Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

sisdis-pr-1

Sistemas Distribuidos

Autor 1:	Toral Pallás, Héctor - 798095
Autor 2:	Pardos Blesa, Javier - 698910
Grado:	Ingeniería Informática
Curso:	2022-2023

23 de Septiembre de 2022

Índice

1. Introducción

1.1. Descripción del problema

El problema en cuestión es el de asignar tareas de una aplicación distribuida a recursos computacionales. Por recurso computacional, se entiende en esencia, CPU, red de comunicación y almacenamiento. El objetivo es utilizar toda la capacidad computacional de los recursos para poder satisfacer los requisitos de la aplicación.

1.2. Descripción de la aplicación

La aplicación consiste en un grupo de clientes que envian peticiones contra el servidor, dichas peticiones contienen el trabajo a procesar, en este caso es la búsqueda de los números primos contenidos en el intervalo enviado por el cliente. Para llevar a cabo el trabajo, se han relizado varias versiones con diferentes arquitecturas (que se detallan en el apartado de Diseño especifico de cada una) de forma que se pueda comparar su rendimiento.

1.3. Recursos computacionales disponibles

En el laboratorio L1.02, donde vamos a desarrollar las prÁcticas, contamos con 20 máquinas

nombre	dirección IP	nombre	dirección IP	nombre	dirección IP
lab102-191	155.210.154.191	lab102-192	155.210.154.192	lab102-193	155.210.154.193
lab102-194	155.210.154.194	lab102-195	155.210.154.195	lab102-196	155.210.154.196
lab102-197	155.210.154.197	lab102-198	155.210.154.198	lab102-199	155.210.154.199
lab102-200	155.210.154.200	lab102-201	155.210.154.201	lab102-202	155.210.154.202
lab102-203	155.210.154.203	lab102-204	155.210.154.204	lab102-205	155.210.154.205
lab102-206	155.210.154.206	lab102-207	155.210.154.207	lab102-208	155.210.154.208
lab102-209	155.210.154.209	lab102-210	155.210.154.210		

Cada maquina tiene una CPU de 6 cores de 64 bits (Intel Core i5-9500) que nos van a permitir ejecutar, por lo tanto, hasta 6 instancias en paralelo de nuestra aplicación. Además estas máquinas cuentan con 32 GB y 2,3 GB para el Swap.

2. Cliente-Servidor secuencial

Esta arquitectura consta de un solo servidor, el cual procesa todas las peticiones que va recibiendo de diferentes clientes.

2.1. Diseño de la arquitectura

A continuación se detalla el diagrama de secuencia de una ejecucion sobre el esquema seguido. Se observa que los clientes envian sus peticiones (com.Request) al servidor, y este las va gestionando en forma de cola FIFO. El servidor usa un único hilo para hacer las operaciones necesarias (con la petición req recibida se ejecuta findPrimes(req.Interval)) que el cliente pide, y finalmente devolver el paquete con la respuesta (com.Reply).

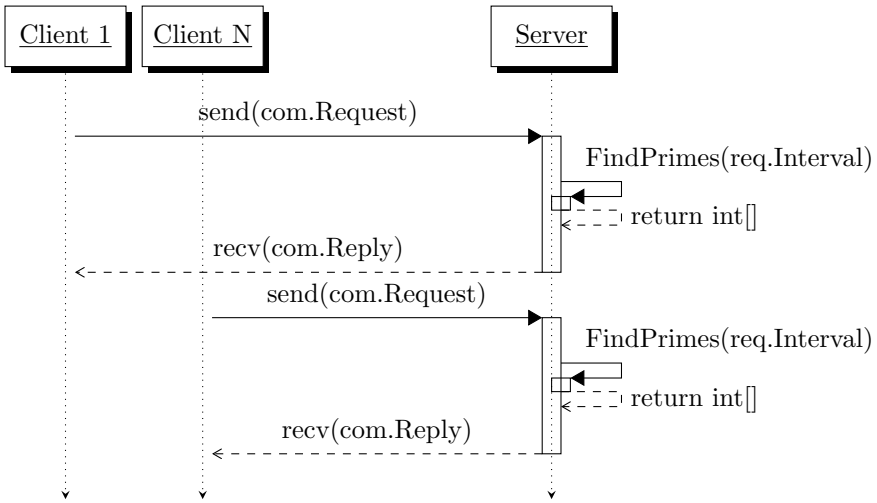

Figura 1: Diagrama de secuencia CS-Secuencial.

