Practical Assignment 4 - Distributed Systems

Raft Part 2

Authors:

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

Delivery date: 12/11/2025

Group: Pair: 1-6

Index

Introduction:

In this exercise, we completed the implementation of the Raft algorithm from the previous exercise, finalizing the leader election and operation replication functionality in a fault-tolerant distributed system. The main objective is to build a replicated key-value store service that maintains consistency through Raft consensus, functioning correctly in scenarios with node failures.

Design of our solution:

Our implementation follows the original Raft specification with the three main states:

- Follower: Responds to RPCs and resets its timeout upon hearing a heartbeat.
- Candidate: Begins elections by soliciting votes.
- Leader: Manages replication of inputs and sends heartbeats.

Regarding the implementation of the previous practice, the method for electing the leader has been completed, including the restriction on term and index number for selecting the best leader. This has been achieved by including the index and term information from their last entry when a replica requests a vote from the others.

Additionally, a Follower will deny its vote if its mandate is greater than the mandate it receives in the request. If the mandates are the same, its response will depend on which of the two has a higher index, thus choosing the node with the most up-to-date information as the leader.

Sequence diagrams:

As sequence diagrams, for this practice we have designed the different scenarios proposed as tests.

The first diagram (sequence image) refers to the consensus of entries with one failed replica, and the second refers to the consensus of entries with two failed replicas. To view the sequence diagrams, see Appendix A.

State machines:

The first state machine designed represents the internal behavior of a Raft node in the consensus algorithm. Its purpose is to manage the roles and transitions between states (Follower, Candidate, and Leader) to coordinate the leader election, replicate inputs, and maintain the coherence of the distributed system, even in the presence of failures.

The second state machine models the operation of the application that uses Raft; that is, the replicated state machine that applies committed operations to the key-value store in memory. Its purpose is to represent how received operations are consistently applied after consensus, ensuring that changes to the local state are made only when they are committed. To view both state machines, see Appendix B.

Evidence

In addition to the tests proposed in this exercise, it has also been verified that those proposed in the previous exercise continue to function correctly.

Test 1: Agreement with a disconnected replica

The objective of this test is to verify that the system reaches consensus with two of the three operational nodes. Since a majority is still present, the entries are committed even if one of the nodes is offline. The leader replicates the entries to the follower that is still online and updates the commitIndex when the majority replicates the entry.

Test 2: No agreement with two disconnected replicas

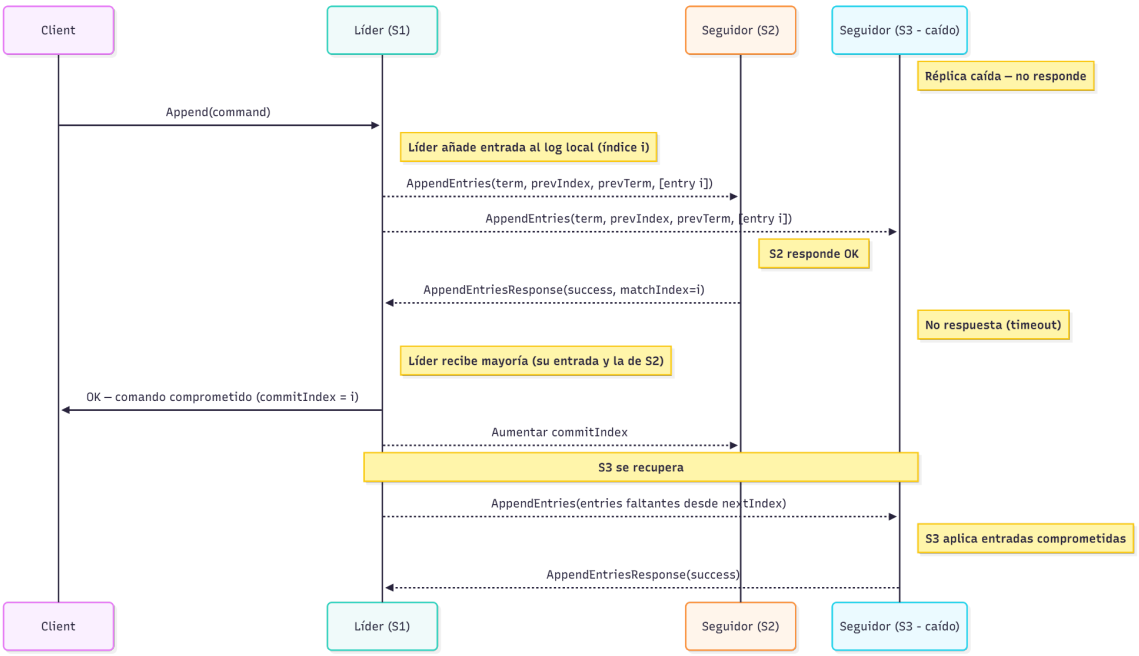
The purpose of this test is to see what happens when two of the three initial nodes disconnect. In this case, trades are not committed because a majority is not reached, and therefore the client receives a timeout or error when attempting to submit trades until a majority is re-established.

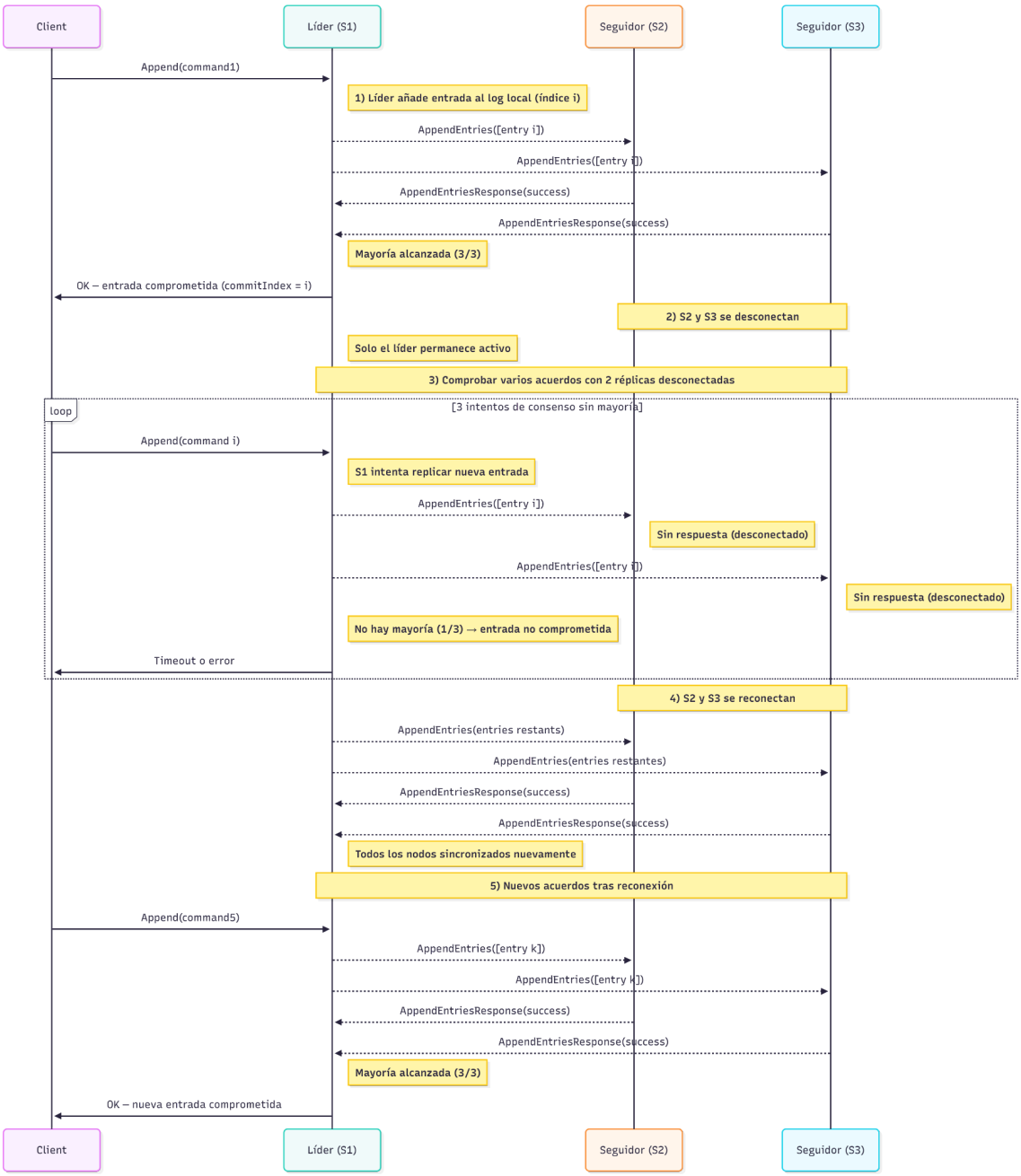
Test 3: Concurrent Operations

The objective of this test is to validate the behavior of our algorithm with multiple concurrent clients. The five proposed concurrent operations are processed correctly, advancing the registration indices consistently across all replicas.

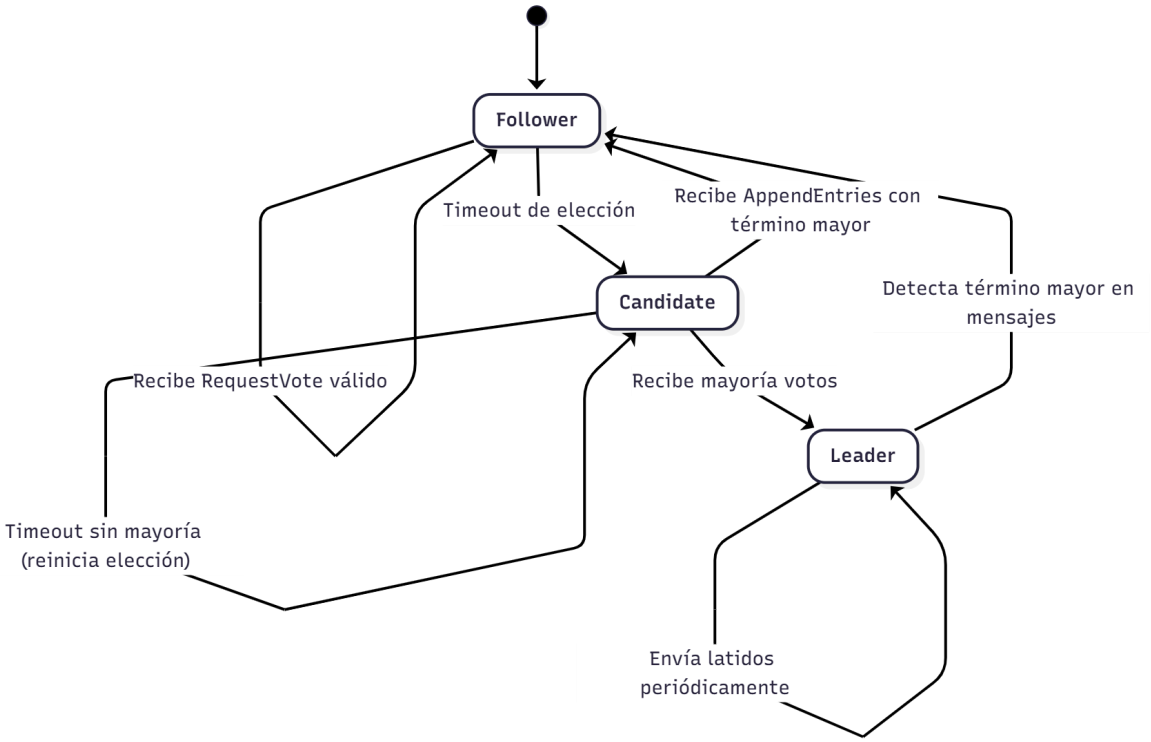
Annex

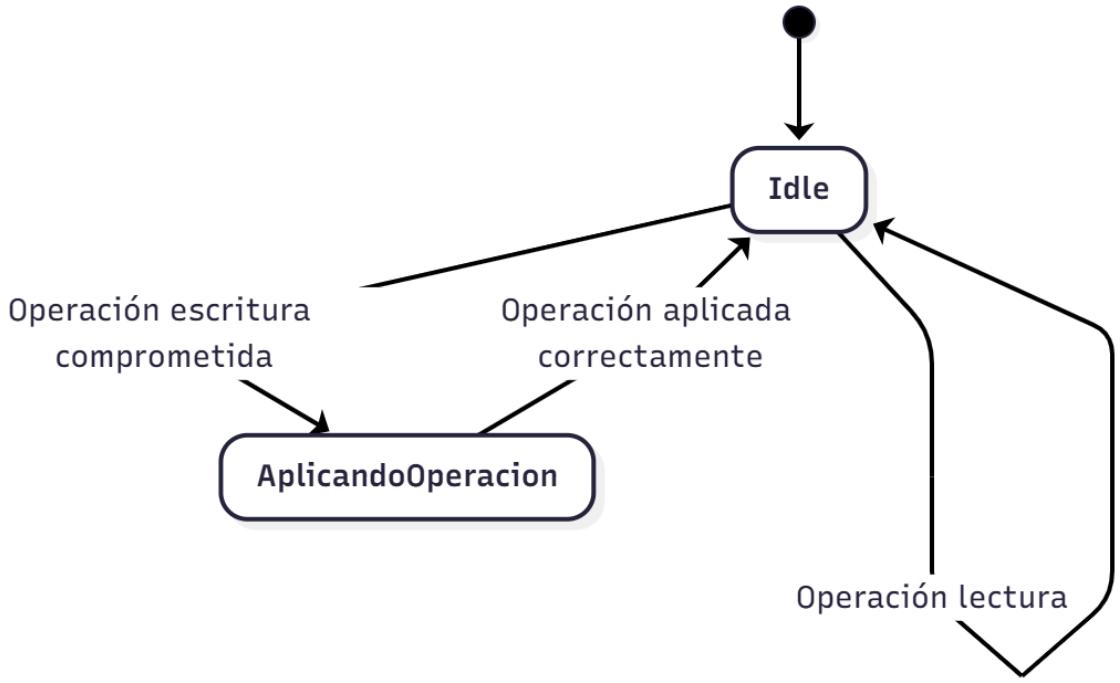
A. Sequence diagrams




Client
Líder (S1)
Seguidor (S2)
Seguidor (S3)
Append(command1)
1) Líder añade entrada al log local (índice i)
AppendEntries([entry i])
AppendEntries([entry i])
AppendEntriesResponse(success)
AppendEntriesResponse(success)
Mayoría alcanzada (3/3)
OK — entrada comprometida (commitIndex = i)
2) S2 y S3 se desconectan
Solo el líder permanece activo
3) Comprobar varios acuerdos con 2 réplicas desconectadas
loop
[3 intentos de consenso sin mayoría]
Append(command i)
S1 intenta replicar nueva entrada
AppendEntries([entry i])
Sin respuesta (desconectado)
AppendEntries([entry i])
Sin respuesta (desconectado)
No hay mayoría (1/3) → entrada no comprometida
Timeout o error
4) S2 y S3 se reconectan
AppendEntries(entries restants)
AppendEntries(entries restantes)
AppendEntriesResponse(success)
AppendEntriesResponse(success)
Todos los nodos sincronizados nuevamente
5) Nuevos acuerdos tras reconexión
Append(command5)
AppendEntries([entry k])
AppendEntries([entry k])
AppendEntriesResponse(success)
AppendEntriesResponse(success)
Mayoría alcanzada (3/3)
OK — nueva entrada comprometida
Client
Líder (S1)
Seguidor (S2)
Seguidor (S3)

