# Entrega Práctica 1 - Sistemas distribuidos

## Conceptos y mecanismos básicos

**Autores:**

Fernando Pastor Peralta (897113)

Javier Murillo Jimenez (897154)

**Fecha de entrega:** 25/09/2025

**Grupo:** Pareja:1-6

***Ingeniería Informática - Universidad de Zaragoza, Curso 2025-2026***

# Índice

# Descripción del problema:

El objetivo propuesto en la práctica es estudiar el comportamiento de distintos modelos de comunicación de red y de ejecución distribuida, utilizando máquinas de distinta arquitectura (Raspberry Pi para la prueba final y los PCs de sobremesa para pruebas intermedias). Para ello:

Podemos dividir la práctica en dos bloques claramente diferenciados:

1. Escenarios de red:
    - Cliente TCP que intenta conectarse a un servidor que no está en marcha (medimos el tiempo hasta recibir el error de la llamada *Dial*).
    - Cliente TCP que se conecta a un servidor ya activo (medimos el tiempo de establecimiento de la conexión).
    - Cliente y servidor UDP, donde se mide el tiempo de ida y vuelta en el envío y recepción de una letra.
    - Comparación de estos tres escenarios ejecutando cliente y servidor en la misma máquina y en máquinas distintas.
2. Arquitecturas concurrentes y distribuidas:
    - Algoritmo de barrera distribuida.
    - Servidor TCP concurrente con múltiples clientes utilizando goroutines.
    - Arquitectura máster-worker con comunicación por SSH para el cálculo de números primos en intervalos distribuidos.

Las pruebas se han realizado con los siguientes recursos computacionales:

- Raspberry Pi 4 Model B con procesador ARM Cortex-A72 con 4 cores y 8 GB de RAM.
- PCs de sobremesa del laboratorio 1.02 con arquitectura AMD64.
- Sistema operativo Linux
- Lenguaje de programación Go.

Con la ejecución de las diversas pruebas realizadas, se busca entender mejor el funcionamiento de los protocolos de comunicación, analizar la latencia en distintos entornos y estudiar cómo las decisiones de diseño en la arquitectura de software influyen en el rendimiento global de las aplicaciones distribuidas.

# Diseño de las arquitecturas:

## TCP:

| | Misma máquina (TCP) | | Distinta máquina (TCP) | |
|---|---|---|---|---|
| | Failed | Dial | Failed | Dial |
| 1º | 540.439 µs | 729.566 µs | 653.659 µs | 1.345355 ms |
| 2º | 764.381 µs | 1.201617 ms | 666.715 µs | 1.273671 ms |
| 3º | 835.232 µs | 908.083 µs | 639.789 µs | 1.202135 ms |
| 4º | 701.862 µs | 1.264597 ms | 679.844 µs | 1.093081 ms |
| 5º | 645.993 µs | 697.696 µs | 631.289 µs | 1.533205 ms |

La operación *Dial* en Go permite al cliente establecer una conexión TCP con el servidor dada su dirección. El tiempo de ejecución de esta operación depende de lo siguiente:

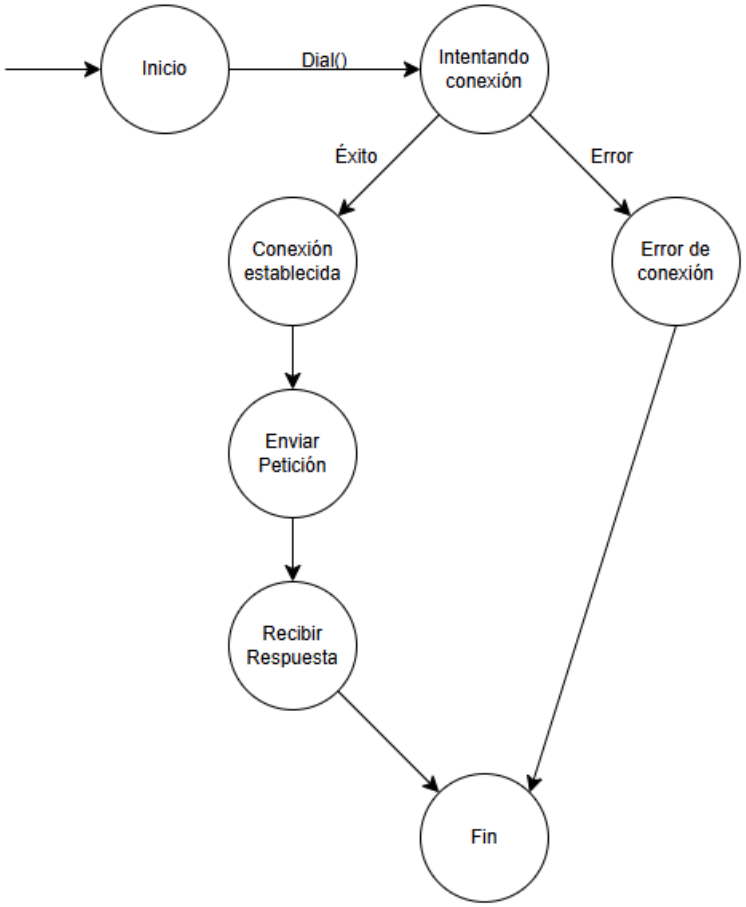
- La resolución de la dirección.
- La creación del socket.
- El intento de conexión: el cliente envía SYN, recibe SYN+ACK del servidor y responde con ACK.
- La entrega a la aplicación,ta

Depende del escenario, el coste de *Dial* puede variar:

- *Dial* falla: si el servidor no está en marcha o el puerto no está abierto, se intenta conectar pero recibe un *connection refused* y el tiempo es menor que cuando la llamada funciona correctamente.
- *Dial* funciona correctamente: el tiempo se limita al establecimiento de la conexión.
- Cliente y servidor en la misma máquina: al no haber transmisión física por la red, el tiempo de ejecución es mínimo.
- Cliente y servidor en distintas máquinas: la latencia aumenta debido a la transmisión por la red y depende del retardo de propagación.

En cambio, cuando *Dial* funciona correctamente el coste corresponde con el tiempo que le cuesta al cliente establecer la conexión con el servidor. Cuando ejecutamos esto en máquinas distintas, al tiempo que le cuesta de media en la misma máquina (~0.7-1.3 ms) habría que añadirle la latencia de propagación de la LAN (~0.3-0.5 ms).

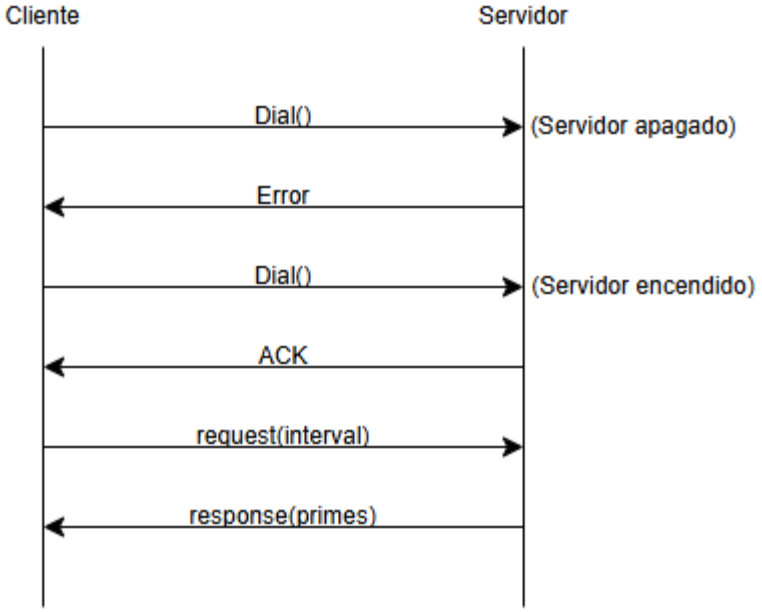
- **Máquina de estados:**




Explicación:

1. **Inicio:** el cliente se prepara para establecer la conexión con el servidor.
2. **Intentando conexión:** el cliente ejecuta *Dial()* para intentar conectarse al servidor.
3. **Conexión establecida:** *Dial()* devuelve un objeto Conn válido.
4. **Enviar Petición / Recibir Respuesta:** el cliente envía el intervalo y recibe los primos calculados por el servidor.
5. **Error de conexión:** *Dial()* devuelve un error porque el servidor aún no está en marcha.
6. **Fin:** el cliente termina su ejecución.

- **Diagrama de Secuencia:**




Explicación:

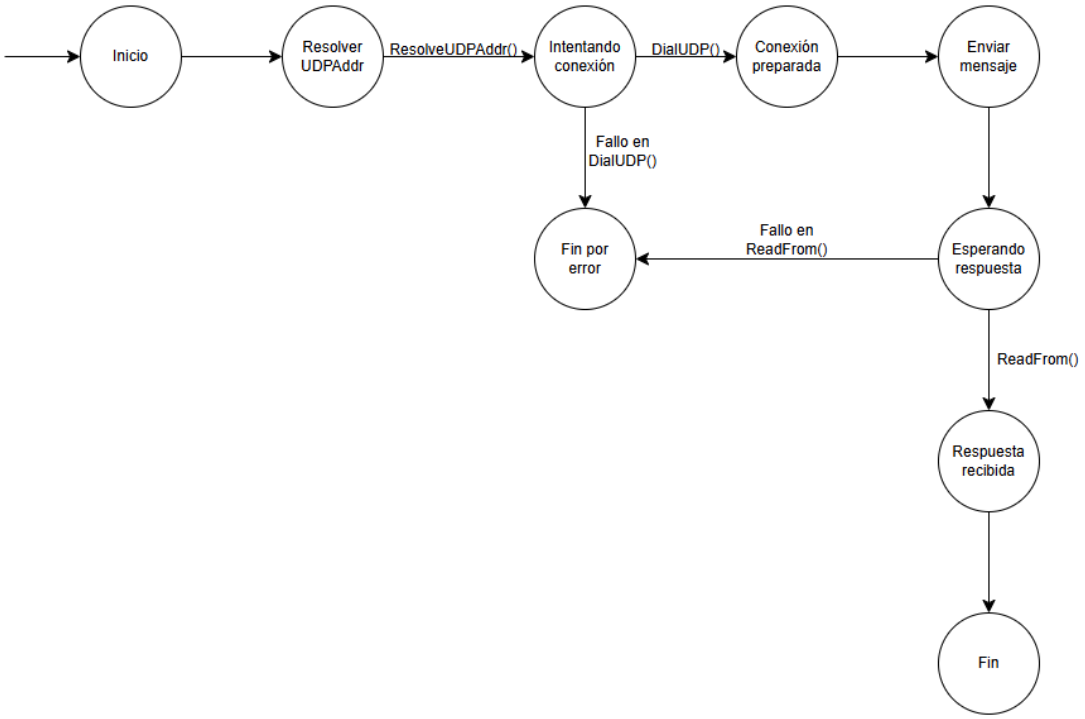
1. El cliente intenta conectarse con el servidor pero este aún no está en marcha por lo que *Dial()* devuelve un error.
2. Lanzamos el servidor y el cliente vuelve a llamar a *Dial()* y esta vez sí que se establece la conexión.
3. El cliente envía el intervalo de números.
4. El servidor calcula los primos en el intervalo y se los devuelve al cliente.
5. El cliente recibe el resultado y finaliza su ejecución mandando close() (nada más se envía desde ese momento).

# UDP:

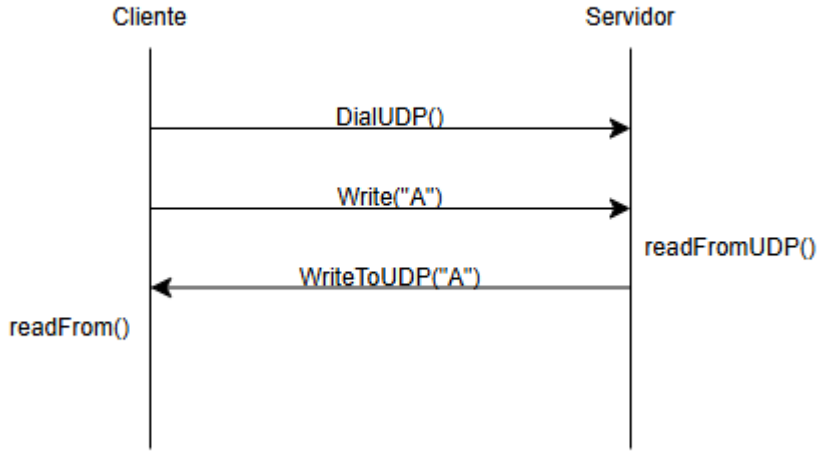
| | Misma máquina (UDP) | Distinta máquina (UDP) |
|---|---|---|
| 1º | 473.142 µs | 693.64 µs |
| 2º | 359.496 µs | 661.344 µs |
| 3º | 345.848 µs | 512.624 µs |
| 4º | 339.182 µs | 491.716 µs |
| 5º | 334.811 µs | 505.754 µs |

Con respecto al coste de ejecutar la transmisión en el escenario de UDP, esto se debe a la preparación y encapsulación del mensaje en el cliente, a la transmisión del datagrama, al procesamiento, al eco del servidor, y a la recepción de la respuesta de vuelta en el cliente.