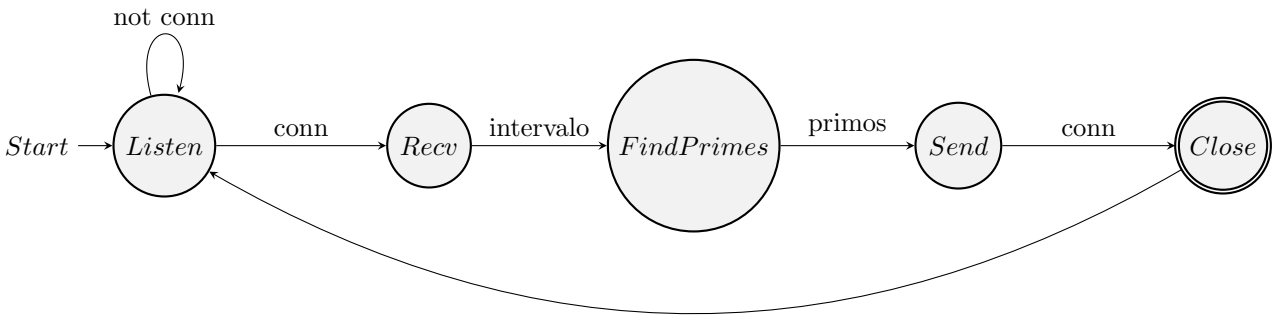

Figura 2: Diagrama de estados CS-Secuencial.

2.2. Ejecución

```
# Cliente
go run ./client/client.go <ip> <port>

# Servidor
go run ./server/server-draft.go -s <ip> <port>
```

2.3. Rendimiento

Respecto al rendimiento de la aplicación con este diseño, se puede apreciar en la gráfica que por cada petición se va aumentando el tiempo que se tarda en ser procesada, pudiendo llegar a tener un coste en tiempo creciente. Se puede validar esta afirmación ya que por cada grupo de peticiones que se lanzan desde el cliente hay un intervalo de 3 segundos de tts (time to sleep) en el que pueden procesarse una parte de estas peticiones pero inevitablemente se va formando una cola que produce que se incremente el tiempo hasta la llegada de la respuesta.

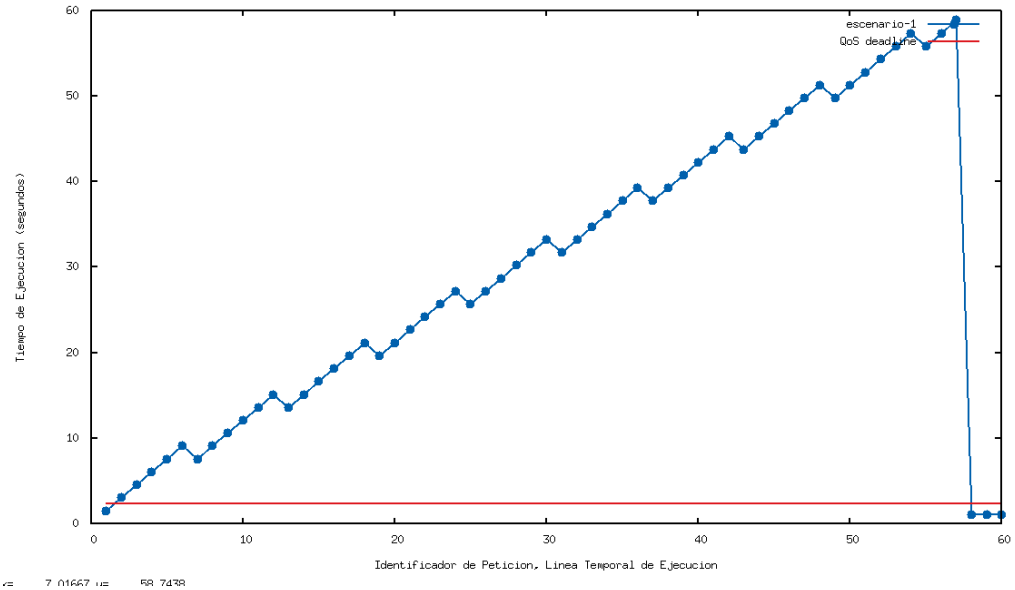

Figura 3: Gráfica cliente-servidor secuencial

3. Cliente-Servidor concurrente con una Goroutine por petición

3.1. Diseño de la arquitectura

Es un esquema que proporciona concurrencia para la gestión de las peticiones, a diferencia del cliente-servidor secuencial. En este caso, por cada cliente que inicia una comunicación con el servidor se crea una Goroutine específica para esa petición pudiendo dejar libre el hilo principal del servidor para seguir gestionando otras peticiones.

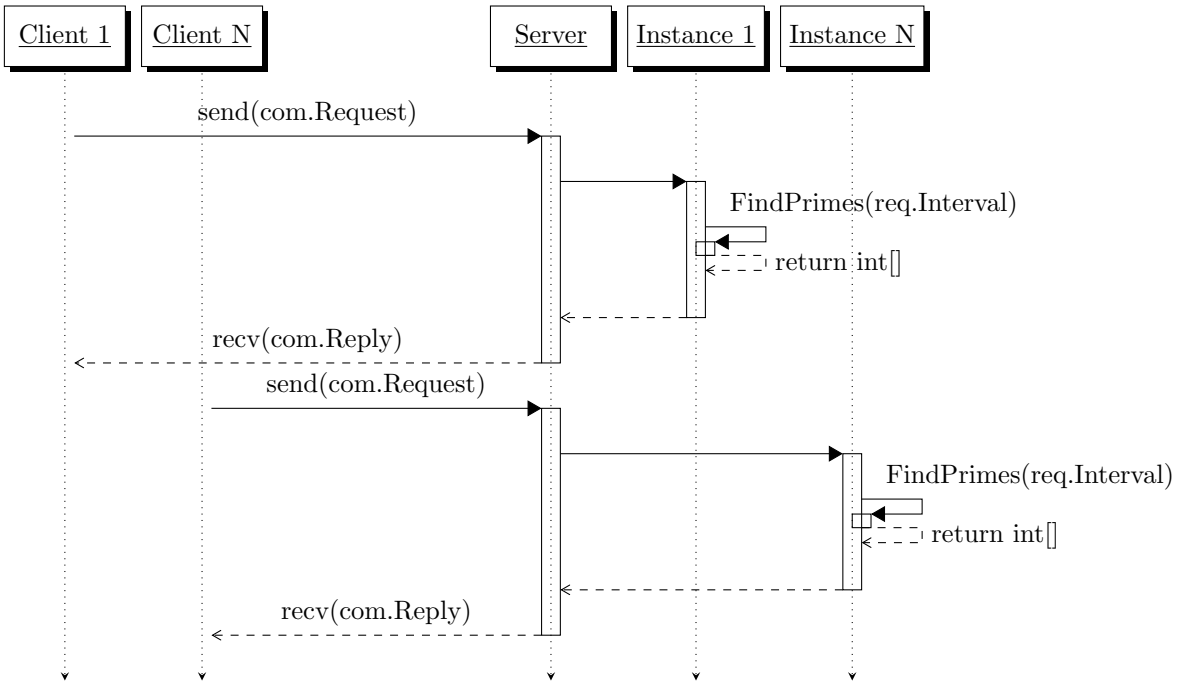

Figura 4: Diagrama de secuencia CSC-Sin pool fijo.

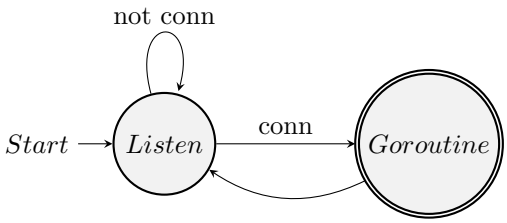

Figura 5: Diagrama de estados CSC-Sin pool fijo.

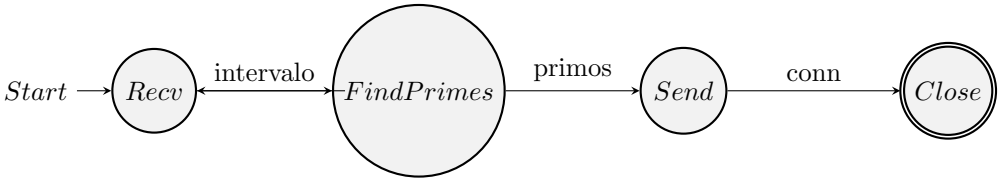

Figura 6: Diagrama de estados CSC-Sin pool fijo. Handler()

3.2. Ejecución

```
# Cliente
go run ./client/client.go <ip> <port>

# Servidor
go run ./server/server-draft.go -cspf <ip> <port>
```

3.3. Rendimiento

Como se puede observar, al no tener una limitación en la creación de Goroutines el resultado de esta prueba no supera el QoS. Esto se produce ya que por cada grupo de peticiones recibidas desde el cliente, el servidor puede atender todas ellas de forma concurrente delegando su ejecución en Goroutines. De esta manera se obtiene una comunicación mas rápida que si se hace de forma secuencial.

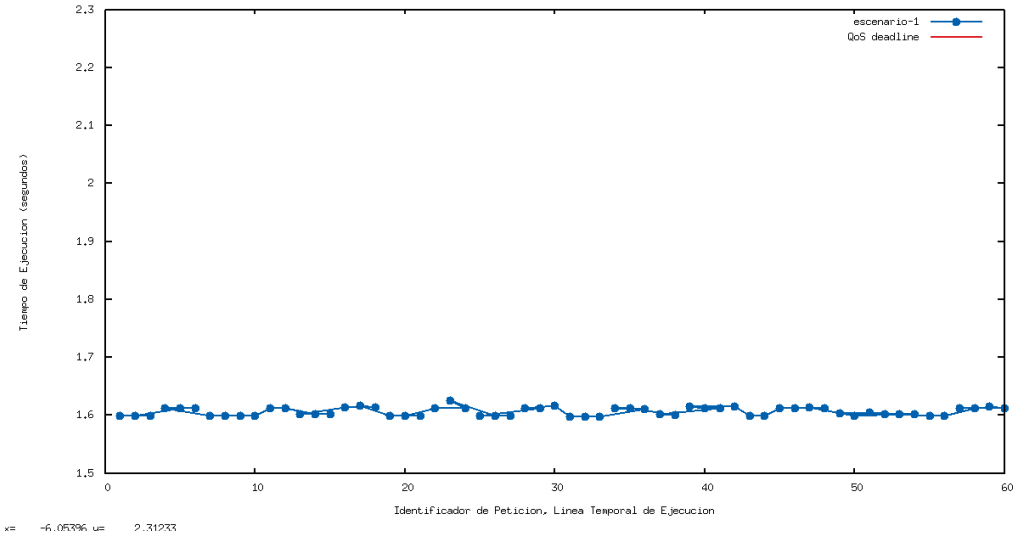

Figura 7: Gráfica CSC-Sin pool fijo

4. Cliente-Servidor concurrentet con un pool fijo de Goroutines

4.1. Diseño de la arquitectura

Este diseño, ofrece la posibilidad de poder tratar las peticiones de forma concurrente. En un primer lugar se instancian las N Gorutines que van a tratar las peticiones entrantes y a continuación se queda escuchando las peticiones de los M clientes que pudiera tener.

La ventaja de este esquema es que se evita el coste computacional de tener que crear una Gorutine cada vez que llega una nueva petición, limitando así el número de peticiones que pueden ser atendidas a la vez pudiendo provocar en algunos casos congestión debido a la alta demanda por parte de los clientes.

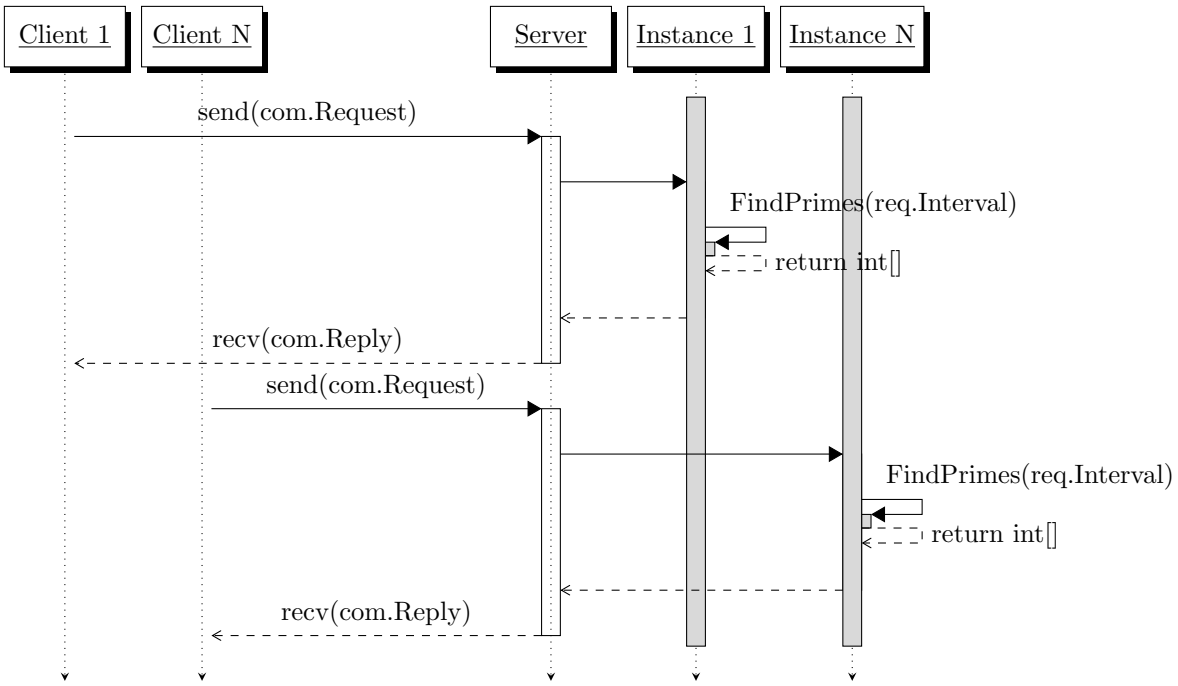

Figura 8: Diagrama de secuencia CSC-Con pool fijo.

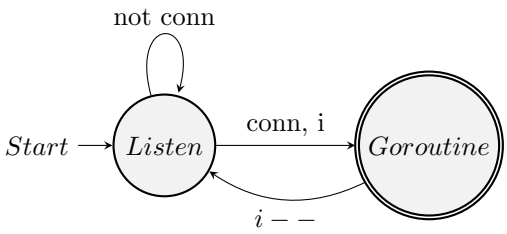

Figura 9: Diagrama de estados CSC-Con pool fijo.

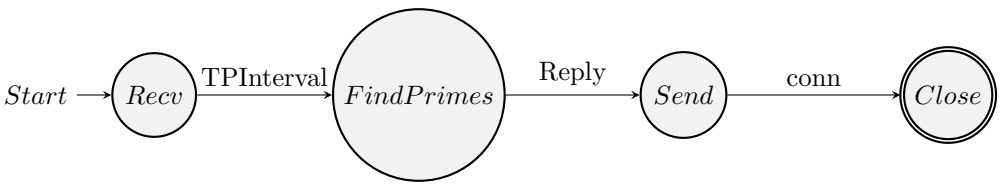

Figura 10: Diagrama de estados CSC-Con pool fijo. Handler()

4.2. Ejecución

```
# Cliente
go run ./client/client.go <ip> <port>

# Servidor
go run ./server/server-draft.go -cpf <ip> <port> <num gorutines>
```

4.3. Rendimiento

Para las pruebas sobre este esquema, se ha definido un pool de hasta 3 Gorutines para hacer frente a los grupos de peticiones que generan los clientes. En este caso se observa que las 3 primeras peticiones se gestionan de forma concurrente dando unos tiempos similares en cambios las 3 siguientes tardan mas tiempo (ya que deben esperar a que algun hilo acabe con las peticones anteriores).

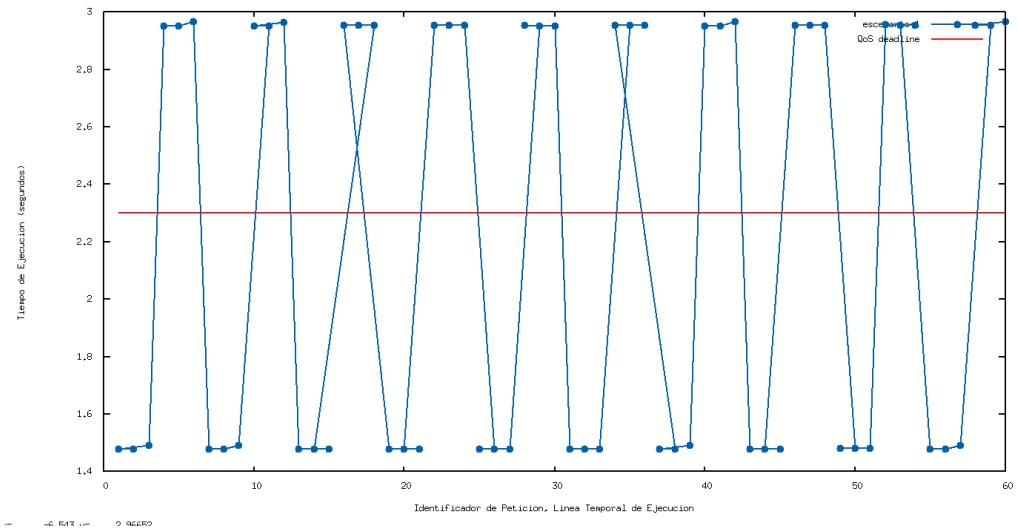

Figura 11: Gráfica CSC-Con pool fijo

5. Master-Worker

5.1. Diseño de la arquitectura

Para el esquema Master-Worker, se aprovecha el poder de cómputo de las máquinas a las que el master puede redirigir las peticiones que recibe de los clientes. En su implementación se ha optado por lo siguiente: El equipo que actua de master analiza las máquinas del laboratorio y elige N máquinas activas para usar como workers. Una vez que el master, tiene la cantidad de máquinas necesarias, lanza los workers mediante ssh y crea un pool fijo de gorutinas que gestionan las peticiones entrantes a los workers de tal manera que se asignen las peticiones al primer worker que este libre.

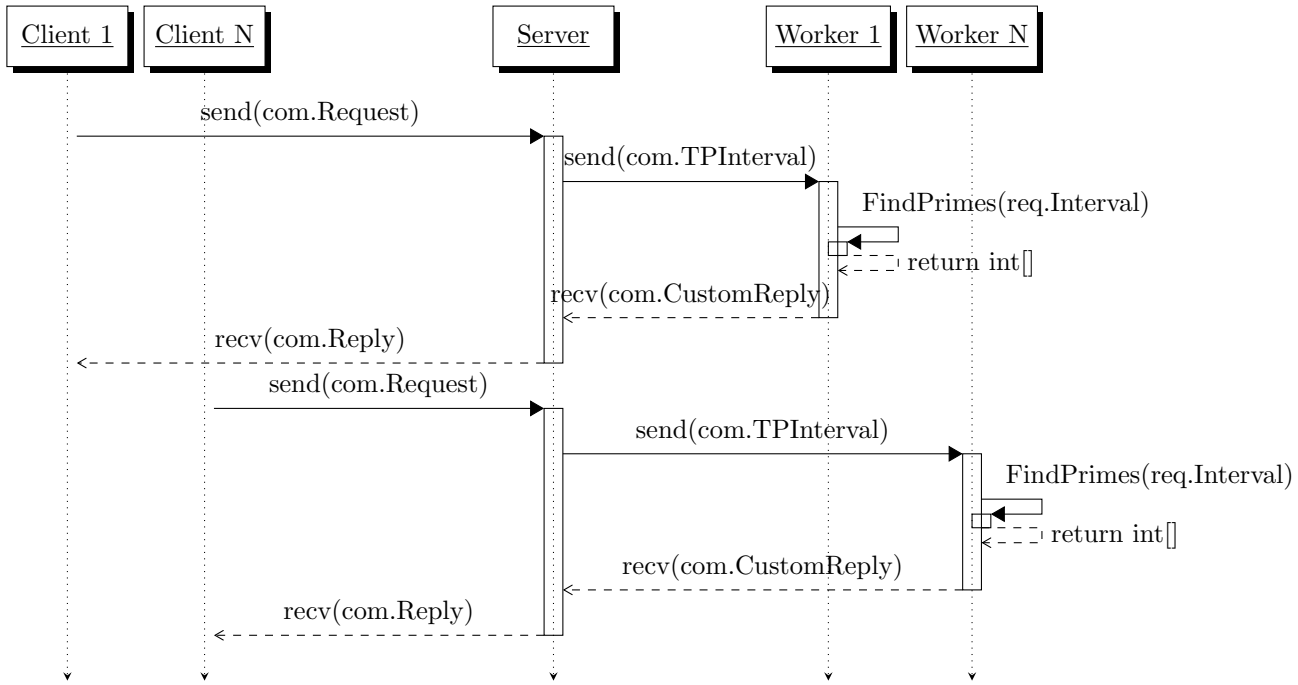

Figura 12: Diagrama de secuencia Master Worker.

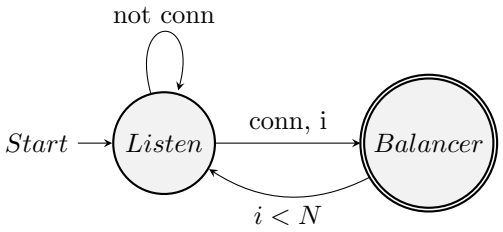

Figura 13: Diagrama de estados balanceador de carga.

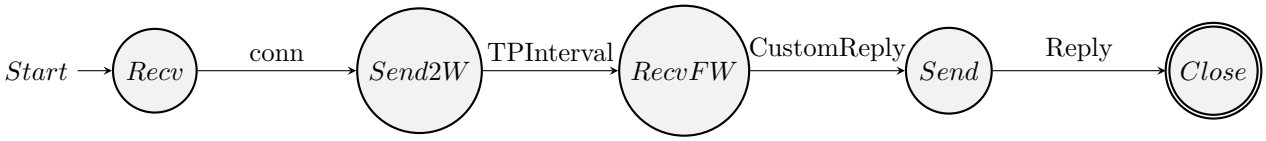

Figura 14: Diagrama de estados Master.

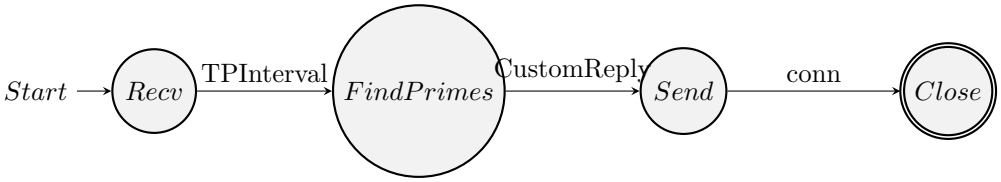

Figura 15: Diagrama de estados Worker.

5.2. Ejecución

Para su ejecución, el usuario debe haber generado la clave pública y tenerla almacenada en la máquina destino para poder ejecutar los comandos de lanzamiento de los workers de forma remota desde el master. Esto se consigue con lo siguientes comandos:

```
# Generar clave publica desde la maquina que se va a lanzar el master
ssh-keygen -t rsa -b 4096

# Copiar la clave publica al fichero ~/.ssh/authorized_keys de la maquina destino
cat id_rsa >> authorized_keys

# Cliente
go run ./client/client.go <ip> <port>
```

Para realizar de forma correcta la ejecución del master la variable constante definida al principio del fichero master.go debe ser igual a la ruta absoluta donde se tenga la copia del repositorio.

```
SRC\_PATH = "/home/aNIP/Desktop/sisdis-pr-1/"
```

Finalmente se puede ejecutar el master que inicializara los workers de forma remota

```
# Master
go run ./master/master.go <ip> <port> <num workers>
```

5.3. Rendimiento

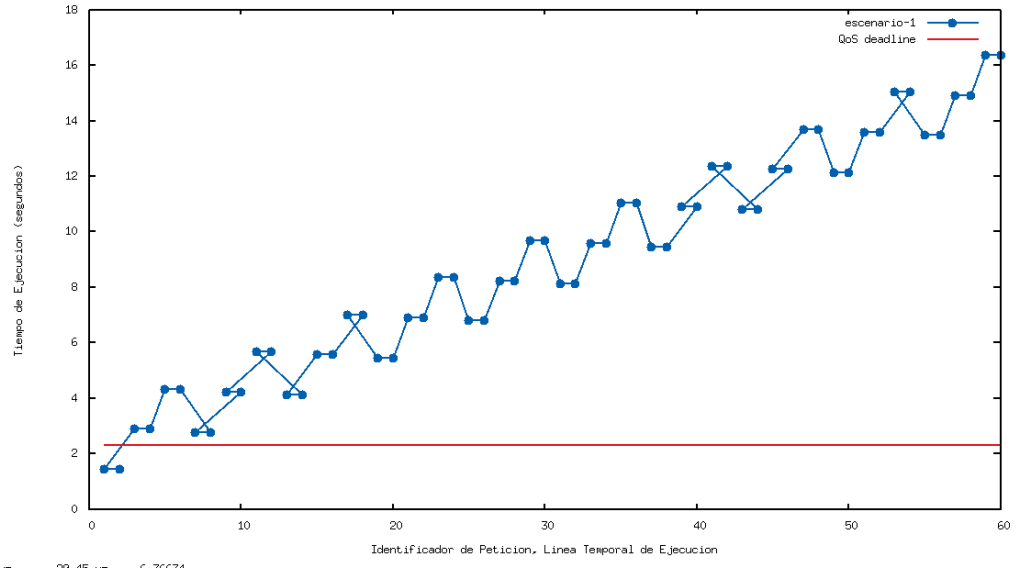

Figura 16: Gráfica Master-Worker