Sistemas distribuidos - Recuerdo practica 1

Esposito Inchiostro Alessio - 847803, Jesús Mendoza Arata - 777524

[1]

En esta memoría se analizará el uso de las Goroutines del lenguaje Golang y de como se puedan utilizar para implementar un servidor que pueda responder a más solicitudes de diferentes clientes a lo mismo tiempo, y cuales son las mejores implementaciónes para garantizar el mejor servicio posible.

1. Descripción del problema

Tenemos un ordenador del lab 1.02 para utilizar como servidor para un servicio de busqueda de numeros primos. Esto recibirà por los clientes mensajes que contienen un ID de la solicitud y dos numeros que delimitan un intervalo en lo que buscar los numeros primos. Tenemos que implementar en maneras diferentes el servicio y analizar cuales son las mejores. Las especificación tecnicás que nos interesan más son el procesor i5-9500 con 6 threads en total, y los 32 GB de RAM del ordernador que se supone que utilizar como servidor.

2. Descripción de las implementaciones

En esta practica hemos implementado cuatro diferentes maneras de servidor, que son las siguentes:

- Arquitectura servidor secuencial, que simplemente procesa una solicitud a la vez.
- Arquitectura servidor concurrente, que genera una Goroutine (la implementación Golang de funciones ejecutadas en manera concurrente) para cada solicitud y las ejecuta todas a lo mismo tiempo.
- Arquitectura servidor concurrente con "Goroutines Fixed Pool", que es una implementación más eficiente de la *Servidor Concurrente* porque tiene un numero fijado de Goroutines (el numero ideál y la razón de esto seran analizadas en esta memoria) que ejecutan lo que es necesario para responder a las solicitudes.
- Arquitectura Master-Worker, en la que hay un servidor (Master) que recibe las solicitudes y comparte en tamaño entre los Worker (otros ordenadores, normalmente en la misma red que solo ejecutan codigo *CPU-intensive* y que "obedecen" al Master

3. Análisis teórico sobre la carga de trabajo maxima

Ya que Golang nos permite de saber cuantos son los threads totales del ordenador que está ejecutando el codigo, vamos a generalizar este numero utilizando el nombre N_t. El tiempo de ejecución efectivo de la solicidud (ignorando el overhead y el tiempo de trasmisión será llamado T_e. Tambien se recuerda che el tiempo acceptable de respuesta a una solicitud es $2 * T_e$.

3.1. Servidor secuenciál

Ya que esta arquitectura solo nos permite de ejecutar una solicitud a la vez, la carga de trabajo maxima C_m será

$$\frac{1 solicitud}{T_e + T_o + T_t} < C_m < \frac{2 solicitudes}{T_e + T_o + T_t}$$

Menos será el overhead y más la carga que el servidor puede elaborar será cercana de $\frac{2 solicitudes}{T_e + T_o + T_t}$.

3.2. Servidor concurrente

Para analizar la carga de trabajo maxima tenemos que entender como funcionan las Goroutines y como estas sono ejecutadas por el ordenador. Golang es un lenguaje hecho a propósito para garantizar un overhead minimo para lo que tiene que ver con la multiprogramación. Sobre internet se pueden buscar muchos debates que confirman que se puedan crear muchas Goroutines sin particular overhead causado por la creación o por el context switch (eso porque es Golang que gestiona completamente las Goroutines). Pero, tambien si es verdad que Golang permita de generar muchisímas Goroutines, eso no significa que efectivamente crearlas sea mejor. De echo, si se generan más Goroutines de N_t voy a perder eficiencia, porque tambien si son muy bajos se causaran overheads. Entonces la carga de trabajo maxima ideal para esta arquitectura será

$$\frac{N_t}{T_e + T_o + T_t} solicitudes$$

Claramente lo que es **ideal** no es *real*, entonces ejecutando un servidor con este tipo de arquitectura tenemos que tener en cuenta que **más solicitudes el servidor recibirá, más estará ralentizado**

3.3. Servidor concurrente con pool fijada de Goroutines

Ya que se ha dibatido en la sección anterior que el numero ideal de Goroutines que se deberían generar es igual a N_t, la carga de trabajo maxima será la misma del servidor concurrente. En este caso pero el servidor no lanzará infinitas Goroutines, simplemente

$$\frac{N_t}{T_e + T_o + T_t} solicitudes$$

La diferencia entre esta y la otra arquitectura es que si el servidor recibe más solicitudes de las que puede procesar no se ralentizará, simplemente siguerá procesando N_t solicitudes a la vez.

3.4. Arquitectura Master-Worker

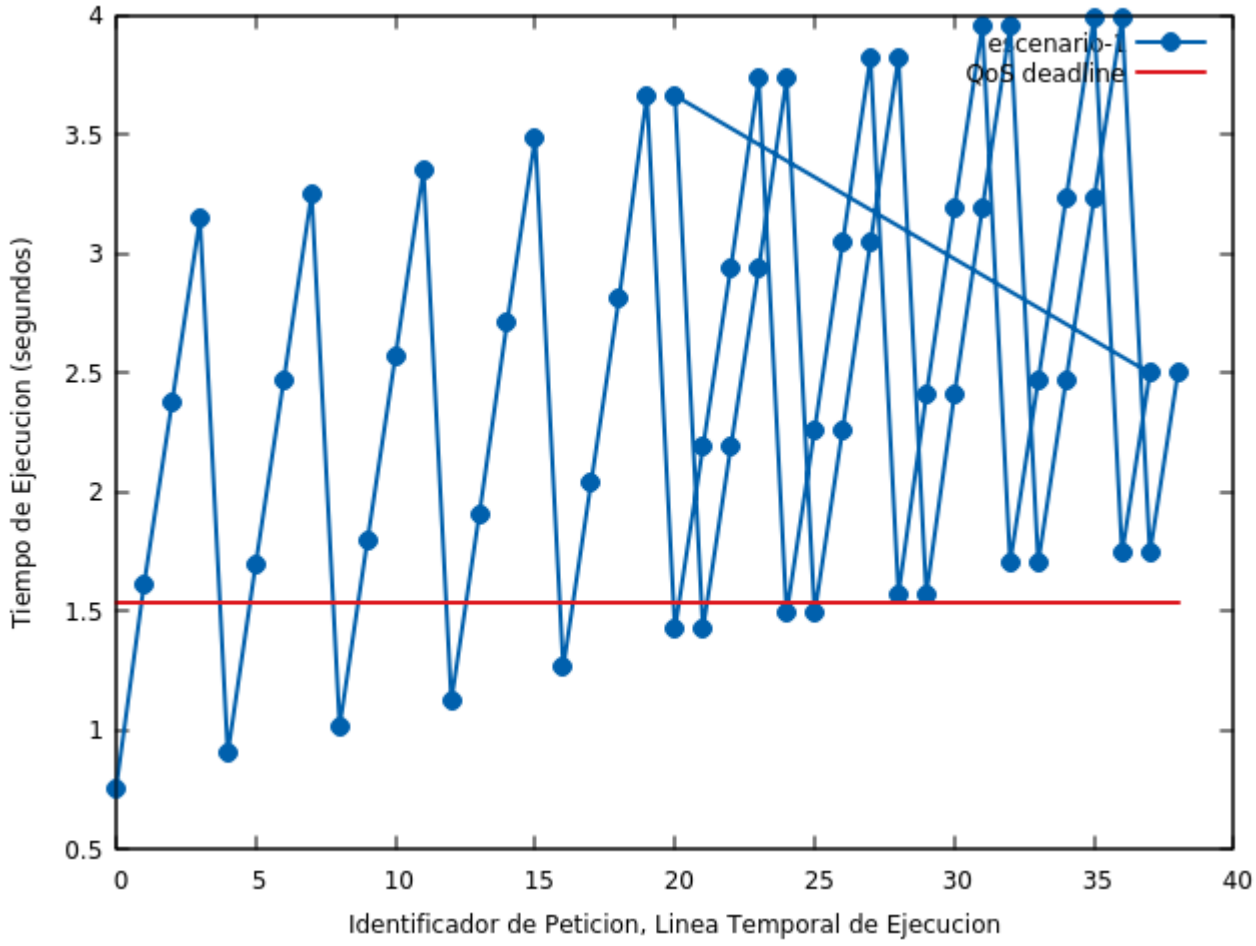
Como se mencionó antes, la arquitectura Master-Worker incluye un servidor master que recibe las solicitudes de los clientes, las envia a los Workers y cuando esos terminan la elaboración envia la respuesta al client. Se puede suponer que los Workers sean N_w ordenadores con pool fijada de Goroutines (ya que esa es la arquitectura más eficiente para un singolo ordenador) y entonces que como carga de trabajo maxima tengan la misma de las arquitecturas servidor concurrente con pool de Goroutines $\frac{N_c}{T_e+T_o+T_t} solicitudes$. La diferencia es que el master tiene disponibles N_w workers, entonces la carga de trabajo maxima será