- **Máquina de estados:**




- **Diagrama de Secuencia:**

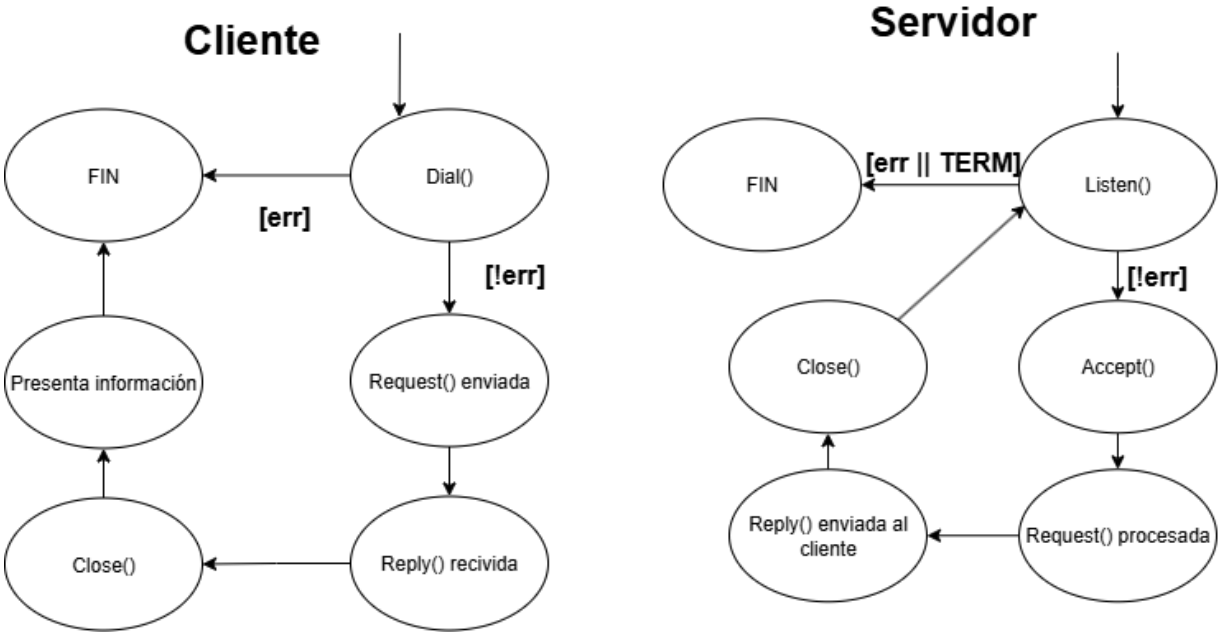



Explicación:

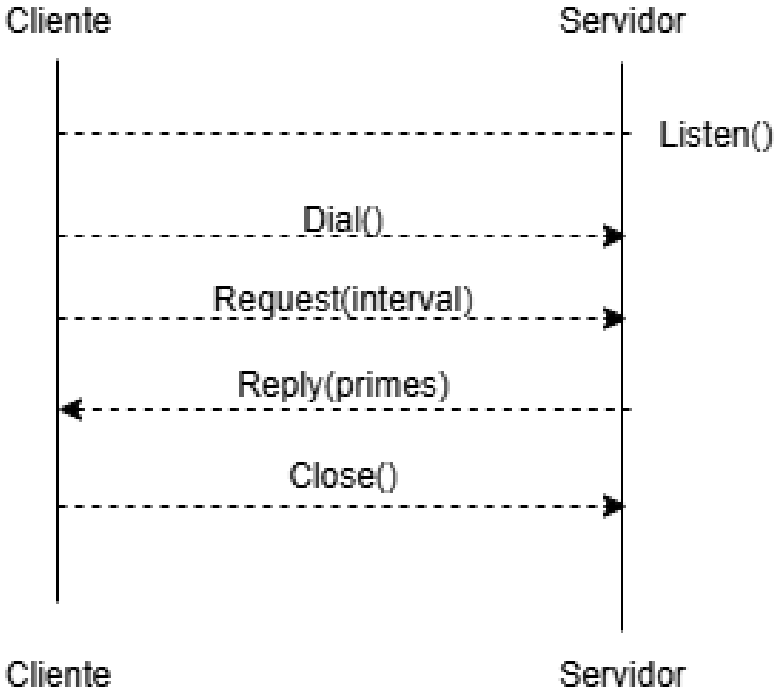
1. El cliente prepara el UDP con DialUDP().
2. Envía una letra al servidor con Write().
3. El servidor recibe el mensaje y se lo reenvía al cliente.
4. El cliente recibe la respuesta y finaliza la ejecución.

