Practical Assignment 1 - Distributed Systems

Basic concepts and mechanisms

Authors:

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

Delivery date: 25/09/2025

Group: Pair: 1-6

Computer Engineering - University of Zaragoza, Academic Year 2025-2026

Index

Problem description:

The objective of this practical exercise is to study the behavior of different network communication and distributed execution models, using machines with different architectures (Raspberry Pi for the final test and desktop PCs for intermediate tests). To achieve this:

We can divide the practice into two clearly differentiated blocks:

1. Network scenarios:
    - TCP client attempting to connect to a server that is not running (we measure the time until receiving the call error)*Dial*).
    - TCP client connecting to an already active server (we measure the connection establishment time).
    - UDP client and server, where the round-trip time for sending and receiving a letter is measured.
    - Comparison of these three scenarios running client and server on the same machine and on different machines.
2. Concurrent and distributed architectures:
    - Distributed barrier algorithm.
    - Concurrent TCP server with multiple clients using goroutines.
    - Master-worker architecture with SSH communication for calculating prime numbers in distributed intervals.

The tests were performed using the following computational resources:

- Raspberry Pi 4 Model B with ARM Cortex-A72 processor with 4 cores and 8 GB of RAM.
- Lab 1.02 desktop PCs with AMD64 architecture.
- Linux operating system
- Go programming language.

The execution of the various tests carried out aims to better understand the operation of communication protocols, analyze latency in different environments, and study how design decisions in software architecture influence the overall performance of distributed applications.

Architectural design:

TCP:

	Same machine (TCP)		Distinct machine (TCP)	
	Failed	Dial	Failed	Dial
1st	540.439 µs	729.566 µs	653.659 µs	1.345355 ms
2nd	764.381 µs	1.201617 ms	666.715 µs	1.273671 ms
3rd	835.232 µs	908.083 µs	639.789 µs	1.202135 ms
4th	701.862 µs	1.264597 ms	679.844 µs	1.093081 ms
5th	645.993 µs	697.696 µs	631.289 µs	1.533205 ms

The operation*Dial* In Go, this allows the client to establish a TCP connection with the server given its address. The execution time of this operation depends on the following:

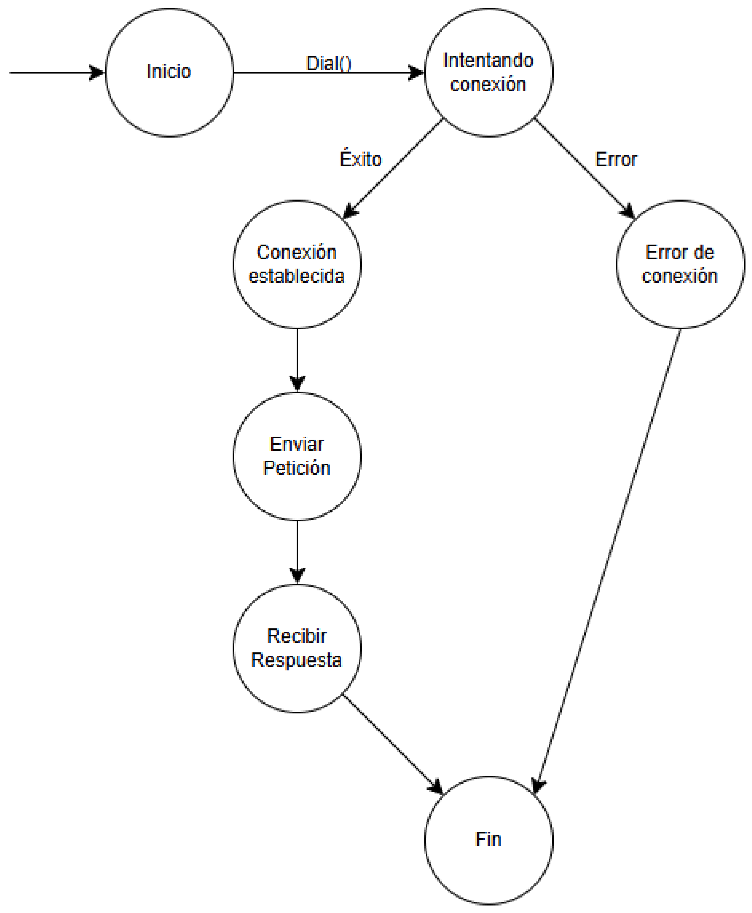
- The management's decision.
- The creation of the socket.
- The connection attempt: the client sends SYN, receives SYN+ACK from the server and responds with ACK.
- The delivery to the application,ta

Depending on the scenario, the cost of*Dial* may vary:

- *Dial*Failure: If the server is not running or the port is not open, an attempt is made to connect but receives an error.*connection refused* and the time is shorter than when the call works correctly.
- *Dial* It works correctly: the time is limited to establishing the connection.
- Client and server on the same machine: since there is no physical transmission over the network, the execution time is minimal.
- Client and server on different machines: latency increases due to transmission over the network and depends on propagation delay.

