Articulo científico

Arquitectura de un servidor HTTP que maneja peticiones concurrentes

Sergio Andrés Rodríguez Torres, Escuela colombiana de ingeniería Julio Garavito

Ingeniería de sistemas, Arquitecturas Empresariales

21 de septiembre2019

Introducción:

Este articulo busca describir y explicar una propuesta de arquitectura de un servidor simple capaz de manejar peticiones de forma concurrente y analizar el impacto que tiene la concurrencia en la capacidad del servidor para manejar grandes volúmenes de peticiones, esto incluye el tiempo de respuesta de las peticiones individuales y la carga que es capaz de soportar el servidor, así como el número total de peticiones manejadas por unidad de tiempo, para realizar dichas pruebas vamos a crear un cliente capaz de realizar peticiones de forma concurrente, dicho cliente va a estar en una instancia de Amazon EC2.

Arquitectura del servidor HTTP:

Partiendo de la arquitectura de un servidor capaz de manejar peticiones HTTP estableciendo conexiones a nivel del protocolo TCP (En el artículo anterior se describe dicha arquitectura).

Vamos a ver cómo hacer que nuestro servidor sea capaz de manejar varias peticiones de forma concurrente.

Antes cabe aclarar que esta implementación está destinada a fines académicos ya que en un ámbito profesional no es la más eficiente y carece de robustez en términos de seguridad y escalabilidad, para eso se recomienda el uso de alguna implementación de un Java Servlet, ya aclarado esto vamos a ver cómo gestionar varias peticiones a la vez, aremos uso de concurrecia a alto nivel implementando un thread pool que se encargue de gestionar las peticiones que lleguen al servidor

Que es un thread pool:

Es un patrón de diseño de software para lograr la ejecución concurrente de un programa, con el fin de evitar la creación y destrucción constante de hilos asignados a tareas de corta duración, gestiona las tareas que se van a realizar concurrentemente, definiendo dos entidades principales:

- Cola de tareas: en dónde se van a encolar las tareas entrantes y van a quedar en espera hasta que alguno de los hilos en el pool de hilos esté libre
- Pool de hilos: estructura que va a mantener múltiples hilos en espera y cuando llegue una tarea a ser ejecutada la asigna un hilo que se encuentre en espera y una vez esté finalizada la tarea va a delegarla a las tareas completadas y volverá a dejar el hilo en espera para recibir una nueva tarea.

Implementando un thread pool:

Implementaremos un thread pool ya definido en el API de Java Executors a partir del método factory newFixedThreadPool, que crea una instancia de Executor que representa thread pool con un número

fijo de threads, no es el más eficiente o flexible, pero es perfecto para realizar pruebas del impacto de la concurrencia en el servidor.

```java
public ServerConnection(int port, boolean isAlive, int nThreads) {
    this.port = port;
    this.isAlive = isAlive;
    this.threadPool = Executors.newFixedThreadPool(nThreads);
}
```

```java
public static void main(String[] args) {
    int port = getPort();
    ServerService serverService = new ServerService(port,  nThreads: 100);
    serverService.initialize();
    serverService.listen();
}
```

Está variable nThreads la vamos a pasar desde ServerController al ServerService hasta la instancia ServerHTTP que pasa esta variable al constructor de la super clase ServerConnection, luego jugaremos con esta variable para modificar el tamaño de nuestro thread pool.

Usando el thread pool:

Ahora vamos a ver cómo usar el thread pool, al iniciar el servidor se crea una instancia de ServerSocket y este al recibe peticiones a través del método accept, este retorna un Socket para comunicarse con el cliente, el socket y estas peticiones las manejaremos en diferentes hilos, el hilo principal lo ocupará el ServerSocket y recibirá nuevas peticiones.

Para la implementación nos vamos a apoyar en el API CompletableFuture de Java 8 que provee métodos para gestionar peticiones asíncronas, el método *runAsync* recibe una instancia de la interfaz funcional Runnable y un Executor, usaremos funciones anónimas lambda de Java 8 para crear una crear una instancia de Runnable, esto es posible ya que al ser Runnable una interfaz funcional (que solo tiene un método abstracto), el compilador de Java creará una instancia de Runnable implementando el método abstracto con la función suministrada a partir de la función lambda, ya que ambas funciones tienen la misma firma, en el segundo parametro pasaremos nuestro ya definido thread pool.

```java
public void serverUp() {
    try {
        serverSocket = getServerSocket();
        do {
            Socket clientSocket = getClientSocket();
            CompletableFuture.runAsync(() -> processInput(clientSocket), threadPool);
        } while (isAlive);
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        closeConnection();
    }
}
```

Dentro del método *processInput* vamos a encargarnos con el Socket del cliente de manejar la petición.

