Entrega Práctica 4 - Sistemas distribuidos

Raft 2ª parte

Autores:

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

Fecha de entrega: 12/11/2025

Grupo: Pareja:1-6

Índice

Introducción:

En esta práctica hemos completado la implementación del algoritmo de Raft de la práctica anterior, completando la funcionalidad de elección de líder y replicación de operaciones en un sistema distribuido tolerante a fallos. El objetivo principal es construir un servicio de almacenamiento clave-valor replicado que mantenga consistencia a través del consenso Raft, funcionando correctamente en escenarios con fallos de nodos

Diseño de nuestra solución:

Nuestra implementación sigue la especificación original de Raft con los tres estados principales:

- Seguidor: Responde a RPCs y reinicia su timeout ante latidos.
- Candidato: Inicia elecciones solicitando votos.
- Líder: Gestiona replicación de entradas y envía latidos.

Con respecto a la implementación de la práctica anterior se ha completado el método de elección de líder, incluyendo la restricción acerca del número de mandato y número de índice para seleccionar al mejor líder. Esto se ha conseguido incluyendo información del índice y mandato de su última entrada cuando una réplica pide voto a las otras.

Adicionalmente, un Follower denegará su voto si su mandato es mayor al mandato que recibe en la petición. En el caso de que el mandato sea el mismo, su respuesta dependerá de cuál de los dos tiene un mayor índice, eligiendo así como líder al nodo con la información más actualizada.

Diagramas de secuencia:

Como diagramas de secuencia, para esta práctica hemos diseñado los distintos escenarios propuestos como pruebas.

El primer diagrama(Imagen de secuencia hace referencia al consenso de entradas con una réplica caída y el segundo hace referencia al consenso de entradas con dos réplica caídas. Para poder ver los diagramas de secuencia, consultar Anexo A.

Máquinas de estados:

La primera máquina de estados diseñada representa el comportamiento interno de un nodo Raft en el algoritmo de consenso. Su propósito es gestionar los roles y transiciones entre estados (Follower, Candidate y Leader) para coordinar la elección de líder, replicar entradas y mantener la coherencia del sistema distribuido, incluso ante la presencia de fallos.

La segunda máquina de estados modela el funcionamiento de la aplicación que utiliza Raft, es decir, la máquina de estados replicada que aplica las operaciones comprometidas al almacenamiento clave-valor en memoria. Su propósito es representar cómo las operaciones recibidas se aplican de forma consistente tras el consenso, asegurando que los cambios al estado local se hacen solo cuando están confirmados (comprometidos). Para poder ver ambas máquinas de estado, consultar Anexo B.

Pruebas

Además de las pruebas propuestas en esta práctica, también se ha comprobado que las propuestas en la práctica anterior sigan funcionando correctamente.

Prueba 1: Acuerdo con una réplica desconectada

El objetivo de esta prueba consiste en verificar que el sistema alcanza consenso con dos de tres nodos operativos. Como sigue habiendo mayoría, las entradas se comprometen igualmente aunque uno de los nodos esté desconectado. El líder replica las entradas en el seguidor que aún está conectado y actualiza el “commitIndex” cuando la mayoría replica la entrada.