In contrast, when*Dial* It works correctly; the cost corresponds to the time it takes the client to establish a connection with the server. When we run this on different machines, the average time it takes on the same machine (~0.7-1.3 ms) would have to be added to the LAN propagation latency (~0.3-0.5 ms).

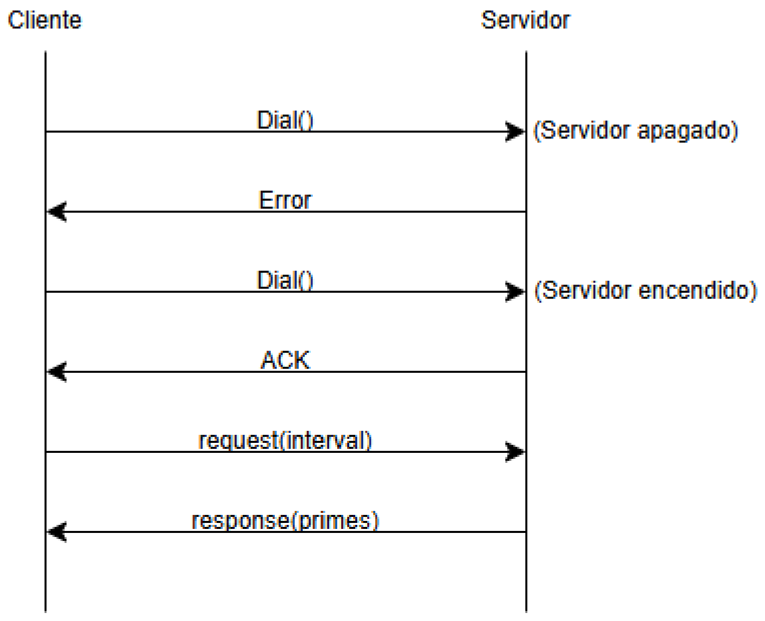
- **State machine:**



Explanation:

1. **Start:**The client prepares to establish a connection with the server.
2. **Attempting connection:**the client executes*Dial()*to attempt to connect to the server.
3. **Connection established:** *Dial()*returns a valid Conn object.
4. **Send Request / Receive Response:**The client sends the interval and receives the primes calculated by the server.
5. **Connection errorifrom:** *Dial()*It returns an error because the server is not yet running.
6. **END:**The client completes its execution.

- **Sequence Diagram:**



Explanation:

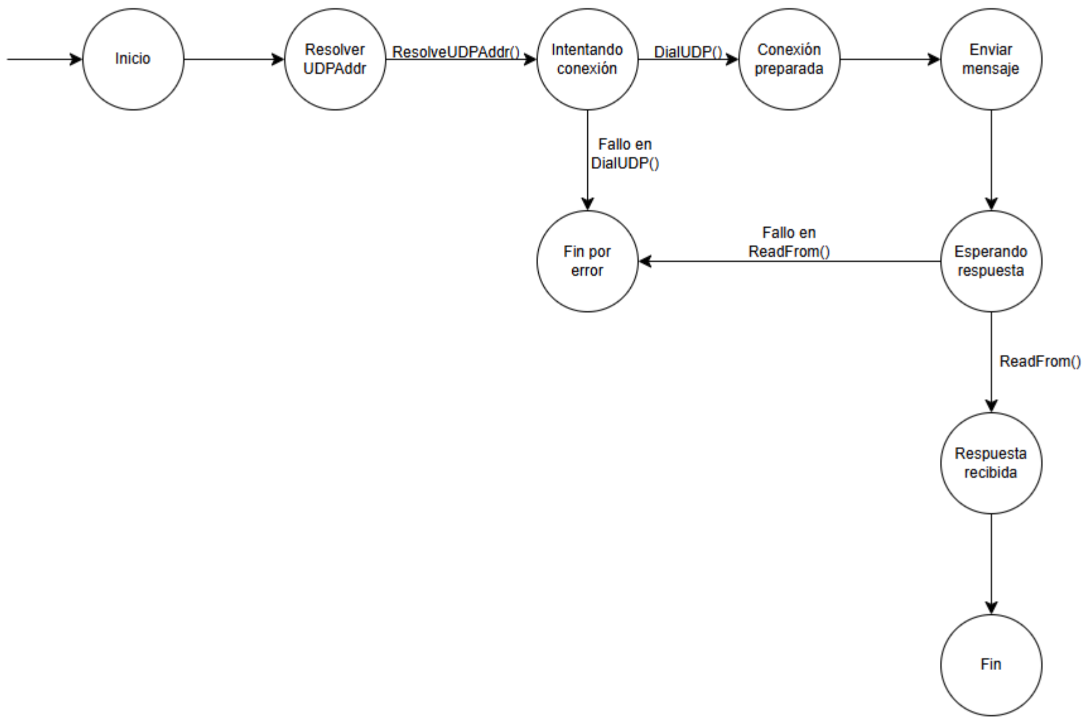
1. The client is trying to connect to the server, but it is not yet running, so*Dial()* returns an error.
2. We launch the server and the client calls again.*Dial()*And this time the connection is established.
3. The client sends the range of numbers.
4. The server calculates the primes in the interval and returns them to the client.
5. The client receives the result and ends its execution by sending close() (nothing else is sent from that moment on).

UDP:

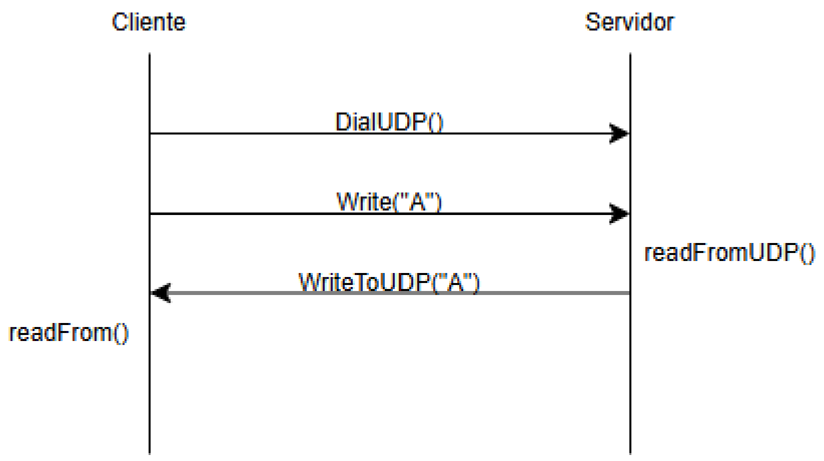
	Same machine (UDP)	Distinct machine (UDP)
1st	473.142 µs	693.64 µs
2nd	359.496 µs	661.344 µs
3rd	345.848 µs	512.624 µs
4th	339.182 µs	491.716 µs
5th	334.811 µs	505.754 µs

Regarding the cost of running the transmission in the UDP scenario, this is due to the preparation and encapsulation of the message on the client, the transmission of the datagram, the processing, the server echo, and the reception of the response back on the client.

