LECTORES Y ESCRITORES DISTRIBUIDO

SISTEMAS DISTRIBUIDOS - PRÁCTICA 1

Lizer Bernad Ferrando - 779035
Lucía Morales Rosa – 81690

Descripción del problema

La aplicación consiste en un sistema de comunicación entre dos procesos que utiliza tanto TCP como UDP para evaluar el rendimiento de ambos protocolos en diferentes escenarios. Los procesos pueden ejecutarse en una misma máquina o en dos máquinas diferentes, conectadas en una red local. El propósito principal es medir los tiempos de respuesta, la gestión de conexiones fallidas y la latencia de transmisión de datos en ambos protocolos, destacando cómo la red afecta su rendimiento en situaciones controladas.

Respecto a los recursos computacionales, la aplicación se ejecuta en un entorno formado por un conjunto de *Raspberry Pi 4 Model B* configuradas como un *cluster* en una red local. Este entorno permite probar el sistema en condiciones reales de red, donde la proximidad física de las máquinas minimiza las fluctuaciones externas y garantiza un análisis más preciso del comportamiento de los protocolos bajo distintas cargas de red. El sistema evalúa tanto el desempeño en una sola máquina como la interacción entre procesos distribuidos en múltiples nodos del *cluster*, brindando una visión clara de los impactos de latencia y congestión de la red sobre los tiempos de respuesta de TCP y UDP.

Análisis de Prestaciones de Red

La conexión TCP muestra un rendimiento similar tanto cuando se ejecuta en una única máquina como en máquinas distintas. Sin embargo, se observa un pequeño incremento en el tiempo que tarda en determinar que la conexión ha sido fallida debido a la ausencia del servidor cuando se utilizan máquinas diferentes. Este comportamiento probablemente se debe al tiempo de propagación del mensaje a través de la red, a pesar de que las máquinas están ubicadas relativamente cerca.

Por otro lado, en las conexiones UDP la diferencia es más significativa, siendo notablemente más lenta al utilizar máquinas distintas. Esto puede deberse a que, a diferencia de TCP, UDP no establece una conexión previa, lo que lo hace más vulnerable a las condiciones de la red, como la congestión. Cabe destacar que, en caso de que no haya un servidor disponible, UDP no experimenta tiempo de espera, ya que no realiza un *handshake* o saludo con el receptor antes de enviar los paquetes. Por lo tanto, no hay un periodo de espera para determinar el fallo de la conexión, como ocurre en TCP.

TCP (µs)				UDP (µs)			
Misma máquina		Distintas máquinas		Misma máquina		Distintas máquinas	
Sin servidor	Con servidor	Sin servidor	Con servidor	Sin servidor	Con servidor	Sin servidor	Con servidor
0,696	1,10	0,727	0,769	-	0,478	-	0,655
0,702	1,15	0,865	1,17	-	0,321	-	0,622
0,800	0,719	0,734	1,17	-	0,337	-	0,525
0,791	1,11	0,733	1,20	-	0,413	-	0,669
0,709	1,22	0,728	1,13	-	0,368	-	0,648
0,724	1,21	0,709	1,10	-	0,428	-	0,606
Valores medios							
0,737	1,08	0,749	1,08	-	0,39	-	0,62
Desviación típica							
0,046	0,186	0,057	0,161	-	0,060	-	0,052

Sección Sincronización Barrera Distribuida

Este programa en Go implementa una barrera de sincronización entre varios procesos utilizando conexiones TCP. Cada proceso carga una lista de direcciones de otros procesos (*endpoints*) desde un archivo, lo que le permite conectarse y enviar mensajes de notificación entre sí. El objetivo es asegurarse de que todos los procesos lleguen a un mismo punto de sincronización antes de proceder.

Cada proceso se convierte en un servidor, escuchando en su propio *endpoint* mientras envía simultáneamente notificaciones a los demás procesos indicando que ha alcanzado la barrera. Cuando un proceso recibe una notificación de otro, actualiza un registro de mensajes recibidos. Una vez todos los procesos han recibido los mensajes de los demás, se desencadena una señal a través del canal "*barrierChan*" que indica que la barrera ha sido alcanzada por todos.

El cierre del programa se maneja de manera controlada. Una vez alcanzada la barrera, el canal "*quitChannel*" se cierra, lo que detiene la escucha de nuevas conexiones. Para garantizar que todas las *goroutines* de envío de notificaciones hayan finalizado correctamente, se utiliza un mecanismo de sincronización con *WaitGroup*, que asegura que el programa no se cierre prematuramente.

Entre los cambios más importantes introducidos en esta versión del programa, destaca la incorporación del canal "*barrierChan*", que maneja la sincronización entre procesos al llegar a la barrera. Además, el uso de *WaitGroup* permite coordinar el cierre adecuado de las *goroutines* encargadas de la notificación entre procesos, evitando condiciones de carrera y asegurando un cierre ordenado. También se ha implementado un mecanismo de cierre controlado del *listener*, que permite detener de manera segura la escucha de nuevas conexiones una vez que todos los procesos han alcanzado la barrera.

En la Ilustración 1 se puede observar el diagrama de secuencia diseñado para llevar a cabo la barrera distribuida. En el diagrama se puede ver como todos los procesos se envían mensajes a los demás procesos. Cuando un proceso ha recibido mensajes de todos los demás, envía otro mensaje notificando que ha llegado a la barrera.

Finalmente, cuando todos los procesos han notificado su llegada a la barrera podrán seguir su ejecución.

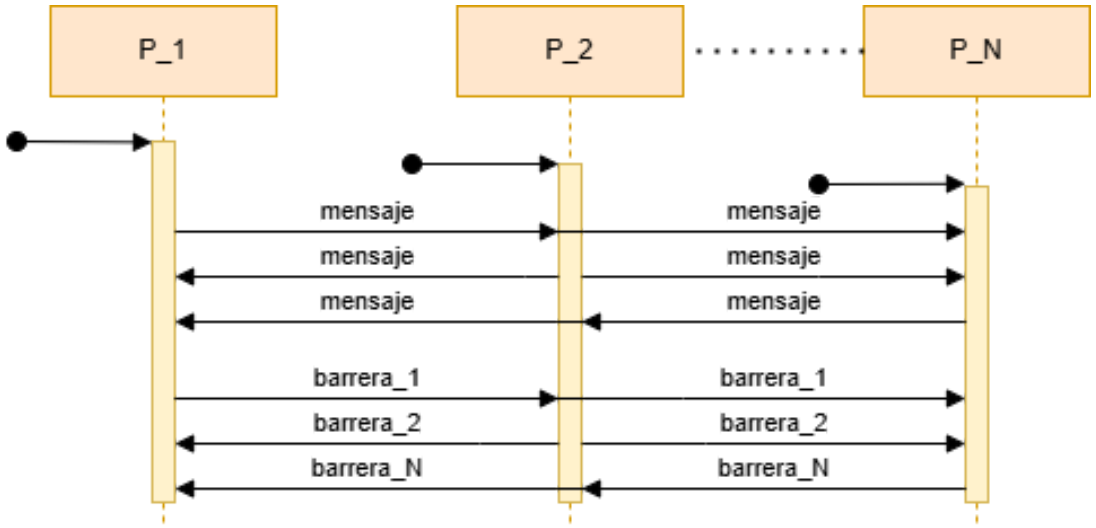

Ilustración 1: Diagrama de secuencia de una barrera distribuida

Diseño de las 4 arquitecturas

Para esta práctica, se han diseñado e implementado las siguientes variantes de la arquitectura cliente-servidor.

Cliente-servidor secuencial

Se ha comenzado con una arquitectura cliente-servidor secuencial donde el servidor atiende las peticiones del cliente de forma secuencial.

Para ello se ha diseñado el diagrama de secuencia de la Ilustración 2. Este diagrama, al igual que la implementación es muy sencillo ya que, cuando el servidor recibe una petición, realiza las operaciones necesarias para calcular la respuesta y se la envía al cliente.

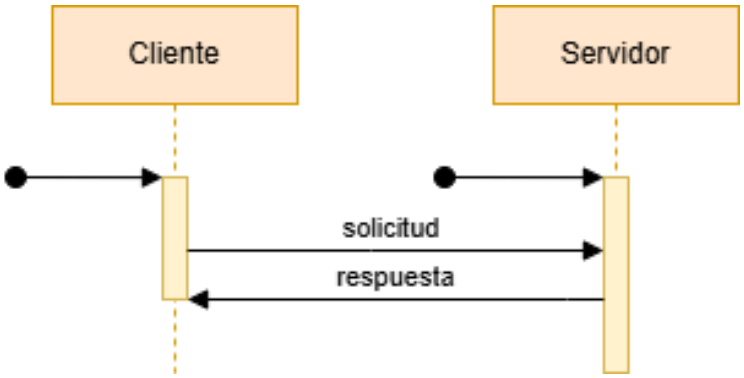

Ilustración 2: Diagrama de secuencia del cliente servidor secuencial

Cliente-servidor concurrente

Para la ejecución de un cliente-servidor concurrente se han empleado las *goroutines* propias del lenguaje Go. De esta forma, cada vez que se recibe una petición de un cliente, se crea una *goroutine* que la trata consiguiendo así realizar varias tareas de forma simultánea y reducir en gran medida los costes temporales de ejecución.

El diseño de esta arquitectura puede observarse en la Ilustración 3.

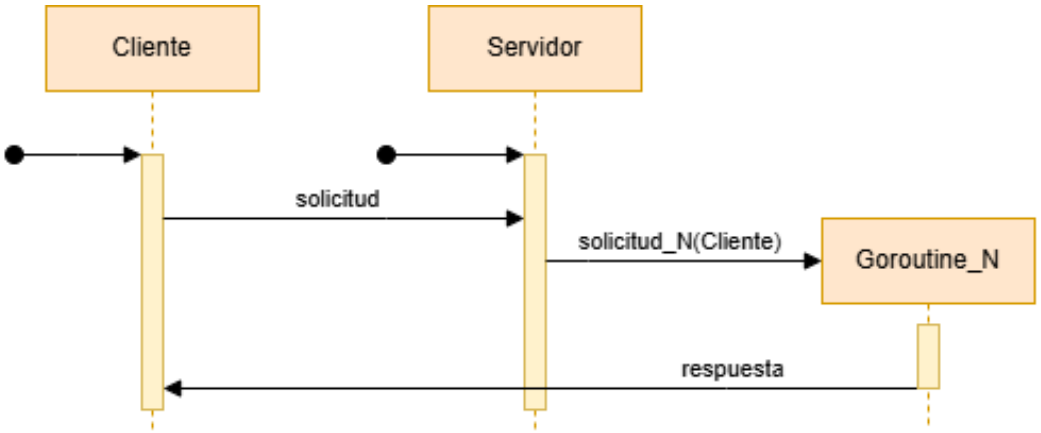

Ilustración 3: Diagrama de secuencia del cliente servidor concurrente

Cliente-servidor con un pool de goroutines

El diseño e implementación de esta versión de cliente servidor es muy similar al anterior. El único cambio es que, en lugar de tener la posibilidad de ejecutar infinitas *goroutines* como en el concurrente, se creará un número de *goroutines* determinado entre las que se repartirán las peticiones de los clientes.

En el diagrama mostrado en la Ilustración 4 se puede observar la creación de "maxWorkers" antes de comenzar a aceptar peticiones de los clientes.

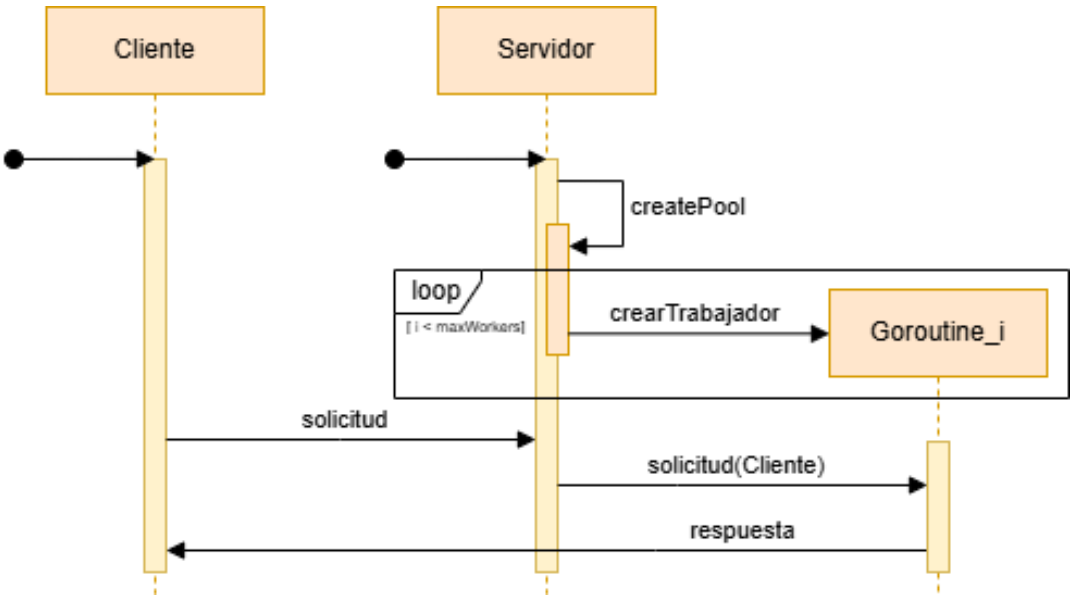

Ilustración 4: Diagrama de secuencia de un cliente servidor con un pool de goroutines

Arquitectura *Master-Worker*

En esta arquitectura se va a hacer uso de máquinas remotas, de forma que no se agotan recursos de la máquina principal. Esta arquitectura por tanto contará con tres tipos de procesos: Cliente, Servidor y Trabajadores.

Además, para repartir la carga de trabajo entre los distintos Trabajadores se ha decidido emplear la política de *Round Robin*, es decir, que se asignarán en orden según el identificador de proceso de cada Trabajador.

Esta arquitectura se puede ver representada en el diagrama de secuencia de la Ilustración 5.

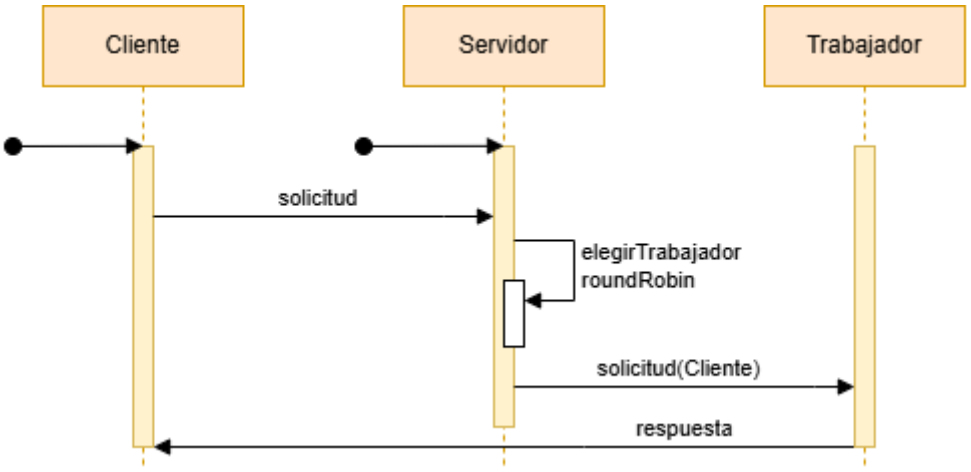

Ilustración 5: Diagrama de secuencia de la arquitectura Master-Worker

Además, para facilitar la ejecución de este programa se ha creado un fichero ejecutable llamado "encenderMaquinas.sh" que lee el contenido de un fichero de texto pasado como parámetro y enciende los Trabajadores cuyas ip y puertos estén especificados en el fichero de texto.