Cliente concurrente en AWS:

Acá vamos a implementar también un cliente concurrente que sea capaz de realizar múltiples peticiones a nuestro servidor HTTP de forma concurrente, también va a implementar un thread pool de tamaño fijo

para tener control del número de hilos que se usan para hacer peticiones al servidor, esto con el objetivo de asegurarnos en que punto tenemos el nivel adecuado de paralelismo en el que no sea nuestro cliente web el cuello de botella y nos permita medir al servidor.

Va a ser una aplicación Java sencilla de consola que se va a ejecutar en una instancia EC2 de AWS, para esto solo pasaremos el archivo compilado *.class* y lo ejecutaremos en la maquina a través de una conexión SSH

Conectándose a la maquina en AWS, después de crea una instancia de EC2 en AWS para poder conectarnos a la maquina a través de una conexión por el protocolo SSH debemos generar un juego de llaves, tenerla llave privada en el mismo directorio en la que se solicita la conexión, para solictar la conexión se usa el siguiente comando.

ssh -i "private-key.pem" ec2-user@ec2-instance-code.compute-1.amazonaws.com

```
sergio@sergio-ThinkPad-E480:~/Documents/escuelaing/arep/aws$ ssh -i "arep-aws.pem"
ec2-user@ec2-3-90-62-27.compute-1.amazonaws.com
The authenticity of host 'ec2-3-90-62-27.compute-1.amazonaws.com (3.90.62.27)' can'
t be established.
ECDSA key fingerprint is SHA256:HUJoZRQH1OxB+QNz9cD7iYuVaBuO4Bdp/jyGzfTELtI.
Are you sure you want to continue connecting (yes/no)? y
Please type 'yes' or 'no': yes
Warning: Permanently added 'ec2-3-90-62-27.compute-1.amazonaws.com,3.90.62.27' (ECD
SA) to the list of known hosts.
Last login: Fri Sep 20 21:52:17 2019 from 45.239.88.78

       __|  __|_  )
       _|  (     /   Amazon Linux AMI
      ___|\___|___|

https://aws.amazon.com/amazon-linux-ami/2018.03-release-notes/
3 package(s) needed for security, out of 7 available
Run "sudo yum update" to apply all updates.
[ec2-user@ip-172-31-28-254 ~]$
```

Una vez establecida la conexión vamos a asegurarnos que la maquina tenga java 8 instalado, por lo general viene con java 7, por lo que tendremos que instalar OpenJDK 8 a través de “yum”

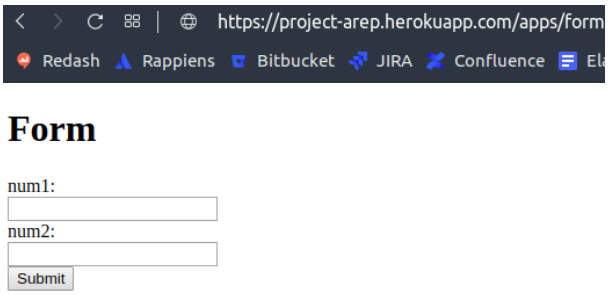
```
[ec2-user@ip-172-31-28-254 ~]$ java -version
openjdk version "1.8.0_222"
OpenJDK Runtime Environment (build 1.8.0_222-b10)
OpenJDK 64-Bit Server VM (build 25.222-b10, mixed mode)
```