- **State machine:**



- **Sequence Diagram:**

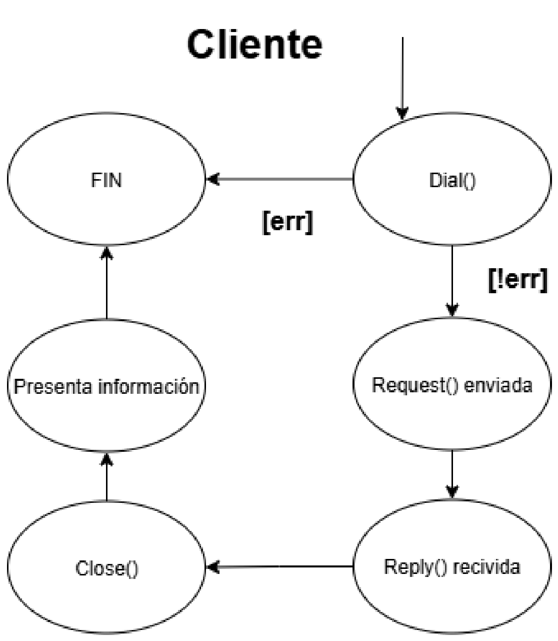


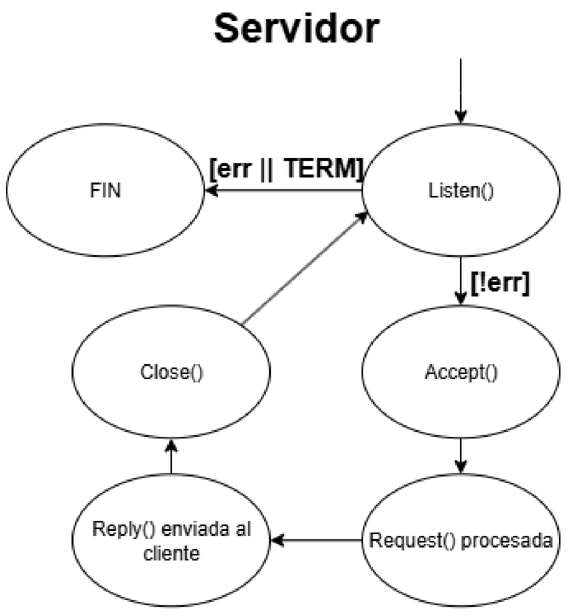
Explanation:

1. The client prepares the UDP with DialUDP().
2. Send a letter to the server with Write().
3. The server receives the message and forwards it to the client.
4. The client receives the response and the execution is complete.

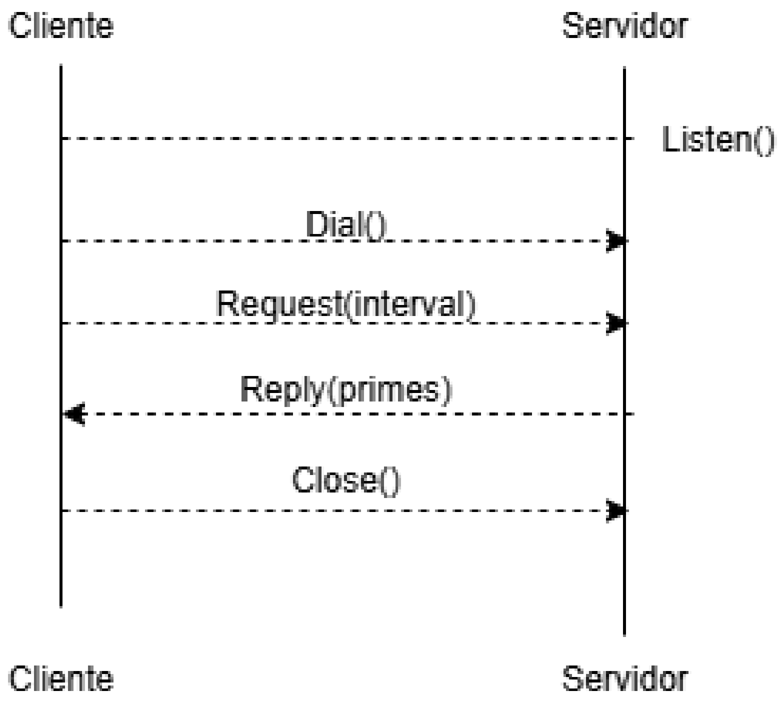
Sequential client-server:

- **State machine:**



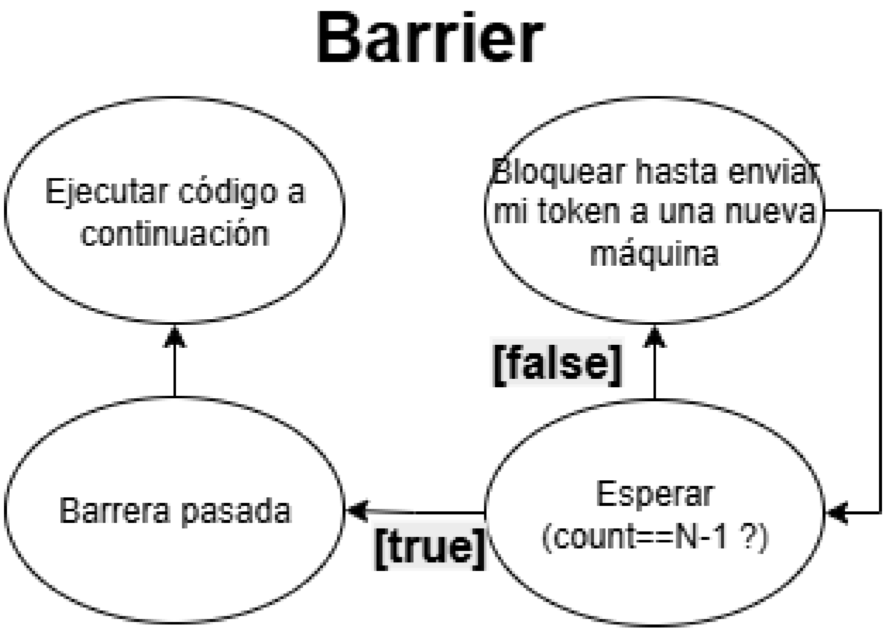


- **Sequence Diagram:**

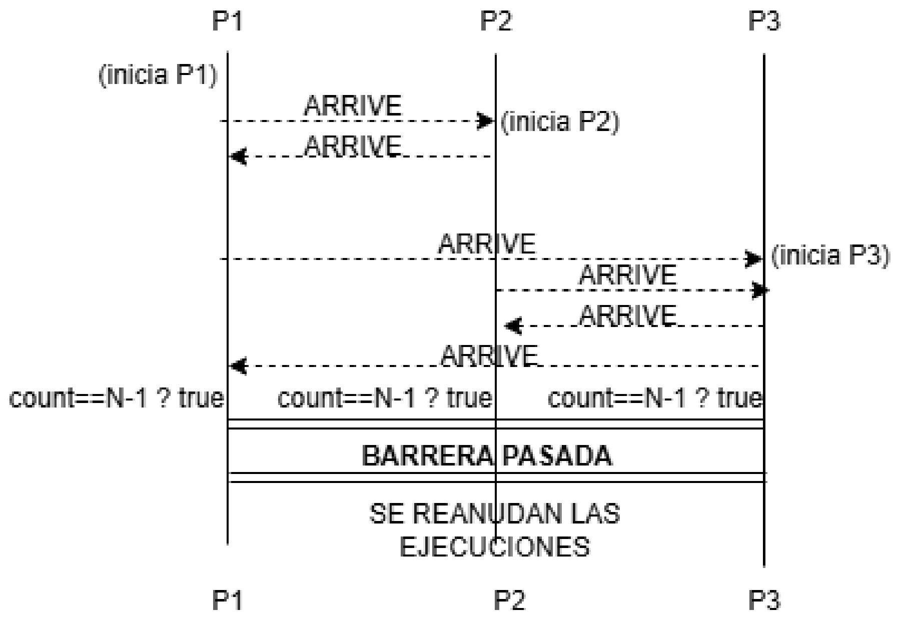


Distributed barrier (barrier):

- **State machine:**


Barrier
Ejecutar código a continuación
Bloquear hasta enviar mi token a una nueva máquina
[false]
Barrera pasada
[true]
Esperar (count==N-1 ?)

- **Sequence Diagram:**



- **Our design:**

In addition to fixing the code's syntax problems, we added a few lines to the end of the main function to make the barrier work correctly:

```
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)
```