# Cliente servidor secuencial:

- **Máquina de estados:**

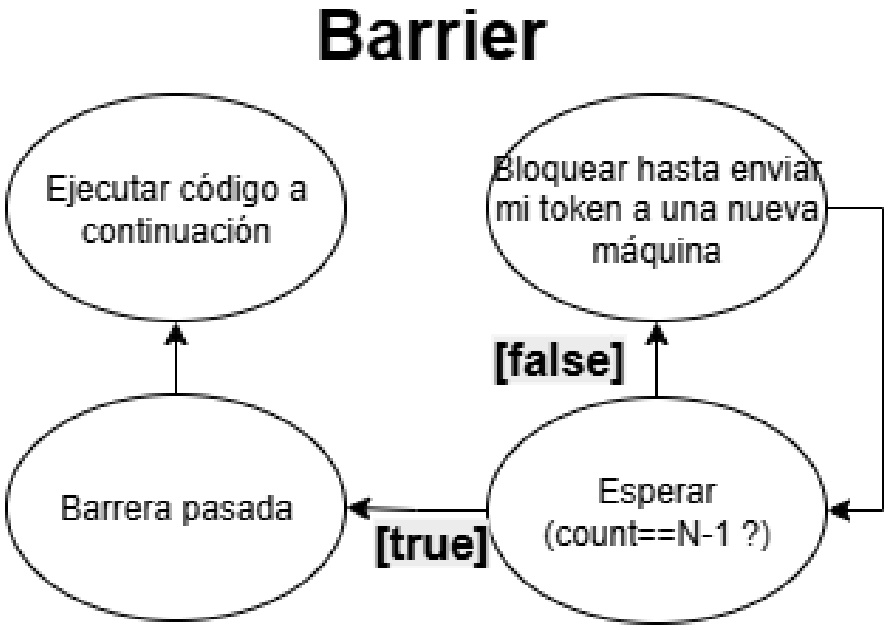



- **Diagrama de Secuencia:**

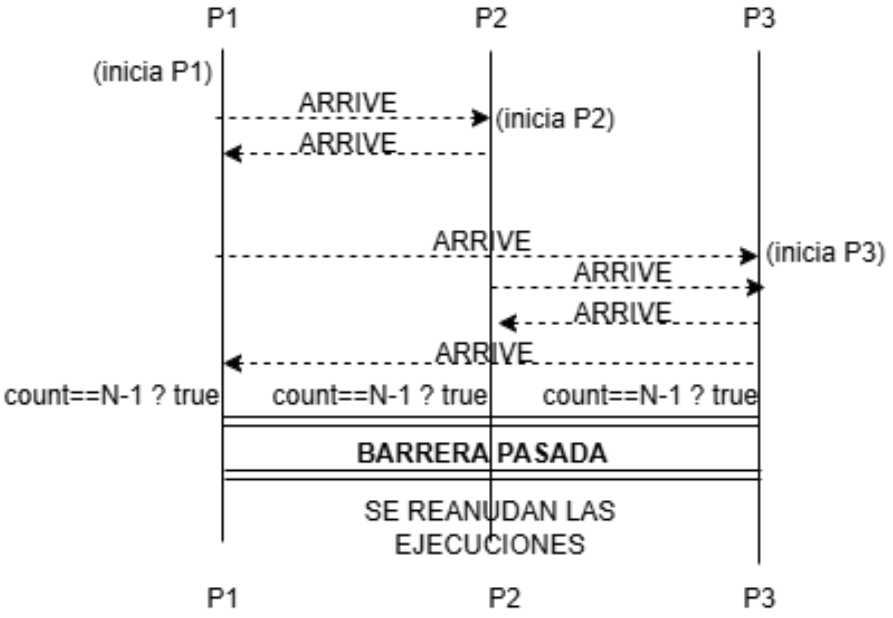


# Barrera distribuida (barrier):

- **Máquina de estados:**




- **Diagrama de Secuencia:**



- ## Nuestro diseño:

Además de solucionar los problemas propios de sintaxis del código, hemos añadido unas líneas al final del main para que el barrier funcionase correctamente:

```
fmt.Println("Waiting for all the processes to reach the barrier")
     <-barrierChan
    fmt.Println("Barrier passed!")

    listener.Close()
    quitChannel <- true
time.Sleep(1 * time.Second)
```