Prueba 2: Sin acuerdo con dos réplicas desconectadas

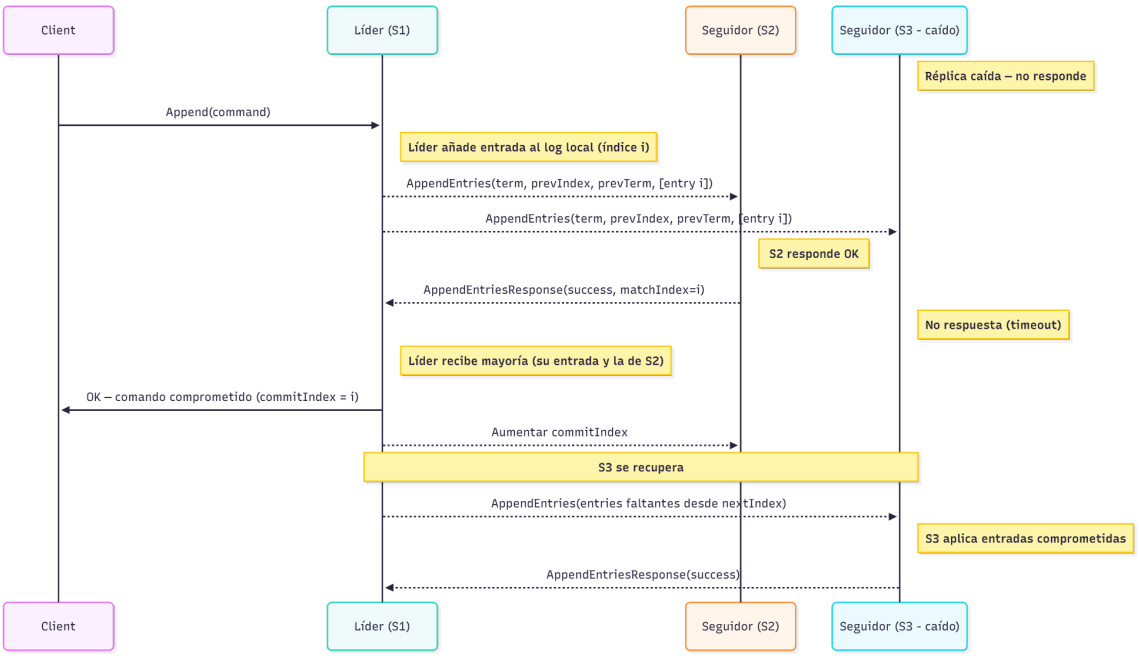
El objetivo de esta prueba es comprobar qué ocurre cuando se desconectan dos de los tres nodos iniciales. En este caso, las operaciones no se comprometen ya que no se alcanza mayoría y, por lo tanto, el cliente recibe un timeout o error al intentar someter operaciones hasta que vuelva a haber mayoría.