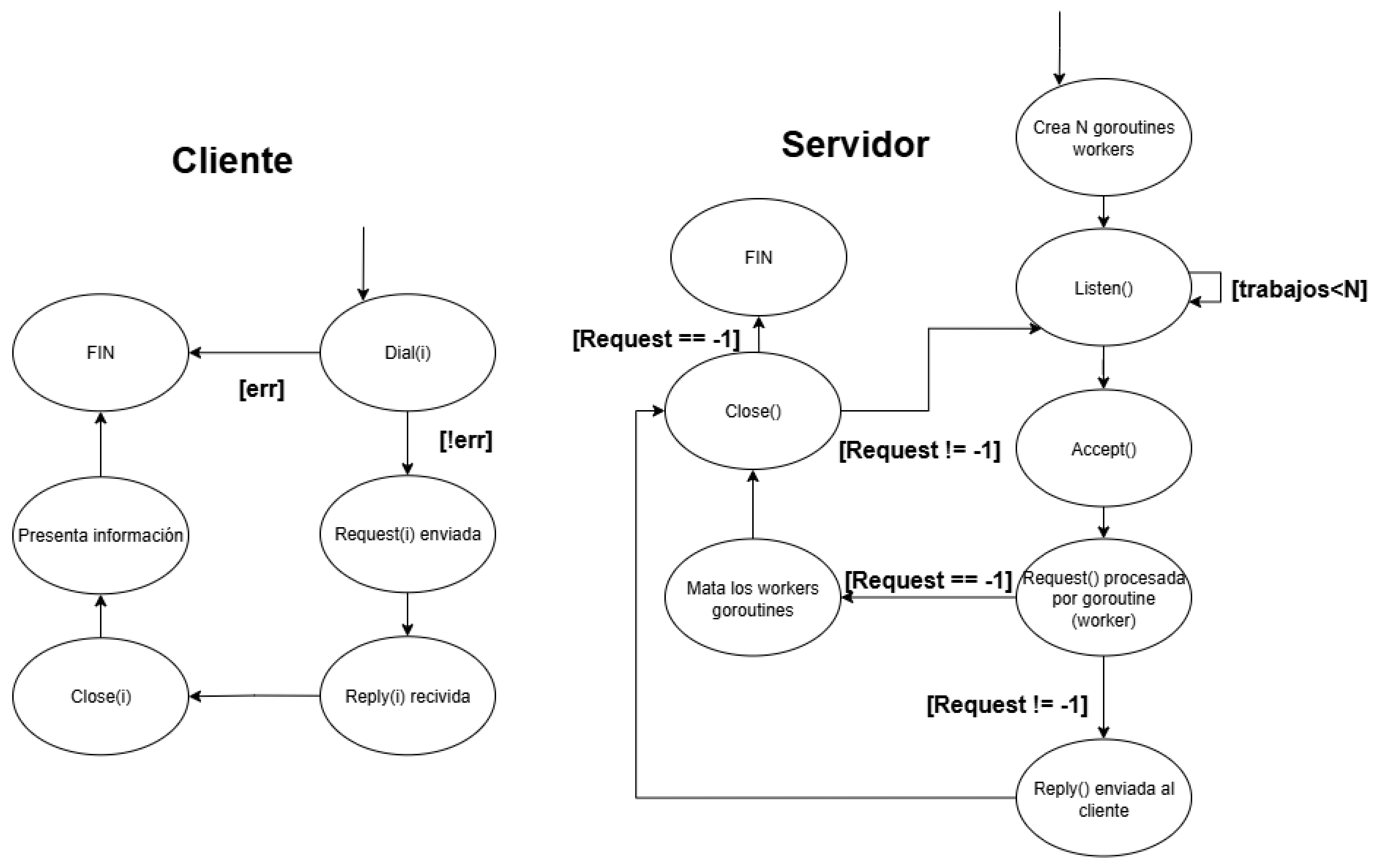
<-barrierChan waits for true to be received through the barrierChan channel, which is only affirmative when all processes have notified ours of their existence, therefore the barrier can now be overcome.

Next, we close the channel we are listening on, and send the value true through the quitChannel channel, which causes acceptAndHandleConnections to exit, thus ceasing to handle connections.

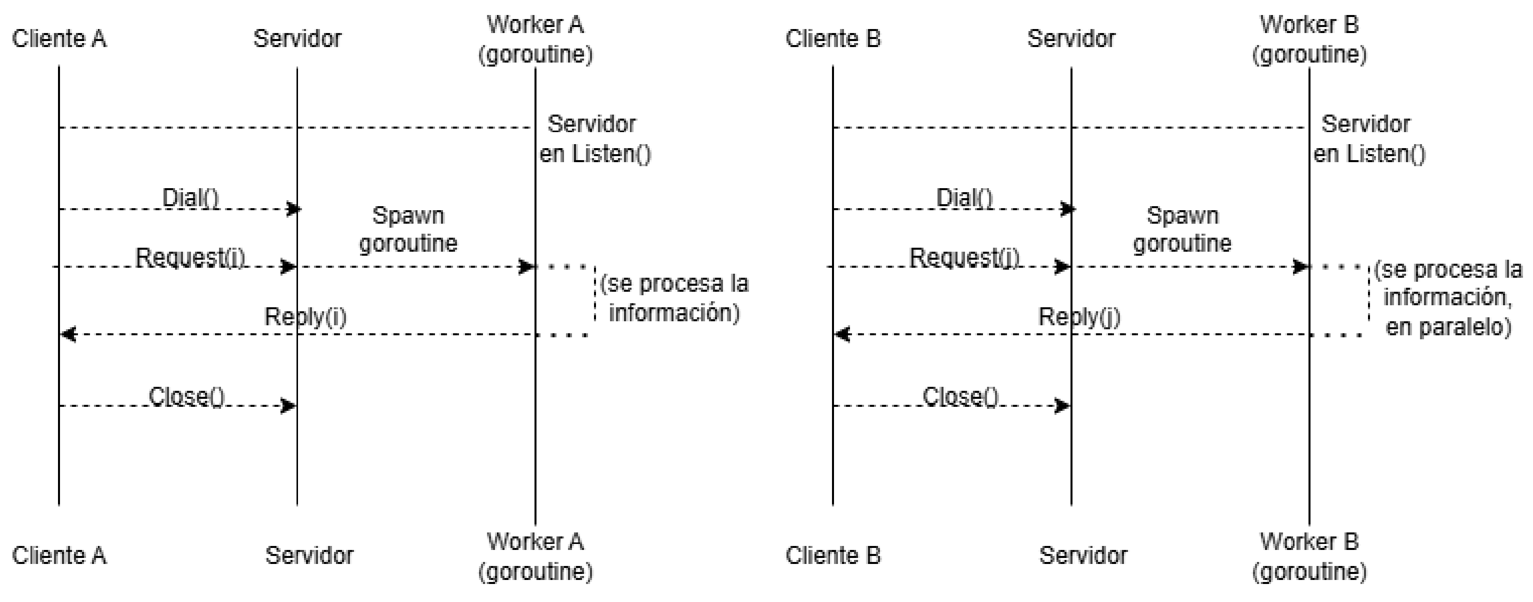
Finally, we put the process to sleep for one second, because otherwise, the last process to be executed might not receive the presence message from the other processes. This is because they finished before sending it; this second gives the other processes time to communicate their presence.

Concurrent client-server:

- **State machine:**



- **Sequence Diagram:**



- Our design:

To create a concurrent client-server environment with a fixed pool of Goroutines, we made the following changes to the provided server code:

- We modified the function*processRequest()* in order to identify the end signal (Id == -1) to close the server.

- We added a function that defines the task of each of the n-workers and another:
```
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
      }
}
```

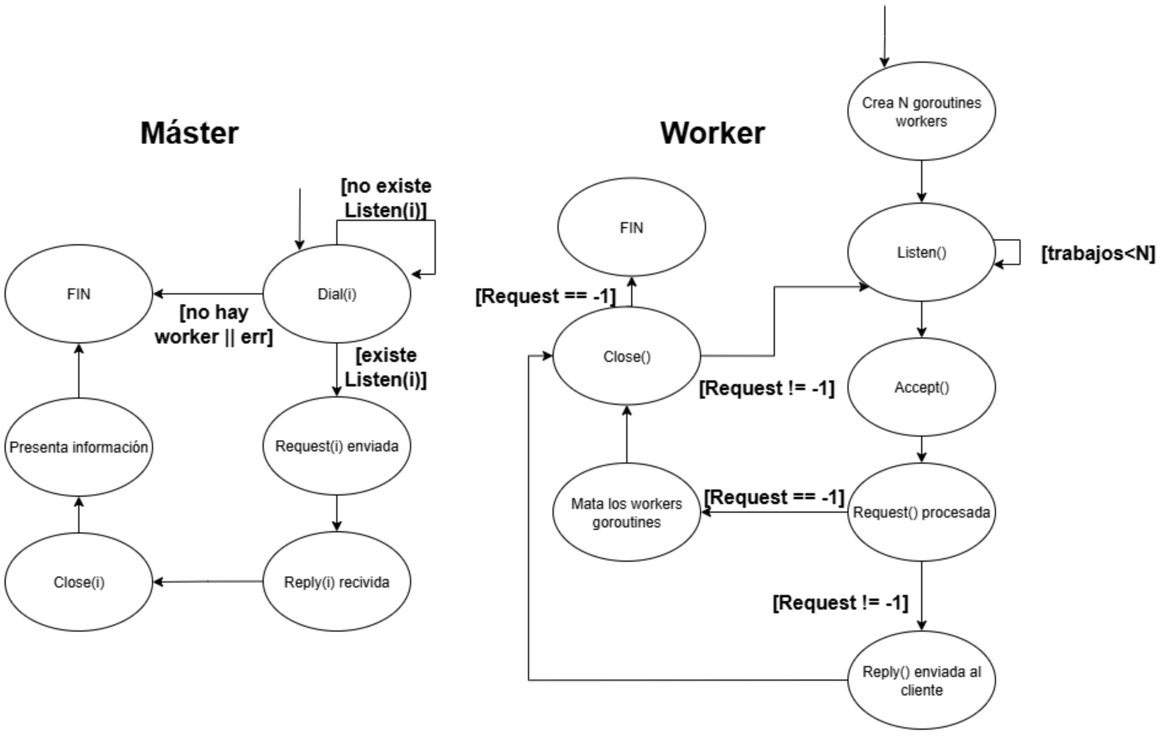
- We changed the connection acceptance loop to inside a goroutine so as not to block the server

- We add the following piece of code:
```
<-done                       //Wait for -1 to stop
log.Println("End signal received. Stopping server…")
_ = listener.Close()              //We close the listener
(causes an error in the connection acceptance function)
<-stop          //We wait for the acceptor to finish
for i := 0; i < NWORKERS; i++ { //Cerramos los workers
           <-workerDone
}
close(results) //We close results so that resultHandler
finishes
log.Println("Server successfully stopped.")
```

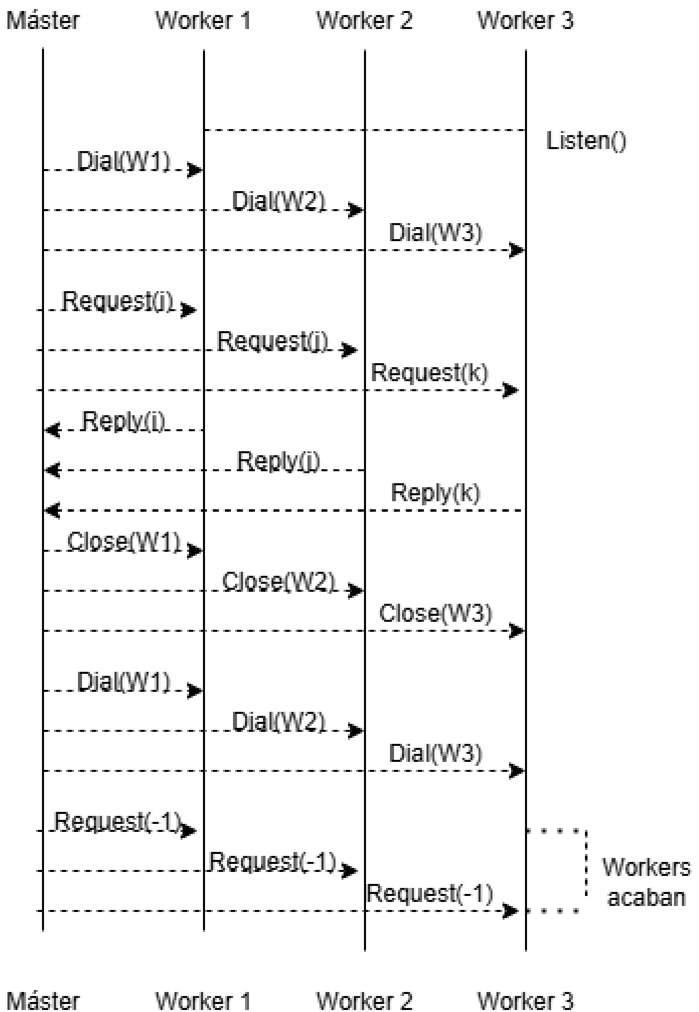
- We reused some functions from other parts of the provided code, slightly modified.

Master Worker:

- **State machine:**



- **Sequence Diagram:**



- Our design:

We have created several functions for the design of the Master's program; we will briefly explain them:

1. readEndpoints: reused function ofbarrier.go, in which the file with the different IPs and ports of the machines we want to connect to is passed to it, and we return them in list format.

2. divideInterval: We divide the interval given in the arguments when calling the process into n equal parts, where n is the number of machines (endpoints) available to use. These parts will serve to distribute the work equally among all available workers.

3. processShipWorker: The information to be processed is sent to the worker, and the worker sends its result.

4. sendStopAll: Sends the termination signal -1 to all wakers.

Finally, in the main function, everything is assembled to function correctly and with proper error handling: parameters are checked, the interval is divided into blocks, and these blocks are sent to each endpoint to process them and provide a response. Finally, all the data is grouped into a result list, and the necessary information is displayed on the screen.