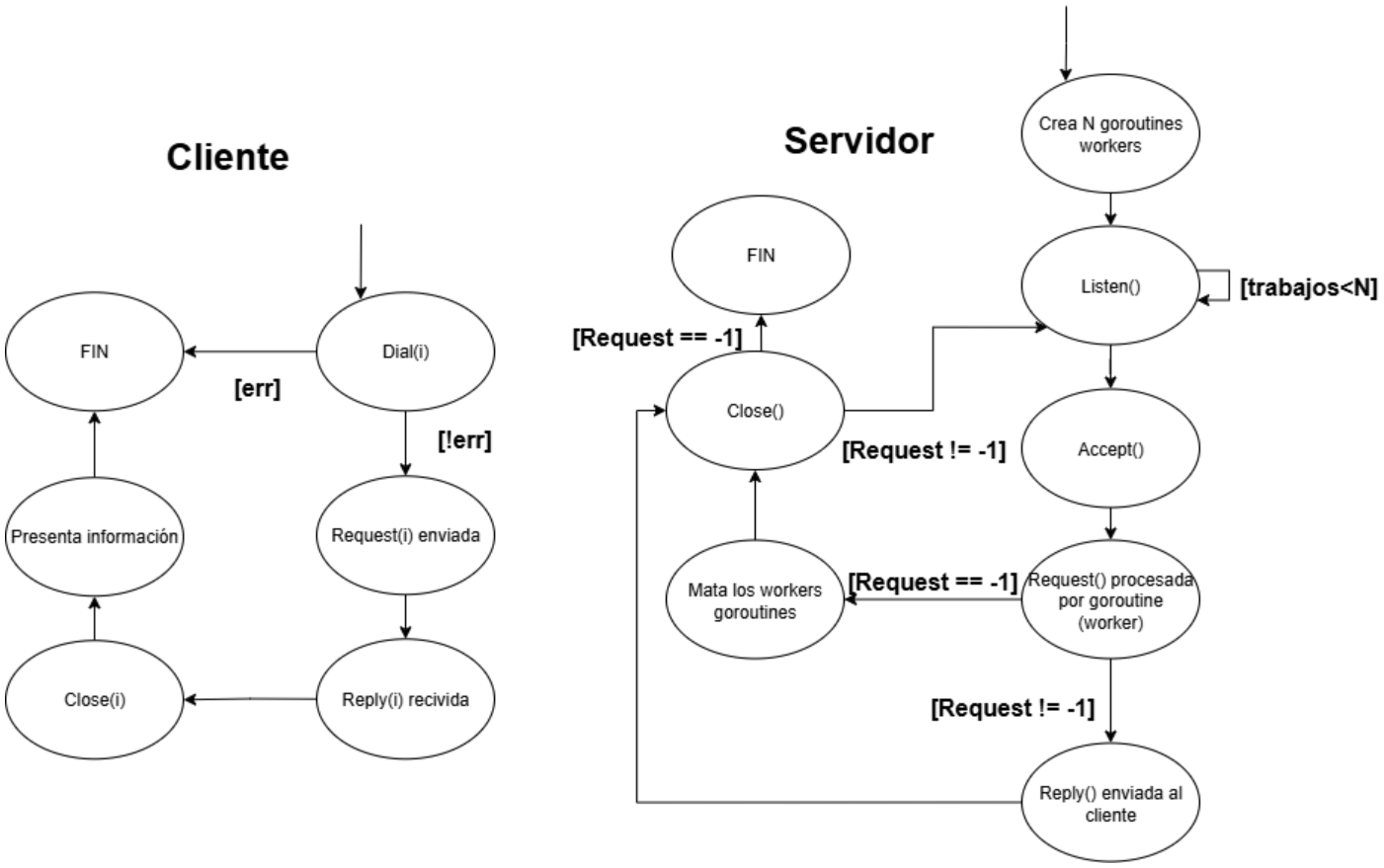
<-barrierChan espera a que se reciba true por el canal barrierChan, que solo es afirmativo cuando todos los procesos han notificado de su existencia al nuestro, por lo tanto ya se puede superar la barrera.

Después, cerramos el canal por el que escuchamos, y mandamos por el canal quitChannel el valor true, que provoca la salida de acceptAndHandleConnections dejando de tratar así conexiones.

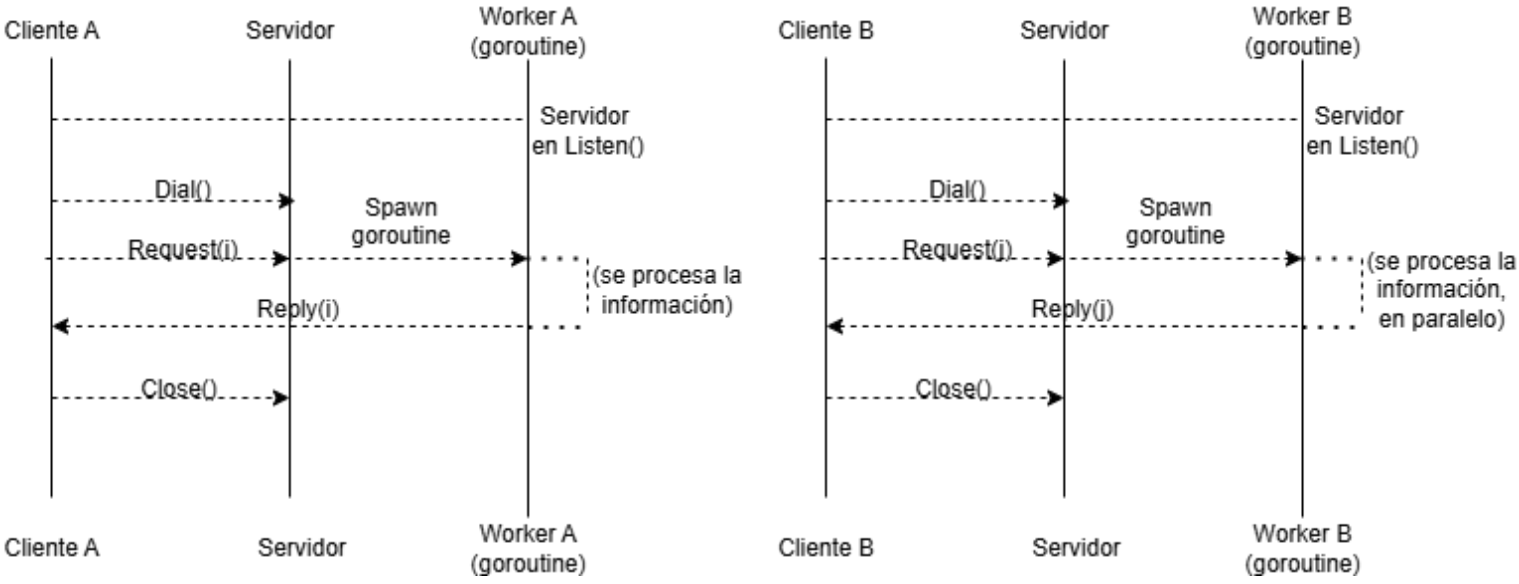
Por último, dormimos el proceso durante un segundo, ya que si no, el último proceso en ser ejecutado puede que no reciba el mensaje de presencia de los demás procesos. Esto se debe a que han terminado antes de mandárselo, con este segundo, a los demás procesos les da tiempo a comunicar su presencia.

