# SISTEMAS DISTRIBUIDOS

## PRÁCTICA 1

Pablo Moreno Muñoz 841972
Andrés Yubero Segura 842236

# ÍNDICE

# INTRODUCCIÓN

En esta memoria, diseñamos e implementamos cuatro arquitecturas para la ejecución de una aplicación distribuida de cálculo de números primos. La tarea principal de la aplicación es encontrar números primos dentro de un intervalo dado [1000, 70000].

En este proyecto, se considera la ejecución en un conjunto de máquinas en el laboratorio L1.02, que cuentan con CPU Intel Core i5-9500 de 6 cores de 64 bits. Cada máquina tiene 32 GB de memoria y 2.3 GB de Swap, lo que proporciona recursos suficientes para realizar estas prácticas.

# DISEÑO DE LAS 4 ARQUITECTURAS

A continuación, presentamos el diseño de las cuatro arquitecturas de cálculo de números primos en un entorno distribuido:

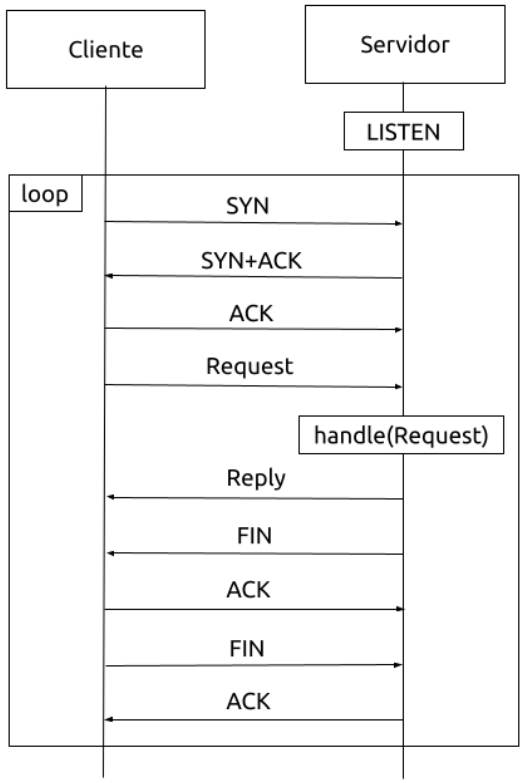
## Arquitectura Cliente-Servidor Secuencial

En esta arquitectura, se utiliza un servidor que atiende peticiones de forma secuencial, procesándolas una a una. Cuando varias peticiones llegan simultáneamente, el servidor las procesa una por una. Esta arquitectura es adecuada cuando los recursos hardware son limitados. Para usarla se invoca a *handleClient()* cada vez que llega una petición del cliente de esta manera:

```
for {
	// Accept incoming client connections
	conn, err := listener.Accept()
	com.CheckError(err)

	// Handle client
	switch mode {
		case 0: // Client - Secuential server
			handleClient(conn)
		. . .
	}
}
```

El diagrama secuencial para esta arquitectura es el siguiente:



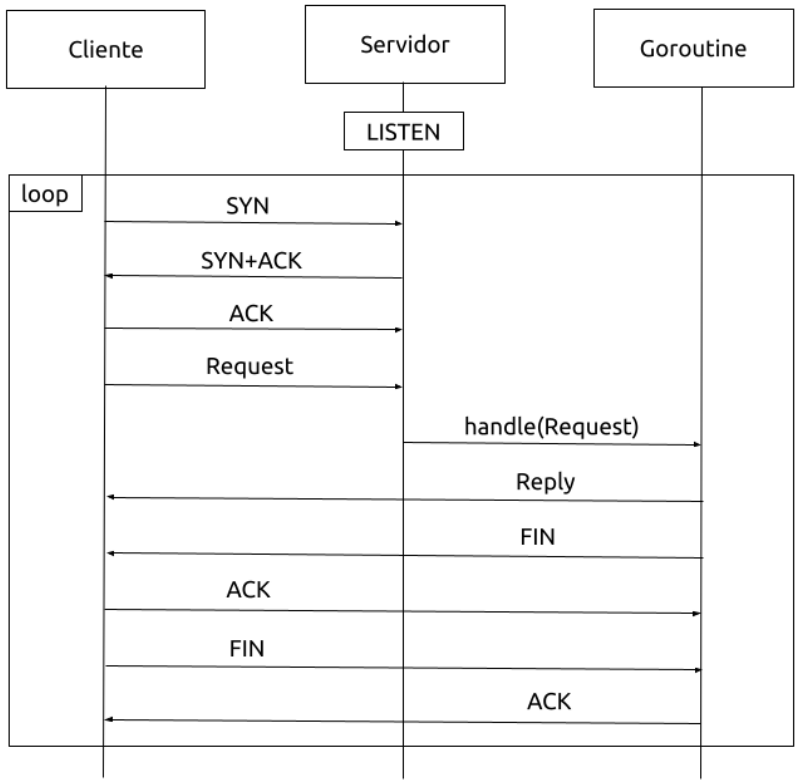

## Arquitectura Cliente-Servidor Concurrente con Goroutines

En esta arquitectura, el servidor es capaz de atender múltiples peticiones en paralelo utilizando el modelo de concurrencia de Goroutines. Cuando una solicitud llega, se crea una nueva Goroutine para procesarla de manera concurrente como se puede ver en el código de abajo. Esto reduce los tiempos de espera de los clientes y permite una mejor utilización de los recursos hardware disponibles.

```
for {
	// Accept incoming client connections
	conn, err := listener.Accept()
	com.CheckError(err)

	// Handle client
	switch mode {
		. . .
		case 1: // Concurrent server with goroutines
			go handleClient(conn)
		. . .
	}
}
```

La diagrama secuencial de esta arquitectura es la siguiente:




## Arquitectura Cliente-Servidor Concurrente con Pool de Goroutines

En esta variante de la arquitectura cliente-servidor concurrente, se utiliza un conjunto fijo de Goroutines que se denomina "pool de Goroutines". Estas Goroutines atienden peticiones y pueden reutilizarse. Las solicitudes se envían a través de canales síncronos, lo que permite una gestión eficiente de las tareas y una comunicación entre las Goroutines.

Primero creamos un canal del que cada *goroutine* irá atendiendo las peticiones de los clientes y después creamos el pool de goroutines:

```
// Create a channel for incoming client connections
clientJobs = make(chan net.Conn)

// Create worker goroutines
for i := 0; i < maxWorkers; i++ {
	go routine(clientJobs)
}
```

Cada *goroutine* irá leyendo del canal clientJobs:

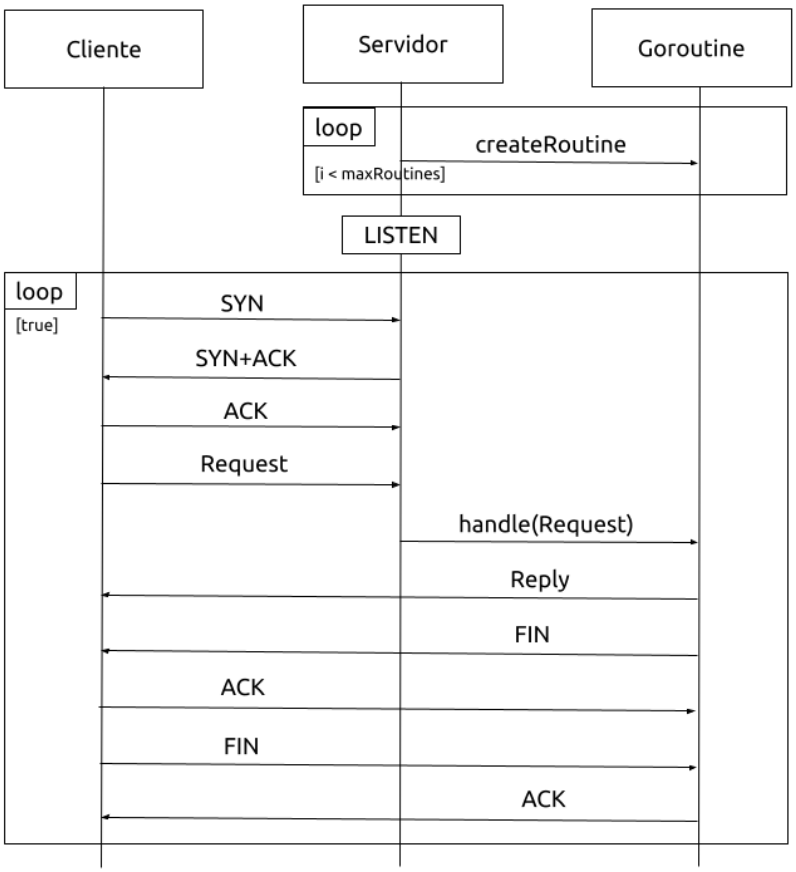
```
func routine(jobs <-chan net.Conn) {
	for conn := range jobs {
		handleClient(conn)
	}
}
```

Cada conexión aceptada es agregada al canal clientJobs:

```
for {
      // Accept incoming client connections
      conn, err := listener.Accept()
      com.CheckError(err)

      // Handle client
      switch mode {
            . . .
            case 2: // Fixed goroutine pool
                  clientJobs <- conn
            . . .
      }
}
```

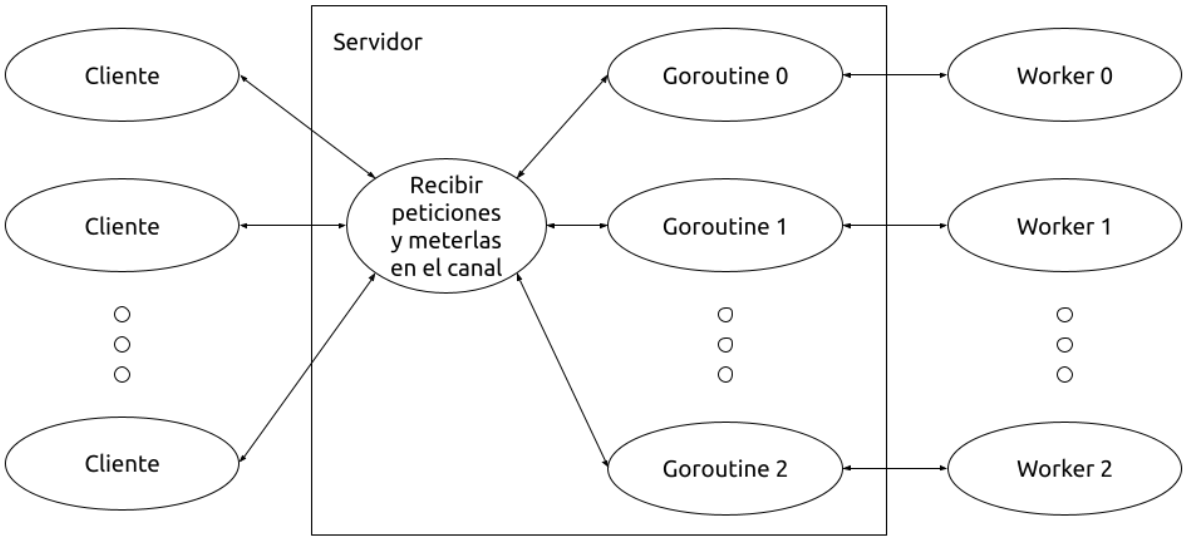
El diagrama de secuencia de esta arquitectura es el siguiente:



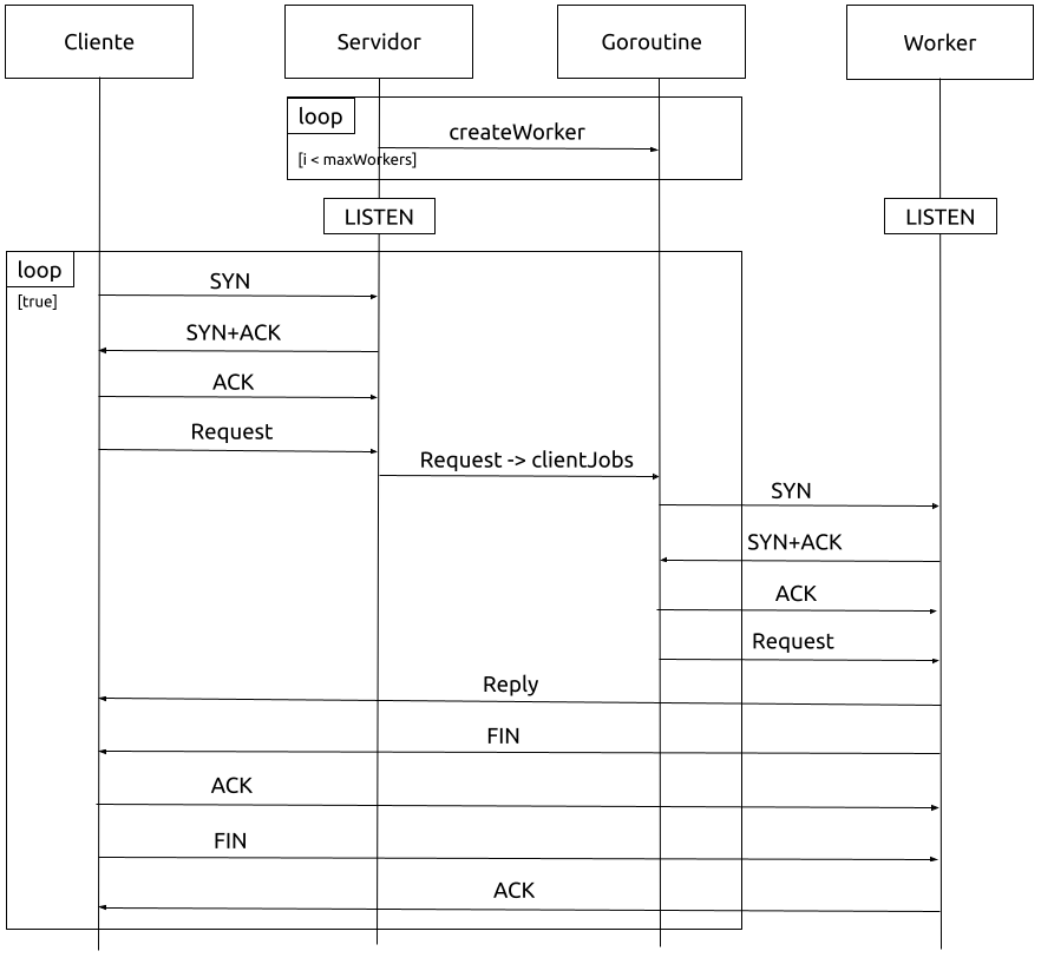

## Arquitectura Máster-Worker

La arquitectura Máster-Worker se basa en el modelo cliente-servidor concurrente con un pool de Goroutines, pero con la diferencia de que en esta arquitectura no se gastan recursos del propio ordenador ya que se realiza en remoto e interactúan las gorrutines con otras máquinas. El Máster coordina las tareas y distribuye el trabajo a los trabajadores.

Para esta arquitectura, se muestra un diagrama de comunicación entre los distintos componentes que participan en el proceso (cliente, servidor y *workers*):




El diagrama de secuencia es el siguiente:



# VALIDACIÓN EXPERIMENTAL

Para calcular de manera experimental cuál es el tiempo de ejecución efectivo (T) de una tarea en cada una de las arquitecturas, hemos medido cuánto tiempo transcurre desde que se envía una petición desde el cliente hasta que se recibe su respuesta. En la tabla se muestran los tiempos en milisegundos para cada una de las arquitecturas en 10 tests distintos y su media.

| | **T1** | **T2** | **T3** | **T4** | **T5** | **T6** | **T7** | **T8** | **T9** | **T10** | **Media** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Secuencial** | 77.0 | 65.9 | 54.8 | 43.6 | 32.5 | 21.1 | 76.5 | 65.4 | 54.0 | 42.9 | 45.952 |
| **Concurrente** | 24.9 | 24.0 | 20.4 | 24.4 | 22.5 | 16.6 | 30.2 | 24.4 | 25.5 | 22.4 | 23.018 |
| **Pool** | 34.1 | 29.2 | 26.4 | 22.3 | 25.0 | 20.4 | 30.2 | 22.5 | 24.9 | 21.8 | 25.066 |
| **Máster-Worker** | 35.5 | 35.5 | 33.9 | 25.1 | 24.3 | 24.1 | 40.8 | 38.4 | 37.8 | 26.2 | 30.909 |

Como se puede apreciar, el salto de secuencial a concurrente es bastante significativo, ya que se consigue reducir casi a la mitad el tiempo de procesamiento de una tarea. Sin embargo, al hacer un pool fijo de *goroutines*, 10 como máximo, el tiempo de ejecución efectivo aumenta ligeramente. Finalmente, se puede observar el tiempo de la arquitectura Máster-Worker que es superior a la concurrente y a la del pool de *goroutines* pero inferior a la secuencial; creemos que el aumento de tiempo se debe al tiempo que se pierde en establecer conexión con los *workers*.