Universidad Zaragoza
1542

30321 – Sistemas Distribuidos
Prácticas de Laboratorio

Grado en Ingeniería Informática

Curso 2020-2021

ESCUELA DE INGENIERÍA Y ARQUITECTURA
Departamento de Informática e Ingeniería de Sistemas

v 1.0
septiembre de 2020

Práctica 1. Introducción a las arquitecturas de Sistemas Distribuidos

1. Objetivos y Requisitos

Esta práctica tiene como objetivo que analicéis la arquitectura cliente servidor con sus variantes y así como la arquitectura master-worker. Para ello, diseñaremos e implementaremos sistemas distribuidos muy sencillos y realizaremos distintos experimentos.
Deberéis entregar el código fuente y una memoria en la que analizaréis el comportamiento del cliente-servidor y resumiréis vuestro diseño del máster worker. **El número máximo de páginas permitido es de 6 en total (4 páginas arquitecturas + 1 experimentos + 1 portada)**.

1.1 Objetivo

- Familiarización con las arquitecturas, cliente-servidor y máster-worker

1.2 Requisitos

- Golang y su entorno de desarrollo

2. Aspectos de Golang necesarios para esta práctica

2.1 Concurrencia en Golang

La concurrencia es una parte inherente del lenguaje de programación Go y se proporcionan dos elementos fundamentales en el lenguaje: 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) de crear una Goroutine es mínimo en comparación con 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*. Puede que solo haya un *thread* en un programa con miles de Goroutines. Las Goroutines se comunican 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. 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.

En el Fragmento de Código 1, puede observarse el mecanismo, muy simple, con el que se diseñaron las Goroutines. 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 en la que se pone en ejecución la función *hello* simultáneamente con el programa principal *main*. Muy probablemente, el programa principal en este caso, terminará antes que la Goroutine, terminando la ejecución de todo el programa, de manera que a la Goroutine no le dará tiempo a ejecutarse completamente.

[1] http://usingcsp.com/

```go
package main

import (
    "fmt"
)

func hello() {
    fmt.Println("Hello world goroutine")
}
func main() {
    go hello()
    fmt.Println("main function")
}
```

Fragmento de código 1: Goroutine[2], que además puede ejecutarse en https://play.golang.org/p/zC78_fc1Hn

En el Fragmento de código 2, 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.

```go
package main

import (
    "fmt"
)

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")
}
```

Fragmento de código 2: Goroutine[3], que además puede ejecutarse en https://play.golang.org/p/I8goKv6ZMF

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*[4] 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

[2] Extraído de https://golangbot.com/goroutines/

[3] Extraído de https://golangbot.com/goroutines/

[4] https://golangr.com/select/

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 3, 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[5].

```
100 func handleMessages(msgchan <-chan string, addchan <-chan Client, rmchan <-chan Client) {
101         clients := make(map[net.Conn]chan<- string)
102 
103         for {
104                 select {
105                 case msg := <-msgchan:
106                         log.Printf("New message: %s", msg)
107                         for _, ch := range clients {
108                                 go func(mch chan<- string) { mch <- "\033[1;33;40m" + msg + "\033[m" }(ch)
109                         }
110                 case client := <-addchan:
111                         log.Printf("New client: %v\n", client.conn)
112                         clients[client.conn] = client.ch
113                 case client := <-rmchan:
114                         log.Printf("Client disconnects: %v\n", client.conn)
115                         delete(clients, client.conn)
116                 }
117         }
118 }
```

Fragmento de código 3: Goroutine de ejemplo que combina canales síncronos y select para garantizar el acceso en exclusión mutua a una variable

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 de añadir clientes al chat, de borrar clientes del chat o de 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, en Golang 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[6].

2.2 Generación de timestamps

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, Golang proporciona el paquete *time*[7].

En esta práctica puede ser de utilidad medir el tiempo de ejecución total de ciertas operaciones. Por ejemplo, en este Fragmento de código 4 Golang se realiza lo siguiente:

```
start := time.Now()
... operation that takes 20 milliseconds ...
t := time.Now()
elapsed := t.Sub(start)
```

Fragmento de código 4: operaciones temporales

[5] https://github.com/akrennmair/telnet-chat/blob/master/03_chat/chat.go

[6] https://blog.golang.org/pipelines

[7] https://golang.org/pkg/time/

La variable *start* almacena el tiempo en el instante de ejecución en que se ejecuta la instrucción *Now*() del paquete *time*. A continuación, se ejecuta cierta operación durante 20 milisegundos, después se vuelve a medir el tiempo y se almacena en la variable t. Finalmente, la variable elapse proporciona el tiempo de ejecución de la operación (20 milisegundos), que se obtiene de restar t – 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.

2.3 Programación Distribuida en Golang

2.3.1 El formato de los datos

La comunicación entre procesos en un sistema distribuido requiere del intercambio de datos[8]. Estos datos pueden estar muy estructurados, pero 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 Golang para este aspecto de la comunicación.

Si bien, típicamente, la comunicación entre procesos tiende a realizarse a través de TCP o UDP, los lenguajes de programación utilizan frecuentemente estructuras de datos para codificar y solucionar problemas computacionales. Los protocolos de transporte TCP o UDP permiten a los procesos enviar información, pero están diseñados para enviar bytes y no consideran la complejidad inherente a las estructuradas de datos. Cuando las estructuras de datos posibilitan que los datos tengan 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 Golang.

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 Golang, pero Golang **utiliza su propio estándar, denominado *Gob*[9]**, que incorpora información de cómo se ha llevado a cabo la codificación (metadatos) dentro del mensaje, esto persigue un marshalling y unmarshalling (codificación y decodificación) más robustas. 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.

2.3.2 Arquitectura Cliente Servidor

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 Golang:

[8] https://ipfs.io/ipfs/QmfYeDhGH9bZzihBUDEQbCbTc5k5FZKURMUoUvfmc27BwL/dataserialisation/index.html

[9] https://golangbot.com/buffered-channels-worker-pools/

- **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, se puede utilizar la arquitectura cliente-servidor concurrente.

- **La arquitectura cliente servidor concurrente** consiste en un servidor que puede atender varias peticiones en paralelo.

    - En Golang 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 Golang en este link[10].
    - *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 Golang. Y estás son técnicas, no solo disponible en su librería estándar, sino utilizadas, directamente, en el runtime de Golang.

- 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. Ejercicios

Junto con este enunciado, tenéis disponible el fichero primes.go 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 número natural n y devuelve un array con todos los números primos entre [1, n]. Con esas funciones, tenéis que realizar los ejercicios siguientes.

3.1 Diseño de cuatro Arquitecturas de SSDD

En la práctica, los sistemas distribuidos se construyen atendiendo a requisitos funcionales (qué debe de realizar el sistema) y requisitos no funcionales (cómo debe realizarlo y cuáles son sus prestaciones o coste). En los sistemas distribuidos actuales, es muy habitual que los requisitos no funcionales se materialicen en un

[10] https://golangbot.com/buffered-channels-worker-pools/

acuerdo (o contrato formal entre el cliente y la empresa que proporciona el servicio) denominado acuerdo del nivel del servicio (*Service Level Agreement* SLA en inglés). Desde un punto de vista cuantitativo, el SLA se especifica en términos de unos indicadores; como, por ejemplo, tiempo de ejecución total, "*throughput"* (número de tareas ejecutadas por unidad de tiempo, consumo energético máximo o coste económico.

En tiempo de ejecución ("*runtime*"), los indicadores del SLA forman parte de un conjunto de variables denominadas "calidad del servicio" ("*Quality of Service"* QoS, en inglés). **El tiempo de ejecución desde el punto de vista del cliente (Texec-cliente) podemos dividirlo en tres factores fundamental mente: Texec-cliente = Txon + Texec + Tw, donde Txon es el tiempo de transmisión de la petición del cliente al servidor y el tiempo de transmisión de los resultados del servidor al cliente, Texec es el tiempo de ejecución efectivo de la petición y Tw es el tiempo de espera o bien tiempo de sobrecarga (overhead) que aparece debido a la gestión que realiza el servidor de la petición**. En esta práctica vamos a hacer uso de un único indicador de SLA, **el tiempo de ejecución**. Y el acuerdo va a consistir en que el **Texec-cliente** de la tarea total tiene que ser **Texec-cliente < Texec * 2,** el tiempo de ejecución total tiene que ser menor que el doble de tiempo de procesamiento efectivo de la tarea de forma aislada.

Podemos construir el sistema distribuido como queramos, siempre y cuando no se "viole" el acuerdo. Vamos a estudiar las arquitecturas cliente servidor secuencia, cliente servidor concurrente (creando una Goroutine por petición y con un pool de Goroutines) y máster worker.

Se pide

Implementar las 4 arquitecturas software descritas en la sección 2.3.2: (i) cliente-servidor secuencial, (ii) cliente servidor concurrente creando una Goroutine por petición, (iii) cliente servidor concurrente con un pool fijo de Goroutines y (iv) máster-worker. A analizar para qué carga de trabajo puede operar cada una de ellas sin violar el QoS (**Texec-cliente < Texec * 2**). Cada petición de cliente (tarea) consiste en encontrar todos los números primos existentes entre [1, 60 000]. La carga de trabajo se mide en términos de número de peticiones (tareas) que un cliente en vía 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. Después, el análisis teórico tiene que estar validado experimentalmente, confirmando vuestros análisis.

Se pide en la memoria:

1. Explicar el análisis teórico para cada arquitectura (1 página para cada arquitectura, incluyéndose los diagramas arquitecturales correspondientes), donde se justifique para qué carga de trabajo puede operar cada arquitectura sin violar el QoS
2. Para cada arquitectura, realizar una batería de experimentos (1 página para todos los experimentos) en los que se demuestre vuestro razonamiento anterior. Es necesario mostrar gráficas con los resultados de ejecución

Nota: Para garantizar la posible variabilidad en las mediciones, deberéis repetir cada medición 10 veces y realizar la media aritmética de todas las mediciones obtenidas.

4. Evaluación

La realización de las prácticas es por parejas, pero los dos componentes de la pareja deberán entregarla de forma individual. En general estos son los criterios de evaluación:

- Deben entregarse *todos* los programas, se valorará de forma negativa que falte algún programa.
- Todos los programas deben compilar correctamente, se valorará de forma muy negativa que no compile algún programa.
- Todos los programas deben funcionar correctamente como se especifica en el problema.

- Todos los programas tienen que seguir el estilo de codificación de Golang, que podéis obtener a través de la herramienta gofmt[11] (un 20% de la nota estará en función de este requisito). Además de cada fichero fuente deberá comenzar con la siguiente cabecera:

```
# AUTORES: nombres y apellidos
# NIAs: números de identificaci'on de los alumnos
# FICHERO: nombre del fichero
# FECHA: fecha de realizaci’on
# TIEMPO: tiempo en horas de codificación
# DESCRIPCI'ON: breve descripci'on del contenido del fichero
```

4.1. 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 rubrica 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.

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.

[11] https://golang.org/cmd/gofmt/

C (deficiente). El software no compila o presenta errores graves. La memoria no presenta una estructura coherente y/o el lenguaje utilizado es pobre y/o contiene errores gramaticales y/o ortográficos. En el caso del código, este no se ajusta exactamente a las guías de estilo propuestas.

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	1C		

Tabla 1. Rúbrica

5. 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 moodle2 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. Para que la práctica sea evaluada, durante la sesión de la práctica 2 se realizará la defensa de la práctica 1.