# Cliente servidor concurrente:

- **Máquina de estados:**




- **Diagrama de Secuencia:**



## - Nuestro diseño:

Para poder hacer un cliente-servidor concurrente con un pool fijo de Goroutines realizamos los siguientes cambios sobre el código proporcionado del servidor:

- Modificamos la función *processRequest()* para poder identificar la señal de fin (Id == -1) para cerrar el servidor.

- Añadimos una función que define la tarea de cada uno de los n-workers y otra :

```
func worker(tasks <-chan net.Conn, results chan<-
com.Reply, done chan<- bool, workerDone chan<- bool) {
      defer func() { workerDone <- true }()
      for conn := range tasks {
            reply, end := processRequest(conn)
            if end {
                  done <- true
                  continue
            }
            results <- reply
      }
}
```

- Cambiamos el bucle de aceptación de conexiones a dentro de una goroutine para no bloquear al servidor

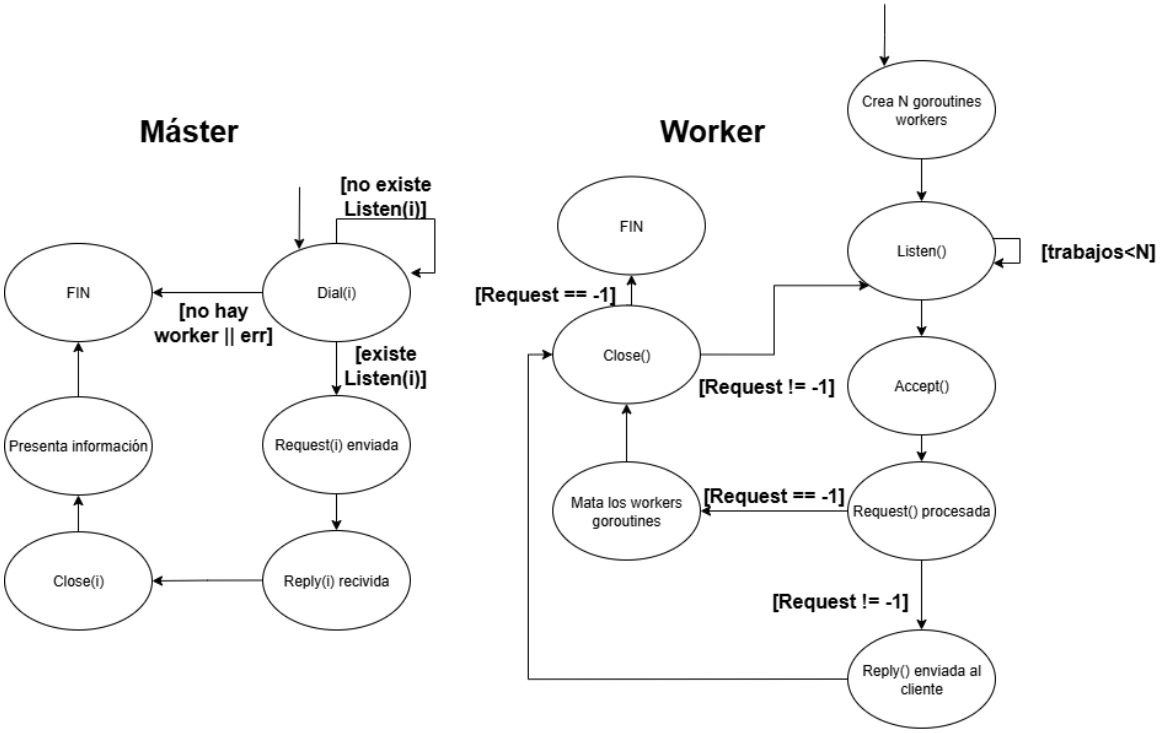
- Añadimos el siguiente trozo de código:

```
<-done                        //Esperar a -1 para parar
log.Println("Señal de fin recibida. Parando servidor…")
_ = listener.Close()              //Cerramos el listener
(provoca error en func de aceptación de conexiones)
<-stop           //Esperamos a que termine el acceptor
for i := 0; i < NWORKERS; i++ { //Cerramos los workers
           <-workerDone
}
close(results) //Cerramos results para que termine
resultHandler
log.Println("Servidor detenido correctamente.")
```

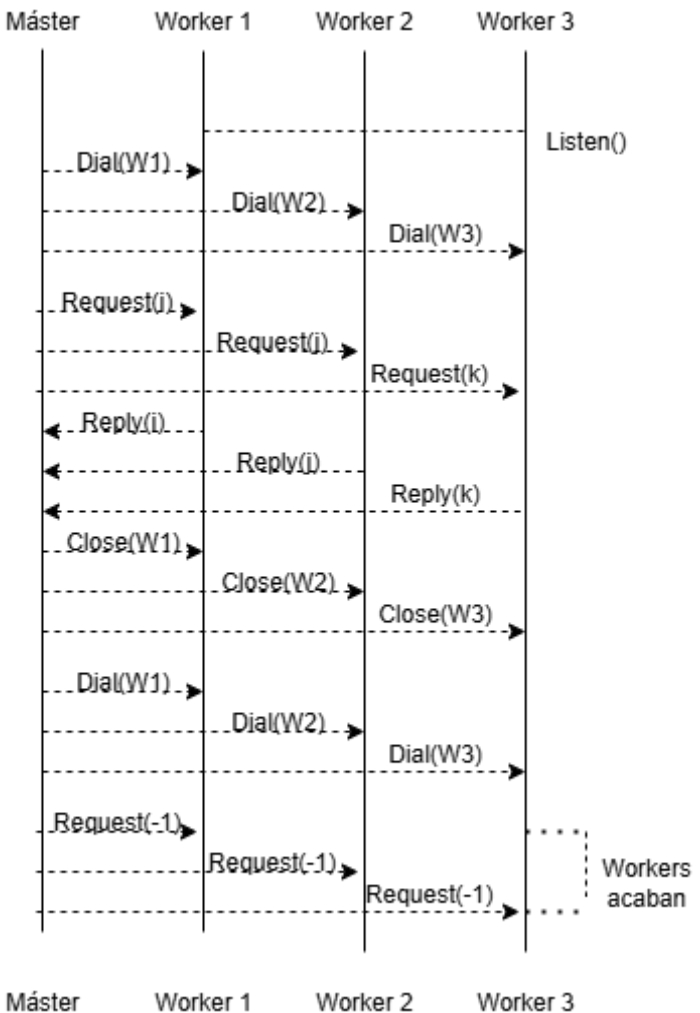
- Reutilizamos algunas funciones de otras partes del código proporcionado, un poco modificadas.

# Máster Worker:

- **Máquina de estados:**




- **Diagrama de Secuencia:**



## - Nuestro diseño:

Hemos creado diversas funciones, para el diseño del Máster, vamos a explicarlas brevemente:

1. readEndpoints: función reutilizada de barrier.go, en la que se le pasa el fichero con las distintas IP y puerto de las máquinas a las que queremos conectarnos, y las devolvemos en formato de lista.

2. dividirIntervalo: dividimos el intervalo dado en los argumentos al llamar al proceso, en n partes iguales, siendo n la cantidad de máquinas (endpoints) disponibles para usar. Estas partes servirán para distribuir el trabajo en partes iguales para todos los workers disponibles.

3. procesarEnvioWorker: Se envía la información para trabajar al worker, y el worker envía su resultado.

4. sendStopAll: Manda la señal de terminación -1 a todos los wokers.

Por último en el main, se ensambla todo para que funcione correctamente y con una correcta gestión de errores: se comprueban los parámetros, se divide el intervalo en bloques, y se envían dichos bloques a cada endpoint para que trabajen con él y nos den una respuesta. Finalmente, se agrupan todos los datos en una lista resultado, y se muestra por pantalla lo que se necesite.