B. State machines





C. Code corresponding to the tests performed

a. Test 1

```Go
//Agreement reached despite follower disconnections -- 3 RAFT NODES
func (cfg *configDespliegue) AgreementDespiteFollower(t *testing.T) {
	//t.Skip("SKIPPED AgreementDespiteFollower")

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

	cfg.startDistributedProcesses()

	// Get a leader
	fmt.Printf("Obtaining initial leader\n")
	leader := cfg.pruebaUnLider(3)
	fmt.Printf("Initial leader: %d\n", leader)

	// Commit an entry
	opInicial := raft.TipoOperacion{
		Operation: "write",
		Clue:     "initial",
		Value:     "test",
	}

	fmt.Printf("Compromising initial entry\n")
	Initial index, mandatoInicial, esLider, idLider, _ :=
		cfg.suberOperacionRemoto(lider, opInicial)

	if !esLider || idLider != leader {
		t.Fatalf("Error in the initial operation")
	}

	fmt.Printf("Committed initial entry: index=%d, command=%d\n",
		Initial index, mandatoInicial)

	// Obtain a leader and then disconnect one of the Raft nodes
	Disconnected follower := (leader + 1) % 3
	fmt.Printf("Disconnecting follower: %d\n", (Disconnected follower)
	cfg.disconnectNode(disconnectedFollower)

	// Check multiple agreements with a disconnected replica
	opsConFallo := []raft.TipoOperacion{
		{Operation: "write", Clue: "k1", Value: "v1"},
		{Operation: "write", Clue: "k2", Value: "v2"},
		{Operation: "write", Clue: "k3", Value: "v3"},
	}

	agreements := 0
```

```
	for i, on := range opsConFallo {
		index, mandate, esLider, idLider, _ :=
			cfg.suberOperacionRemoto(lider, on)

		if esLider && idLider == leader {
			agreements++
			fmt.Printf("Agreement %d reached: index=%d,
mandate=%d\n",
				i+1, index, mandate)
		} else {
			fmt.Printf("Agreement %d failed\n", i+1)
		}

		time.Sleep(300 * time.Millisecond)
	}

	// Verify that agreements were reached
	if agreements < only(opsWithFail)-1 {
		t.Fatalf("Not enough agreements were reached: %d/%d",
			agreements, only(opsWithFail))
	}
	fmt.Printf("%d/%d agreements were reached with disconnected
replication\n",
		agreements, only(opsWithFail))

	// Reconnect previously disconnected Raft node and check various
agreements
	fmt.Printf("Reconnecting follower: %d\n", (Disconnected follower)
	cfg.reconnectNode(followerDisconnected)

	// Wait for the system to stabilize
	time.Sleep(2 * time.Second)


	// Verify that agreements can still be reached
	opsPostReconexion := []raft.TipoOperacion{
		{Operation: "write", Clue: "post_reconnection1", Value: "v1"},
		{Operation: "write", Clue: "post_reconnection2", Value: "v2"},
	}

	for i, on := range opsPostReconexion {
		index, mandate, esLider, idLider, _ :=
			cfg.suberOperacionRemoto(lider, on)

		if !esLider || idLider != leader {
			t.Fatalf("Post-reconnection agreement error %d", i+1)
		}
```

```
			fmt.Printf("Post-reconnection agreement %d: index=%d,
mandate=%d\n",
				i+1, index, mandate)

			time.Sleep(200 * time.Millisecond)
		}

		cfg.stopDistributedProcesses()
		fmt.Printf("All post-reconnection agreements successfully completed")
		fmt.Println(".............", t.Name(), "Overcome")
}
```

b. Test 2

Go

```
//Test 2
//No agreement reached as most followers disconnect -- 3 RAFT NODES
func (cfg *configDespliegue) SinAcuerdoPorFallos(t *testing.T) {
		//t.Skip("SKIPPED No Agreement Due to Failures")
		fmt.Println(t.Name(), ".....................")
		cfg.startDistributedProcesses()

		// Commit an entry
		leader := cfg.pruebaUnLider(3)
		fmt.Printf("Initial leader: %d\n", leader)

		opInicial := raft.TipoOperacion{
			Operation: "write",
			Clue:     "initial_entry",
			Value:     "initial_value",
		}

		Initial index, mandatoInicial, esLider, idLider, _ :=
			cfg.suberOperacionRemoto(lider, opInicial)

		if !esLider || idLider != leader {
			t.Fatalf("Error compromising initial input")
		}
		fmt.Printf("Initial entry committed: index=%d, term=%d\n",
			Initial index, mandatoInicial)


		// Obtain a leader, then disconnect 2 of the Raft nodes
		disconnected replicas := []int{(leader + 1) % 3, (leader + 2) % 3}
		fmt.Printf("Disconnecting 2 replicas: %v\n", (Disconnected replicas)
```

```
	for _, replica := range disconnected replicas {
		cfg.disconnectNode(replica)
	}

	// Check multiple agreements with 2 disconnected replicas
	opsIntento := []raft.TipoOperacion{
		{Operation: "write", Clue: "intento1", Value: "fallo1"},
		{Operation: "write", Clue: "intento2", Value: "fallo2"},
		{Operation: "write", Clue: "attempt 3", Value: "fallo3"},
	}

	agreements := 0
	for i, on := range opsIntento {
		index, _, esLider, idLider, _ :=
			cfg.suberOperacionRemoto(lider, on)

		// Without a majority, it shouldn't be compromised
		if esLider && idLider == leader && index > Initial index {
			agreements++
			fmt.Printf("Unexpected %d Agreement: index=%d\n", i+1,
index)
		} else {
			fmt.Printf("Agreement %d NOT reached\n", i+1)
		}

		time.Sleep(500 * time.Millisecond)
	}

	// Verify that no agreements were reached
	if agreements > 0 {
		t.Logf("%d agreements were reached without a majority",
agreements)
	} else {
		fmt.Printf("Correct behavior: 0 agreements without a majority")
	}

	// Reconnect the two disconnected Raft nodes and test various
agreements
	fmt.Printf("Reconnecting 2 replicas: %v\n", (Disconnected replicas)
	for _, replica := range disconnected replicas {
		cfg.reconnectNode(replica)
	}

	// Wait for stabilization
	time.Sleep(3 * time.Second)

	// Verify that the agreement is recovered
	nuevoLider := cfg.pruebaUnLider(3)
```

```
	fmt.Printf("New leader after reconnection: %d\n", nuevoLider)

	recovery := raft.TipoOperacion{
		Operation: "write",
		Clue:     "recovery",
		Value:     "ok",
	}

	indexRec, mandatoRec, esLider, idLider, _ :=
		cfg.suberOperacionRemoto(nuevoLider, (Recovery)


	if !esLider || idLider != nuevoLider {
		t.Fatalf("The agreement was not recovered after reconnection")
	}

	cfg.stopDistributedProcesses()

	fmt.Printf("Recovered agreement: index=%d, mandate=%d\n",
		indexRec, mandatoRec)
	fmt.Println(".............", t.Name(), "Overcome")
}
```

c. Test 3

```Go
// 5 operations are submitted concurrently -- 3 RAFT NODES
func (cfg *configDespliegue) SubmitConcurrently Operations(t *testing.T) {
	//t.Skip("SKIPPED SubmitConcurrentlyOperations")
	fmt.Println(t.Name(), ".....................")
	cfg.startDistributedProcesses()

	// To stabilize the execution
	time.Sleep(2 * time.Second)

	// Obtain a leader and then submit a transaction
	leader := cfg.pruebaUnLider(3)
	fmt.Printf("Stable leader: %d\n", leader)

	opInicial := raft.TipoOperacion{
		Operation: "write",
		Clue:     "concurrent_start",
		Value:     "reference",
	}

	Initial index, _, esLider, _, _ :=
```

```
		cfg.suberOperacionRemoto(lider, opInicial)

	if !esLider {
		t.Fatalf("Initial operation error")
	}

	fmt.Printf("Initial operation: index=%d\n", Initial index)

	// Submit 5 concurrent operations
	Concurrent operations := []raft.TipoOperacion{
		{Operation: "write", Clue: "c1", Value: "v1"},
		{Operation: "write", Clue: "c2", Value: "v2"},
		{Operation: "write", Clue: "c3", Value: "v3"},
		{Operation: "write", Clue: "c4", Value: "v4"},
		{Operation: "write", Clue: "c5", Value: "v5"},
	}

	// Definition of the structure for the result
	type resultOperation struct {
		idx    int
		index int
		err    error
	}

	// Channel for concurrent results
	results := make(chan resultOperation, only(opsConcurrent))

	// Launch concurrent operations
	for i, on := range Concurrent operations {
		go func(idx int, operation raft.TipoOperacion) {
			index, _, _, _, err :=
				cfg.suberOperacionRemoto(lider, operation)
			results <- resultOperation{
				idx:    idx,
				index: index,
				err:    err,
			}
		}(i, on)
	}

	// Collect results
	indices := make([]int, only(opsConcurrent))
	for i := 0; i < only(opsConcurrent); i++ {
		select {
		case res := <-results:
			if res.err != nil {
				t.Fatalf("Error in operation %d: %v", res.idx,
res.err)
```

```
                        }
                        indices[res.idx] = res.index
                        fmt.Printf("Operation %d → index %d\n", res.idx,
res.index)
                case <-time.After(10 * time.Second):
                        t.Fatalf("Timeout in concurrent operations")
                }
        }

        // Check Raft node states (command and index of each one)
        // they should be identical to each other
        fmt.Println("Checking consistency between replicates...")
        time.Sleep(1 * time.Second)

        for i := 0; i < cfg.numReplicas; i++ {
                idNode, mandate, _, idLider := cfg.obtenerEstadoRemoto(i)

                // Verify that everyone knows the same leader
                if idLider != leader {
                        t.Fatalf("Replica %d has an inconsistent leader",
idNode)
                }

                fmt.Printf("Replica %d: mandate=%d, leader=%d\n",
                        idNode, mandate, idLider)
        }

        // Check index progress
        maxIndice := indices[0]
        for _, idx := range indices {
                if idx > maxIndice {
                        maxIndice = idx
                }
        }

        fmt.Printf("Index advanced from %d to %d\n",
                Initial index, maxIndice)
        fmt.Printf("5 operations competitors completed")
        fmt.Println("............", t.Name(), "Overcome")
}
```