PRÁCTICA 1

Golang y las Arquitecturas Cliente Servidor y Master Worker

Aarón Ibáñez Espés 779088
Ángel Espinosa Gonzalo 775750

Introducción

El problema planteado en la práctica consiste en calcular todos los números primos que se encuentran en un intervalo dado. Para ello se pide implementar 4 arquitecturas distribuidas diferentes, para cada una de las cuales se ha realizado un análisis teórico y un conjunto de pruebas que refuerzan el resultado del análisis teórico.

1. Diseño Arquitectural

La comunicación entre el cliente y el servidor en todas las arquitecturas descritas a continuación se realiza mediante el paso de mensajes asíncronos por la red utilizando el protocolo de comunicación TCP/IP.

Cliente-Servidor Secuencial:

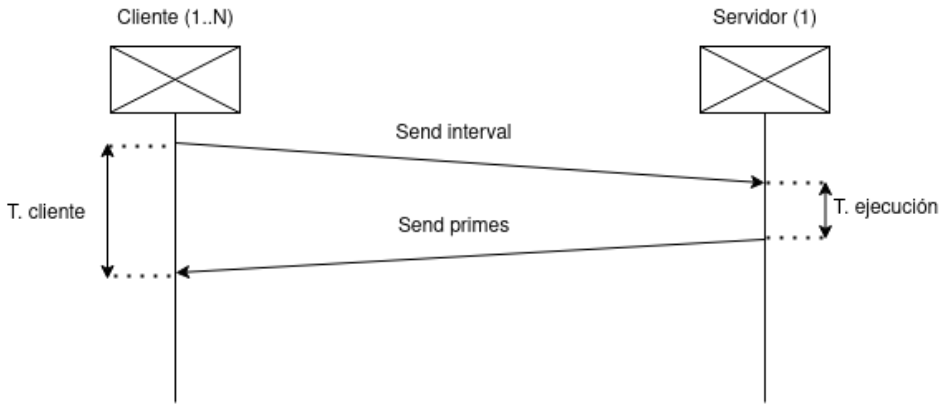


Figura 1. Diagrama de secuencias cliente servidor secuencial

El primer modelo de arquitectura implementado ha sido un modelo cliente servidor secuencial, es decir, el servidor solamente es capaz de atender a una petición simultáneamente, por lo que por más peticiones que lleguen estas van a ejecutarse una detrás de otra, generalmente, en orden de llegada.

Cliente-Servidor Concurrente:

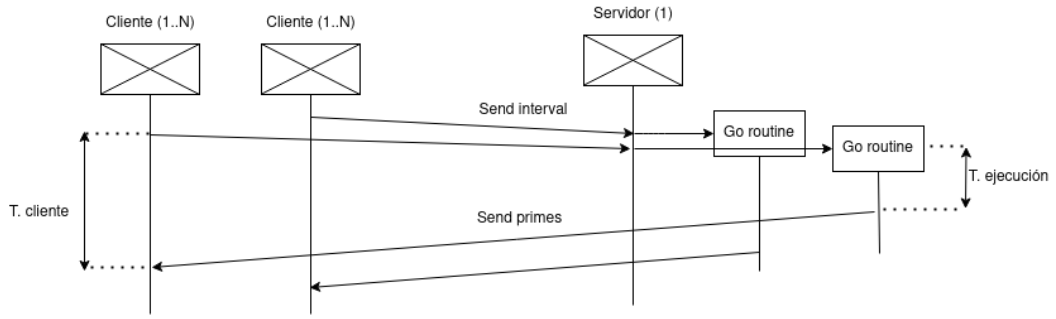


Figura 2. Diagrama de secuencias cliente servidor concurrente

La segunda arquitectura implementada ha sido la de cliente servidor concurrente. Esta arquitectura permite al servidor atender varias peticiones en paralelo. Para ello, el servidor lanza una Gorutine por cada una de las peticiones que le llegan, para que ésta la procese y devuelva el resultado al cliente, por lo que el servidor es capaz de procesar varias peticiones al mismo tiempo, es decir, de manera concurrente.

