30221/39521 - Sistemas Distribuidos
Universidad de Zaragoza, curso 2024/2025

Práctica 1: **Conceptos y Mecanismos Básicos**

Objetivos y requisitos

Objetivos

Uno de los problema básicos de los sistemas distribuidos consiste en asignar las 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. Por tanto, es fundamental conocer los recursos computacionales para poder construir una arquitectura software distribuida de forma adecuada. En esta práctica vamos a analizar y diseñar arquitecturas cliente servidor y master-worker, para una aplicación muy sencilla que calcula los números primos en un intervalo dado. Para ello, analizaremos las características de la aplicación y los recursos computacionales, fundamentalmente el cluster de Raspberry Pis, del Laboratorio L1.02. Estos son los objetivos de esta práctica en particular:

- Familiarizarse con el lenguaje de programación Golang
- Familiarización con la arquitectura cliente servidor, secuencial y concurrente.
- Familiarización con la arquitectura master-worker.
- Análisis de la Calidad de Servicio (Quality of Service, QoS) en sistemas distribuidos y gestión de recursos computacionales

Requisitos

- Golang versión ¿= go1.22 linux/ARM64/amd64[1]

[1] Disponible en `https://golang.org/doc/install`

- Seguir, de forma rigurosa, la metodología de programación estructurada y organizada, con nombres significativos (tipos, variables, parámetros, funciones), y con funciones cortas (no más de 12 líneas de instrucciones)
- El protocolo *ssh* para ejecutar comandos remotos
- Los ficheros de apoyo, que acompañan al guion, con un esqueleto de código fuente en el que inspirarse y apoyarse.

1. Ejercicios

En esta práctica tenéis que realizar los ejercicios siguientes (no necesariamente en este orden):

Ejercicio 1.

Obligatorio
Análisis de las prestaciones de red descritas en la Sección 2.2:

- ejecutar repetidamente los escenarios descritos en la Sección 2.2, recoger los tiempos derivados de la ejecución en todos los escenarios y crear una tabla para la memoria de esta práctica.
- explicar a qué actividades se debe el coste de ejecutar la operación Dial de TCP en Golang y estudiar la variación en el tiempo de ejecución, cuando Dial falla, cuando funciona correctamente y cuando el cliente y el servidor están en la misma máquina y en distinta máquina.
- explicar a qué actividades se debe el coste de ejecutar la transmisión en el escenario de UDP.

Entrega: Tabla con los tiempos y explicación de a qué se deben, considerando TCP y UDP

Ejercicio 2.

Obligatorio
En el código adjunto a esta memoria hay un esqueleto que implementa una barrera distribuida. Este ejercicio consiste en completar el código.

Entrega: Diagrama de secuencia y fichero de código completo barrier.go

Ejercicio 3.

Obligatorio
Diseñar (diagramas de secuencia) e Implementar (en Go) las 3 arquitecturas software descritas en la Sección 2.3:

- cliente servidor concurrente creando una Goroutine por petición
- cliente servidor concurrente con un pool fijo de Goroutines
- máster-worker. En este caso, el máster, mediante comandos ssh, arrancará los workers antes de operar. De manera que no los tendréis que arrancar vosotros manualmente. El máster obtendrá de un fichero un listado de máquinas (IPs + puertos) donde puede lanzar workers.

En todas ellas hay que utilizar el paquete gob para serializar los mensajes.

Entrega: Diagramas de secuencia y código de servidores, máster y worker

Ejercicio 4.

Opcional
Analizar para qué carga de trabajo puede operar cada una de ellas sin violar el QoS ($t_{ex} + t_{xon} + t_o < 2 * t_{ex}$). Cada petición de cliente (tarea) consiste en encontrar todos los números primos existentes entre [1000, 70000]. La carga de trabajo se mide en términos de número de peticiones (tareas) que un cliente envía al servidor por unidad de tiempo (segundo). Se espera que para cada arquitectura realicéis un análisis teórico de lo que es esperable, teniendo en cuenta la carga de trabajo, el software en ejecución y el hardware disponible en cada caso.
Entrega: en la memoria, incluir análisis

Ejercicio 5.

Opcional
Para cada arquitectura, ejecutar una serie de experimentos que demuestren que el análisis realizado es correcto. Notar que cuando se ejecuta el cliente, este escribe por la salida estándar por cada tarea: su identificador y el tiempo total de ejecución visto por el cliente. En el código proporcionado puede encontrarse un script de gnuplot [a] que permite visualizar gráficamente que efectivamente no se viola el QoS.
Entrega: en la memoria, incluir descripción experimental y figuras

[a] `http://www.gnuplot.info/`

Ejercicio 6.

Obligatorio
Escribir una memoria donde se muestren todos los resultados y que obligatoriamente tendrá estas secciones:

- Sección de Introducción o Descripción del Problema, descripción de la aplicación y de los recursos computacionales disponibles
- Sección Análisis de Prestaciones de Red con tabla de tiempos y explicación causas de resultados.
- Sección Sincronización Barrera Distribuida con explicación concisa y diagrama de secuencia.
- Sección de Diseño de las 4 arquitecturas (diagramas de secuencia, si fuera necesario añadir diagrama de componentes).
- Sección de análisis teórico de cuál es la carga de trabajo máxima que puede soportar cada arquitectura
- Sección de validación experimental donde se describen qué experimentos se han realizado en cada caso y donde se muestren las gráficas generadas mediante gnuplot.

Entrega: memoria en pdf (plantilla word o latex)

1.1. Instrucciones de Entrega

Deberéis entregar un fichero zip que contenga: (i) los fuentes para las 4 arquitecturas, y (ii) la memoria en pdf. La entrega se realizará a través de moodle en la actividad habilitada a tal efecto. **La fecha de entrega será no más tarde del día anterior anterior al comienzo de la siguiente práctica**.

2. Descripción del Problema de esta Práctica

Una de las características fundamentales de los sistemas distribuidos es la Calidad del Servicio. El tiempo de procesamiento y la red de comunicación juegan un papel importante en ese aspecto. En la primera parte de la práctica vamos a analizar y estudiar el tiempo necesario para establecer una conexión en TCP, así como la latencia, entendida como el tiempo necesario para enviar y recibir un paquete.

En la segunda parte de esta práctica, vamos a considerar distintas formas de organizar el código, lo que denominamos arquitecturas de sistemas distribuidos, que hacen que el throughput del sistema distribuido sea diferente.

2.1. Especificación Técnica de las Máquinas del L1.02

Para las prácticas de esta asignatura utilizaremos el cluster de Raspberry Pi 4 Model B del Laboratorio 1.02. Para pruebas intermedias, podéis utilizar también las 20 máquinas de sobremesa del laboratorio L1.02, con arquitectura hardware AMD64.

El procesador de una Raspberry Pi 4 Model B (ARM Cortex-A72) tiene 4 cores con 1 hilo por core, que nos van a permitir ejecutar, por lo tanto, hasta 4 instancias en paralelo de nuestra aplicación (la especificación técnica de la CPU podéis obtener de forma detallada si ejecutáis la instrucción lscpu en el sistema operativo Linux). Estas máquinas cuentan con 7.6 GB.

2.2. Estudio de la Red de Comunicación

Para estudiar las características de la red de comunicación que tienen las Raspberry Pis vamos a realizar distintos experimentos que debéis reflejar en la memoria. En los escenarios vamos a considerar los protocolos TCP y UDP, por tanto, para poder explicar las mediciones que obtengamos, tendremos que comprender cómo se comportan los distintos protocolos.

Se proporciona el código de un cliente TCP (en la carpeta network), que primero se conecta a un servidor que no está en marcha y luego lanzaremos el servidor para que interactúen. Se trata de medir el tiempo que tarda en devolver el error la llamada "Dial" en el primer caso y el tiempo que tarda "Dial" en el segundo caso. Comprobar si se obtienen los mismos valores cuando el cliente y el servidor están en la misma o en distinta máquina.

El segundo escenario se corresponde a un cliente y servidor UDP sencillos, se trata de medir el tiempo que le cuesta al cliente el envío de una letra y la recepción de otra letra. Comprobar si se obtienen los mismos valores cuando el cliente y el servidor están en la misma o en distinta máquina.

2.3. Modelo de Aplicación

El modelo de aplicación que utilizaremos en esta práctica es uno de los más sencillos posibles desde un punto de vista de los sistemas distribuidos. La aplicación consiste en una única tarea que recibe la entrada, proporciona una salida y termina. En particular, la aplicación de esta práctica consiste en encontrar los números primos dentro de un intervalo [1000,70000] dado como argumento. Es importante resaltar que al haber una única tarea, no hay interdependencias con otras tareas. Por eso, es uno de los tipos de aplicación más sencillos desde un punto de vista de los sistemas distribuidos. La aplicación es sobre todo intensiva en CPU, puede no requiere de mucha red de comunicación y no requiere de almacenamiento.

Junto con este enunciado, tenéis disponible unos ficheros de ayuda:

- Un cliente ya terminado y que no hay que cambiar que envía peticiones / tareas con un intervalo que siempre es el mismo [1000, 70000]. Simplemente lo invocaremos con distintos parámetros para que genere cargas de trabajo diferentes.

- Un servidor secuencial que contiene una función para resolver esta práctica. En particular: IsPrime y FindPrimes. La primera determina si un número entero dado es primo o no y devuelve true o false, respectivamente. La segunda toma como entrada un intervalo y devuelve un array con todos los números primos en el intervalo.

2.4. Arquitecturas Cliente-Servidor y Máster Worker

La arquitectura cliente servidor es una de las más utilizadas en sistemas distribuidos. En esencia, consiste en un proceso servidor, que aglutina la mayor parte de la funcionalidad, y un conjunto de procesos clientes que solicitan al servidor esa funcionalidad mediante el intercambio de mensajes. Existen distintas variantes de esta arquitectura, así como distintas posibilidades de implementación que nos proporciona Go:

1. La arquitectura cliente servidor secuencial consiste en un servidor que atiende peticiones de forma secuencial, de una en una, de manera que cuando llegan varias peticiones, atiende una de ellas (a menudo la primera en llegar) y, una vez terminada, atiende la siguiente. Para reducir el tiempo de espera de los clientes, siempre que haya recursos hardware suficientes en el servidor y siempre que la aplicación lo permita, se puede utilizar la arquitectura cliente-servidor concurrente.

2. La arquitectura cliente servidor concurrente consiste en un servidor que puede atender varias peticiones en paralelo. En Go esto puede implementarse de varias formas:

   - La más sencilla consiste en que inicialmente el servidor espera a que llegue una petición, una vez recibida se crea una Goroutine y se le pasa la petición para que la procese y devuelva el resultado.
   - Crear una Goroutine por cada petición conlleva un sobrecoste en tiempo y, por tanto, otra opción podría ser tener un conjunto fijo de Goroutines (Goroutine pool en inglés, patrón software pool de elementos) que atienden peticiones y se pueden reutilizar. Las Goroutines se comunican con el proceso principal servidor a través de dos canales síncronos: un canal donde el programa principal envía las tareas que se reciben y las Goroutines leen todas de ese canal y van extrayendo los datos de él (se realiza de forma oportunista, esto es, la primera que consigue obtener la tarea se la lleva) y otro canal donde las Goroutines escriben los resultados para que el programa principal del servidor los recoja. Alternativamente, podríamos prescindir de este canal de resultados y que las Goroutines enviaran directamente el resultado al cliente correspondiente. Podéis encontrar información al respecto y código fuente de ejemplo en Go en este link 2.
   - La tercera y última opción consiste en utilizar la función Select del paquete socket, que es equivalente a la función Select del lenguaje C. Esta funcionalidad tiene su origen en el SO UNIX. Hay llamadas especializadas en algunos

sistemas operativos, como "epoll" en Linux o "kqueue" en FreeBSD, que optimizan aún más el multiplexado en accesos a descriptores de sockets y ficheros. Los diferentes elementos que operan sobre la técnica "epoll" de Linux están también disponibles en el package "syscall" de Go. Y estás son técnicas, no solo disponible en su librería estándar, sino utilizadas, directamente, en el runtime de Go.

3. La Arquitectura Máster Worker puede verse como una extensión de la arquitectura cliente-servidor concurrente con un pool de Goroutines, en la que existe un programa principal (máster) que reparte las tareas a un conjunto de procesos (workers). Sin embargo, a diferencia del cliente servidor, en el master worker, cada Goroutine no hace uso de los recursos propios de la máquina sino que interactúa con otro proceso remoto en otra máquina. Esta característica nos proporciona la ventaja de posibilitar la escalabilidad.

3. Compilación y Ejecución en Golang

3.1. Compilación y ejecución

En las Raspberry Pis del Lab 1.02 está instalado el compilador de Golang v1.18. Esta versión del compilador será la referencia durante la asignatura. Para utilizarlo en las máquinas del laboratorio, podéis obtener la ruta en la que está instalado:

```
rafaelt@r07:~$ which go
/usr/bin/go
```

Si la ruta no está en vuestra variable de entorno $PATH$, podéis añadirla de esta forma:

```
rafaelt@r07:~$ export PATH=$PATH=:/usr/bin/go/bin/
```

Para configurarlo de forma permanente en vuestra cuenta, podéis añadir la ruta de instalación de Golang a vuestro fichero *.bashrc*.

Además, también podéis descargarlo e instarlo en vuestras máquinas, aquí las versiones del compilador para las distintas plataformas: `https://go.dev/dl/`.

Para la compilación de fuentes Golang, de manera que se genera un ejecutable a partir de los fuentes de un directorio, se puede ejecutar en el directorio:

```
rafaelt@r07:~$ go build
```

También puede ejecutarse un fichero fuente directamente:

```
rafaelt@r07:~$ go run <nombre_fuente.go>
```

A modo de ejemplo, se puede considerar el siguiente código HolaMundo en Golang, hello-world.go:

```
package main
import "fmt"
func main() {
    fmt.Println("hello world")
}
```

Si se ejecuta go run hello-world.go, se obtiene lo siguiente:

```
rafaelt@r07:~$ go run hello-world.go
hello world
```

Alternativamente, el código se puede compilar de manera que se genere el ejecutable:

```
rafaelt@r07:~$ go build hello-world.go
rafaelt@r07:~$ ls
hello-world    hello-world.go
We can then execute the built binary directly.

rafaelt@r07:~$ ./hello-world
hello world
```

3.2. Paquetes y Módulos en Golang

En Golang, la unidad básica de organización y acceso de código es el "paquete" (package), también para nuevos programas cuando su tamaño crece y se desea estructurar. Cada paquete se ubica en su propio directorio cuyo nombre es idéntico al paquete. Para simplificar el acceso y utilización de paquetes locales a un programa, es aconsejable organizarlos dentro de un "módulo" Golang, como elemento organizativo que agrega paquetes y, si es necesario crear programas ejecutables, varios paquetes main. El acceso del código main a los paquetes internos al módulo requiere explicitar un camino local de importación de dichos paquetes con el formato:

import " $< nombremodulo > / < pathdeaccesohastadirectoriodepaquete >$ ".

El código que acompaña este guión se puede ver como un ejemplo simple de esta metodología [2]. Dicho código está ubicado en un modulo llamado "practica1". El fichero "go.mod" que define este modulo lo tenéis ya generado mediante la ejecución del comando "go mod init practica1", una vez ubicados en el directorio "practica1". Ahora los paquetes y códigos main internos utilizaran Paths locales a dicho modulo. Para compilar y ejecutar el programa principal de cliente en el esqueleto suministrado, primero os hay que situarse en el directorio "practica1" y después ejecutar:

```
~/ go build cmd/client/main.go
~/ ./client
```

o también:

```
~/ go run cmd/client/main.go
```

La compilación de ambos ejecutables, servidor y cliente, con destino al directorio relativo bin, lo podéis realizar con el comando:

```
~/ go build -o bin ./...
```

El acceso a la librería "com" puesta a disposición desde fuera de dicho paquete, pero dentro del modulo "practica1" se efectúa mediante:

[2] Otro ejemplo significativo de uso de modulo lo podeis ver en `https://www.digitalocean.com/community/tutorials/how-to-use-go-modules`

```
import "practica1/com"
```

Finalmente, para reejecución automática de un comando shell (por ejemplo, "go run ./cmd/server/main.go") tras el evento de guardar un fichero de código ".go", tras haber sido modificado, podéis utilizar, entre otros, la herramienta "arelo" que podéis encontrar en `https://github.com/makiuchi-d/arelo`. Tenéis un par de ficheros de ejemplo, que podéis utilizar, en el código asociado a este guion. Para que funcione, debe estar instalado inotify en Linux.

3.3. Depuración en Golang

Golang dispone de depuradores de estilo tradicional como Delve [3]. Se puede integrar a VScode. Pero para depuración de programas distribuidos es más adecuado obtener trazas con estampillas de tiempo, mediante sistemas de registro de eventos o "logging", para poder obtener un orden de ejecución de las partes distribuidas, y en particular poder analizar problemas de paso de mensajes y sincronización. Los mensajes mostrados permiten realizar un análisis de una traza de ejecución una vez el programa se ha ejecutado. Una forma rudimentaria de registrar eventos consisten en escribir mensajes por pantalla (bien salida estándar o bien en la salida de error), haciendo uso de las bibliotecas de entrada salida. Sin embargo, los lenguajes de programación modernos incorporan librerías específicas para registrar eventos (tiempo, fichero, línea de fichero). En Golang, está el paquete "log", que puede ser útil para depurar programas. Podéis ver su API aquí:

4. El Despliegue del Software en un Sistema Distribuido

El despliegue del software en un sistema distribuido es un aspecto muy complejo, sobre todo cuando son necesarios muchas bibliotecas o componentes software que pueden tener distintas versiones. Tal es así que sistemas como *Docker*[4] surgieron motivados para simplificar las tareas de despliegue.

El desarrollo y pruebas iniciales de vuestra solución se puede realizar de forma local en un portátil u ordenador de sobremesa. Pero las pruebas y evaluaciónn finales deberá ser realizado, en todas las prácticas de la asignatura, en el clúster de 20 máquinas (raspberries -ARM64-), con Ubuntu server 22.04, que se encuentra en el laboratorio 1.02, y que nos permiten un entorno dedicado y controlado de ejecución distribuida.

El cluster consta de 20 máquinas, accesibles remotamente, con direcciones IP privadas en el rango 192.168.3.1 a 192.168.3.20, todas ellas con SO Ubuntu server 22.4. Solo puede accederse a ellas a través de ssh y desde el servidor central.cps.unizar.es. Tenéis que utilizar vuestras mismas cuentas de hendrix (mismo nombre y contraseña). Vuestras cuentas de usuario tienen un home local con una cuota máxima de 300 MB. Configurar en ellas la autentificación ssh/scp mediante clave pública (authorized keys). No se debe utilizar el servidor central para ejecutar vuestros programas, utilizarlo, sólo, para realizar

[3] Documentación de Delve `https://github.com/go-delve/delve/tree/master/Documentation`
[4] `https://www.docker.com/`

copias de ficheros y pasarela de acceso ssh al cluster de 20 máquinas. Cada pareja de prácticas tiene un rango exclusivo de 10 puertos TCP/UDP, asignado en un documento que tenéis disponible en moodle, para no interferir entre parejas. Para distribuir el uso de las 20 máquinas del cluster, cada pareja puede utilizar hasta 4 máquinas, cuyo rango está también definido en el mismo documento en moodle. El número de máquina en el rango se utiliza como última cifra de su dirección IP (máquina 7, IP 192.168.3.7).

Además, cada alumno dispone de su propio directorio:
$/misc/alumnos/sd/sd2425/aXXXXXX$
(mediante el Sistemas de Ficheros Distribuido NFS), que está compartido entre las 41 máquinas de laboratorio 1.02 (las 20 de sobremesa, 20 ARMs del cluster y el servidor central). Esto nos permite copiar datos y código entre las máquinas del laboratorio y el clúster de Raspberry Pis. Por ejemplo, para copiar el directorio auxiliar de la práctica 1 desde central a una Raspberry pi del clúster, puede ejecutarse este comando:

```
$ scp -r practica1 central.cps.unizar.es:/misc/alumnos/sd/sd2425/aXXXXXX
```

Es importante notar que si compiláis el código go en una máquina con arquitectura ARM64, el ejecutable generado no va a funcionar en AMD64 y viceversa. No obstante, se puede, también, realizar una *compilación cruzada* en una máquina AMD64 para generar binario ARM64 mediante:
$GOOS = linux GOARCH = arm64 go build main.go$

Para pruebas intermedias, podéis utilizar también las 20 máquinas AMD64 de sobremesa del laboratorio L1.02.

4.1. La Ejecución Remota de Scripts en Unix: SSH

Secure Shell (SSH) es, por un lado, un protocolo de transmisión de red que permite acceder a un servidor de forma remota y segura. Una aplicación muy habitual de SSH es propocionar un acceso a la línea de comandos del servidor de forma remota, pero también permite ejecutar comandos shell de forma remota, sin utilizar la línea de comandos.

Por ejemplo, si se ejecuta el comandos SSH de OpenSSH como en el siguiente fragmento de código:

```
1 $>ssh user1@server1 command1
```

se consigue que el usuario $user1$ ejecute el comando $command1$ de forma remota en el servidor $server1$. Por defecto, ssh solicitará al usuario $user1$ que introduzca su password.

4.2. Ejecución Remota de Scripts sin Password

Es posible iniciar sesión en un servidor Linux remoto sin tener que introducir interactivamente la contraseña. Esto permite automatizar la ejecución de determinados scripts. Para ello, hay que realizar estos 3 pasos, usando ssky-keygen y ssh-copy-id:

Vamos a asumir que estamos trabajando en una máquina cuya IP es 155.210.154.200 (local-host) y que queremos conectarnos remotamente a la máquina 155.210.154.201. Paso 1: Desde la línea de comandos de 155.210.154.200 vamos a crear las claves pública y privada mediante ssh-key-gen

```
rafaelt@local-host$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/rafaelt/.ssh/id_rsa):[Enter key]
Enter passphrase (empty for no passphrase): [Press enter key]
Enter same passphrase again: [Pess enter key]
Your identification has been saved in /home/rafaelt/.ssh/id_rsa.
Your public key has been saved in /home/rafaelt/.ssh/id_rsa.pub.
The key fingerprint is:
33:b3:fe:af:95:95:18:11:31:d5:de:96:2f:f2:35:f9 rafaelt@local-host
```

Paso 2: Copiamos la clave pública en nuestra cuenta del servidor remoto (155.210.154.201) utilizando ssh-copy-id:

```
rafaelt@local-host$ ssh-copy-id -i ~/.ssh/id_rsa.pub 155.210.154.201
rafaelt@155.210.154.201's password:

Now try logging into the machine, with "ssh 'remote-host'", and check in:

.ssh/authorized_keys

to make sure we haven't added extra keys that you weren't expecting.
```

Paso 3: Ahora ya se pueden utilizar ssh sin necesidad de introducir el password de forma interactiva

```
rafaelt@local-host$ ssh 155.210.154.201 command1
Last login: Sun Nov 16 17:22:33 2008 from 192.168.1.2
```

De manera que ejecutará el comando *command*1 en el servidor 155.210.154.201.

4.3. SSH desde Golang

Go cuenta con una biblioteca estándar que implementa la funcionalidad de SSH, lo cual posibilita la interacción con servidores SSH desde código Go. Por ahora no lo utilizaremos, pero puede tener interés si se necesita mayor integración de funcionalidades.

5. Concurrencia en Golang

La concurrencia es una parte inherente del lenguaje de programación Go y para ello el lenguaje proporciona dos elementos fundamentales: las Goroutines y los canales síncronos. Las Goroutines son funciones o métodos que se ejecutan al mismo tiempo que otras funciones o métodos. Los goroutines pueden considerarse hilos (*threads*), pero en realidad la sobrecarga (*overhead* en inglés) asociada a una Goroutine es mínimma en comparación con la de un thread. Por lo tanto, es común que las aplicaciones Go tengan miles de Goroutines ejecutándose al mismo tiempo, de hecho, están diseñadas para tal fin. Las

Goroutines se multiplexan en una menor cantidad de threads. Incluso, podría darse el caso en que solo hubiera un thread en un programa con miles de Goroutines.

En el Fragmento de Código Gorutina simple [5], puede observarse el mecanismo, muy simple, con el que se diseñaron las Goroutines. En el código existen dos funciones hello() y main(), el programa principal. Una vez que se ejecuta main, en la línea 11 se ejecuta go hello, lo que hace que se cree una Goroutine que ejecuta la función hello simultáneamente con el programa principal main. Muy probablemente, el programa principal terminará antes que la Goroutine, finalizando la ejecución de todo el programa, de manera que a la Goroutine no le dará tiempo a ejecutarse completamente.

```
1  package main // Código Gorutina simple
2
3  import (
4      "fmt"
5  )
6
7  func hello() {
8      fmt.Println("Hello world goroutine")
9  }
10 func main() {
11     go hello()
12     fmt.Println("main function")
13 }
```

Las Goroutines, *dentro del mismo programa*, pueden comunicarse entre sí mediante canales síncronos, inspirados en el paradigma Communicating Sequential Processes (CSP, de Hoare) [1]. Los canales se pueden considerar como una tubería a través de la cual se comunican las Goroutines. Desde un punto de vista semántico, los canales síncronos bloquean al emisor y al receptor hasta que ambos estén en ejecución simultánea en el canal, esto es, si un proceso escribe en el canal, este se quedará bloqueado hasta que el proceso receptor lea del canal. Y al revés, si un proceso receptor intenta leer de un canal antes de que el proceso productor escriba, también se quedará bloqueado.

En el Fragmento de Código Gorutina simple en orden [6], puede verse cómo pueden combinarse las Goroutines y los canales. El programa principal crea un canal de booleanos en la línea 12 y lanza la Goroutine hello en la línea 13, pasándole el canal como argumento. A partir de ahí, la Goroutine y el programa principal se sincronizarán a través del canal. Una vez que hello termine su ejecución, en la línea 9 le enviará el booleano true al programa principal. Nótese que el flujo de ejecución del programa principal estaba bloqueado, esperando, en la línea 14, hasta que la Goroutine hello escribiera en el canal.

```
1  package main // Código Gorutina simple en orden
2
3  import (
4      "fmt"
```

[5] Disponible y ejecutable en `https://play.golang.org/p/zC78_fc1Hn`
[6] Disponible y ejecutable en `https://play.golang.org/p/I8goKv6ZMF`

```
)

func hello(done chan bool) {
    fmt.Println("Hello world goroutine")
    done <- true
}
func main() {
    done := make(chan bool)
    go hello(done)
    <-done
    fmt.Println("main function")
}
```

Debido a la semántica de los canales síncronos, mediante la cual bloquean tanto al lector como al escritor, para aquellos escenarios en los que una Goroutine tiene que utilizar varios canales simultáneamente es necesaria una estructura de control que permita bloquearse en todos ellos y despertarse en cuanto uno esté disponible. La estructura de control Select [7] permite a una Goroutine esperar en múltiples canales simultáneamente, tanto en escritura como en lectura. Podemos tener cualquier número de declaraciones de casos dentro de Select, siempre habrá un canal en la guarda de cada declaración. Por tanto, al invocar Select, si ningún canal está listo, Select bloqueará al proceso invocante.

En cuanto haya una declaración activa (un canal listo para leer o escribir) se desbloqueará. Si hubiera varios activos a la vez, el sistema elige uno aleatoriamente.

En el Fragmento de código de la función handleMessages, hay un ejemplo ilustrativo de cómo se pueden combinar las Goroutines y los canales síncronos para solucionar el problema de la sección crítica. El fragmento corresponde a un servidor de un chat grupal centralizado que podéis encontrar en esta dirección [8].

```
func handleMessages(msgchan <-chan string,
        addchan <-chan Client,
        rmchan <-chan Client) { // Función handleMessages

    clients := make(map[net.Conn]chan<- string)
    for {
        select {
        case msg := <-msgchan:
            log.Printf("New message: %s", msg)
            for _, ch := range clients {
                go func(mch chan<- string) {
                    mch <- "\033[1;33;40m" + msg + "\033[m"
                    }(ch)
            }
        case client := <-addchan:
            log.Printf("New client: %v\n", client.conn)
            clients[client.conn] = client.ch
        case client := <-rmchan:
```

[7] https://golangr.com/select/
[8] https://github.com/akrennmair/telnet-chat/blob/master/03_chat/chat.go

```
19                log.Printf("Client disconnects: %v\n", client.conn)
20                delete(clients, client.conn)
21            }
22        }
23    }
```

La Goroutine handleMessages gestiona la lista de personas que están activas en el chat durante la ejecución (variable clients), que es una tabla hash, una estructura de datos que almacena los participantes en el chat. A la Goroutine le llegan peticiones para añadir clientes al chat, borrar clientes del chat o para enviar mensajes a todos los participantes del chat. Notar que, aunque las peticiones pueden llegar a la vez, se atienden de una en una. Por último y no menos importante, múltiples Goroutines pueden leer simultáneamente del mismo canal hasta que este se cierre y múltiples Goroutines pueden escribir simultáneamente en el mismo canal hasta que este se cierre. Estas dos formas de utilizarlos se denominan Fan-out y Fan-in respectivamente. Tenéis más información al respecto en este link [9], estos patrones pueden ser muy útiles para construir la arquitectura máster-worker de esta práctica.

6. Programación Distribuida en Go

6.1. El formato de los datos

La comunicación entre procesos en un sistema distribuido requiere del intercambio de mensajes y estos a su vez se estructuran en datos. Estos datos deben serializarse para su transporte a través de la red de comunicación. En esta sección vamos a reseñar brevemente cuál es la funcionalidad que proporciona el lenguaje Go para este aspecto de la comunicación.

Si bien cuando la comunicación entre procesos se realiza a través de TCP o UDP, los procesos se intercambian información en forma de secuencias de bytes, los lenguajes de programación utilizan frecuentemente estructuras de datos para codificar y solucionar problemas computacionales. Cuando las estructuras de datos son multidimensionales o cuando muestran un tamaño variable, la transmisión de la información a través de la red se convierte en un reto, que aumenta a medida que aumenta el número de procesos del sistema y su heterogeneidad (sistema operativo, arquitectura hardware, etc.).

Una opción para poder realizar la comunicación, quizá la más básica, es que tanto el emisor como el receptor de los mensajes acuerden exactamente cómo se va a efectuar la serialización de las estructuras de datos en los mensajes (por serialización se entiende, cómo transformar una estructura de datos a una secuencia de bytes). Ningún tipo de descripción acerca de la codificación (marshalling en inglés) se incluye en el mensaje. A esta aproximación se le denomina implícita u opaca, porque tanto emisor como receptor tienen que saber cómo decodificar (unmarshalling en inglés) la información recibida a partir de una secuencia de bytes. Esta opción puede utilizarse en Go.

[9] `https://go.dev/blog/pipelines`

Frente a la estrategia implícita u opaca, se establece una aproximación que describe la estructura de los datos y se incorpora en los mensajes (metadatos); de manera que el proceso decodificador (unmarshaller) utilizará los metadatos del mensaje para extraer los datos de la secuencia binaria. Esta es la aproximación que se puede utilizar en Go; además Go utiliza su propio estándar, denominado Gob [10], que incorpora información sobre cómo se ha llevado a cabo la codificación (metadatos) dentro del mensaje, esto persigue un marshalling y unmarshalling (codificación y decodificación) más robustos. En la especificación del paquete Gob podéis encontrar ejemplos de utilización. Puede ser muy útil a la hora de realizar los ejercicios de esta práctica, de lo contrario el paso de mensajes puede convertirse en una labor muy tediosa.

7. Calidad de Servicio (Quality of Service)

La calidad de servicio (QoS) es la medición cuantitativa del desempeño general de un sistema distribuido. Para ello, a menudo se consideran aspectos relacionados con la red de comunicación (tales como la pérdida de paquetes, la tasa de bits, el rendimiento, el retardo de transmisión, la disponibilidad, etc.), con el procesamiento (uso de memoria, tiempo de respuesta, throughput, consumo energético o coste económico) y con el almacenamiento.

Normalmente, estos requisitos varían de una aplicación a otra y, en ocasiones, también dependen de las preferencias del usuario. Por ejemplo, en una aplicación de videoconferencia, un usuario puede querer una frecuencia de cuadros de 12 fps y una frecuencia de muestreo de audio de 44 khz. Otro usuario puede querer una velocidad de fotogramas de vídeo de 15 fps y audio de 32 khz. Del mismo modo, puede haber otros requisitos, como que se requiera cifrado o no. La asignación de recursos suficientes a diferentes aplicaciones para satisfacer estas limitaciones es un problema de gestión de recursos y de QoS.

En esta práctica, vamos a considerar el QoS para el cliente, de manera que el tiempo total de ejecución T de una tarea es:

$$T = t_{ex} + t_{xon} + t_o \tag{7.1}$$

donde t_{ex} es el tiempo de ejecución efectivo de la tarea en la máquina donde se ejecute, en condiciones ideales, esto es, de forma aislada sin ningún otro proceso que interaccione y desvirtúe la ejecución; t_{xon} es el tiempo de transmisión requerido al intercambiar los mensajes en la red para realizar la ejecución y t_o es el tiempo de overhead que surge cuando aparecen tiempos de espera u otro tipo de interferencia en las prestaciones debidos a la gestión de la ejecución. En esta práctica, para simplificar el problema, *nuestras tareas van a ser siempre las mismas, de manera que, en condiciones ideales,* t_{ex} *es constante.*

Como métrica de QoS consideraremos que T no puede ser mayor en ningún caso que el doble del t_{ex}. Dicho de otro modo, intentaremos que se cumpla lo siguiente:

[10] `https://pkg.go.dev/encoding/gob`

$$t_{ex} + t_{xon} + t_o < 2 * t_{ex}$$

En esta práctica se proporciona el cliente que mide T. En primer lugar, realizaremos un análisis para determinar cuántas peticiones puede atender un servidor manteniendo el QoS y posteriormente validaremos nuestro análisis experimentalmente.

7.1. Análisis del QoS de la Arquitectura Cliente Servidor

Para realizar un análisis de cuál es el QoS que cada arquitectura puede soportar, hay que calcular el throughput del sistema, esto es, el número de tareas por unidad de tiempo que cada sistema es capaz de procesar.

En primer lugar, estudiaremos y calcularemos cuál es el t_{ex}. Para el cálculo del tiempo de ejecución efectivo, deberéis ejecutar un número significativo de veces (por ejemplo 10 veces) una tarea en condiciones ideales, esto es, de forma aislada y sin que se estén ejecutando otros procesos en la máquina (aparte del SO, claro está) . Con todas esas mediciones calcularéis la media aritmética y ese será vuestro t_{ex}, tiempo de ejecución efectivo.

Si tomamos como ejemplo la arquitectura cliente servidor secuencial, el throughput (μ) del servidor, esto es, el número máximo de tareas por unidad de tiempo que puede procesar viene dado por:

$\mu = 1/t_{ex}$

asumiendo que el t_{xon} y t_o son prácticamente despreciables frente a t_{tex} y que *el servidor secuencial solo es capaz de procesar una tarea a la vez.*

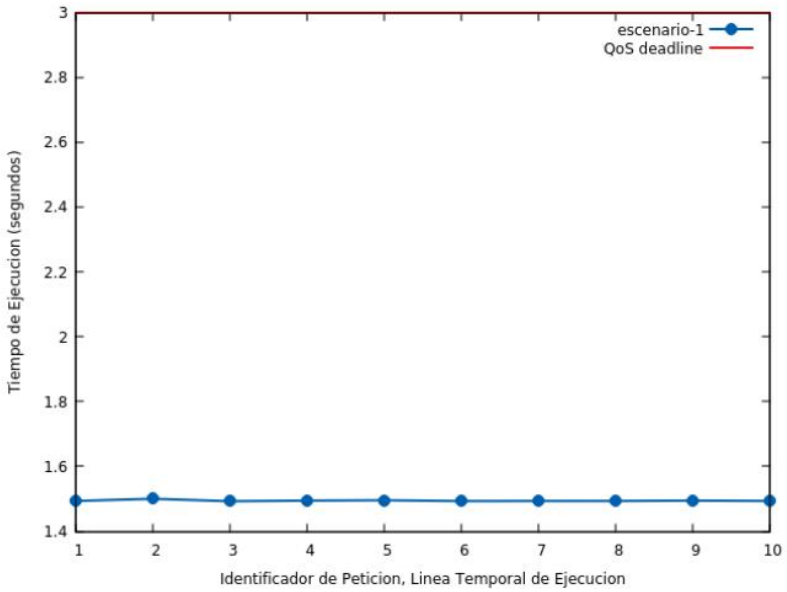
Por tanto, para cargas de trabajo menores o iguales a μ el servidor será capaz de mantener el QoS. Sin embargo, para cargas de trabajo mayores a μ, comienza a aparecer un tiempo de espera, que hará que no se cumpla el QoS.

7.2. Validación Experimental

Para verificar esto experimentalmente se puede ejecutar la arquitectura cliente servidor secuencial proporcionada y generar cargas de trabajo que sean menores y mayores que μ. El número mínimo de experimentos que pueden realizarse para verificarlo dos:

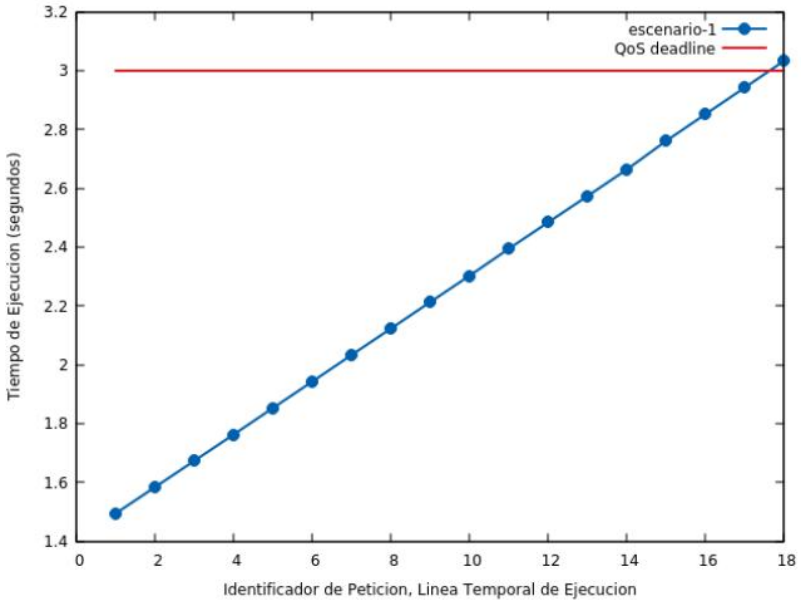
- carga de trabajo igual a μ
- carga de trabajo mayor que μ.

En el primer caso, se puede observar cómo se mantiene el QoS, mientras que no sucede en el segundo. El cliente proporcionado escribe por salida estándar el tiempo T observado para cada petición. Se proporciona un script de gnuplot que permite visualizar gráficamente la salida de un experimento. La Figura 1 muestra un experimento para el cliente servidor secuencial en el que la carga de trabajo es igual al throughput. En la Figura 2, la carga de trabajo es ligeramente mayor que el throughput. Los experimentos demuestran que el análisis era correcto.



Gráfica 1
servidor secuencial
carga = 0.67 peticiones/s

Figura 1: Experimento con $\lambda = \mu$, carga de trabajo igual al throughput del sistema



Gráfica 2
servidor secuencial
carga = 0.71 peticiones/s

Figura 2: Experimento con $\lambda > \mu$, carga de trabajo mayor que el throughput del sistema

7.3. Generación de Timestamps para Medir el Tiempo de Ejecución

Los sistemas operativos proporcionan cierto soporte para poder medir el tiempo de ejecución de las operaciones, así como para cualquier otra funcionalidad relacionada con aspectos temporales. Apoyado en esas llamadas al sistema, Go proporciona el paquete time. En esta práctica puede ser de utilidad medir el tiempo de ejecución total de ciertas operaciones. En el fragmento de código del cliente de esta práctica, que se proporciona, se puede ver un ejemplo de utilización.

```
1  // fragmento de código correspondiente al cliente de esta práctica
2      start := time.Now()
3      id := 1
4      err = encoder.Encode(int64(40000))
5      if err != nil {
6          log.Fatal("encode error:", err)
7      }
8
9      var pisequence string
10     err = decoder.Decode(&pisequence)
11     end := time.Now()
12     fmt.Println(id, "\t", end.Sub(start))
```

La variable start almacena el tiempo en el instante de ejecución en que se ejecuta la instrucción Now() del paquete time. A continuación, se ejecutan una secuencia de operaciones, después se vuelve a medir el tiempo y se almacena en la variable end. Finalmente, se imprime por pantalla el tiempo transcurrido desde start a end, que se obtiene de restar end – start, operación proporcionada por el paquete time. De esta forma se puede medir el tiempo de ejecución de una secuencia de operaciones y esta técnica se conoce como habitualmente instrumentación del código y en este caso es intrusivo.

8. Rúbrica

Con el objetivo de que, tanto los profesores como los estudiantes de esta asignatura por igual, puedan tener unos criterios de evaluación objetivos y justos, se propone la siguiente rúbrica en la Tabla 1. Los valores de las celdas son los valores mínimos que hay que alcanzar para conseguir la calificación correspondiente y tienen el siguiente significado:

- A+ (excelente). En el caso de software, conoce y utiliza de forma autónoma y correcta las herramientas, instrumentos y aplicativos software necesarios para el desarrollo de la práctica. Plantea correctamente el problema a partir del enunciado propuesto e identifica las opciones para su resolución. Aplica el método de resolución adecuado e identifica la corrección de la solución, sin errores. En el caso de la memoria, se valorará una estructura y una presentación adecuadas, la corrección del lenguaje, así como el contenido explica de forma precisa los conceptos involucrados en la práctica. En el caso del código, este se ajusta exactamente a las guías de estilo propuestas.

Tabla 1: Rúbrica para la Práctica

Calificación	Arquitectura	Código	Memoria
10	A+	A+	A+
9	A+	A	A
8	A	A	A
7	B	A	A
6	B	B	B
5	B-	B-	B-
suspenso	C		

- A (bueno). En el caso de software, conoce y utiliza de forma autónoma y correcta las herramientas, instrumentos y aplicativos software necesarios para el desarrollo de la práctica. Plantea correctamente el problema a partir del enunciado propuesto e identifica las opciones para su resolución. Aplica el método de resolución adecuado e identifica la corrección de la solución, con ciertos errores no graves. Por ejemplo, algunos pequeños casos (marginales) no se contemplan o no funcionan correctamente. En el caso del código, este se ajusta casi exactamente a las guías de estilo propuestas.

- B (suficiente). En el caso de software, conoce y utiliza de forma autónoma y correcta las herramientas, instrumentos y aplicativos software necesarios para el desarrollo de la práctica. No plantea correctamente el problema a partir del enunciado propuesto y/o no identifica las opciones para su resolución. No aplica el método de resolución adecuado y / o identifica la corrección de la solución, pero con errores. En el caso de la memoria, bien la estructura y / o la presentación son mejorables, el lenguaje presenta deficiencias y / o el contenido no explica de forma precisa los conceptos importantes involucrados en la práctica. En el caso del código, este se ajusta a las guías de estilo propuestas, pero es mejorable.

- B- (suficiente, con deficiencias). En el caso de software, conoce y utiliza de forma autónoma y correcta las herramientas, instrumentos y aplicativos software necesarios para el desarrollo de la práctica. No plantea correctamente el problema a partir del enunciado propuesto y/o no identifica las opciones para su resolución. No se aplica el método de resolución adecuado y/o se identifica la corrección de la solución, pero con errores de cierta gravedad y/o sin proporcionar una solución completa. En el caso de la memoria, bien la estructura y / o la presentación son manifiestamente mejorables, el lenguaje presenta serias deficiencias y / o el contenido no explica de forma precisa los conceptos importantes involucrados en la práctica. En el caso del código, hay que mejorarlo para que se ajuste a las guías de estilo propuestas.

Referencias

[1] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.