$$\frac{N_w * N_t}{T_e + T_o + T_t} solicitudes$$

4. Resultados prácticos

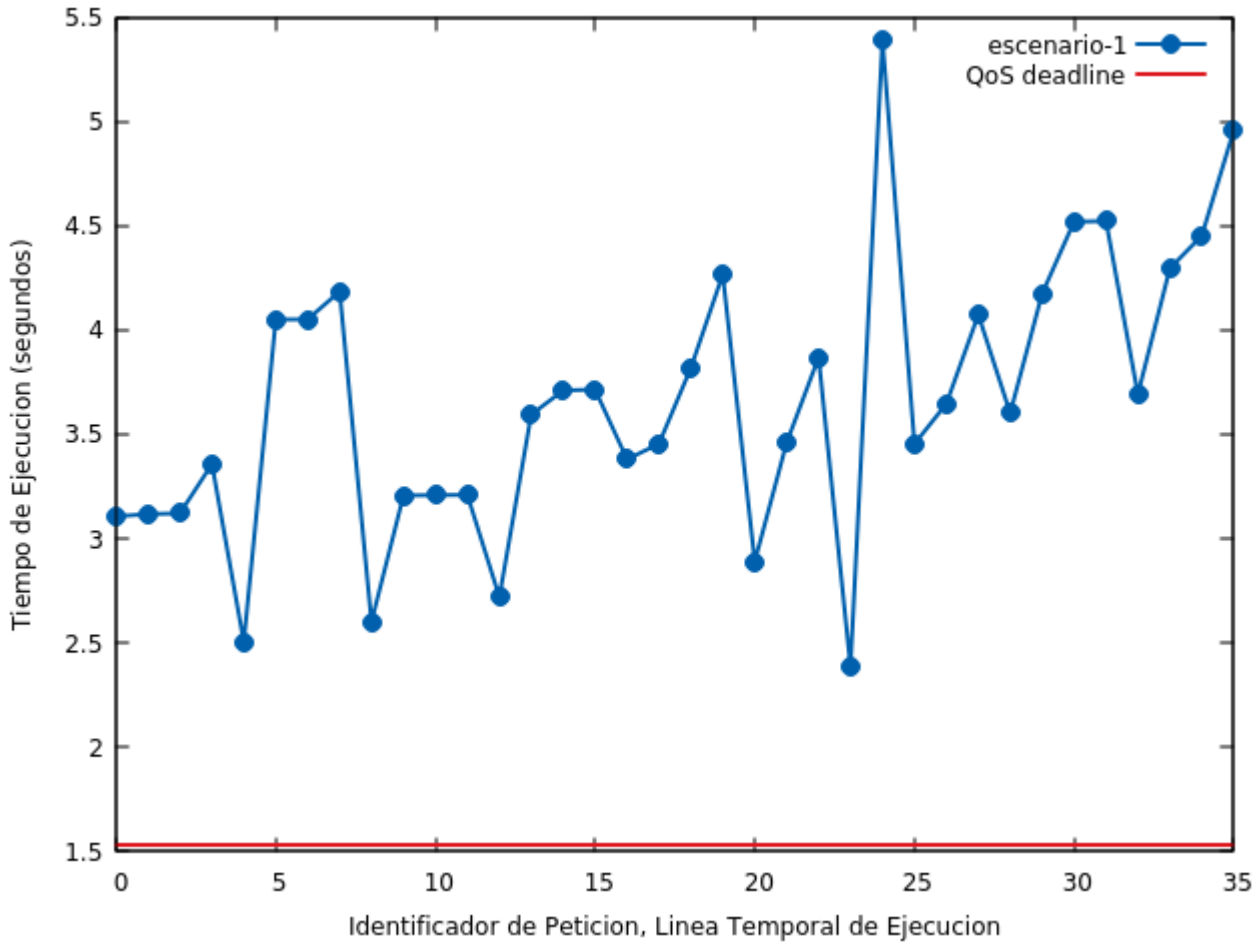
Ya que los ordenadores del Lab1.02 no son siempre disponibles, hemos utilizado los nuestros ordenadores para ejecutar los servidores y mirar a los resultados. Hemos creado uno script, benchmark.go que ejecuta 50 veces findPrimes(1000, 70000) y nos dice el tiempo medio de ejecuciòn. Desde esto podemos calcular $2Te$ para el QoS. Tambien para simular en manera mejor un entorno de ejecución màs similar a lo del Lab1.02 hemos utilizado la funciòn runtime.GOMAXPROCS(0) para obtener el numero de threads totales del ordenador y enviar un numero proporcional de solicitudes al numero de threads.

4.1. Servidor secuenciál



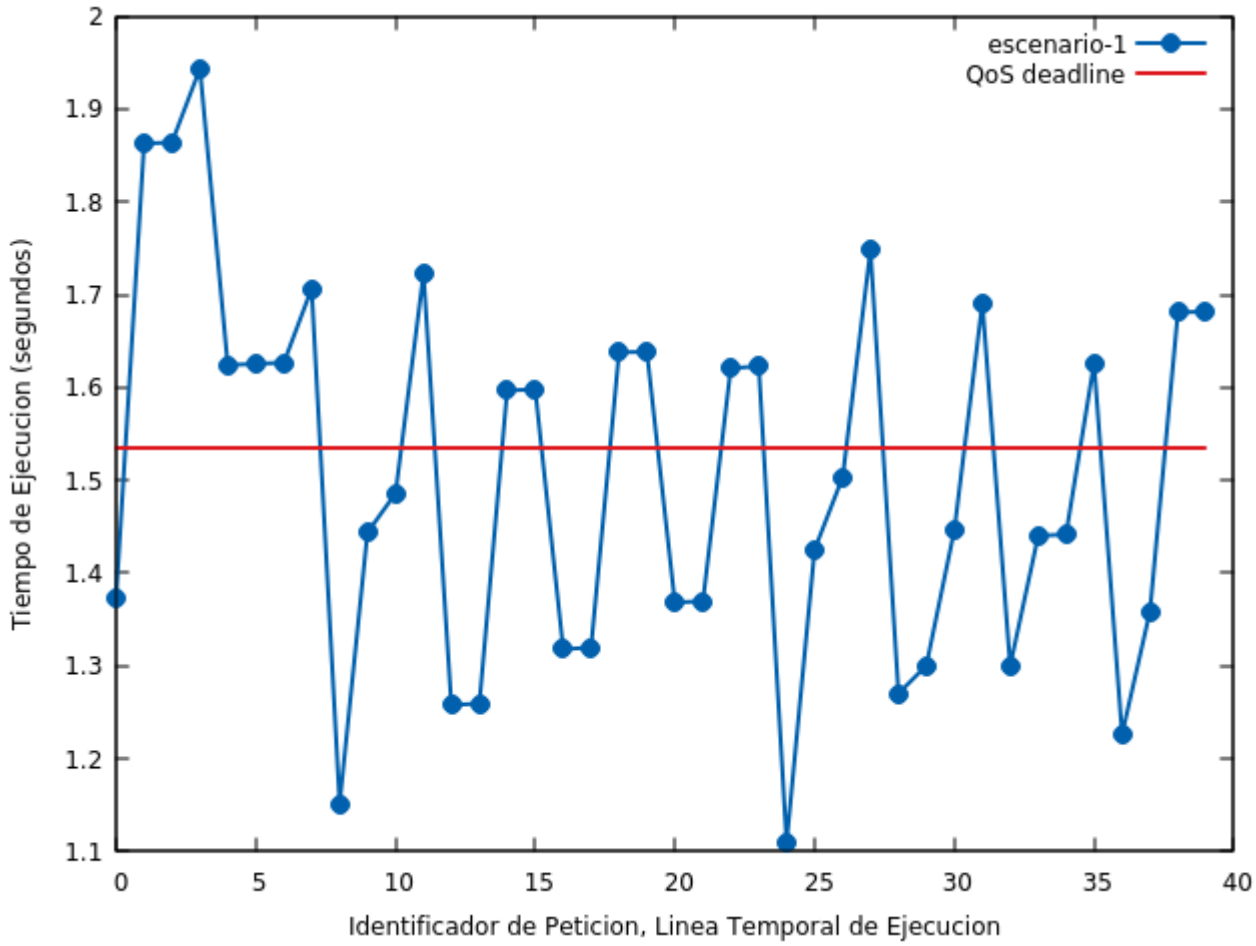
El servidor secuencial solo satisface la primera solicitud de cada burst de antes de romper la restricción de la QoS, pero las solicitudes siguentes hacen cola hasta que ninguna recuesta satisface el vinculo de QoS.

4.2. Servidor concurrente



Como ya ha sido analizado, más solicitudes el servidor concurrente recibe más allá de su carga maxima, y mas se ralentizará. El resultado nos confirma que esta arquitectura es menos eficiente de la secuencial.

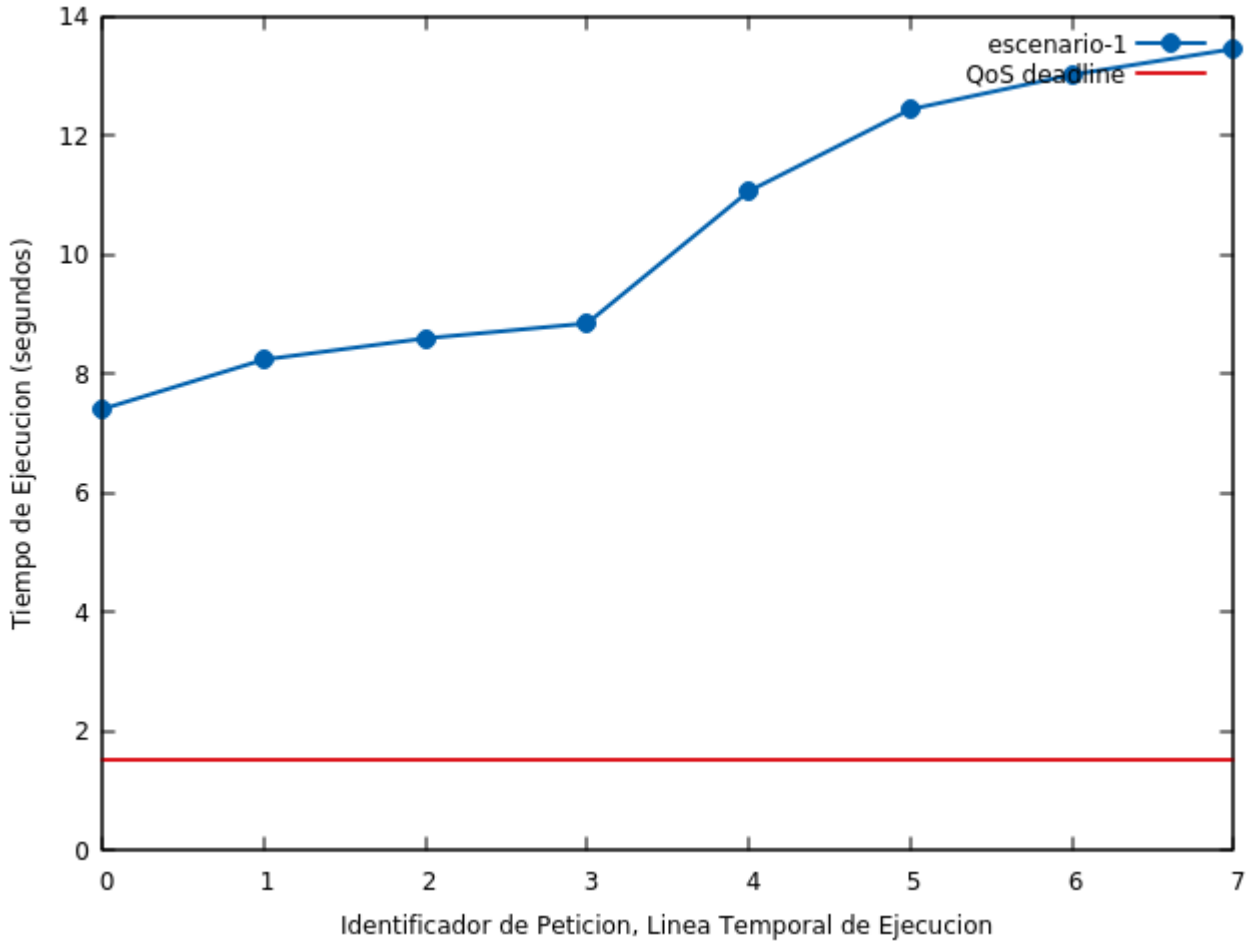
4.3. Servidor concurrente con pool fijada de Goroutines



El servidor contesta a un numero fijado maximo de solicitudes, el numero efectivo de solicitudes que recibe es irrelevante. Claramente si recibe más solicitudes de las que puede procesar va a romper la restricción de la QoS. Muchas de las respuestas en retraso estàn a 0.15 segundos de la linea de QoS. No es perfecto pero un retraso equivalente al 10% de $2Te$ es todavìa acceptable.

4.4. Arquitectura Master-Worker

La arquitectura con pool de Goroutines tiene un defecto: se podrìa utilizar el ordenador màs rapido del mundo, pero en algún momento no serà màs posible responder a todas las peticciònes si esas aumentarìan infinitamente. Esto significa que la arquitectura no es **scalable**. Hemos implementado una versiòn funcionante de la metodologia master-worker pero probablemente el no utilizar gob para el encoding y decoding nos ha costado mucho tiempo de overhead, entonces el servidor no satisface los requisitos.


escenario-1
QoS deadline
Tiempo de Ejecucion (segundos)
Identificador de Peticion, Linea Temporal de Ejecucion
0
2
4
6
8
10
12
14
0
1
2
3
4
5
6
7