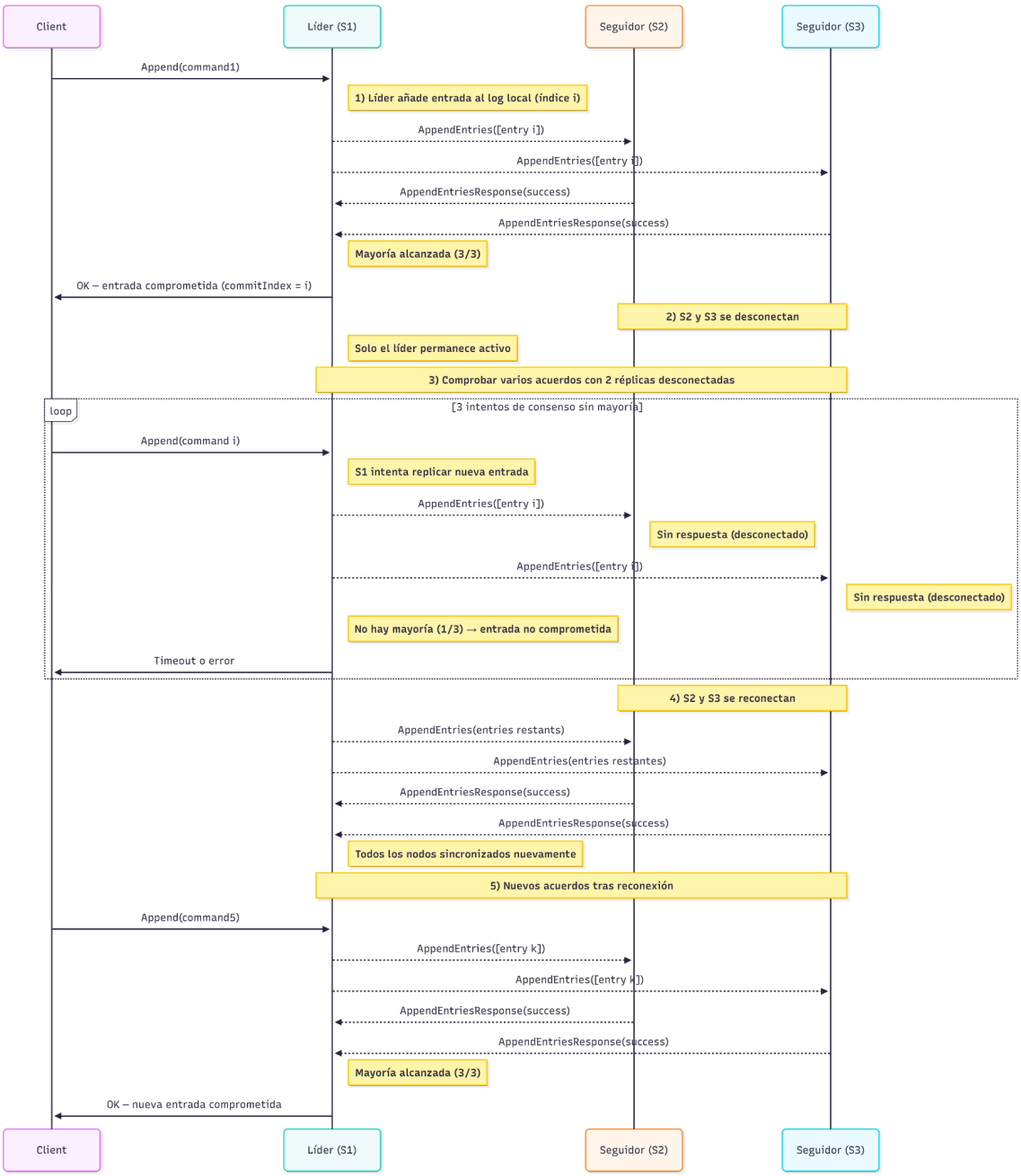
Prueba 3: Operaciones concurrentes

El objetivo de esta prueba es validar el comportamiento de nuestro algoritmo con múltiples clientes concurrentes. Las cinco operaciones concurrentes propuestas se procesan correctamente avanzando los índices de registro de forma consistente en todas las réplicas.