Moveremos el ejecutable a la máquina virtual EC2, lo aremos a través de SFTP

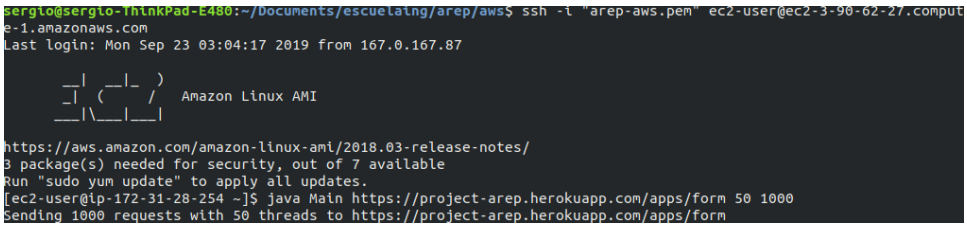
```
sergio@sergio-ThinkPad-E480:~/Documents/escuelaing/arep/aws$ sftp -i "arep-aws.pem"
 ec2-user@ec2-3-90-62-27.compute-1.amazonaws.com
Connected to ec2-3-90-62-27.compute-1.amazonaws.com.
sftp> lcd ..
sftp> lcd ClientAWS/
```

```
sftp> put Main.class
Uploading Main.class to /home/ec2-user/Main.class
Main.class                                    100% 2895    11.4KB/s   00:00
```

La página que vamos a usar para las pruebas es la siguiente:



Y realizaremos la prueba de la siguiente forma



Resultados de las pruebas

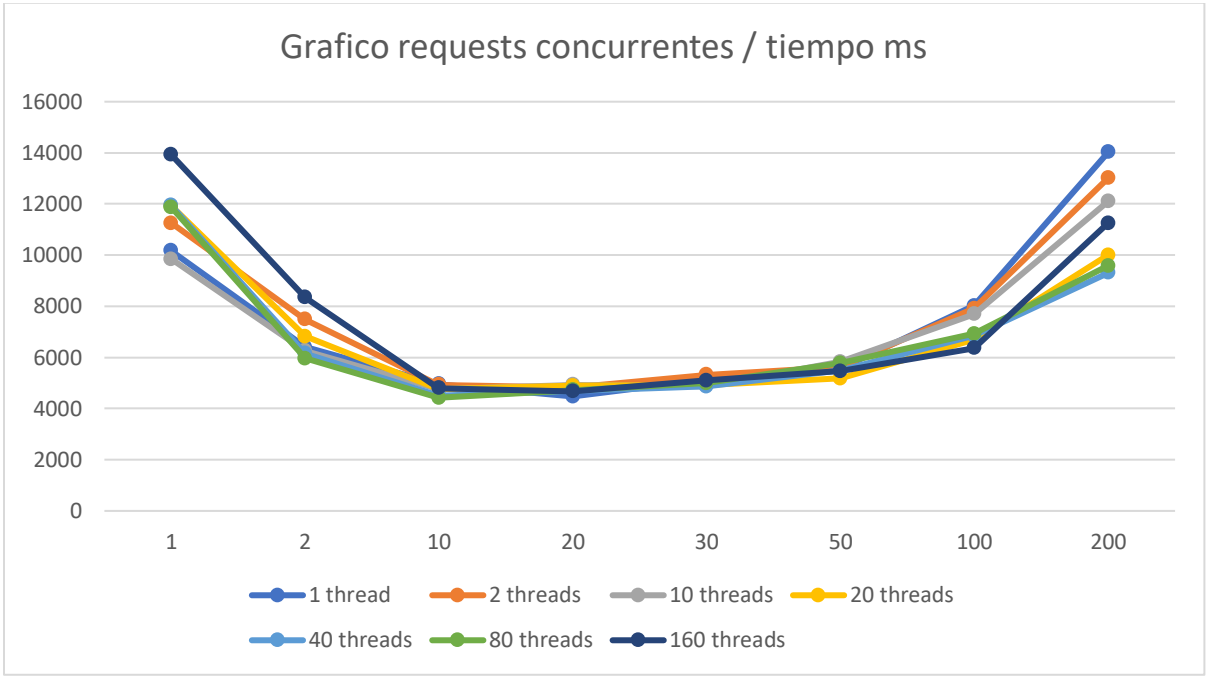
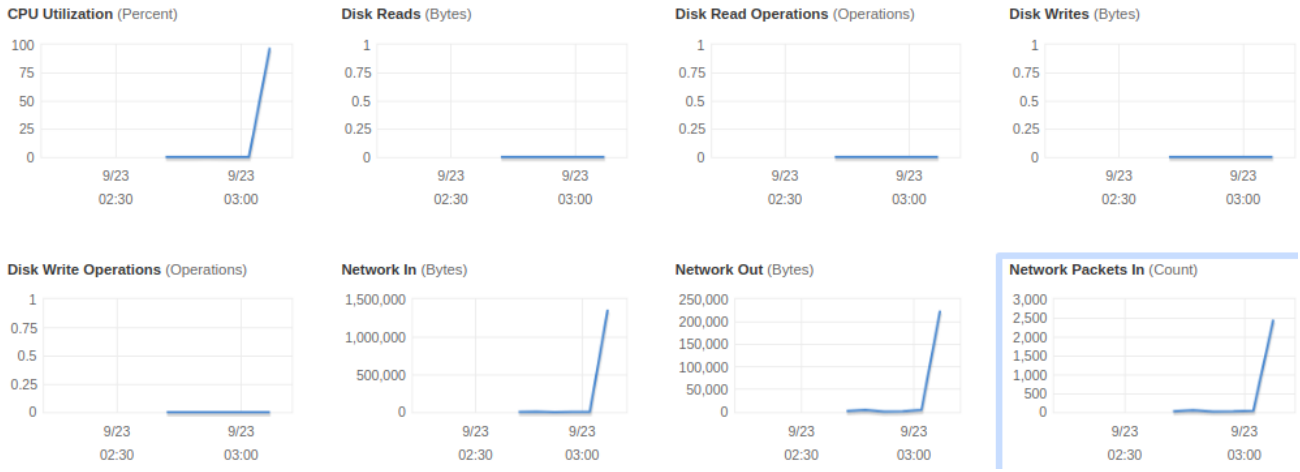


Tabla requests concurrentes / tiempo ms

	1 thread	2 threads	10 threads	20 threads	40 threads	80 threads	160 threads
1	10167	11245	9848	11929	11941	11874	13922
2	6402	7482	6259	6816	6151	5959	8342
10	4949	4931	4756	4763	4506	4426	4793
20	4477	4827	4935	4884	4696	4722	4671
30	5131	5325	4882	4915	4870	4991	5098
50	5575	5619	5830	5172	5493	5770	5469
100	8020	7920	7686	6685	6841	6929	6367
200	14020	13012	12102	9995	9303	9585	11249

En estos graficos vemos el número de threads del servidor (N thread) vs el numero de requests concurrentes que realiza el cliente Web y como resultado el tiempo en ms.

Como podemos ver el uso de nuestra maquina EC2 es evidente



Conclusiones

Teniendo en cuenta el error en los datos por agentes externos como la red y la estabilidad del servidor en heroku, es consistente el comportamiento, a una baja cantidad de hilos del servidor, al recibir muchas peticiones es considerablemente más lento una aplicación multi-thread, no obstante cuando recibe muy pocas peticiones es capaz de manejar incluso más rápido las peticiones que una que su contraparte multi.thread, esto se debe al overhead que introduce el scheduler y el ciclo de los threads, por ejemplo el servidor con un solo thread es capaz de responder más rápido que cualquiera de las otras configuraciones cuando se le hace una petición a la vez, pero cuando subimos hasta 400 peticiones concurrentes es cuando más notamos la diferencia siendo positivo el cambio para aumentar el nivel de paralelismo también podemos observar qué al subir demasiado el nivel de paralelismo empezamos a perder eficiencia y los requests tardan más tiempo, esto se debe a que nuestra máquina no tiene la capacidad suficiente como para manejar tantos hilos al tiempo, recordemos que estos no son hilos del procesador si no que son hilos a nivel de la JVM que se comunican con el scheduler del sistema operativo, por lo que en algún punto se debe realizar interliving para simular 160 hilos o más. Estos hilos consumen más recursos, CPU al evitar tiempos en desuso y más memoria RAM ya que cada uno tiene

que tener almacenado el estado en el que se encuentra el stack y todo el entorno en el que se esté ejecutando la computación de ese hilo. Por lo que llegado un punto nuestra pequeña máquina en Heroku empieza a perder eficiencia después de una cantidad determinada de hilos, eso también lo podemos observar con la máquina de EC2 de AWS, en la que si hacemos 2000 llamados concurrentes directamente se bloquea al quedarse sin memoria para almacenar tantos procesos al mismo tiempo.