Cliente-Servidor Concurrente (Pool de Gorutines):

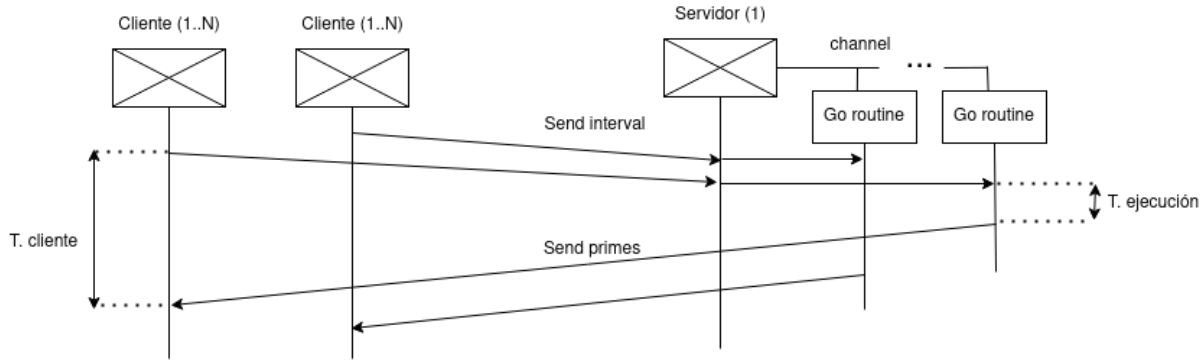


Figura 3. Diagrama de secuencias cliente servidor concurrente con pool de gorutines

La tercera arquitectura implementada es una arquitectura cliente servidor, en la que el servidor dispone de un conjunto (pool) de Gorutines fijo a las cuales asignar cada una de las peticiones que recibe. Para implementar la comunicación entre las Gorutines y el servidor se utiliza un canal síncrono, en el cual el servidor publica las peticiones conforme le llegan y las Gorutines las extraen del canal, las procesan y devuelven el resultado al cliente.

Máster-Worker:

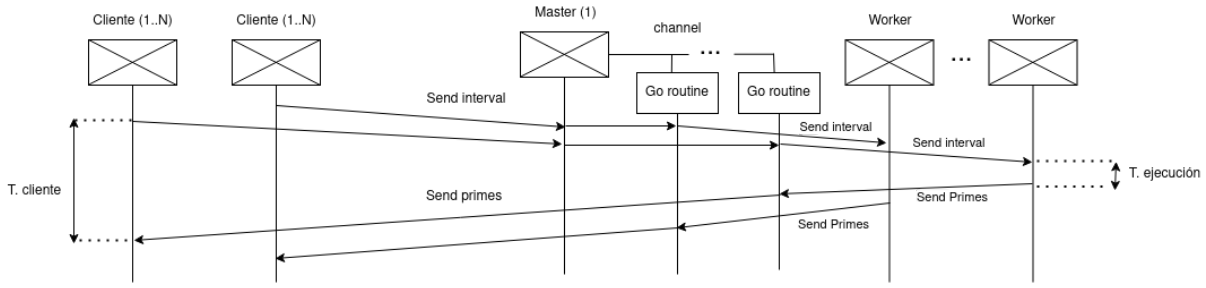


Figura 4. Diagrama de secuencias master-worker

La última arquitectura consiste en una arquitectura master-worker. Cliente y master se comunican mediante el paso asíncrono de mensajes. A su vez el master se comunica con un conjunto de Goroutines de forma síncrona y las Goroutines se comunican con los workers mediante el intercambio de mensajes asíncronos.

