Millisecond)
		}
	}
}
```

Implementation of RPCs

Ask for Votes RPC

```Go
func(nr *RaftNode) RequestVote(request *ArgsRequestVote,
	reply *ResponsePetitionVote)error {
	nr.Mux.Lock()
	defer nr.Mux.Unlock()

	if peticion.Term < nr.currentTerm { // The reply is not granted a vote if term
< currentTerm.
		reply.Term = nr.currentTerm
		reply.ConcederVoto = false
		return nil
	} else if peticion.Term > nr.currentTerm {
		nr.currentTerm = peticion.Term
		nr.estado = Follower
		no.votedFor =-1
		nr.enEleccion = false
	}

	reply.Term = peticion.Term
```

```
        if(no.votedFor ==-1 || nr.votedFor == request.CandidateId) &&
nr.logIsUpdated(request.LastLogIndex, request.LastLogTerm) {
                nr.votedFor = peticion.IdCandidato
                reply.ConcederVoto = true
                select {
                casenr.latidos <- true:
                default:
                }
        } else {
                reply.ConcederVoto = false
        }

        return nil
}
```

AppendEntries RPC

```
Go
func (no. *NodoRaft) AppendEntries(args *ArgAppendEntries, results *Results) error {
          nr.Mux.Lock()
        defer nr.Mux.Unlock()

        // default value.
        results.Term = nr.currentTerm
results.Success = false

        // Reject if the leader's Term is old.
        if args.Term < nr.currentTerm {
                return nil
        }

        // If the leader has a newer term, we update.
        if args.Term > nr.currentTerm {
                nr.currentTerm = args.Term
nr.estado = Follower
no.votedFor =-1
                nr.inElection = false
        }

        // We already know who the current leader is.
no.IdLider = args.IdLider
        nr.inElection = false

        select {
        casenr.latidos <- true:
```

```
        default:
        }

        // We check the consistency of the previous log.
        if args.PrevioLogIndice > 0 {
                // Not enough log.
                if only(nr.log) < args.PrevioLogIndice {
                        return nil
                }
                // I have that entry, but the Term doesn't match, this is conflicting.
                if nr.log[args.PrevioLogIndice-1].Term != args.PrevioLogTerm {
                        return nil
                }
        }

        // Insert/overwrite new leader entries.
        if only(args.Entries) >0 {
                // Cut my log up to PreviousLogIndex
                nr.log = nr.log[:args.PrevioLogIndice]
                // Add new entries
                for_, e :=rangeargs.Entries {
no.log =append(no.log, e)
                }
        }

        // Update commitIndex following the leader.
        if args.LeaderCommit > nr.commitIndex {
                if args.LeaderCommit > only(no.log) {
                        nr.commitIndex = only(log no.)
                } else {
                        nr.commitIndex = args.LeaderCommit
                }
        }

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

}
```

Timeout Management

Timeout settings

```Go
// Random timeouts to prevent simultaneous selections
timeout := time.Duration(150+rand.Intn(150)) * time.Millisecond

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

// Global election timeout: 2.5 seconds
election_time := time.Since(nr.inicioEleccion)
if election_time > 2500*time.Millisecond {
	no.Logger.Printf("Error:No leader could be chosen in %v. Aborting.\n",
election_time)
	nr.Mux.Unlock()
	os.Exit(1)
}
```

Public API

Initializing a node

```Go

// Returns the initialized Raft node and launches the state management gouroutine.
func NewNode(nodes []rpctimeout.HostPort, them int,
	channelApplyOperation chan (Applies Operation) *NodoRaft {
	No. := &NodoRaft{}
	No. Transfer = nodes
	nr.Yo = them
	no.IdLider = -1

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

		fmt.Println("LogPrefix: ", logPrefix)

		if kLogToStdout {
			No.Logs = log.New(os.Stdout, nodeName+" -->> ",
				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())
                        }
                        No.Logs = log.New(logOutputFile,
                                logPrefix+" -> ", log.Lmicroseconds|log.Lshortfile)
                }
                no.Logger.Println("logger initialized")
        } else {
                No.Logs = log.New(ioutil.Discard, "", 0)
        }

        nr.currentTerm = 0
        no.votedFor = -1
        no.log = make([]Prohibited, 0, 64)
        nr.commitIndex = 0
        no.lastApplied = 0
        no.nextIndex = make([]int, only(nodes)
        no.matchIndex = make([]int, only(nodes)
        nr.state = Follower
        number of heartbeats = make(chan bool, 1)
        nr.votosRecibidos = 0
        finalElection number = make(chan bool, 1)
        nr.quit = make(chan bool, 1)
        nr.enEleccion = false
        nr.startElection = time.Now()
        nr.channelApplyOperation = channelApplyOperation

