SSDD

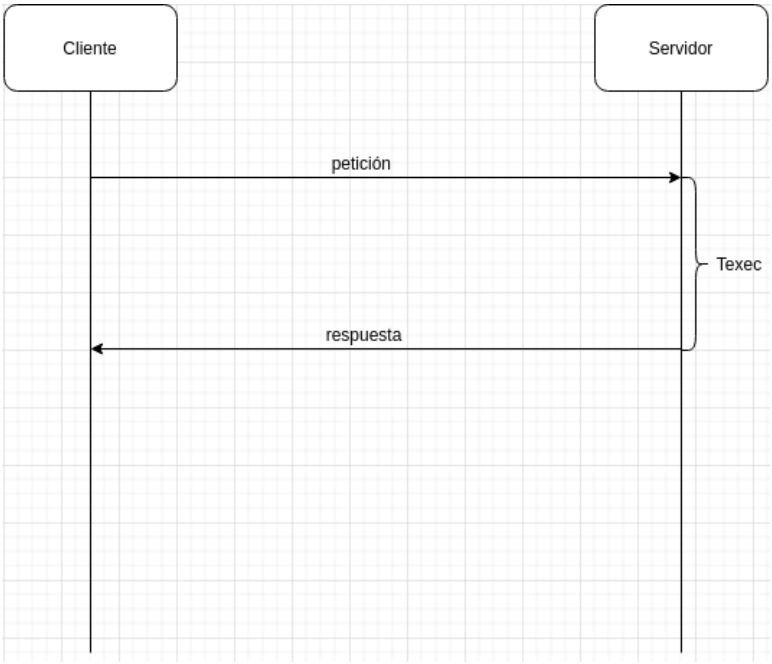
- práctica 1 -

Por:

Diego Caballé Casanova (738712)

Análisis teórico de las arquitecturas

Cliente-Servidor secuencial



Como nos dice el apartado 2.3.2 consiste en un servidor que atiende peticiones de forma secuencial, es decir, que tenemos **un proceso** que está trabajando en responder a las peticiones de los clientes.

Esta aproximación es, sin duda, la peor de todas, el hecho de que un sólo proceso tenga que atender a varios clientes producirá que los clientes tengan que esperar (de forma virtual) en una cola a ser atendidos. Es decir, el primer cliente en enviar la petición sólo tendrá que esperar a que se responda su petición, pero el segundo tendrá que esperar a la finalización del primer cliente y después al procesado de su petición. Así consecutivamente contra más clientes añadamos.

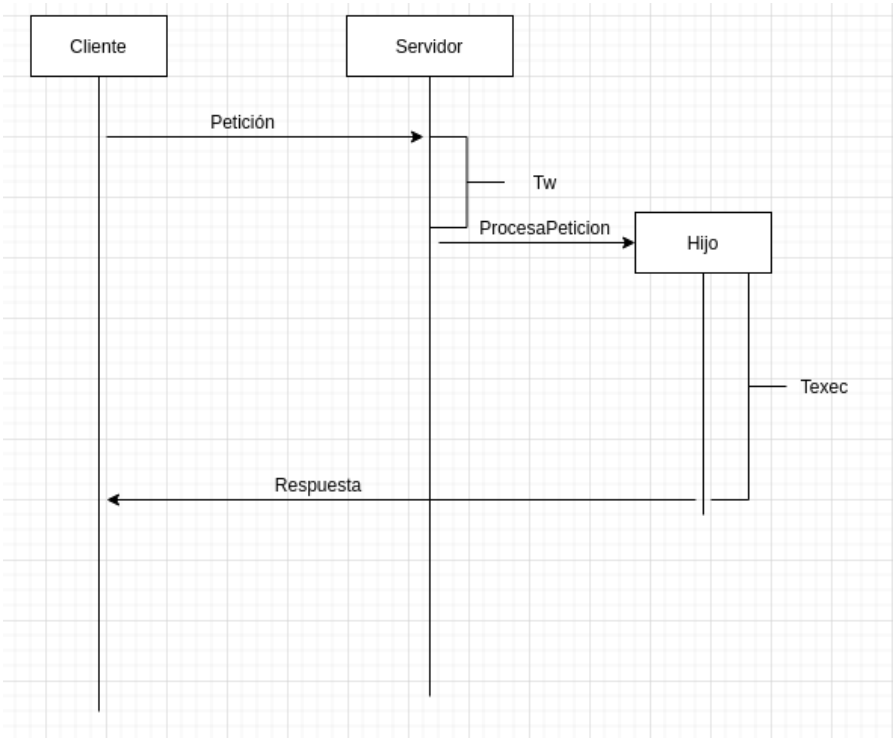
Para no violar el QoS tenemos que cumplir que el tiempo del enviado de la petición, procesado (con su overhead) y respuesta sea menor a dos veces el tiempo del procesado efectivo de la tarea de forma aislada.

En concreto en esta arquitectura, tenemos el problema de que, desde el punto de vista del cliente, cuando enviamos la petición comienza el tiempo de espera, ya que no sabemos si estamos siendo atendidos o no. Así que si somos el segundo cliente en enviar la petición, a nuestro tiempo de ejecución total se le añade el tiempo de ejecución total del primer cliente.

Asumiendo que Texec-cliente > Texec, 2*Texec-cliente > 2*Texec. Luego la carga de trabajo que puede operar esta arquitectura será de **1 cliente**. Es decir, podrá tratar a un cliente, y mientras está tratando a ese cliente, los demás clientes no podrán ser atendidos, para no incumplir el QoS.

Cliente servidor concurrente creando una Goroutine por petición

Esta solución es bastante mejor que la anterior, aquí nos aprovechamos de las ideas de la concurrencia para poder tratar a varios clientes a la vez. Ante una petición de cliente, tenemos un proceso que va creando Goroutines que realizarán el procesamiento a la vez de varias peticiones, pudiendo atender a varios clientes simultaneamente, olvidandonos de la cola virtual de la solución anterior.



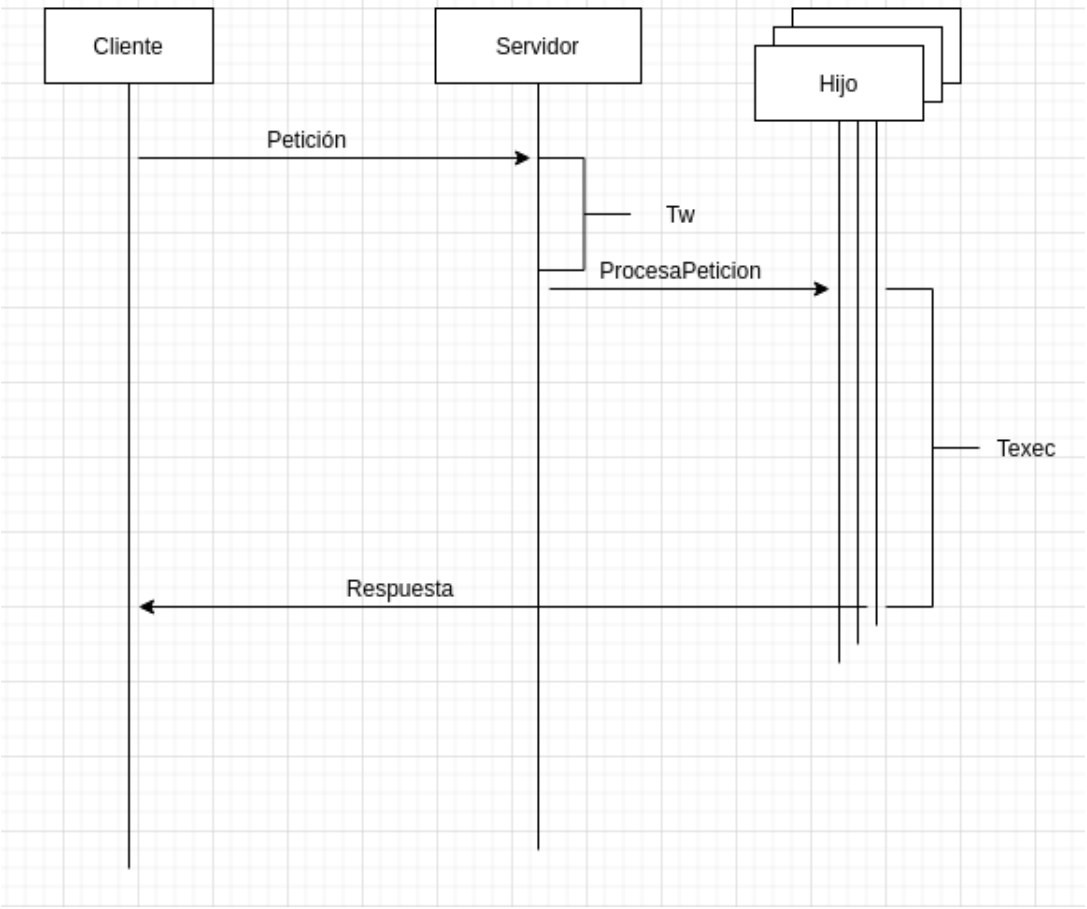
*Nota: a más clientes, más hijos

A pesar de todo, esta solución también tiene sus problemas, volviendo a la ecuación del Texec-cliente tenemos los valores del Txon, Texec y Tw. Txon es constante en el protocolo de comunicación que utilizamos y Texec en el algoritmo que procesa la petición, en este caso, como nos han dado el código de primes asumiremos, también, que es un valor constante. Pero la variable Tw es variable en la arquitectura elegida. En este caso particular, Tw será el coste de crear una Goroutine bajo petición + el tiempo de procesado del código del servidor, pero además, la existencia de varios procesos concurrentes hará que lentamente, el Tw aumente si el número de clientes es muy alto. Aunque Go puede aprovechar muy eficientemente los núcleos de nuestro ordenador, es inevitable que ante más procesos concurrentes menos tiempo tendrá el proceso en el núcleo y tendrá un overhead de cambio de estado y espera a otros procesos que hará aumentar el tiempo total.

La carga de trabajo que soporta es complicado de deducir teorícamente sin buscar datos de eficiencia, pero se entiende que Tw irá aumentando contra más procesos creemos, y el límite será cuando Tw + Txon sea igual a Texec.

Cliente servidor concurrente con un pool fijo de Goroutines

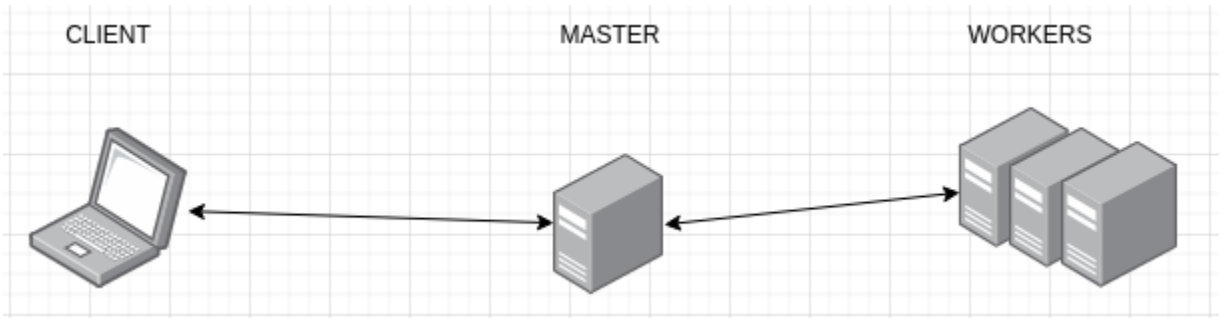
En esta nueva aproximación, intentamos minimizar el tiempo de generar un nuevo Goroutine, teniendo ya creada una pool de Goroutine, estos procesos se comunicaran con el servidor vía canales también previamente creados que las Goroutines estarán escuchando (trabajará la goroutine más rápida en leer el canal)



*La respuesta puede ser al servidor y el servidor responder al cliente.

En este caso, de Tw eliminamos la constante de crear las Goroutines (para una carga de trabajo = al número de procesos), pero, en este caso la solución es menos escalable (salvo que permitamos la creación de nuevas Goroutines si nuestro pool está todo trabajando). Entonces, igual que antes, tendremos que tener cuidado con Tw, pues los procesos sobre un mismo ordenador hará que haya sobrecarga y se esperen entre ellos. Pero eso es mejor que las peticiones se tengan que esperar entre sí.

Arquitectura Máster – Worker



Aquí mejoramos la versión anterior con las distribución de los procesos, en una máquina tendremos el servidor y en otras distintas tendremos los procesos workers, de esta manera el solapamiento de los procesos se reduce y sólo se verá limitado en el número de máquinas que tendremos, que nos dirá cuantos núcleos podremos ocupar sin solapamiento.

Además, ante la escasez de workers, esta solución es **escalable**, es decir, que sin necesidad de cerrar el servidor podemos aumentar el número de workers para responder a la demanda

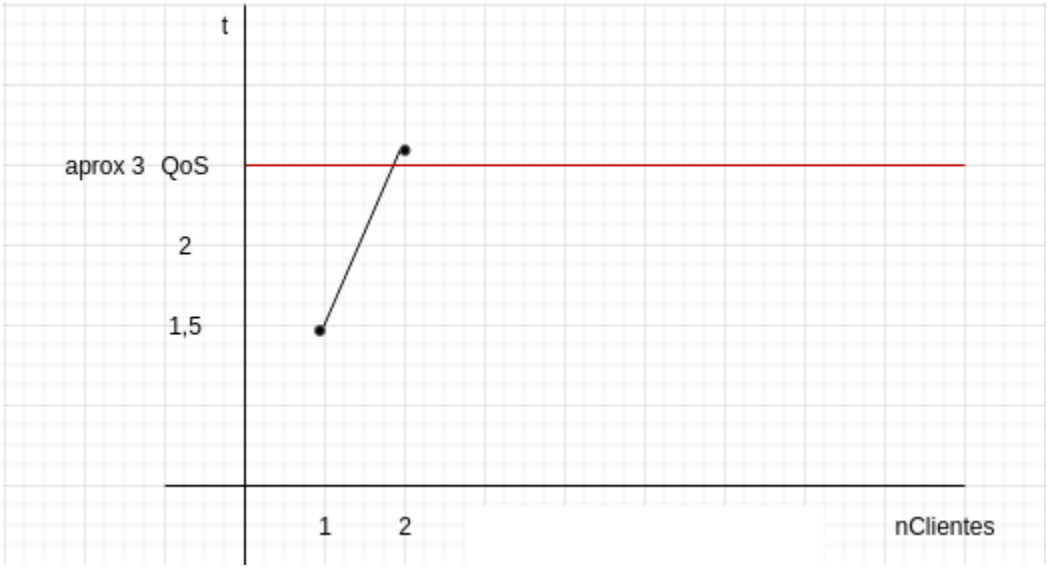
Esta solución será la más rápida, podrá soportar gran cantidad de peticiones sin penalización, teniendo el Texec-cliente mínimo (es decir, Tw siendo la sobrecarga del código simplemente). Una vez todas las máquinas estén ocupadas, podremos añadir una máquina y añadir los nuevos procesos. Todo esto en función de la carga de trabajo y nuestra capacidad de ir encendiendo nuevas máquinas.

Análisis empírico de las arquitecturas

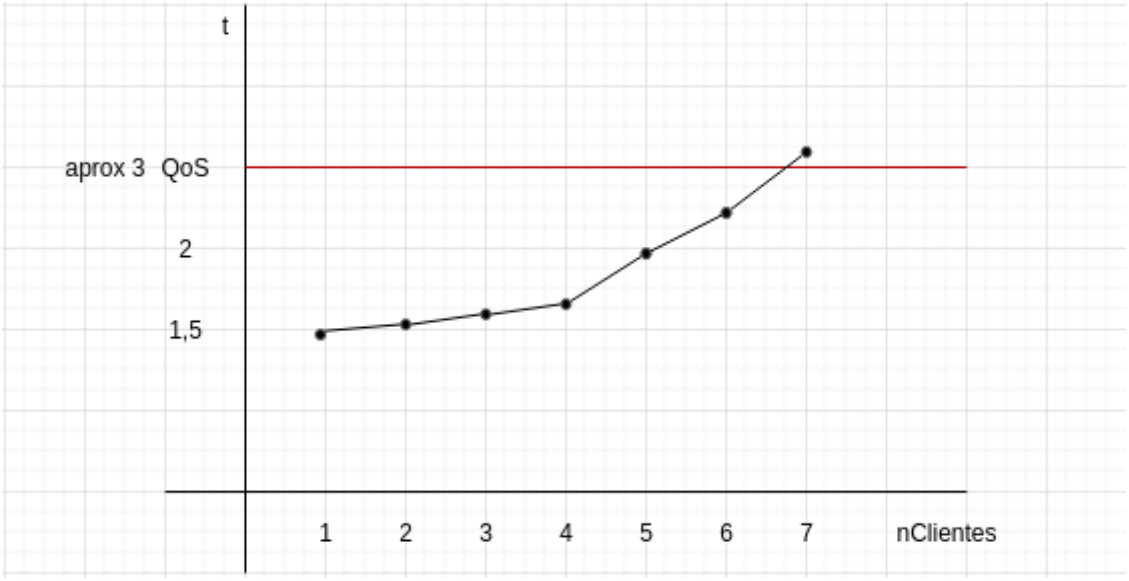
Las gráficas representan el peor tiempo registrado para un número de clientes.

Cliente-Servidor secuencial

Comenzando por la primera arquitectura, se cumple lo que se había analizado en el análisis teórico

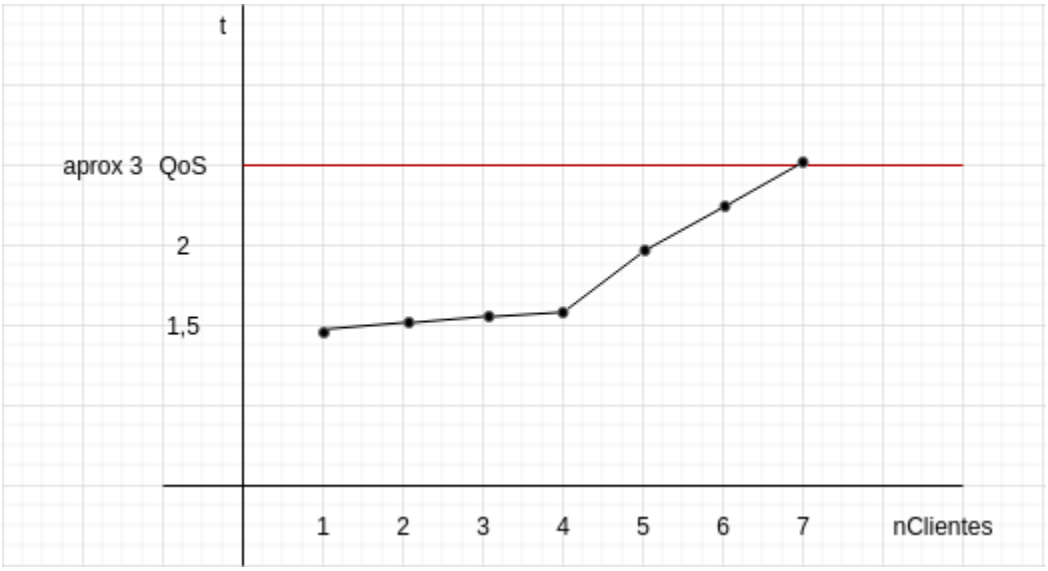


Cliente servidor concurrente creando una Goroutine por petición



Cabe apuntar que con 7 clientes, la respuesta que recibe el tercero de los clientes ya supera el QoS.

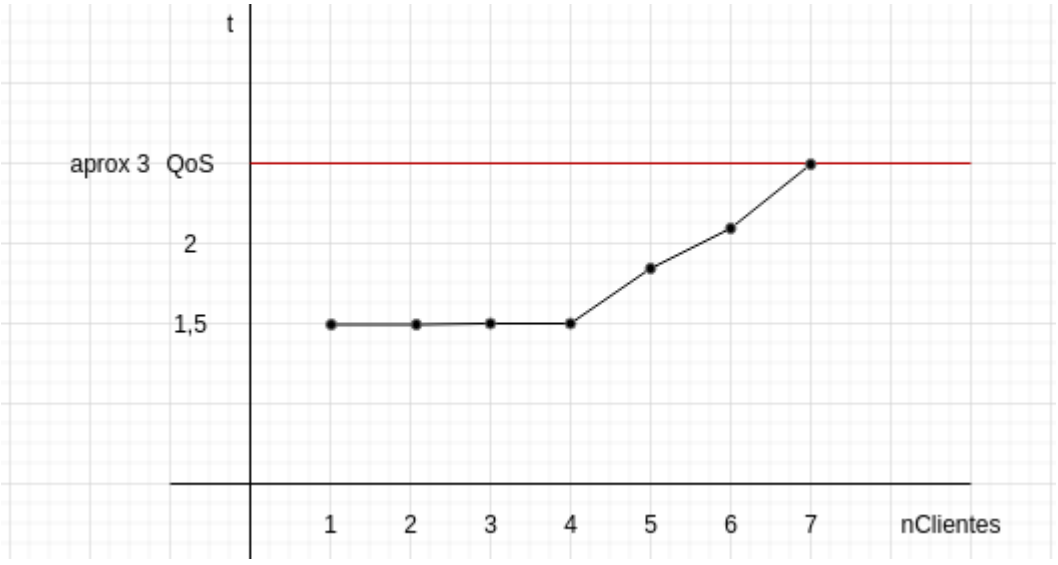
Cliente servidor concurrente con un pool fijo de Goroutines



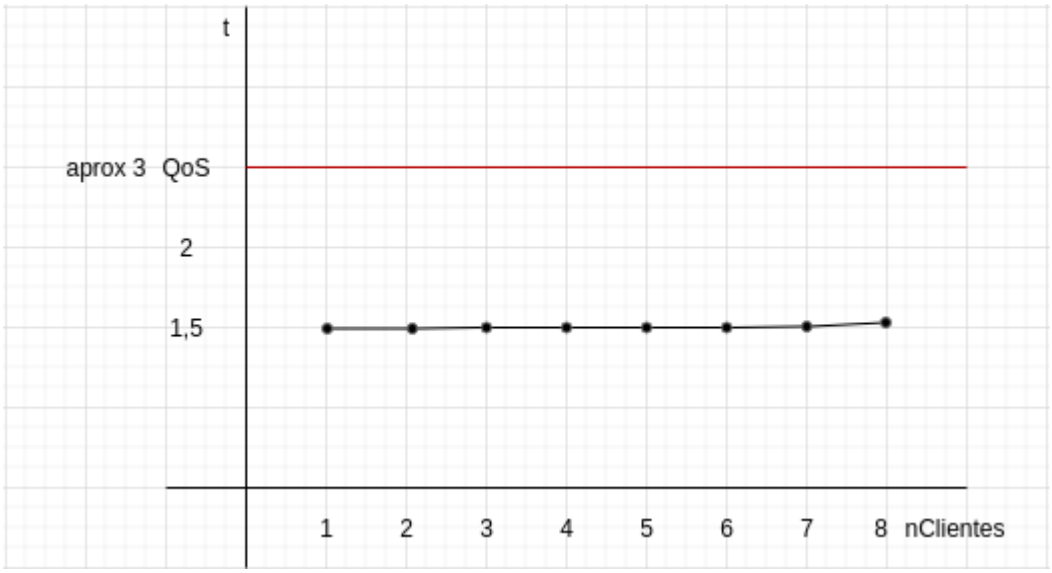
Como habíamos anticipado en el análisis, y como anotación que no puede representar esta gráfica, si la carga es mayor al número de workers, los resultados son los de la primera arquitectura, donde un cliente espera a la resolución del anterior.

La mejora frente a la anterior arquitectura es ligera en los números pequeños de clientes, y se va notando más cuando el número de clientes es alto, en el último caso, 7 clientes y 7 workers no consiguen respetar el QoS por centesimas de segundo, obteniendo 2,77 en la sexta respuesta, pero la septima respuesta obtiene 2,9 segundos así que lo infringe en el último de los clientes.

Arquitectura Máster – Worker



En este caso tenemos 1 máquina worker, para responder al nClientes > 4 hemos puesto en la misma máquina varias gorutinas.



2 máquinas con 4 workers cada una.