El máster dispone de un canal síncrono en el que están escuchando un número determinado de Goroutines. Cuando las Goroutines escuchan del canal pasan los parámetros recibidos a los workers que ejecutan el cálculo solicitado y se lo devuelven a la Goroutine. Una vez la Goroutine dispone del resultado se lo devuelve de nuevo al cliente. El master está continuamente atendiendo las peticiones mientras haya alguna Goroutine disponible para atenderla, en caso contrario se queda bloqueado intentando pasar los parámetros por el canal hasta que alguna Goroutine se quede libre y pueda atender la petición.

2. Análisis Teórico QoS

En este apartado se realiza el análisis teórico de rendimiento de cada una de las arquitecturas descritas en el apartado anterior. En este analisis se estima la carga a la que es capaz de ser sometida cada una de las arquitecturas sin violar el Quality of Service(QoS), es decir, (tex + txon + to < 2 ∗tex).

Durante el análisis y posterior validación experimental a la suma (tex + txon + to) se le llama tiempo total de cliente.

Cliente-Servidor Secuencial:

Esta arquitectura viola el QoS para cualquier número de clientes ya que solamente es capaz de atender 1 petición de forma simultánea, por lo que cualquier otra petición que llegue deberá de esperar a que termine el procesamiento de la anterior para comenzar con el suyo, esto supone un tiempo de espera y que el tiempo total de cliente sea superior al doble del tiempo de ejecución del servidor no cumpliendo así el QoS.

Existe un caso en el que el QoS se puede cumplir y es el caso en el que las peticiones lleguen con un lapso de tiempo superior al tiempo de ejecución del servidor, por lo que así, estas no tienen que esperar a ser procesadas y por lo tanto el QoS se cumpliría.
En la validación experimental realizada posteriormente se muestran los resultados obtenidos al probar esta arquitectura atendiendo peticiones de un cliente de prueba dado por el profesor.

Cliente-Servidor Concurrente:

Esta arquitectura no cumple con el QoS puesto que (tex + txon + to < 2 ∗tex) ya que al lanzar una Gorutine por cada petición que le llega al servidor estas se ejecutan concurrentemente. Sin embargo, cuando el número de Gorutines que se lanzan es superior al número de threads que el servidor es capaz de ejecutar el tiempo de ejecución del servidor se incrementa notablemente.

Esto es debido a que el scheduler para atender a todos los procesos que se ejecutan en paralelo de forma equitativa otorga cpu a unos y se la arrebata a otros lo que provoca el incremento en los tiempos mencionado anteriormente. Mientras el número de clientes sea menor que los threads que el procesador puede lanzar, habrá suficientes recursos para atender las peticiones con un tiempo de ejecución razonable.

En la validación experimental realizada posteriormente se muestran los resultados obtenidos al probar esta arquitectura con un cliente de prueba dado por el profesor y ligeramente modificado para mostrar ejemplos en los que se cumple y en lo que no se cumple el QoS.

Cliente-Servidor Concurrente (Pool de Gorutines):

El tamaño del conjunto de Gorutines que se ha establecido para esta arquitectura ha sido de 6, ya que los ordenadores del lab 1.02 tienen 6 cores.

Con el cliente de prueba que se ha proporcionado el QoS solamente se cumple si el número de clientes es menor o igual a 2, debido a que cada uno de los clientes realiza 6 peticiones de manera casi simultánea, que son asignadas cada una a una de las Gorutines de la pool. En el momento en el que se conectan más de 2 clientes, la pool no es suficiente para atender todas las peticiones que están llegando por lo que las peticiones se acumulan en el canal síncrono de comunicación esperando a su procesado, lo que conlleva que el cliente tenga que esperar a que su petición sea procesada y se viole el QoS siendo el tiempo final (denominado T.cliente en el diagrama) muy superior a 2*Tex.

En la validación experimental realizada posteriormente se muestran los resultados obtenidos al probar esta arquitectura atendiendo peticiones de un cliente de prueba.

Máster-Worker:

Se cumple el QoS para esta arquitectura mientras el número de peticiones no supere el número de workers disponibles, cuando esto suceda los clientes que intenten ser atendidos cuando no haya ningún worker disponible quedarán bloqueados hasta que se quede alguno libre.
Estos bloqueos aumentaran tex + txon + to y no se cumplirá el QoS. Por lo tanto se esperan buenos resultados con un número bajo de clientes.