        // We initialize nextIndex and matchIndex for leaders.
        for i := range no.nextIndex {
                nr.nextIndex[i] = 1
                no.matchIndex[i] = 0
        }

        go nr.updateStates()
        return No.
}
```

Submission of Operations

```
Go
func (no. *NodoRaft) SubmitOperationRaft(operation TypeOfOperation, reply *Remote
Result* error

func (no. *NodoRaft) submitOperation(operation TypeOfOperation) (int, int,
        bool, int, string) {
```

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

        //We check if we are the leader.
        if nr.state != Leader {
                return -1, nr.currentTerm, false, no.IdLider, ""
        }
        // Add entry to log.
        newEntry := Prohibited{
                Term:      nr.currentTerm,
                Index:    only(log no.) + 1,
                Operation: operation,
        }
        no.log = append(no.log, newEntry)

        // We updated our internal replication indexes for "me".
        nr.matchIndex[nr.Yo] = only(log no.)
        nr.nextIndex[nr.Yo] = only(log no.) + 1

        // Replicate this post to ALL followers.
        acks := 1 // myself
        most := only(No. Nodos)/2 + 1

        for i := range No. Transfer {
                if i == nr.Yo {
                        continue
                }
                if nr.replicaSeguidor(i, newEntry.Index) {
                        acks++
                }
        }

        // If there is a majority, we raise commitIndex to that entry.
        if acks >= most {
                if NewEntry.Index > nr.commitIndex {
                        nr.commitIndex = NewEntry.Index
                }
        }

        return newEntry.Index, nr.currentTerm, true, nr.Yo, ""
}

/*
* Assistant to submit operation:
* Try to ensure that "node" has at least the entry "uptoIndex".
* Returns true if the follower reaches that input.
 */
func (no. *NodoRaft) follower replica(node int, hastaIndice int) bool {
        for {
                // Returns true if already replicated
                if no.matchIndex[node] >= hastaIndice {
                        return true
                }
```

```
                // Calculate PreviousLogIndex, PreviousLogTerm and pending entries.
                IndexPrevious := nr.nextIndex[nodo] - 1
                TermPrevio := 0
                if IndexPrevious > 0 && IndexPrevious <= only(log no.) {
                        TermPrevio = nr.log[IndicePrevio-1].Term
                }

                tickets := append([]Prohibited(nil), no.log[no.nextIndex[node]-1:]...)

                args := ArgAppendEntries{
                        Term:            nr.currentTerm,
                        IdLider:         nr.Yo,
                        PrevioLogIndice: IndexPrevious,
                        PrevioLogTerm:   TermPrevio,
                        Tickets:        tickets,
                        LeaderCommit:    nr.commitIndex,
                }

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

                if !ok {
                        return false
                }

                // If the follower returns a larger term, we have ceased to be the
leader.
                if reply.Term > nr.currentTerm || nr.state != Leader {
                        nr.currentTerm = reply.Term
                        nr.state = Follower
                        no.votedFor = -1
                        return false
                }

                if Reply. Success {
                        // We update nextIndex and matchIndex of that follower.
                        no.matchIndex[node] = args.PrevioLogIndice + only(args.Inputs)
                        nr.nextIndex[nodo] = no.matchIndex[node] + 1

                        done := no.matchIndex[node] >= hastaIndice
                        if done {
                                return true
                        }

                } else {
                        // Consistency failure: go back to nextIndex and retry.
                        if nr.nextIndex[nodo] > 1 {
                                nr.nextIndex[nodo]--
                        }
                        // loop and retry.
                }
        }
}
```

Get status

```Go
func (no. *NodoRaft) GetNodeState(args Empty, reply *RemoteState) error;
func(nr *RaftNode) getState() (int, int, bool, int) {
        nr.Mux.Lock()
        defer nr.Mux.Unlock()
        ourthemint = nr.Yo
        ourmandateint = nr.currentTerm
        our esLider bool= (nr.state == Leader)
        ouridLiderint=no.IdLider
        returnI, command, isLeader, idLeader
}
```

Stop a node

```Go
func (no. * NodoRaft) ParaNode(args Empty, reply *Empty) error;
func (no. *NodoRaft) to() {
        nr.quit <- true
        go func() {time.Sleep(5 * time.Millisecond); os.Exit(0) } ()
}
```

Implementation characteristics

With our implementation we can ensure that the following characteristics are met:

- Random timeouts to prevent simultaneous leader elections.
- Periodic heartbeats every 50 ms.
- Global timeout control of 2.5 seconds per choice.
- Log verification to ensure consistency.
- Robust RPC communication with appropriate timeouts.

