Sistemas Distribuídos 2021-2022

- Práctica 1 -

Introducción a las Arquitecturas de Sistemas Distribuídos

Jorge Lisa 774248
David Zandundo 780500

Introducción del Problema

Para la realización de la práctica se pide desarrollar cuatro arquitecturas de tipo cliente-servidor.

La arquitectura cliente-servidor consiste en un proceso servidor, que aglutina la mayor parte de la funcionalidad, y un conjunto de procesos clientes que solicitan al servidor esta funcionalidad mediante el intercambio de mensajes.

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 dado como argumento. La aplicación es sobre todo intensiva en CPU, puede no requiere de mucha red de comunicación y no requiere de almacenamiento.

Para calcular el tiempo de ejecución, se va a usar un programa auxiliar llamado tex.go que es capaz de medir distintos costes.

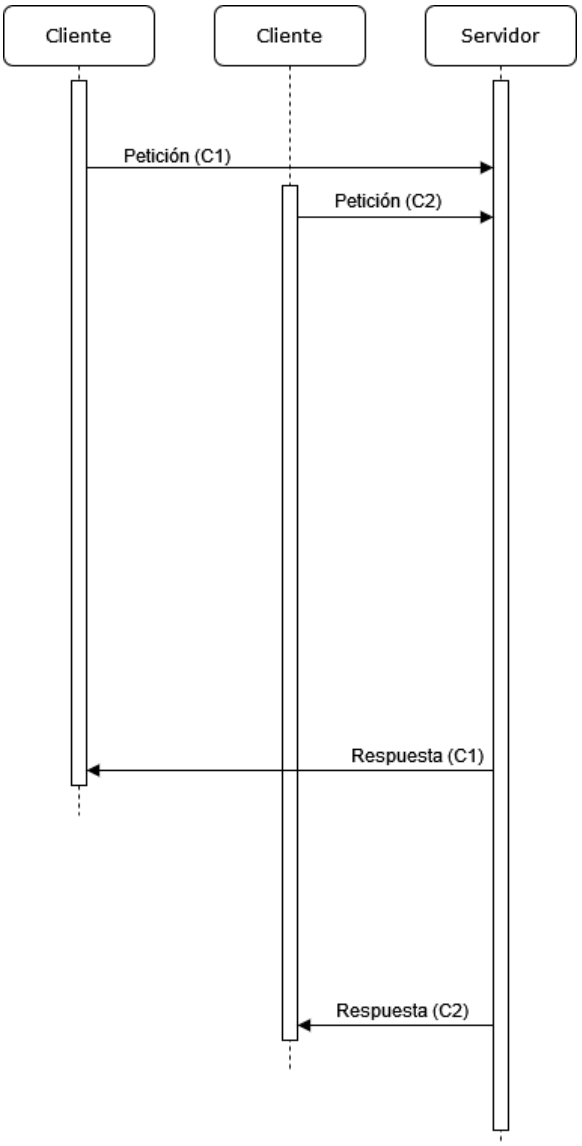
Para la comunicación a través de canales (o a través de funciones) en las arquitecturas concurrentes y master worker, se ha creado un struct llamado “Mensaje” que guarda el encoder del cliente (para que la rutina haga el trabajo de enviar a través del encoder) y el intervalo que debe calcular.

Por último, se ha modificado el cliente con un canal de bool que permite la espera de todas las peticiones realizadas y así poder realizar mejores gráficas analíticas de tiempos de ejecución.

Diseño de las Arquitecturas

Servidor Secuencial

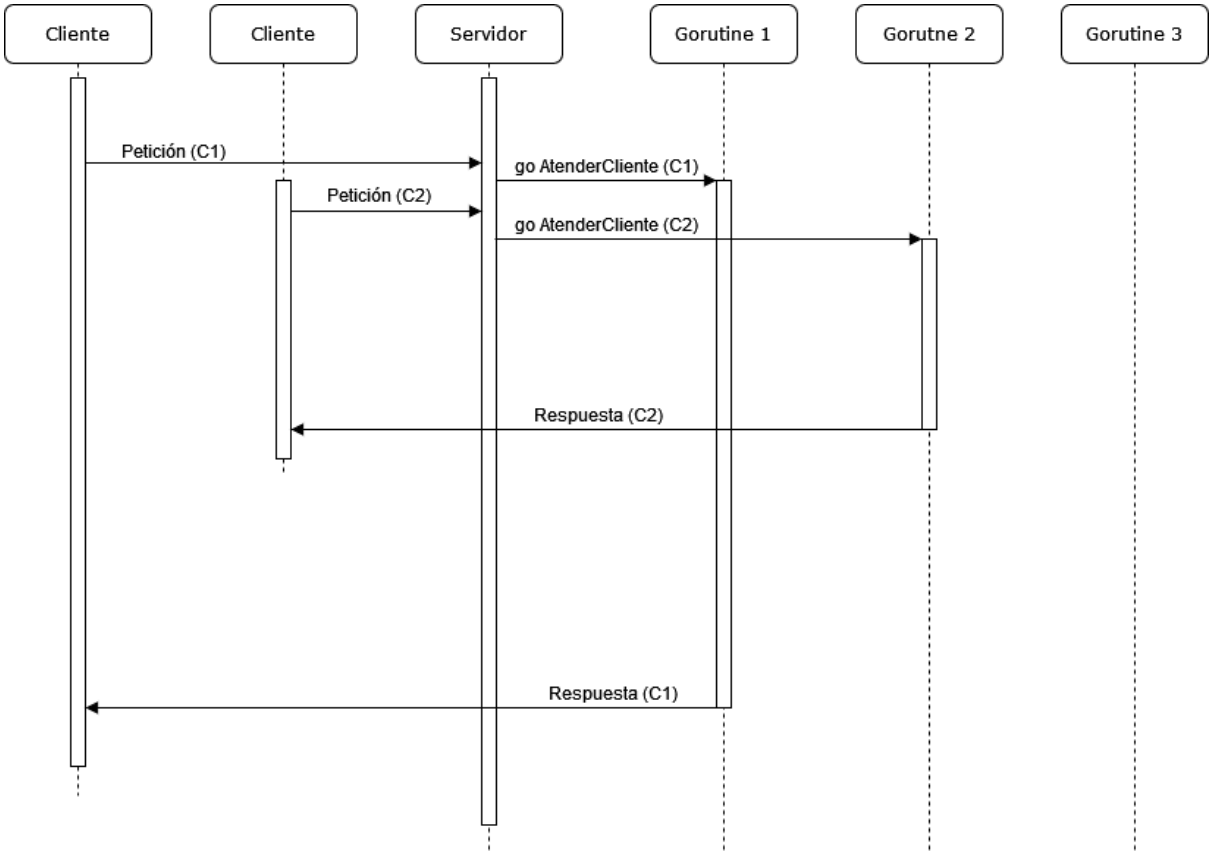
La arquitectura cliente servidor secuencial consiste en un servidor que atiende peticiones de forma secuencial, de una en una, de manera que cuando llegan varias peticiones, atiende una de ellas (a menudo la primera en llegar) y, una vez terminada, atiende la siguiente.
El esquema de la arquitectura será por lo tanto el siguiente:



Servidor con Gorutines dinámicas

La arquitectura cliente servidor concurrente consiste en un servidor que puede atender varias peticiones en paralelo. Para ello, 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.

El esquema de la arquitectura será por lo tanto el siguiente:

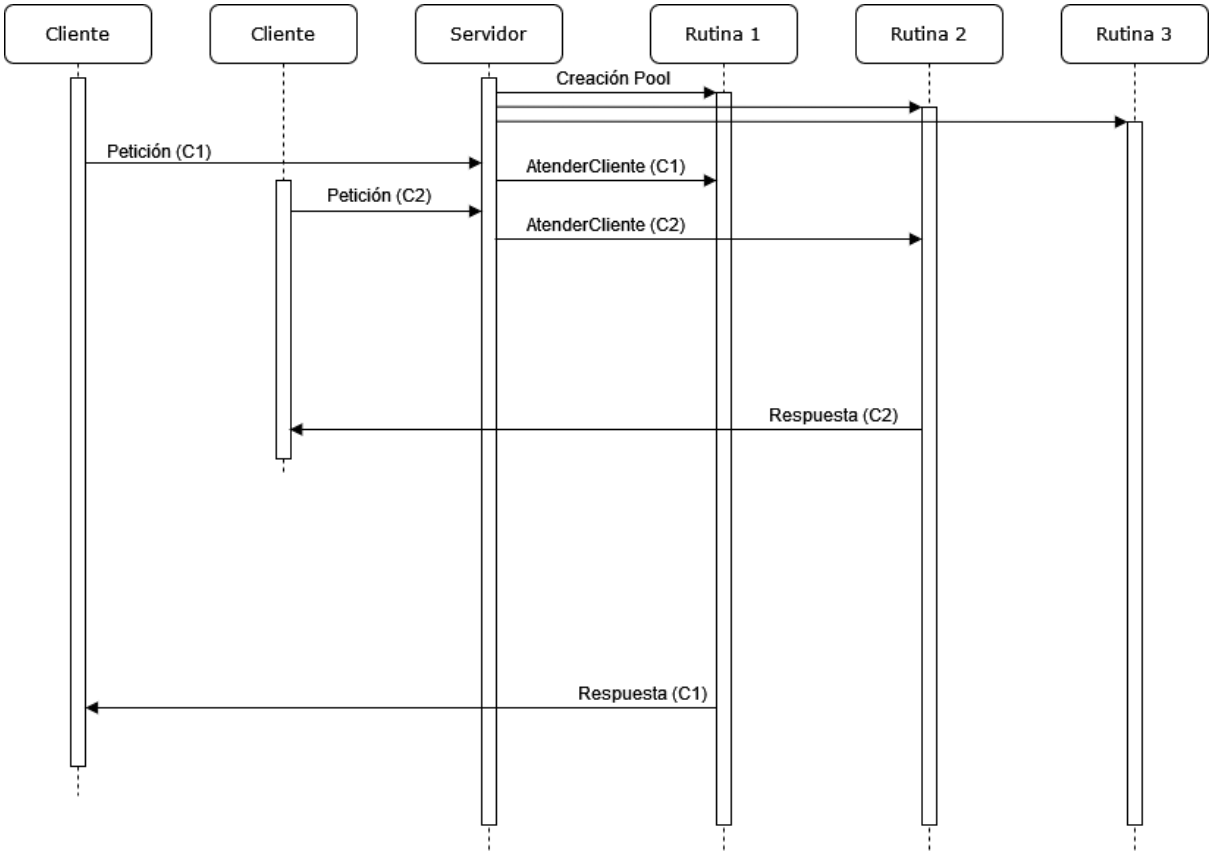


Servidor con Gorutines fijadas por pool

La arquitectura cliente servidor concurrente por pool consiste en un servidor que puede atender a un número fijo de peticiones en paralelo, ya que al crear una Goroutine por cada petición conlleva un sobrecoste en tiempo.

Para ello, se tiene un conjunto fijo de Goroutines 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.

El esquema de la arquitectura será por lo tanto el siguiente:

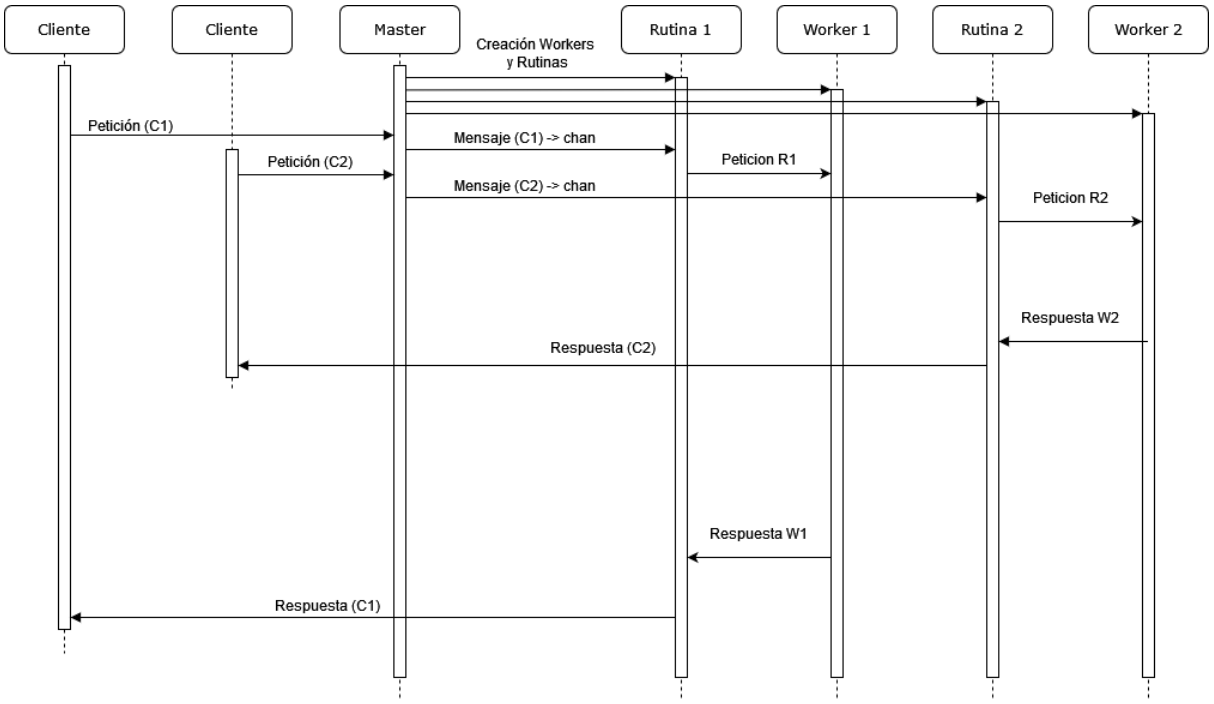


Hay que recalcar que al hacerse la comunicación a través de un canal, cualquier rutina puede coger el Mensaje, sin prioridad.

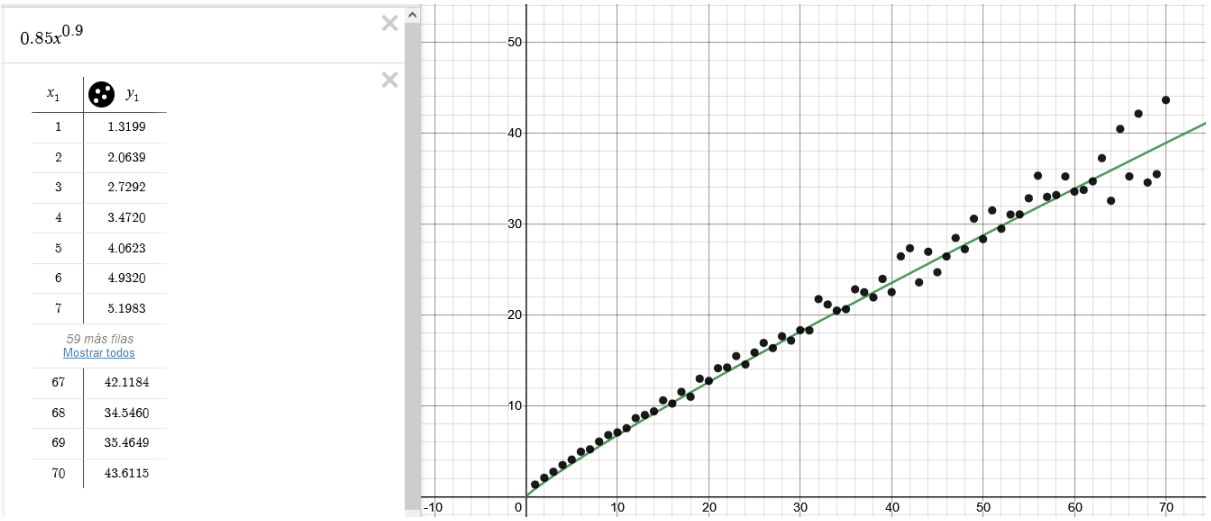
Master - Worker

La Arquitectura Master-Worker puede verse como una extensión de la arquitectura cliente-servidor concurrente con un pool de Goroutines, en la que existe un programa principal (master) 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.

El esquema de la arquitectura será por lo tanto el siguiente:



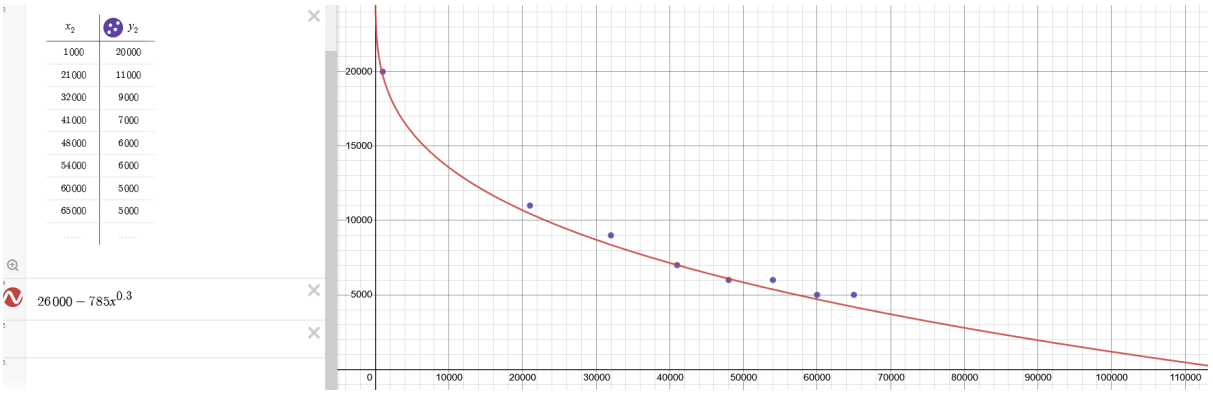
Para la arquitectura Master-Worker, se ha probado a diseñar una manera de separar la carga de trabajo entre los trabajadores de una forma equitativa. Como el intervalo a calcular es fijo, primero se ha partido el intervalo en trozos más pequeños de 1000 números cada uno y se han buscado sus primos. Con esto obtenemos el coste en tiempo que tarda cada intervalo, se ha hecho una gráfica y obtenido una función aproximada:



La función elegida que mejor ajusta los puntos es $0.85x^{0.9}$. Posteriormente, se han integrado por partes para obtener un área entre puntos lo más parecida posible, empezando por 1000. Tras la integración, se obtiene que una forma subóptima de dividir el intervalo es la siguiente:

1.000 -> 21.000 -> 32.000 -> 41.000 -> 48.000 -> 54.000 -> 60.000 -> 65.000 -> 70.000

Por último se ha buscado una función que aproxime estos puntos:



Con esto obtenemos una función capaz de obtener el final de un sub-intervalo dando el principio del mismo. Esta función es 26.000 - $785x^{0.3}$, la cual va a ser usada en go.

Si se desease reducir el coste en tiempo, valdría con elegir resultados de la integración más pequeños, pero gracias a este análisis, obtenemos que cada intervalo tarda en ser calculado unos 0.16 segundos.

Esto acabó quedandose en un concepto capaz de mejorar el rendimiento, ya que no conseguimos encontrar la forma de saber desde el master si un worker esta disponible para la lectura sin realizar esperas activas con llamadas dial (las cuales costaban más que realizar el findPrimes de todo el intervalo) o como construir canales distribuídos, los cuales permitirían hacer lo mismo que la pool de gorutines, pero de forma distribuída.

Al final optamos por la opción más sencilla, que cada gorutine tenga asociada un worker, donde un canal de mensajes envía a la rutina el trabajo y el canal de respuesta y cada worker calcula dicho trabajo.

Análisis de la carga de trabajo

Teniendo en cuenta que: $t_{ex} + t_{xon} + t_o < 2 * t_{ex}$ para tener un buen Quality of Service, se han calculado el tiempo de ejecución para cada arquitectura

Cálculo del Tiempo de Ejecución t_{ex}

Para obtener el cálculo del tiempo de ejecución, se ha creado un programa que calcula los primos en la misma máquina.
En hendrix, ejecutando 25 veces la búsqueda del intervalo dado, muestra una media de 1.489118664 segundos a las 14:30 de un domingo de pilares (solo yo estaba conectado).

Cálculo del Tiempo de Overhead t_o

Este valor varía dependiendo de la arquitectura, por lo tanto, sabiendo el coste de obtención de los primos, podemos obtener de forma teórica este tiempo:

Arquitectura Secuencial

En la arquitectura secuencial, el tiempo de overhead sería igual al número de procesos en cola por el tiempo de ejecución:

$$t_o = n * 1{,}489118664s$$

Donde n equivale al número de procesos antes que el actual.
Esto es un gran problema ya que si hay 2 procesos esperando, suponiendo que estamos en condiciones ideales con $t_{xon} = 0$, obtendríamos que $t_{ex} + t_o = 1{,}489118664 + 2 * 1{,}489118664 = 4.4673$, por lo tanto la desigualdad que marca el QoS se rompe, ya que 4,467355992 ≮ 2,978237328, sin condiciones ideales, sobraría con 1.

Arquitectura Concurrente de Gorutines Dinámicas

En esta arquitectura, el tiempo de Overhead no se tiene en cuenta mientras haya un número de procesos en ejecución menor o igual al número de núcleos de la máquina.
Pero, también hay que tener en cuenta que la creación de Gorutines tras aceptar un cliente tiene un coste que se debe sumar al tiempo de ejecución.
Se ha modificado el programa anterior para que lance una gorutine a una función sin contenido, teniendo en cuenta que la máquina con la que estamos trabajando tiene un número de núcleos limitado.
Con esto en cuenta obtenemos que el tiempo de ejecución en esta arquitectura es:

$$tex = 1{,}489118664s + 360\ ns = 1{,}489119024s$$

Por lo tanto, este tiempo se debe calcular como:

$$t_o = 0 \qquad n <= Nk$$
$$t_o \cong n * 1{,}489119024 \qquad n > Nk$$

Donde n es igual al número de procesos en espera y Nk es el número de kernels que tiene la máquina.
En esta arquitectura, la QoS se debería romper cuando tenemos 2 o más procesos esperando que el número de núcleos que tiene la máquina (6 en nuestro sistema).
Hay que tener en cuenta que los procesos se van liberando y no siempre deben esperar los 1,48 segundos, puede variar dependiendo de cuando salga el proceso, he de ahí el igual aproximado. Esto seguirá ocurriendo en las próximas arquitecturas.

Arquitectura Concurrente de Gorutines Fijas

En esta arquitectura tenemos el mismo problema que con con la anterior, pero eliminando el coste de creación de las Gorutinas.

$$t_o = 0 \qquad n <= Pool$$
$$t_o \cong n * 1{,}489118664 \qquad n > Pool$$

Donde n es igual al número de procesos en espera y Pool es el número de Gorutines que creamos en el lanzamiento del servidor y las cuales dejamos abiertas.

Si observamos los valores de esta arquitectura y la anterior, vemos que la diferencia del tiempo de ejecución es mínima, haciendo el cambio poco notable, ya que la pool de gorutines debe ser menor o igual al número de núcleos de la máquina.
El problema con el QoS que teníamos en la anterior tampoco lo perdemos, pero esta vez en vez de tener en cuenta los núcleos, tenemos en cuenta la Pool de gorutines asignada.

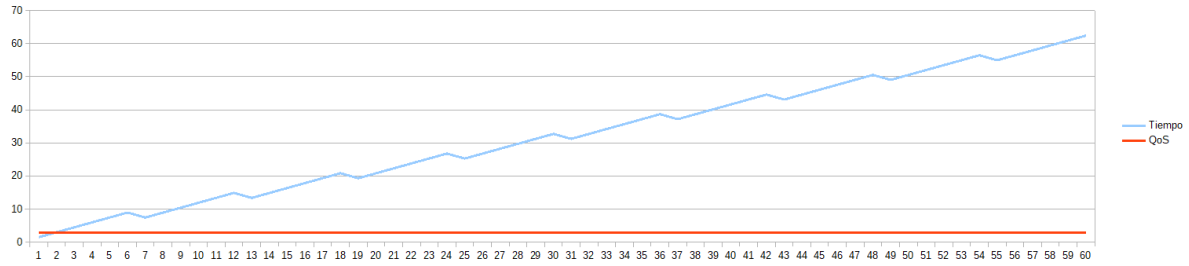
Arquitectura Master - Worker

Con esta arquitectura es un poco más difícil asumir un tiempo de transmisión ideal, ya que primero un cliente se comunica con el master, luego, éste con un worker el cual le devuelve la respuesta para finalmente devolvérsela al cliente.
Aún así, se siguen necesitando gorutinas en el master por lo tanto hay un máximo número de clientes a la espera de sus respuestas; la mayor diferencia con el anterior es que las tareas no se ejecutan en la máquina del master, sino en otras distintas.
El valor que se espera del tiempo de overhead es exactamente igual al de la pool de gorutinas.
La única diferencia con este sería el tiempo de comunicación, que va a ser mayor.

Validación Experimental

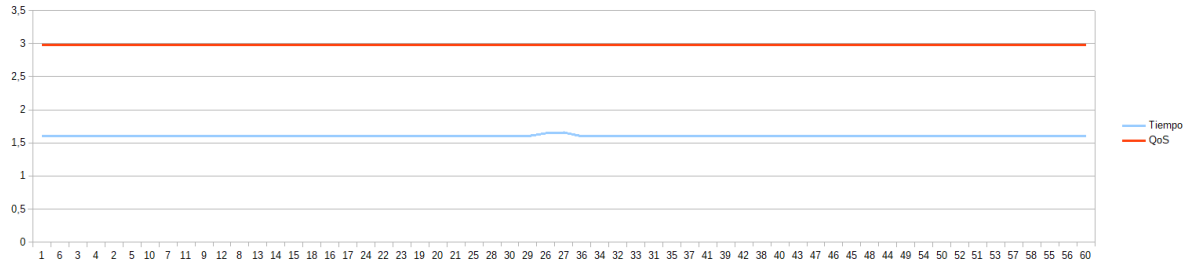
Tras la ejecución de cada arquitectura desde las máquinas del laboratorio, se han obtenido las siguientes gráficas:

Servidor Secuencial



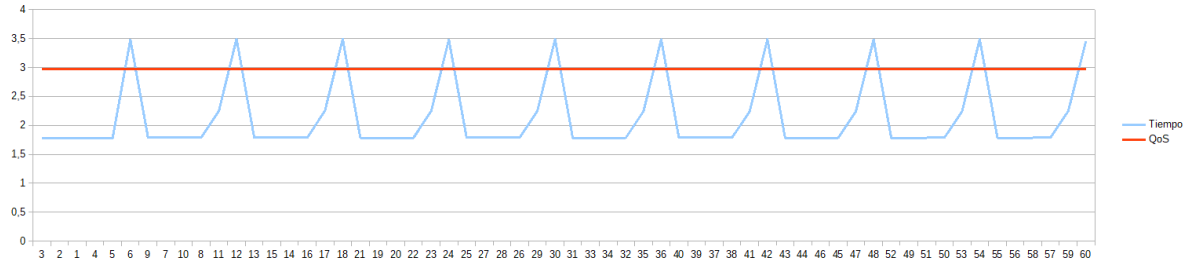
Se esperaba que el tiempo total de esta arquitectura sería sin duda el peor, ya que no hay concurrencia en ningún momento. La gráfica forma casi una función lineal, con pequeñas variaciones cada séptima petición. Esto se debe a que es justo cuando el cliente hace la espera en el bucle y da algo de tiempo a la petición 6 para avanzar en el cálculo antes de iniciar la séptima. Si no fuese por esto, sería exactamente igual a una recta.

Servidor Concurrente de Gorutines Dinámicas



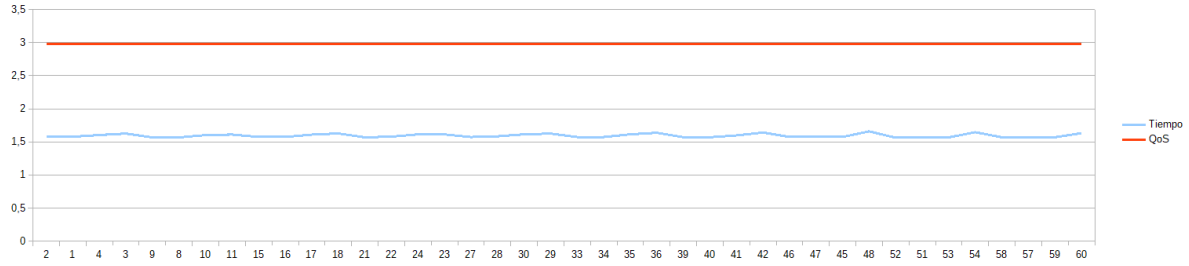
Con esta gráfica, observamos que es una arquitectura estable para nuestro problema en concreto, ya que contamos con 6 cores y el cliente solo ejecuta de forma simultánea 6 peticiones, ya que tras estas realiza una espera que permite a las rutinas terminar el trabajo antes de realizar más peticiones.

Servidor Concurrente con Pool de 5 Gorutines



Como el cliente ejecuta de forma simultánea 6 peticiones y este servidor cuenta con una pool de gorutines de 5, la sexta petición se debe mantener a la espera. Esto causa un tiempo de overhead mayor a 0 y como no nos encontramos en condiciones ideales, habiendo retardos en la comunicación y otros factores, se sobrepasa el límite establecido por el QoS.
Se ha elegido una pool menor al número de núcleos y mayor al de trabajos simultáneos pedidos para que se vea el problema con mayor claridad. Si se hubiese hecho una pool de 6, la gráfica sería igual a la anterior.

Master-Worker



Hay que recalcar que ha habido problemas con esta arquitectura, ya que otros alumnos no mataban los procesos servidores que se quedaban ejecutando y utilizaban gran parte del procesamiento, habiendo pérdida de paquetes y aumento en el tiempo de ejecución.

El master-worker, se ha desarrollado de una forma similar a la de pool de gorutines pero de forma distribuída y con una pool de 6 en vez de 5, acercándose más a la arquitectura de gorutinas dinámicas. Por esto, la gráfica obtenida es muy parecida a la anteriormente nombrada y se puede considerar apta para este problema en concreto.

Problemas Encontrados

Nuestros mayores problemas estaban relacionados con los otros alumnos, los cuales no mataban sus procesos y los dejaban en ejecución durante muchos minutos o incluso horas. Esto nos provocaba pérdida de paquetes en la arquitectura master-worker y aumento parcial del tiempo de conexión y ejecución. De estos 2 últimos factores no es algo de lo que nos podamos quejar, ya que nos lo vamos a seguir encontrando, además de que nosotros también molestamos a otros grupos, pero lo primero no es permisible y al tener que ejecutar varias instancias de los workers, sobrecargaba la conexión y hacía imposible la ejecución correcta de los mismos. Posteriormente se muestran capturas de la ejecución del comando “ps -lu aXXXXXX” de 2 alumnos distintos en una sóla máquina, en el mismo día y hora.

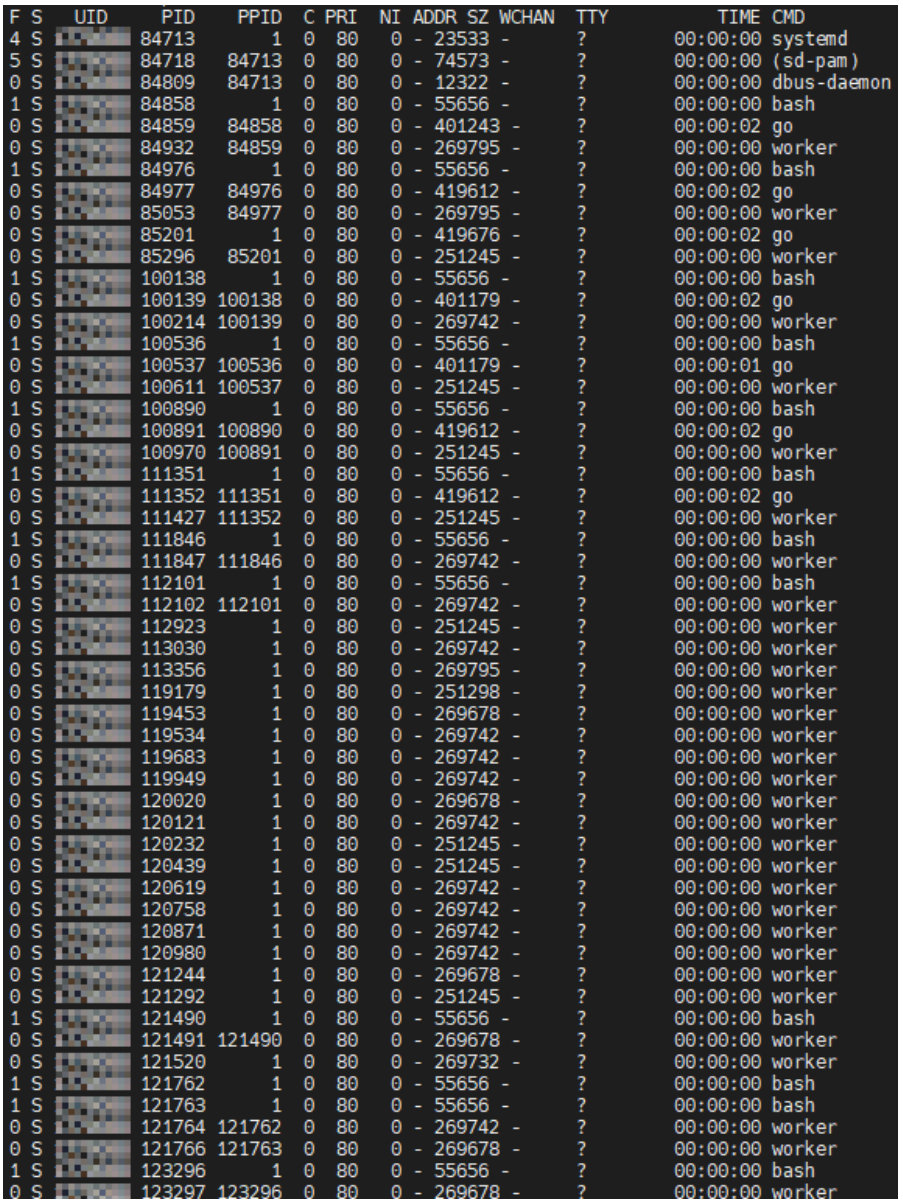
No hemos mostrado los tiempos de los paquetes perdidos, pero salían tiempos de cientos de miles de horas, aunque eran recibidos en escasas décimas de segundo.

También hemos tenido problemas con el ssh desde golang, el cual ha sido difícil saber porqué había veces que no se ejecutaban los workers, pero otras sí sin haber modificado el código entre pruebas.

Alumno número 1

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
4	S	[redacted]	264015	1	0	80	0	-	23559	-	?	00:00:00	systemd
5	S	[redacted]	264018	264015	0	80	0	-	74639	-	?	00:00:00	(sd-pam)
0	S	[redacted]	264034	1	0	80	0	-	419676	-	?	00:00:05	go
0	S	[redacted]	264074	264015	0	80	0	-	12322	-	?	00:00:00	dbus-daemon
0	S	[redacted]	264140	264034	0	80	0	-	251245	-	?	00:00:00	worker
0	S	[redacted]	264783	1	0	80	0	-	419612	-	?	00:00:07	go
0	S	[redacted]	264867	264783	0	80	0	-	251245	-	?	00:00:00	worker
0	S	[redacted]	266262	1	0	80	0	-	401243	-	?	00:00:04	go
0	S	[redacted]	266338	266262	0	80	0	-	269742	-	?	00:00:00	worker
0	S	[redacted]	266617	1	0	80	0	-	401179	-	?	00:00:05	go
0	S	[redacted]	266694	266617	50	80	0	-	362195	-	?	09:39:56	worker
0	S	[redacted]	269338	1	0	80	0	-	401243	-	?	00:00:08	go
0	S	[redacted]	269439	269338	49	80	0	-	362195	-	?	08:39:21	worker
0	S	[redacted]	269567	1	0	80	0	-	401243	-	?	00:00:14	go
0	S	[redacted]	269648	269567	0	80	0	-	269742	-	?	00:00:00	worker
0	S	[redacted]	270226	1	0	80	0	-	401179	-	?	00:00:05	go
0	R	[redacted]	270300	270226	49	80	0	-	362259	-	?	08:23:04	worker
0	S	[redacted]	270509	1	0	80	0	-	401179	-	?	00:00:03	go
0	R	[redacted]	270590	270509	49	80	0	-	343698	-	?	08:20:36	worker
0	S	[redacted]	270955	1	0	80	0	-	401115	-	?	00:00:02	go
0	S	[redacted]	271035	270955	48	80	0	-	362195	-	?	08:14:10	worker
0	S	[redacted]	271435	1	0	80	0	-	437981	-	?	00:00:08	go
0	R	[redacted]	271510	271435	48	80	0	-	362195	-	?	08:11:48	worker
0	S	[redacted]	271559	1	0	80	0	-	438109	-	?	00:00:04	go
0	S	[redacted]	271635	271559	48	80	0	-	362195	-	?	08:08:27	worker
0	S	[redacted]	272239	1	0	80	0	-	419676	-	?	00:00:05	go
0	S	[redacted]	272340	272239	48	80	0	-	362195	-	?	07:59:20	worker
0	S	[redacted]	272576	1	0	80	0	-	419612	-	?	00:00:04	go
0	S	[redacted]	272666	272576	48	80	0	-	362259	-	?	07:55:17	worker
0	S	[redacted]	277117	1	0	80	0	-	419676	-	?	00:00:07	go
0	R	[redacted]	277206	277117	46	80	0	-	362195	-	?	06:47:20	worker
0	S	[redacted]	290068	1	0	80	0	-	419612	-	?	00:00:04	go
0	S	[redacted]	290155	290068	41	80	0	-	362195	-	?	03:30:37	worker
0	S	[redacted]	290938	1	0	80	0	-	401179	-	?	00:00:06	go
0	S	[redacted]	291024	290938	41	80	0	-	362195	-	?	03:07:16	worker
0	S	[redacted]	291060	1	0	80	0	-	382746	-	?	00:00:05	go
0	S	[redacted]	291150	291060	40	80	0	-	343634	-	?	03:04:50	worker

Alumno número 2



```
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S [redacted] 84713       1  0  80   0 - 23533 -      ?        00:00:00 systemd
5 S [redacted] 84718   84713  0  80   0 - 74573 -      ?        00:00:00 (sd-pam)
0 S [redacted] 84809   84713  0  80   0 - 12322 -      ?        00:00:00 dbus-daemon
1 S [redacted] 84858       1  0  80   0 - 55656 -      ?        00:00:00 bash
0 S [redacted] 84859   84858  0  80   0 - 401243 -     ?        00:00:02 go
0 S [redacted] 84932   84859  0  80   0 - 269795 -     ?        00:00:00 worker
1 S [redacted] 84976       1  0  80   0 - 55656 -      ?        00:00:00 bash
0 S [redacted] 84977   84976  0  80   0 - 419612 -     ?        00:00:02 go
0 S [redacted] 85053   84977  0  80   0 - 269795 -     ?        00:00:00 worker
0 S [redacted] 85201       1  0  80   0 - 419676 -     ?        00:00:02 go
0 S [redacted] 85296   85201  0  80   0 - 251245 -     ?        00:00:00 worker
1 S [redacted] 100138      1  0  80   0 - 55656 -      ?        00:00:00 bash
0 S [redacted] 100139 100138  0  80   0 - 401179 -     ?        00:00:02 go
0 S [redacted] 100214 100139  0  80   0 - 269742 -     ?        00:00:00 worker
1 S [redacted] 100536      1  0  80   0 - 55656 -      ?        00:00:00 bash
0 S [redacted] 100537 100536  0  80   0 - 401179 -     ?        00:00:01 go
0 S [redacted] 100611 100537  0  80   0 - 251245 -     ?        00:00:00 worker
1 S [redacted] 100890      1  0  80   0 - 55656 -      ?        00:00:00 bash
0 S [redacted] 100891 100890  0  80   0 - 419612 -     ?        00:00:02 go
0 S [redacted] 100970 100891  0  80   0 - 251245 -     ?        00:00:00 worker
1 S [redacted] 111351      1  0  80   0 - 55656 -      ?        00:00:00 bash
0 S [redacted] 111352 111351  0  80   0 - 419612 -     ?        00:00:02 go
0 S [redacted] 111427 111352  0  80   0 - 251245 -     ?        00:00:00 worker
1 S [redacted] 111846      1  0  80   0 - 55656 -      ?        00:00:00 bash
0 S [redacted] 111847 111846  0  80   0 - 269742 -     ?        00:00:00 worker
1 S [redacted] 112101      1  0  80   0 - 55656 -      ?        00:00:00 bash
0 S [redacted] 112102 112101  0  80   0 - 269742 -     ?        00:00:00 worker
0 S [redacted] 112923      1  0  80   0 - 251245 -     ?        00:00:00 worker
0 S [redacted] 113030      1  0  80   0 - 269742 -     ?        00:00:00 worker
0 S [redacted] 113356      1  0  80   0 - 269795 -     ?        00:00:00 worker
0 S [redacted] 119179      1  0  80   0 - 251298 -     ?        00:00:00 worker
0 S [redacted] 119453      1  0  80   0 - 269678 -     ?        00:00:00 worker
0 S [redacted] 119534      1  0  80   0 - 269742 -     ?        00:00:00 worker
0 S [redacted] 119683      1  0  80   0 - 269742 -     ?        00:00:00 worker
0 S [redacted] 119949      1  0  80   0 - 269742 -     ?        00:00:00 worker
0 S [redacted] 120020      1  0  80   0 - 269678 -     ?        00:00:00 worker
0 S [redacted] 120121      1  0  80   0 - 269742 -     ?        00:00:00 worker
0 S [redacted] 120232      1  0  80   0 - 251245 -     ?        00:00:00 worker
0 S [redacted] 120439      1  0  80   0 - 251245 -     ?        00:00:00 worker
0 S [redacted] 120619      1  0  80   0 - 269742 -     ?        00:00:00 worker
0 S [redacted] 120758      1  0  80   0 - 269742 -     ?        00:00:00 worker
0 S [redacted] 120871      1  0  80   0 - 269742 -     ?        00:00:00 worker
0 S [redacted] 120980      1  0  80   0 - 269742 -     ?        00:00:00 worker
0 S [redacted] 121244      1  0  80   0 - 269678 -     ?        00:00:00 worker
0 S [redacted] 121292      1  0  80   0 - 251245 -     ?        00:00:00 worker
1 S [redacted] 121490      1  0  80   0 - 55656 -      ?        00:00:00 bash
0 S [redacted] 121491 121490  0  80   0 - 269678 -     ?        00:00:00 worker
0 S [redacted] 121520      1  0  80   0 - 269732 -     ?        00:00:00 worker
1 S [redacted] 121762      1  0  80   0 - 55656 -      ?        00:00:00 bash
1 S [redacted] 121763      1  0  80   0 - 55656 -      ?        00:00:00 bash
0 S [redacted] 121764 121762  0  80   0 - 269742 -     ?        00:00:00 worker
0 S [redacted] 121766 121763  0  80   0 - 269678 -     ?        00:00:00 worker
1 S [redacted] 123296      1  0  80   0 - 55656 -      ?        00:00:00 bash
0 S [redacted] 123297 123296  0  80   0 - 269678 -     ?        00:00:00 worker
```