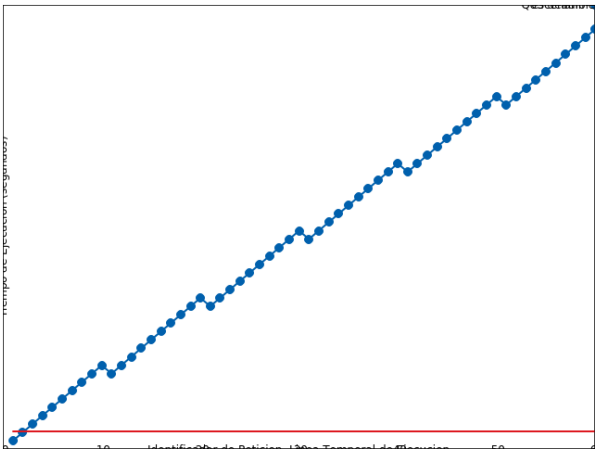
3. Validación experimental

A continuación se detallan con unos ejemplos gráficos los resultados obtenidos al realizar pruebas a todas las arquitecturas implementadas utilizando los ordenadores del Lab 1.02 y el cliente de prueba proporcionado por los profesores.

Durante la validación experimental a la suma (tex + txon + to) se le llama tiempo total de cliente.

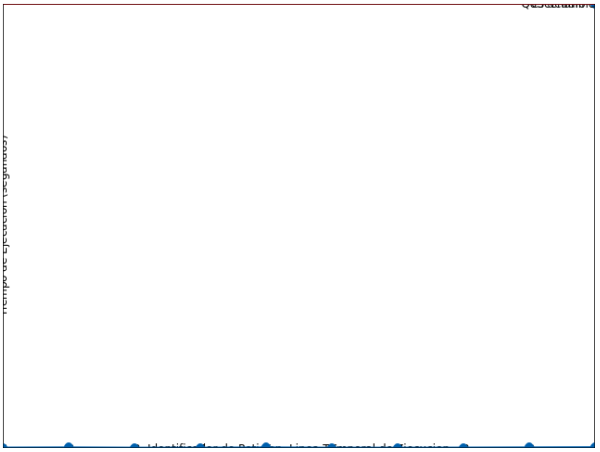
Para la generación de las gráficas utilizadas como apoyo en la valoración experimental se ha utilizado la herramienta GNU Plot, más concretamente el script plot.sh, proporcionado por los profesores, estableciendo el QoS en 3,22 s, ya que el tex medio es de 1,61 s.

Cliente-Servidor Secuencial:



Gráfica 1. Validación experimental servidor secuencial

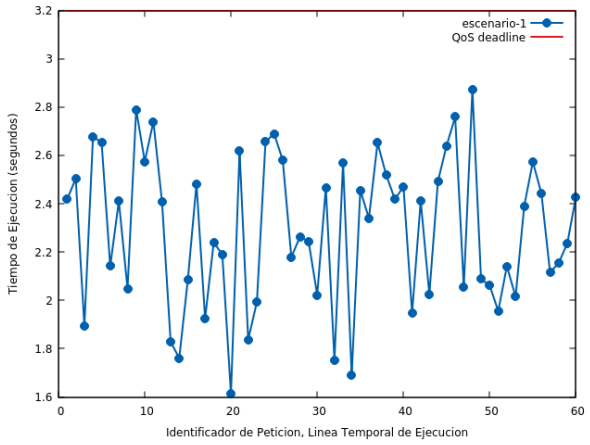
En la Gráfica 1 se muestran los resultados obtenidos al ejecutar el servidor secuencial con 1 cliente de prueba. En ella se puede ver representado con la línea roja el umbral máximo del QoS y con la línea azul el tiempo total de cliente. Se observa como el QoS se viola con solamente un cliente debido a que al realizar tantas peticiones de manera muy seguida estas tienen que esperar a que acabe la que se está procesando para poder ser atendida, lo que implica que el tiempo de espera de la petición sea muy alto y por ello se viole el QoS. Cabe destacar que cada 6 peticiones se observa en la gráfica un pequeño decremento del tiempo del cliente, esto es debido a que el cliente lanza las peticiones en “paquetes” de 6 esperando un tiempo entre ellos.