Evidence

Our tests are based on two pillars:

- Logging system: It was already largely implemented; we have placed several log messages in different areas of the code to help us verify the program's correct operation.
- Test program: The one originally provided contained 3 pre-created tests:
    1. Test that only starts and stops one node.
    2. Test in which it is verified that the first leader is chosen.
    3. Test in which, after a system failure, a new leader is chosen.

    We have also implemented a fourth test:

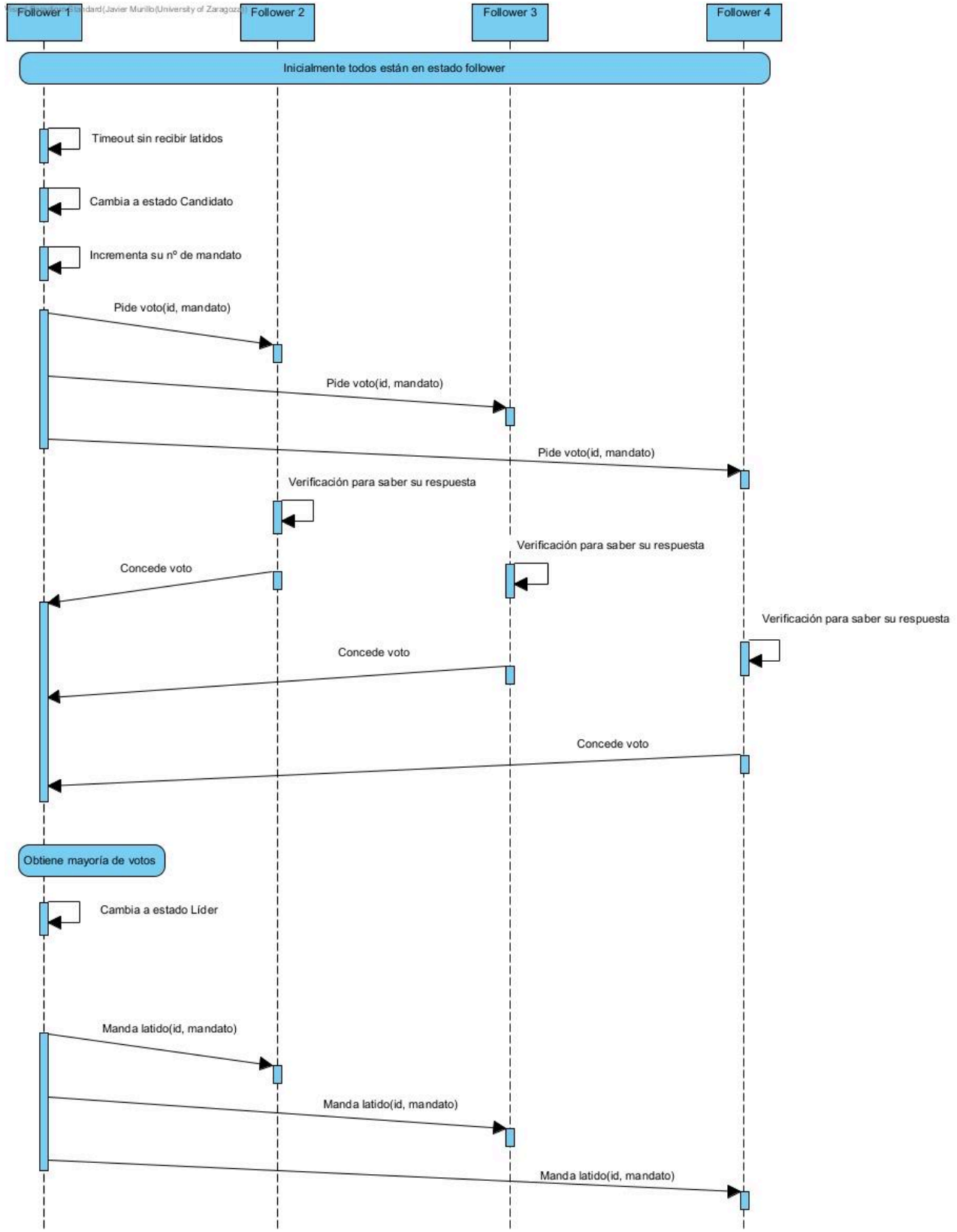
    4. Stability test of three committed operations in our system.

    The design of the latter is basically an identification of the leader, the creation of a client using the cltraft.go package, the submission of three operations, and the detection of errors when committing each of them.

    All 4 tests came back positiveIt is important to keep in mind thatTo verify the first test, due to the nature of our solution, which automatically initializes voting as soon as the node is created, we have created a constant called AutoElectionsOnStart, which must be set to false on all raft servers to test this test. For the other three tests (2-4), the constant must be set to true.

Algorithm design:

Leader Selection Sequence Diagram:



Operations Replication Sequence Diagram:

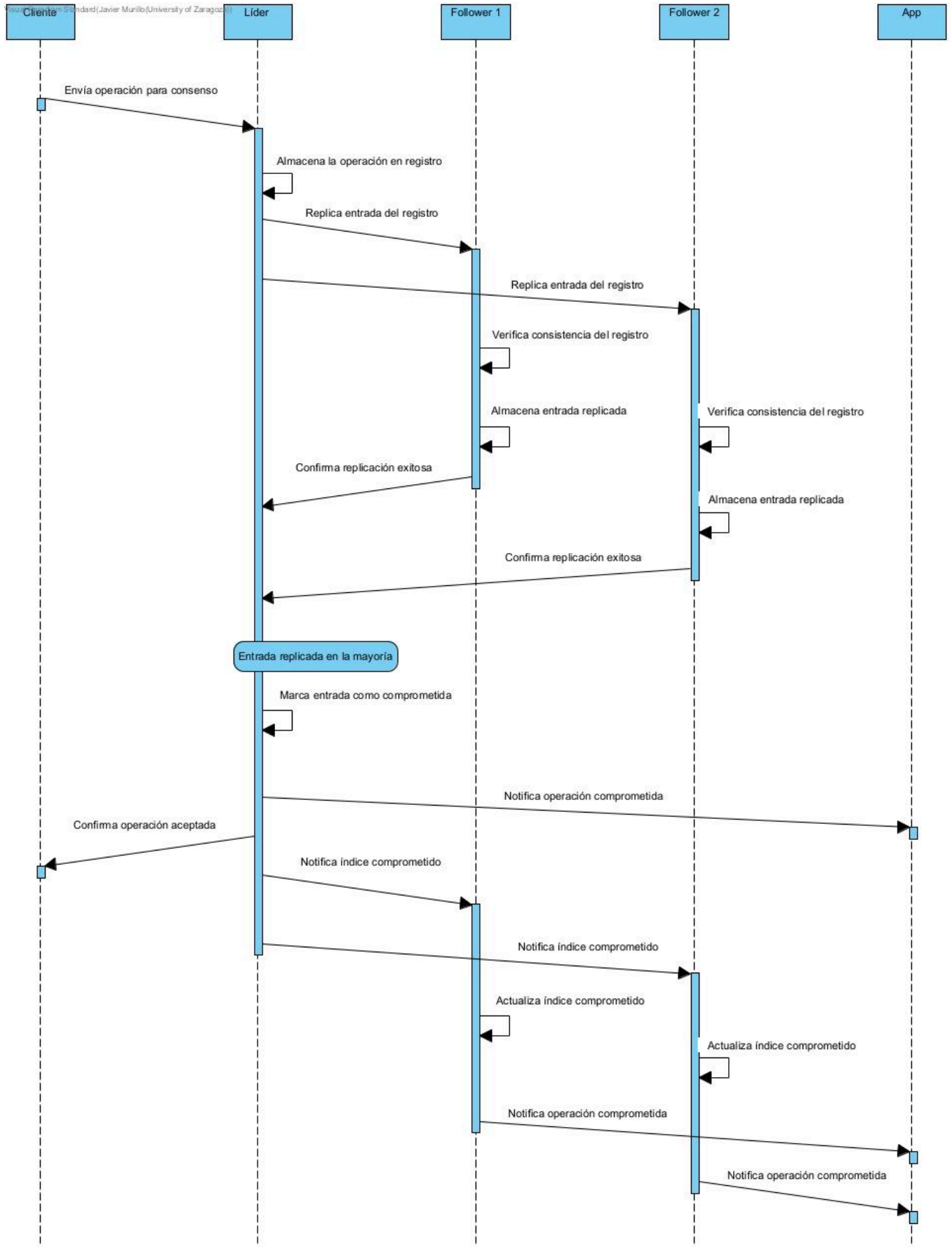
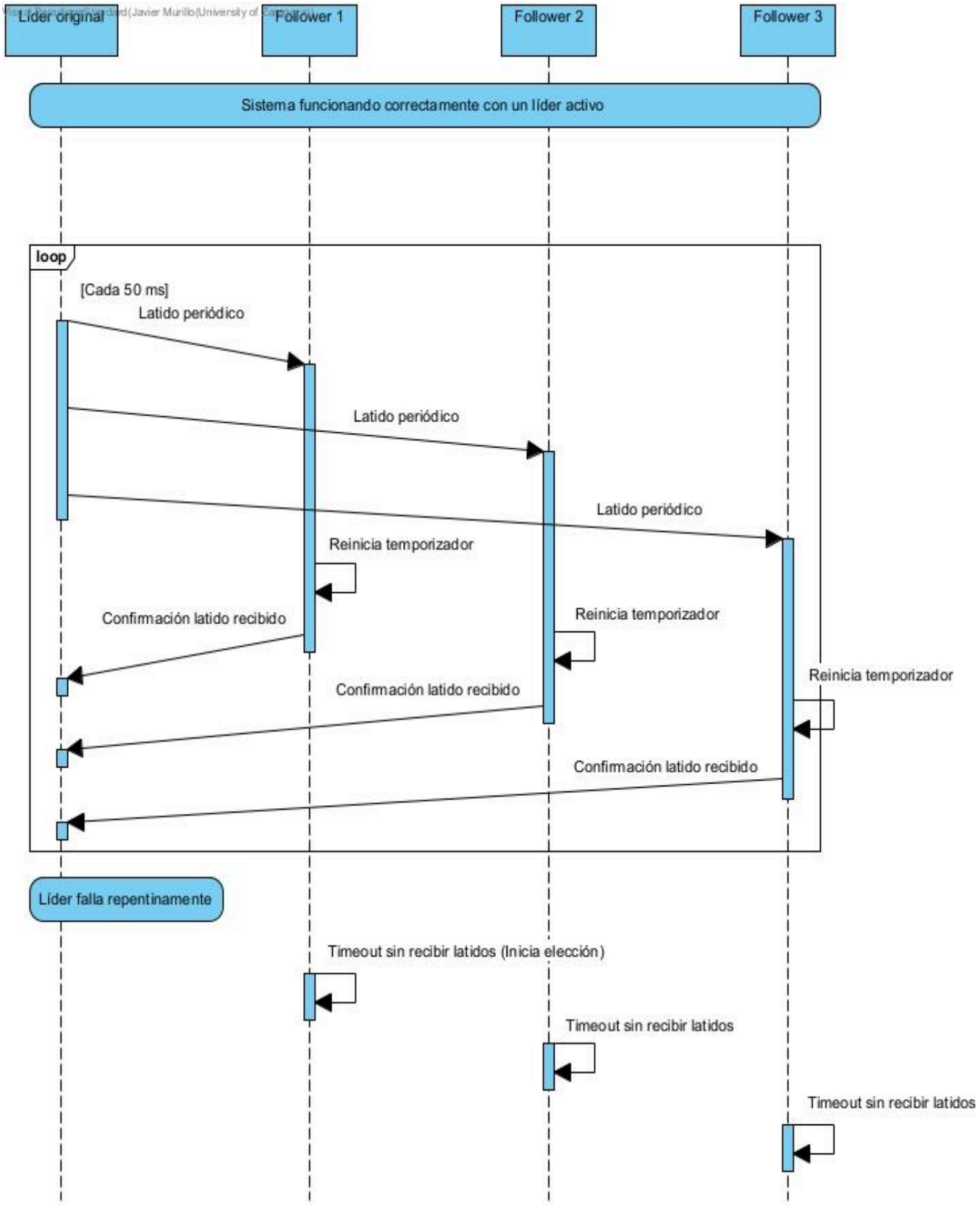


Diagram of periodic heartbeat sequence and fault detection:



Simultaneous Election Sequence Diagram: