SISTEMAS DISTRIBUIDOS

PRÁCTICA 1

874055 - Ariana Porroche Llorén
871627 - Irene Pascual Albericio

ÍNDICE

Introducción **2**
Análisis de Prestaciones de Red **2**
Sincronización Barrera Distribuida **3**
Diseño de las 3 arquitecturas **5**
1. Cliente-Servidor concurrente con una Goroutine por petición 5
2. Cliente-Servidor concurrente con un pool fijo de Goroutines 6
3. Máster-Worker 7
Análisis teórico de la carga de trabajo **9**
Validación experimental **9**

Introducción

En esta práctica, el objetivo es diseñar y probar el envío de mensajes y el establecimiento de conexiones entre máquinas usando un programa llamado Go.
El código utilizado para los ejercicios 2 y 3, consiste en el cálculo de números primos dentro de un rango. Para ir realizando las diferentes pruebas, tenemos acceso a 20 máquinas y a unos determinados puertos. Estas máquinas están conectadas entre sí, por lo que podemos observar cómo se reparten las tareas, se envían los datos, se ejecutan…
Para establecer conexión de forma remota, hacemos uso del comando “SSH”.

Análisis de Prestaciones de Red

Hemos hecho varias pruebas midiendo los tiempos para establecer conexiones con TCP y UDP. Estas pruebas tuvieron lugar en la misma máquina y entre distintas máquinas. A continuación los resultados obtenidos:

TCP:

Prueba	**Tiempo**
Dial falla, misma máquina	511.184 microsegundos
Dial falla, distinta máquina	607.091 microsegundos
Dial funciona, misma máquina	1.168016 ms
Dial funciona, distinta máquina	1.260256 ms

UDP:

Prueba	**Tiempo**
Misma máquina	309.219 microsegundos
Distinta máquina	600.199 microsegundos

Los resultados muestran que en TCP, cuando el servidor falla, la conexión se corta rápido. En cambio, cuando se establece la conexión, el tiempo aumenta, especialmente si están en distintas máquinas ya que hay más retraso por la red (latencia de la red).
Con UDP, observamos que es mucho más rápido en la misma máquina, pero al igual que en la TCP, aumenta cuando es en máquinas distintas.
En cuanto a la diferencia en tiempo entre TCP Y UDP, esto se debe a que TCP es más lento porque es más seguro, ya que antes de enviar los datos, primero debe establecer una conexión y mantener una especie de diálogo para asegurarse que los datos van a llegar correctamente.
En cambio, UDP es más rápido porque simplemente envía los datos sin verificar si llegaron o si están en el orden correcto. Por tanto, UDP es más rápido pero menos seguro al mismo tiempo.

Sincronización Barrera Distribuida

Para este apartado, hemos modificado el código que nos proporcionaban para poder implementar una barrera distribuida, donde varios procesos deben esperar a que todos hayan alcanzado un punto en común (la barrera).

La función **readEndPoints** simplemente se encarga de leer un archivo que contiene las direcciones y devuelve un array donde cada componente es una dirección leída.

La función **getEndPoints** tiene como objetivo devolver el vector nombrado anteriormente (ya que desde aquí se llama a la función readEndPoints), y el número de línea donde aparece la máquina que está ejecutando (se comprueba que sea mayor que 0 y que esté dentro del rango de número de líneas del archivo).

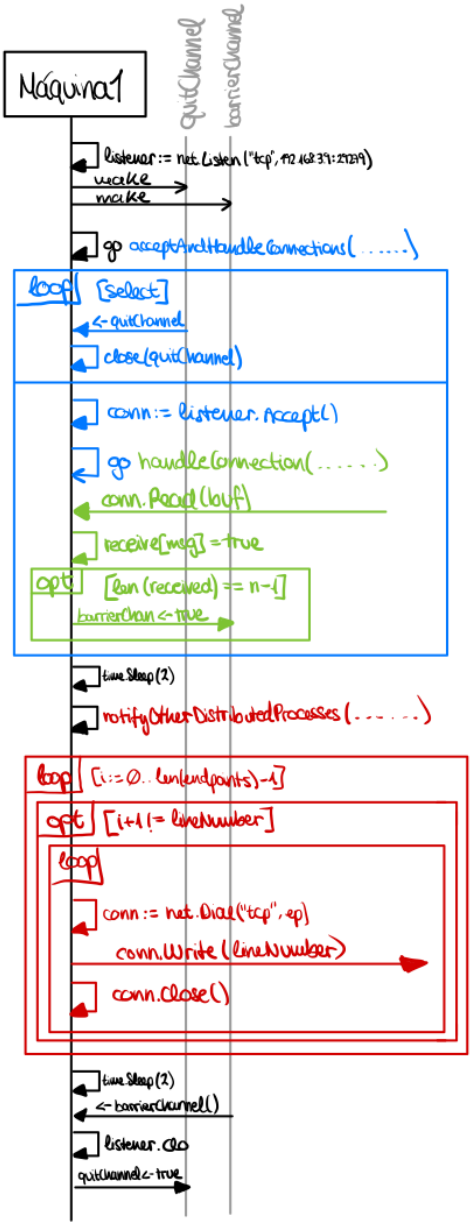
Con respecto a **handleConnection**, maneja cada conexión entrante de otros procesos. Hace uso de un mutex para acceder a la "sección crítica", donde guarda el mensaje recibido. Cuando ya se han recibido los mensajes de todos los otros procesos (n-1), se envía la señal a barrierChan, para indicar que la barrera ya se ha alcanzado.

La función **acceptAndHandleConnections** acepta conexiones entrantes en bucle mediante el listener, y cuando la señal de que la barrera ha sido alcanzada se activa, se detiene el listener. Cada conexión entrante se realiza con una goroutine separada para poder llevar múltiples conexiones en paralelo.

La función **notifyOtherDistributedProcesses** sirve para notificar a los otros procesos que el proceso actual ha llegado a la barrera.

En el **main**, lo primero que se hace es comprobar que los argumentos se hayan pasado correctamente. Después, se leen las direcciones de todos los procesos y se identifica cuál corresponde al proceso actual. Luego, se crea un listener TCP que escucha señales de las otras máquinas. También se configuran los canales necesarios: uno para detener el listener (quitChannel) y otro para controlar la sincronización en la barrera (barrierChan). A continuación, se lanza una goroutine que acepta conexiones de otros procesos y las maneja en paralelo. Después, el proceso avisa a los demás que ha llegado a la barrera y se queda esperando a que todos los procesos lleguen al mismo punto antes de continuar.

Diagrama de secuencia:



Diseño de las 3 arquitecturas

1. Cliente-Servidor concurrente con una Goroutine por petición

Resumen:

En esta versión, el cliente se conecta al servidor y cada vez que hace una petición, se crea una Goroutine para procesar su solicitud. El servidor escucha en un puerto y cuando recibe la petición, comienza a calcular los número primos solicitados y envía la respuesta. Todo esto ocurre en una goroutine independiente por cada conexión. El problema de esta versión es que si hay muchas peticiones a la vez, se sobrecargaría el servidor ya que tendría que crear muchas Goroutines.

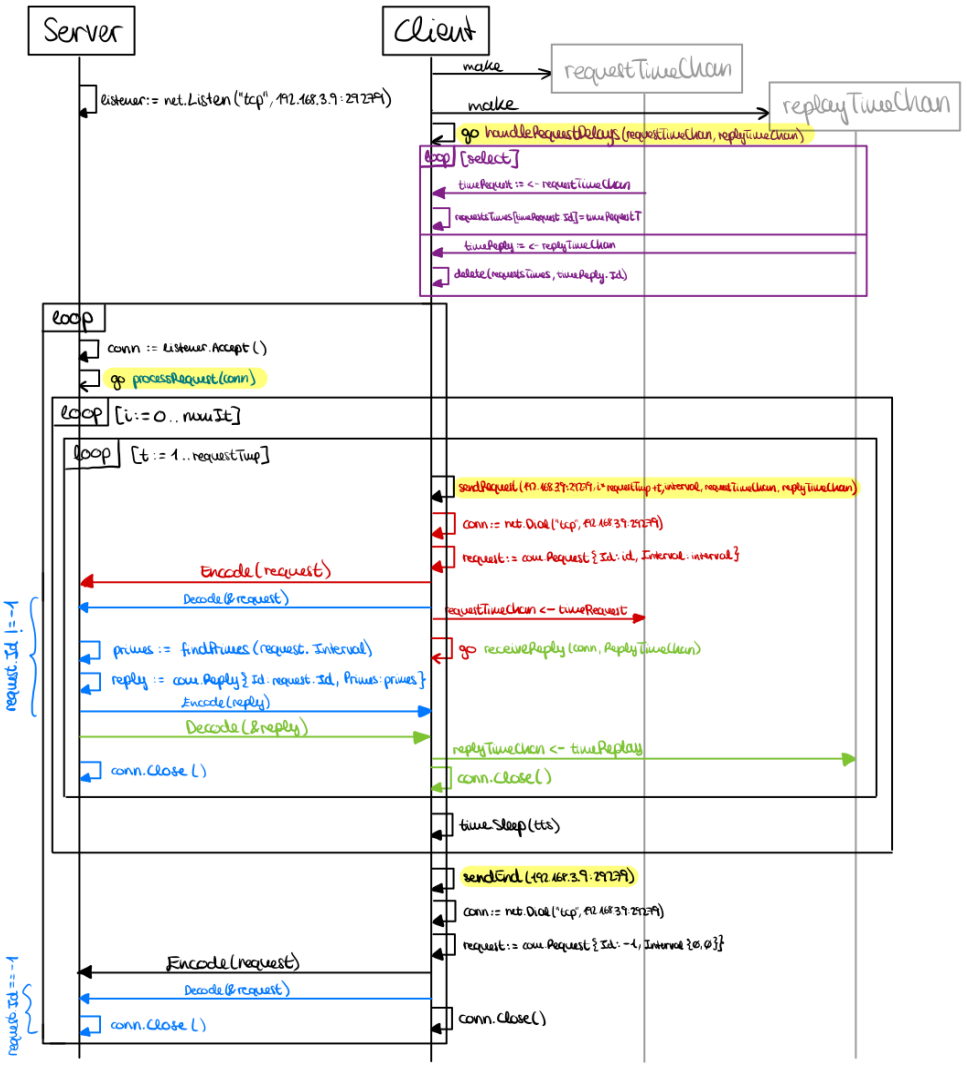
Ejecutar:

Para ejecutar esta arquitectura, abrimos dos terminales en la máquina 192.168.3.9, por ejemplo. Primero, para lanzar el servidor, nos colocamos en el directorio "/misc/alumnos/sd/sd2425/a874055/practica1/cmd/server-draft" y ejecutamos el comando "go run server1.go 192.168.3.9:29279". El servidor se pondrá a escuchar en el puerto 29279. Para el cliente, nos colocamos en el directorio "/misc/alumnos/sd/sd2425/a874055/practica1/cmd/client" y ejecutamos "go run main.go 192.168.3.9:29279". El cliente se conectará al servidor y enviará petición por petición al servidor, obteniendo una respuesta y calculando el tiempo transcurrido entre el envío de la petición y el recibimiento de la respuesta. Estas respuestas no se recibirán secuencialmente, ya que al crear Goroutines aplicamos concurrencia.

Pruebas:

Hemos probado a ejecutar el cliente y el servidor en la misma máquina como hemos comentado anteriormente. Pero también podemos ejecutar el cliente en una máquina distinta que el servidor. Así observamos que el tiempo de delay entre la petición y la respuesta aumenta con respecto a cuando ejecutamos cliente y servidor en la misma máquina.

Diagrama de secuencia:



2. Cliente-Servidor concurrente con un pool fijo de Goroutines

Resumen:

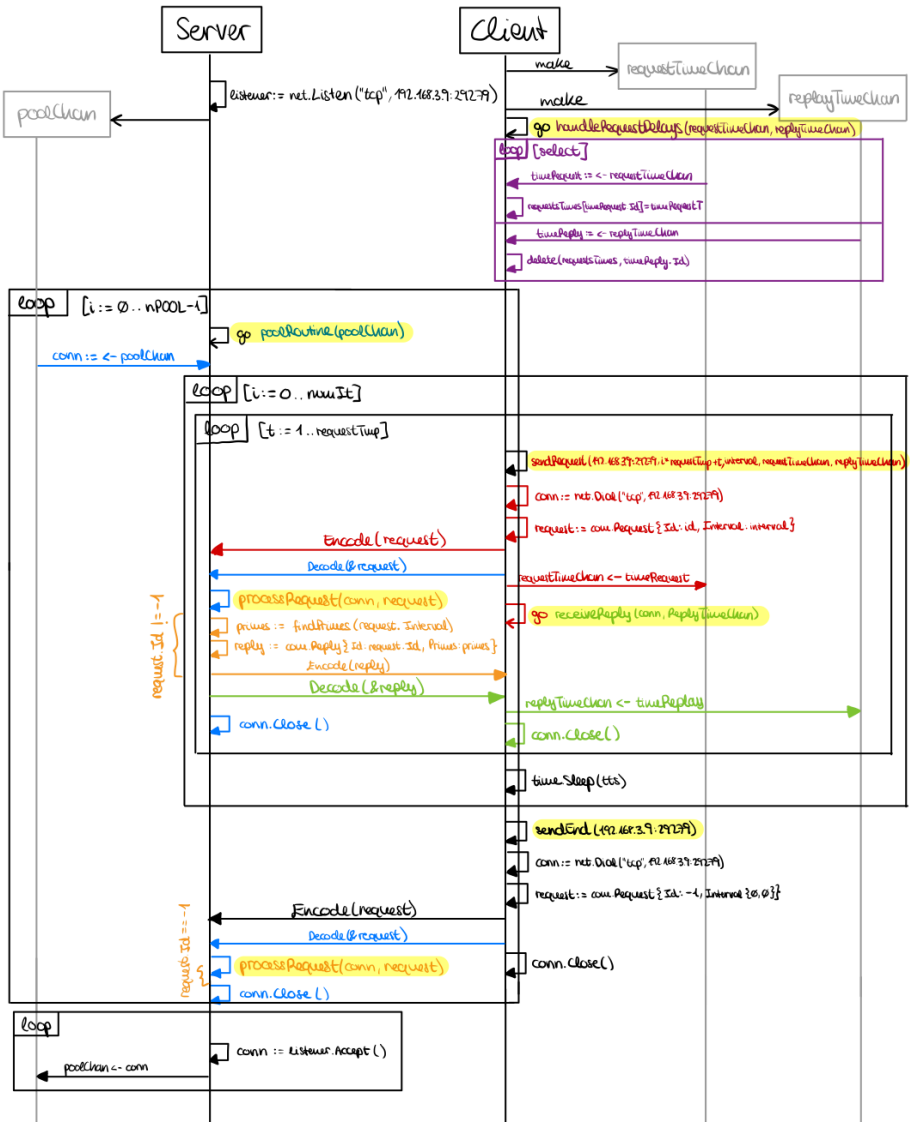
Esta segunda versión de la arquitectura cliente-servidor concurrente utilizando un pool de Goroutines, mejora el problema anterior de la sobrecarga. En este caso, en lugar de crear una Goroutine por cada petición del cliente, tenemos un pool de Goroutines (en nuestro código 6 Goroutines), que son las que atienden las peticiones de los clientes. El servidor, cuando recibe una petición del cliente, la manda al canal de tareas. Las Goroutines van cogiendo estas peticiones del canal y las van atendiendo a medida que se quedan libres.

Ejecutar:

Esta arquitectura se ejecutaría igual que la anterior. Primero abrimos dos terminales, por ejemplo en la máquina 192.168.3.9. Para el servidor, nos colocamos en el directorio “/misc/alumnos/sd/sd2425/a874055/practica1/cmd/server-draft” y ejecutamos el comando “go run server2.go 192.168.3.9:29279”. Para el cliente, nos colocamos en el directorio “/misc/alumnos/sd/sd2425/a874055/practica1/cmd/client” y ejecutamos el comando “go run main.go 192.168.3.9:29279”.

Pruebas:
Al igual que en la anterior arquitectura, si ejecutamos el servidor y el cliente en una máquina distinta, notamos como el delay entre la petición del cliente y la respuesta del servidor aumenta con respecto a ejecutarlos en la misma máquina.

Diagrama de secuencia:



3. Máster-Worker

Resumen:
En la arquitectura máster-worker, el máster es el encargado de coordinar todo. Se pone a escuchar en un puerto, esperando a que se le conecte algún cliente y le envíe peticiones. Cuando le llegue alguna petición, la pondrá en el canal de peticiones. A su vez, se encarga de lanzar los workers que le hemos indicado en el fichero pasado como parámetro, iniciándolos remotamente mediante el comando “ssh”. Cada petición que extrae del canal, se la pasa al worker para que la atienda y nos devuelva un resultado. Este resultado lo reenvía al cliente. Así conseguimos reducir la carga de trabajo del máster, limitándolo a un rol más de coordinador y delegando estas tareas en los workers.

Ejecutar:
Esta arquitectura se ejecuta en 2 terminales, en nuestro caso hemos seleccionado la máquina 192.168.3.9. Para lanzar el máster, nos colocamos en el directorio "/misc/alumnos/sd/sd2425/a874055/practica1/cmd/server-draft" y ejecutamos el comando "go run master3.go 192.168.3.9:29279 fichWorkers". Al lanzar el máster, este se encargará de lanzar automáticamente los workers (fichero worker3.go). Entonces lanzamos el cliente, para lo que vamos al directorio "/misc/alumnos/sd/sd2425/a874055/practica1/cmd/client" y ejecutamos "go run main.go 192.168.3.9:29279". Según el fichero fichWorkers que hemos definido, el máster lanzará 2 workers en las direcciones 192.168.3.10:29270 y 192.168.3.11:29271.

Pruebas:
Hemos realizado las pruebas en las máquinas indicadas anteriormente.

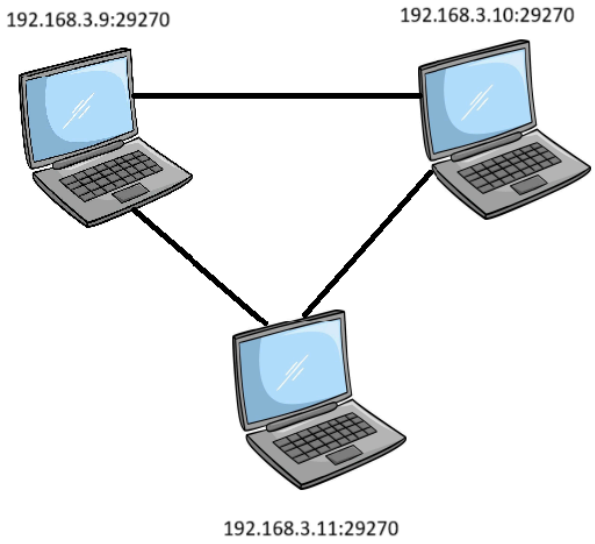
Diagrama de secuencia:

Análisis teórico de la carga de trabajo

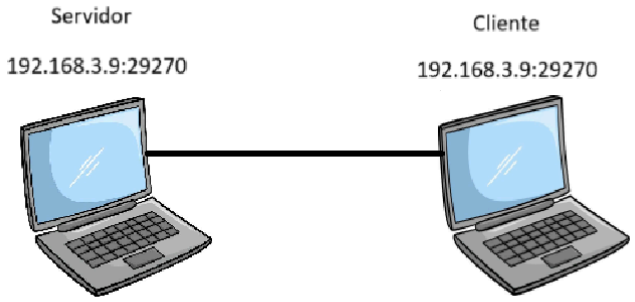
Con respecto a las 3 arquitecturas propuestas, cabe destacar que la primera es la que mayor carga de trabajo tiene, ya que crea una Goroutine por cada petición. La segunda arquitectura, reduce esta carga creando un pool de Goroutines, por lo que se reparten el trabajo entre las 6 Goroutines que hemos creado en nuestro caso. Por último, el máster-worker es el que más mensajes distribuye entre el cliente, el máster y los workers, pero delega el trabajo en los workers, que se ejecutan en máquinas distintas.

Validación experimental

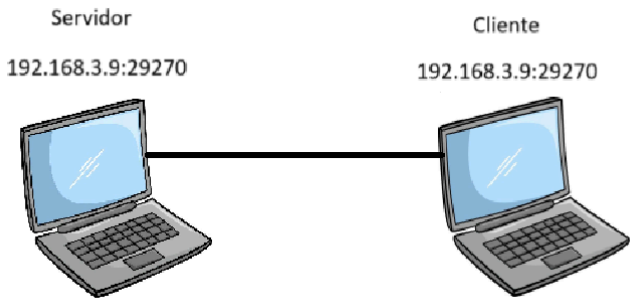
Hemos probado cada una de las arquitecturas explicadas anteriormente, definiendo nuestra propia infraestructura de red.
Barrera:



Cliente-servidor concurrente con una Goroutine por petición:



Cliente-servidor concurrente con un pool de Goroutines:



Máster-worker: