Memoria práctica 1

Sistemas distribuidos

Curso 2019-2020

Juan José Tambo - 755742

Pedro Tamargo - 758267

1. Introducción

La práctica consiste en una comunicación cliente-servidor, donde se le envía al servidor una petición para que aplique la función fibonacci a una lista que se le envía al mismo tiempo.
Existen tres escenarios de ejecución posibles:

- UNO: Se envían dos peticiones cada 2 segundos.
- DOS: Se envían cuatro peticiones cada 2 segundos.
- TRES: Se envían 8 peticiones cada 1,17 segundos (en media), ya que es variable en función del tiempo.

Para garantizar el Qos, es necesario que el tiempo total de respuesta sea menor que 1,5 veces el tiempo de ejecución de forma aislada (1,44 segundos).
En la memoria, describiremos los mecanismos con los que hemos solucionado los distintos problemas.

2. Modelado de cada escenario

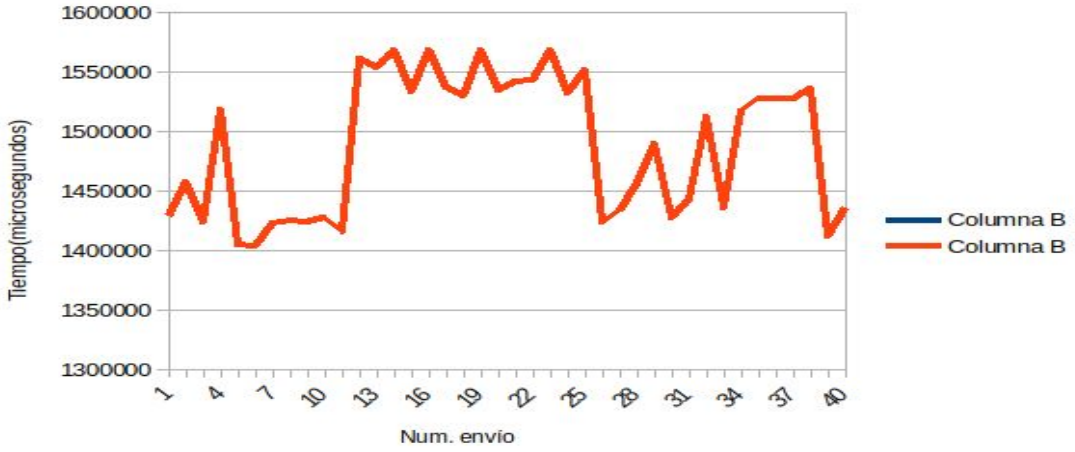
Teniendo en cuenta lo mencionado anteriormente, hemos decidido las siguientes configuraciones para cada escenario:

- **Escenario 1:** Lo hemos modelado a modo cliente-servidor asíncrono. Inicialmente, la estructura era de master-worker, pero consideramos que no era necesario ya que la carga de trabajo no es excesiva.

  ¿Cómo funciona?
  El cliente únicamente se encarga de enviar dos peticiones cada dos segundos, mientras lanza un thread para escuchar la respuesta del servidor, ya que sino se bloquearía esperando la misma.
  El servidor, por otro lado, tiene inicialmente una lista de disponibles, con 4 pid's que representan los 4 núcleos que tienen los ordenadores del laboratorio(ese pid es el de la máquina servidor), una lista de ocupados y otra de pendientes, donde almacenará los pid's de los posibles clientes que no se han podido atender si no hay ningún núcleo libre.
  Comienza realizando una escucha bloqueante, donde puede recibir una petición del cliente o bien la indicación de que uno de los núcleos ha acabado.
  Si se trata de una petición, comprueba si hay algún núcleo libre en la lista de disponibles. Si lo hay, modifica la lista eliminando un elemento de la misma y añadiendolo a la de ocupados. Después, invoca la

función “comunicar” mediante un thread, la cual se encarga de procesar la petición del cliente y enviar el resultado al mismo, además de indicar su finalización al servidor.
Si no hay disponibles, añade el pid del cliente que ha enviado la solicitud a la lista de pendientes, para atenderle posteriormente y vuelve a escuchar.
Por otro lado, si ha recibido la finalización de un núcleo, comprueba si hay algún pendiente para poder solicitar su petición. Si no lo hay, añade el núcleo a la lista de disponibles.

Resultados:



Para que se cumpla el Qos, el tiempo total debe ser menor que 2,16 segundos.
Como se puede observar, en todo momento se mantiene el Qos, aunque aparezcan algunos puntos donde la carga es mayor.

- **Escenarios 2 y 3:** En estos casos, hemos visto conveniente aplicarles la estructura master-worker, con pool de workers. Con esto garantizamos que el Qos se cumpla en ambos escenarios.
  El escenario 2 posee un worker, mientras que el escenario 3 necesita 12 workers, para garantizar el correcto funcionamiento del sistema, evitando en todo momento almacenar peticiones pendientes.

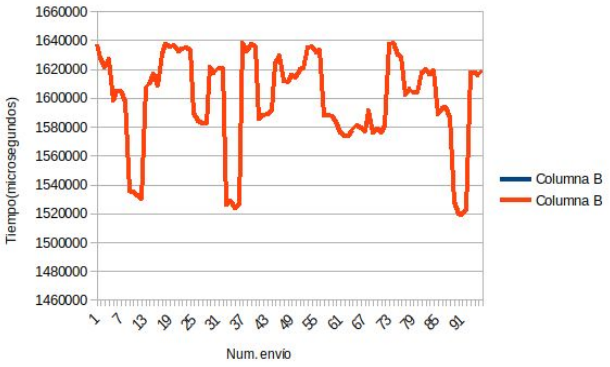
  ¿Cómo funciona?
  El cliente funciona de la misma manera que en la arquitectura descrita anteriormente, mientras que la parte del servidor consiste en el propio servidor(master), un pool de workers y los propios workers, cada uno ejecutado en una máquina distinta.

El máster, se encarga de recibir las peticiones del cliente y comunicarse con el pool y los workers. Para evitar bloqueos, lanza un thread con la función comunicar, la cual se comunica con el pool para pedirle un worker(una vez ha recibido la petición del cliente) y se bloquea a la espera de que le envíe el pid de uno que esté libre, mientras que el hilo principal se encarga de escuchar las peticiones del cliente.
Una vez recibe el pid del worker, realiza un Node.spawn(para que se ejecute en la máquina indicada), con la función “worker”, pasando como argumentos el pid del thread que se encarga de la operación comunicar, el pid del máster, el pid del pool, la operación y la lista correspondiente.
Posteriormente, se bloquea a la espera de que el worker realice la operación y devuelva el resultado, para seguidamente enviarselo al cliente.

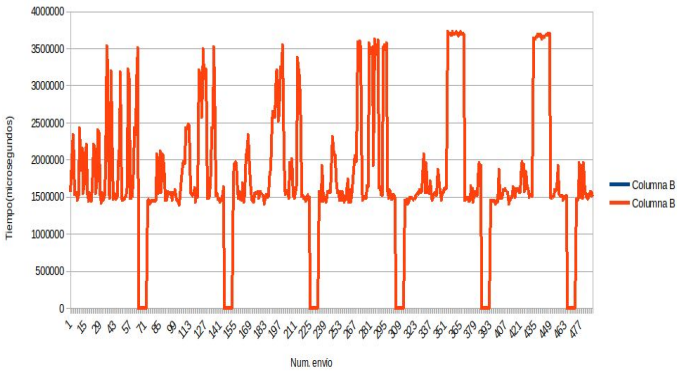
El worker, únicamente se encarga de ejecutar la operación y, una vez acabado enviar el resultado a máster e indicar a pool que ha acabado.

El pool, se encarga de controlar los workers. Para ello posee una lista de workers disponibles, ocupados y de peticiones de pendientes, igual que en la estructura cliente-servidor descrita anteriormente.
Para cada worker, se almacena 4 veces su pid, para representar los 4 núcleos que tiene cada equipo. Además, se intercalan los pids de las diferentes máquinas para que el sistema sea más equitativo.
El pool puede recibir dos tipos de mensaje: la petición de un worker por parte del máster o la indicación de que ha finalizado la ejecución de un worker. Si es una petición del máster, comprueba si hay algún worker disponible y le envía su pid. En caso contrario añade a la lista de pendientes el pid del thread que se ha comunicado, para atenderle posteriormente.
Si lo recibido es una finalización de un worker, comprueba la lista de pendientes. Si hay algún pendiente, le envía el pid del worker que acaba de finalizar. En caso contrario, añade ese pid a disponibles.

Resultados:



esc2



esc 3

En el escenario 2, se ejecuta siempre la operación fibonacci. Por eso, para que se cumpla el Qos, el tiempo de todos envíos debe ser menor que 1,5 veces el tiempo de ejecución aislada(que hemos considerado de 1,44 segundos), por lo tanto, debe ser menos que 2,16, lo que se puede observar que se cumple.
En el escenario 3, se utiliza la operación fibonacci y la fibonacci_tr, cuyo tiempo de ejecución en media es de 0,06 segundos.
En la gráfica se observa que, de media, se garantiza el Qos, aunque se generen ciertos picos en los cuales el tiempo de resolución aumenta. Esto es debido a la saturación de la red, ya que a realizar las pruebas, con 12 workers, observamos que no había ningún cliente en la cola de espera, pero debido a la cantidad de mensajes enviados, la red tardaba en responder. Se pueden ver ciertos mínimos, los que representan la ejecución de la operación fib_tr.

3. Conclusiones

En esta práctica hemos aprendido a adaptar diversos sistemas en base a ciertos requisitos, como lo son las limitaciones técnicas de los dispositivos o la carga de trabajo generada.
Además, hemos conseguido desarrollar un sistema escalable, que funciona para diferentes escenarios.

Por último, hemos fijado ciertos conocimientos acerca del manejo de elixir.

4. Bibliografía

- https://elixirschool.com/es/lessons/basics/collections/
- https://elixir-lang.org/getting-started/processes.html/