Gráfica 2. Validación experimental servidor secuencial

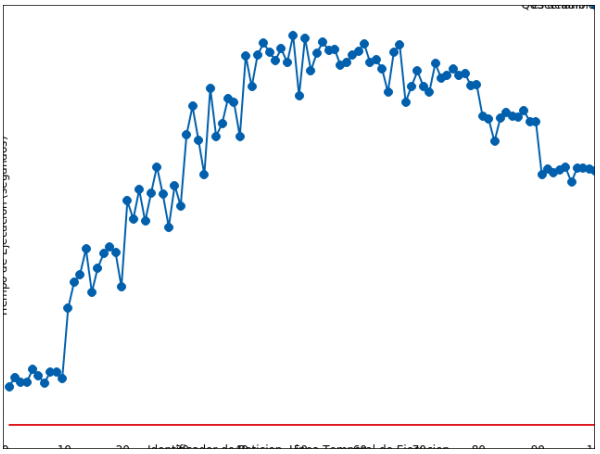
En la gráfica 2 se muestra un ejemplo en el que el servidor secuencial no viola el QoS. Esto se debe a que se ha modificado el cliente de prueba proporcionado para que envíe las peticiones cada 2 segundos, tiempo superior al tiempo de procesamiento de la petición en el servidor, por lo que así no se acumulan las peticiones y se cumple el QoS.

Cliente-Servidor Concurrente:



Gráfica 3. Validación experimental servidor concurrente

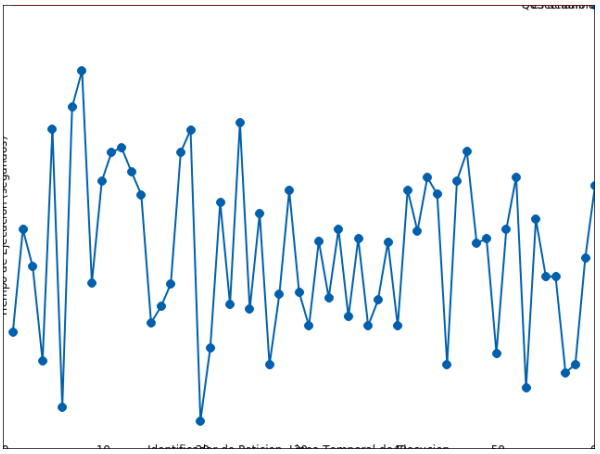
En la Gráfica 2 se muestran los resultados obtenidos al ejecutar el servidor concurrente con un cliente de prueba. En ella se puede observar representado en azul el tiempo de ejecución de cada petición realizada por el cliente de prueba y con la línea roja, el umbral del QoS. Esto se debe a que el cliente de prueba envía las peticiones de 6 en 6 y el servidor es capaz de procesar las 6 de forma concurrente, ya que dispone de 6 cores.



Gráfica 4. Validación experimental servidor concurrente

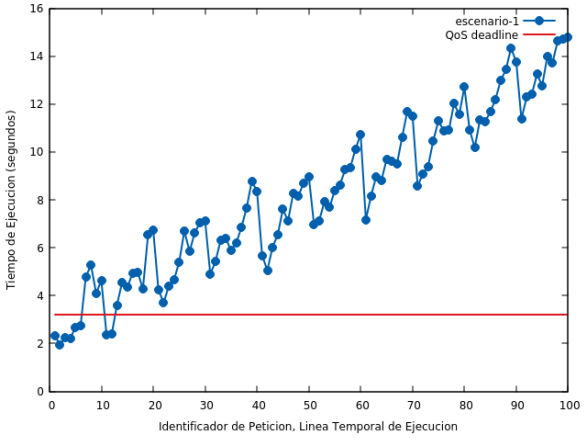
En la gráfica 4 se muestran los resultados obtenidos de modificar el cliente para que el número de peticiones que se envíen en cada bloque pase de 6 a 10 (RequestTMP: 10). Por ello se observa como en este caso el servidor no es capaz de procesar las 10 peticiones de manera concurrente, ya que no dispone de recursos suficientes para ello y el tiempo de procesamiento de cada petición aumenta considerablemente violando así el QoS.

Cliente-Servidor Concurrente (Pool de Gorutines):



Gráfica 5. Validación experimental servidor pool gorutines

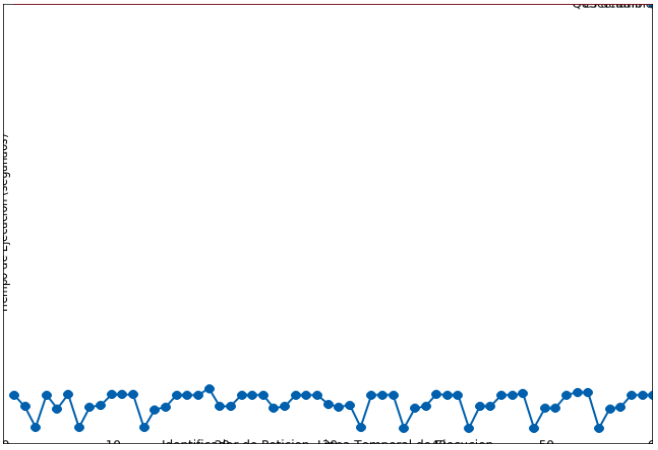
En la Gráfica 5 se muestran los resultados obtenidos al ejecutar el servidor concurrente con un conjunto (pool) de Gorutinas fijo con un cliente de prueba. En la gráfica se puede ver representado en azul el tiempo de ejecución de cada una de las peticiones que realiza el cliente y con la línea roja (apreciable en la parte superior) el umbral del QoS. En esta prueba se cumple con el QoS, ya que el cliente envía las peticiones en bloques de 6, siendo este el tamaño de la pool establecido en el servidor, por lo que cada una de las peticiones es procesada por una de las gorutines y no hay tiempo de espera para que se procesen.



Gráfica 6. Validación experimental servidor pool gorutines

En la gráfica 6 se muestran los resultados obtenidos de modificar el cliente para que el número de peticiones que se envíen en cada bloque sean 10 (RequestTMP: 10). Esto hace que la pool de gorutines sea inferior al número de peticiones que se envían y por ello el servidor no sea capaz de procesarlas concurrentemente teniendo que esperar para ser procesadas y violando así el QoS.

Máster-Worker:



Gráfica 7. Validación experimental master worker

En la Gráfica 7 se muestran los resultados obtenidos al ejecutar el cliente con el despliegue de 12 workers. En la gráfica se puede ver representado en azul el tiempo de ejecución de cada una de las peticiones que realiza el cliente y con la línea roja (apreciable en la parte superior) el umbral del QoS. En esta prueba se cumple con el QoS, ya que el cliente envía las peticiones en bloques de 6 y se disponen de 12 workers para procesar las peticiones, por lo que los recursos del servidor son más que suficientes para atender todas las peticiones.

4. Conclusiones

Finalmente se puede concluir que el análisis teórico realizado para las arquitecturas secuencial, concurrente y concurrente con pool de Gorutinas ha sido correcto, habiendo fallado en el análisis de la arquitectura master worker.
Además, se concluye que la arquitectura distribuida más útil para el procesamiento de peticiones a gran escala es la arquitectura master worker debido a que se puede lanzar en máquinas diferentes y optimizar así en mayor medida los recursos disponibles.