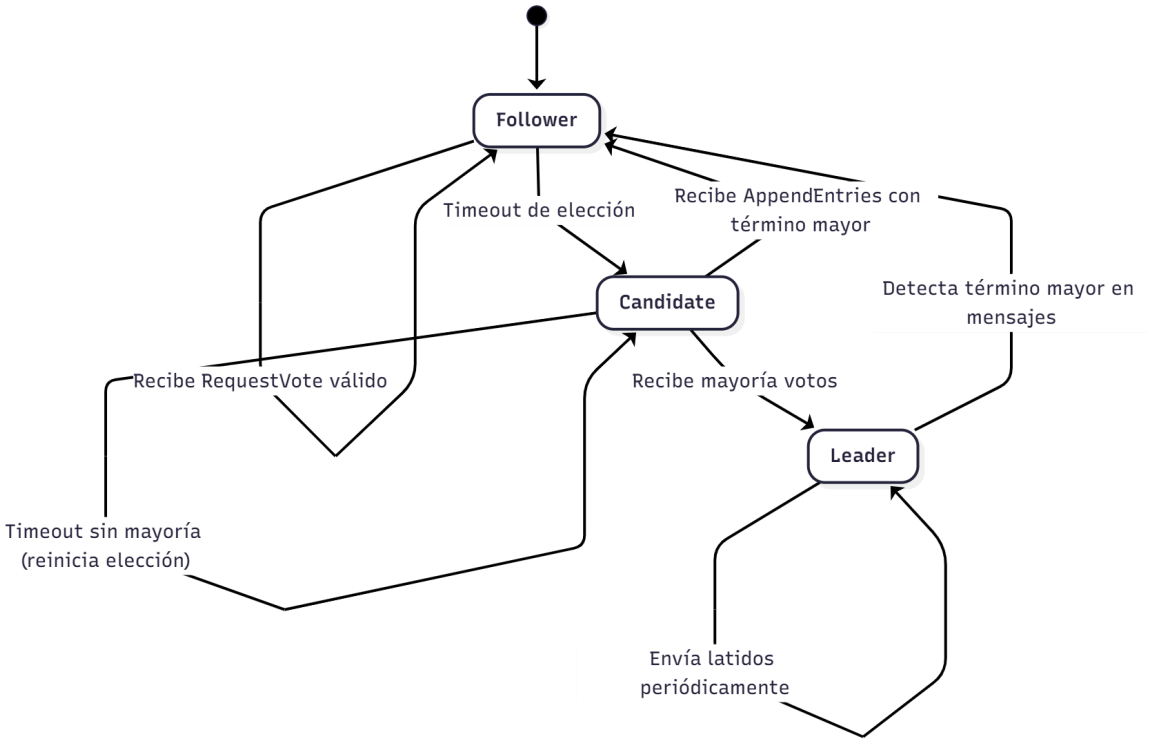
Anexo

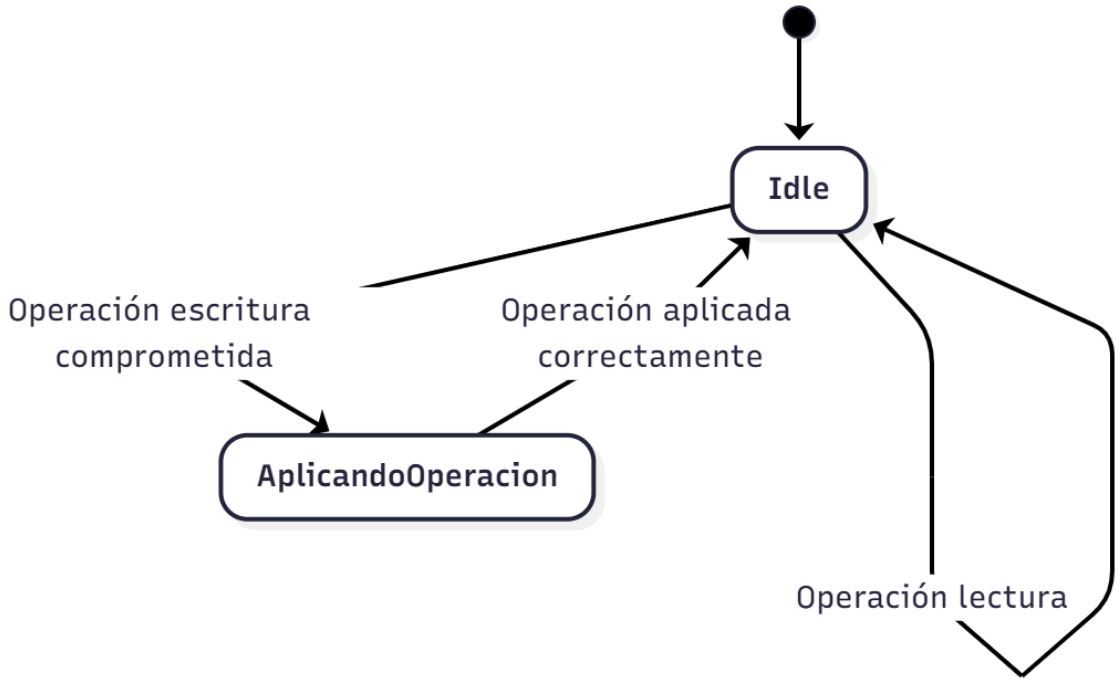
A. Diagramas de secuencia




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. Máquinas de estado





C. Código correspondiente a las pruebas realizadas

a. Prueba 1

Go

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

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

	cfg.startDistributedProcesses()

	// Obtener un lider
	fmt.Printf("Obteniendo lider inicial\n")
	lider := cfg.pruebaUnLider(3)
	fmt.Printf("Lider inicial: %d\n", lider)

	// Comprometer una entrada
	opInicial := raft.TipoOperacion{
		Operacion: "escribir",
		Clave:     "inicial",
		Valor:     "test",
	}

	fmt.Printf("Comprometiendo entrada inicial\n")
	indiceInicial, mandatoInicial, esLider, idLider, _ :=
		cfg.someterOperacionRemoto(lider, opInicial)

	if !esLider || idLider != lider {
		t.Fatalf("Error en la operación inicial")
	}

	fmt.Printf("Entrada inicial comprometida: indice=%d, mandato=%d\n",
		indiceInicial, mandatoInicial)

	// Obtener un lider y, a continuación desconectar una de los nodos
Raft
	seguidorDesconectado := (lider + 1) % 3
	fmt.Printf("Desconectando seguidor: %d\n", seguidorDesconectado)
	cfg.desconectarNodo(seguidorDesconectado)

	// Comprobar varios acuerdos con una réplica desconectada
	opsConFallo := []raft.TipoOperacion{
		{Operacion: "escribir", Clave: "k1", Valor: "v1"},
		{Operacion: "escribir", Clave: "k2", Valor: "v2"},
		{Operacion: "escribir", Clave: "k3", Valor: "v3"},
	}
```

```
	acuerdos := 0
	for i, op := range opsConFallo {
		indice, mandato, esLider, idLider, _ :=
			cfg.someterOperacionRemoto(lider, op)

		if esLider && idLider == lider {
			acuerdos++
			fmt.Printf("Acuerdo %d conseguido: índice=%d,
mandato=%d\n",
				i+1, indice, mandato)
		} else {
			fmt.Printf("Acuerdo %d falló\n", i+1)
		}

		time.Sleep(300 * time.Millisecond)
	}

	// Verificar que se consiguieron acuerdos
	if acuerdos < len(opsConFallo)-1 {
		t.Fatalf("No se consiguieron suficientes acuerdos: %d/%d",
			acuerdos, len(opsConFallo))
	}
	fmt.Printf("Se consiguieron %d/%d acuerdos con réplica
desconectada\n",
		acuerdos, len(opsConFallo))

	// reconectar nodo Raft previamente desconectado y comprobar varios
acuerdos
	fmt.Printf("Reconectando seguidor: %d\n", seguidorDesconectado)
	cfg.reconectarNodo(seguidorDesconectado)

	// Esperar a que el sistema se estabilice
	time.Sleep(2 * time.Second)


	// Comprobar que se pueden seguir alcanzando acuerdos
	opsPostReconexion := []raft.TipoOperacion{
		{Operacion: "escribir", Clave: "post_reconexion1", Valor:
"v1"},
		{Operacion: "escribir", Clave: "post_reconexion2", Valor:
"v2"},
	}

	for i, op := range opsPostReconexion {
		indice, mandato, esLider, idLider, _ :=
			cfg.someterOperacionRemoto(lider, op)

		if !esLider || idLider != lider {
```

```
                        t.Fatalf("Error en acuerdo post-reconexión %d", i+1)
                }
                fmt.Printf("Acuerdo post-reconexión %d: índice=%d,
mandato=%d\n",
                        i+1, indice, mandato)

                time.Sleep(200 * time.Millisecond)
        }

        cfg.stopDistributedProcesses()
        fmt.Printf("Todos los acuerdos post-reconexión completados
correctamente\n")
        fmt.Println(".............", t.Name(), "Superado")
}
```

b. Prueba 2

Go

```
//Pueba 2
// NO se consigue acuerdo al desconectarse mayoría de seguidores -- 3 NODOS
RAFT
func (cfg *configDespliegue) SinAcuerdoPorFallos(t *testing.T) {
        //t.Skip("SKIPPED SinAcuerdoPorFallos")
        fmt.Println(t.Name(), ".....................")
        cfg.startDistributedProcesses()

        // Comprometer una entrada
        lider := cfg.pruebaUnLider(3)
        fmt.Printf("Líder inicial: %d\n", lider)

        opInicial := raft.TipoOperacion{
                Operacion: "escribir",
                Clave:     "entrada_inicial",
                Valor:     "valor_inicial",
        }

        indiceInicial, mandatoInicial, esLider, idLider, _ :=
                cfg.someterOperacionRemoto(lider, opInicial)

        if !esLider || idLider != lider {
                t.Fatalf("Error comprometiendo entrada inicial")
        }
        fmt.Printf("Entrada inicial comprometida: índice=%d, mandato=%d\n",
                indiceInicial, mandatoInicial)
```

```
	//  Obtener un lider y, a continuación desconectar 2 de los nodos Raft
	replicasDesconectadas := []int{(lider + 1) % 3, (lider + 2) % 3}
	fmt.Printf("Desconectando 2 réplicas: %v\n", replicasDesconectadas)

	for _, replica := range replicasDesconectadas {
		cfg.desconectarNodo(replica)
	}

	// Comprobar varios acuerdos con 2 réplicas desconectada
	opsIntento := []raft.TipoOperacion{
		{Operacion: "escribir", Clave: "intento1", Valor: "fallo1"},
		{Operacion: "escribir", Clave: "intento2", Valor: "fallo2"},
		{Operacion: "escribir", Clave: "intento3", Valor: "fallo3"},
	}

	acuerdos := 0
	for i, op := range opsIntento {
		indice, _, esLider, idLider, _ :=
			cfg.someterOperacionRemoto(lider, op)

		// Sin mayoría, no debería comprometerse
		if esLider && idLider == lider && indice > indiceInicial {
			acuerdos++
			fmt.Printf("Acuerdo %d INESPERADO: índice=%d\n", i+1,
indice)
		} else {
			fmt.Printf("Acuerdo %d NO conseguido\n", i+1)
		}

		time.Sleep(500 * time.Millisecond)
	}

	// Verificar que no se consiguieron acuerdos
	if acuerdos > 0 {
		t.Logf("Se consiguieron %d acuerdos sin mayoría", acuerdos)
	} else {
		fmt.Printf("Comportamiento correcto: 0 acuerdos sin mayoría\n")
	}

	// reconectar los 2 nodos Raft desconectados y probar varios acuerdos
	fmt.Printf("Reconectando 2 réplicas: %v\n", replicasDesconectadas)
	for _, replica := range replicasDesconectadas {
		cfg.reconectarNodo(replica)
	}

	// Esperar reestabilización
	time.Sleep(3 * time.Second)
```

```
	// Verificar que se recupera el acuerdo
	nuevoLider := cfg.pruebaUnLider(3)
	fmt.Printf("Nuevo líder después de reconexión: %d\n", nuevoLider)

	opRecuperacion := raft.TipoOperacion{
		Operacion: "escribir",
		Clave:     "recuperacion",
		Valor:     "ok",
	}

	indiceRec, mandatoRec, esLider, idLider, _ :=
		cfg.someterOperacionRemoto(nuevoLider, opRecuperacion)


	if !esLider || idLider != nuevoLider {
		t.Fatalf("No se recuperó el acuerdo después de reconexión")
	}

	cfg.stopDistributedProcesses()

	fmt.Printf("Acuerdo recuperado: índice=%d, mandato=%d\n",
		indiceRec, mandatoRec)
	fmt.Println(".............", t.Name(), "Superado")
}
```

c. Prueba 3

```
Go
// Se someten 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()

	// Para estabilizar la ejecucion
	time.Sleep(2 * time.Second)

	// Obtener un lider y, a continuación, someter una operacion
	lider := cfg.pruebaUnLider(3)
	fmt.Printf("Líder estable: %d\n", lider)

	opInicial := raft.TipoOperacion{
		Operacion: "escribir",
		Clave:     "inicio_concurrente",
		Valor:     "referencia",
```

```
}

indiceInicial, _, esLider, _, _ :=
	cfg.someterOperacionRemoto(lider, opInicial)

if !esLider {
	t.Fatalf("Error en operación inicial")
}

fmt.Printf("Operación inicial: índice=%d\n", indiceInicial)

// Someter 5  operaciones concurrentes
opsConcurrentes := []raft.TipoOperacion{
	{Operacion: "escribir", Clave: "c1", Valor: "v1"},
	{Operacion: "escribir", Clave: "c2", Valor: "v2"},
	{Operacion: "escribir", Clave: "c3", Valor: "v3"},
	{Operacion: "escribir", Clave: "c4", Valor: "v4"},
	{Operacion: "escribir", Clave: "c5", Valor: "v5"},
}

// Definicion de la estructura para el resultado
type resultadoOperacion struct {
	idx    int
	indice int
	err    error
}

// Canal para resultados concurrentes
resultados := make(chan resultadoOperacion, len(opsConcurrentes))

// Lanzar operaciones concurrentes
for i, op := range opsConcurrentes {
	go func(idx int, operacion raft.TipoOperacion) {
		indice, _, _, _, err :=
			cfg.someterOperacionRemoto(lider, operacion)
		resultados <- resultadoOperacion{
			idx:    idx,
			indice: indice,
			err:    err,
		}
	}(i, op)
}

// Recoger resultados
indices := make([]int, len(opsConcurrentes))
for i := 0; i < len(opsConcurrentes); i++ {
	select {
	case res := <-resultados:
```

```
                        if res.err != nil {
                                t.Fatalf("Error en operación %d: %v", res.idx,
res.err)
                        }
                        indices[res.idx] = res.indice
                        fmt.Printf("Operación %d → índice %d\n", res.idx,
res.indice)
                case <-time.After(10 * time.Second):
                        t.Fatalf("Timeout en operaciones concurrentes")
                }
        }

        // Comprobar estados de nodos Raft (mandato e indice de cada uno
        // deberían ser identicos entre ellos
        fmt.Println("Comprobando consistencia entre réplicas...")
        time.Sleep(1 * time.Second)

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

                // Verificar que todas conocen al mismo líder
                if idLider != lider {
                        t.Fatalf("Réplica %d tiene líder inconsistente", idNodo)
                }

                fmt.Printf("Réplica %d: mandato=%d, líder=%d\n",
                        idNodo, mandato, idLider)
        }

        // Verificar avance del índice
        maxIndice := indices[0]
        for _, idx := range indices {
                if idx > maxIndice {
                        maxIndice = idx
                }
        }

        fmt.Printf("Índice avanzó de %d a %d\n",
                indiceInicial, maxIndice)
        fmt.Printf("5 operaciones concurrentes completadas\n")
        fmt.Println(".............", t.Name(), "Superado")
}
```