LABORATORIO 4

Michael Steven Osorio   
Carlos Mario Alba  
Juan Camilo Bojaca

Contenido

[PREGUNTA 1 1](#_Toc4360320)

[TCP Autentificación 1](#_Toc4360321)

[Diseño 1](#_Toc4360322)

[Resultados 1](#_Toc4360323)

[UDP Streaming 2](#_Toc4360324)

[Pregunta 2 3](#_Toc4360325)

[Pregunta 3 3](#_Toc4360326)

[Pregunta 4 3](#_Toc4360327)

[Pregunta 5 4](#_Toc4360328)

# 

# PREGUNTA 1

## TCP Autentificación

### Diseño

Debido a resultados de desempeño vistos anteriormente mientras se desarrollaba el proyecto se supo que la capacidad en términos de concurrencia es alta llegando hasta a 250 usuarios por segundo manejando el posible ataque DDOS. Sin embargo, como se deseaba un desempeño mínimo de 150 usuario de forma concurrente se eligieron escenarios de 60, 100 y 150 para la elaboración del informe. Para el ramp-up se eligió un segundo para los diferentes tamaños de muestras de usuarios con el fin de probar el servidor de manera exhaustiva.

### Resultados

Se tienen las gráficas de los tiempos promedio de procesamiento para iteración con señalando la cantidad de usuarios y el tiempo promedio obtenido.

La primera gráfica muestra los puntos por cada iteración y la segunda muestra el promedio de las dos iteraciones realizadas por cada set de usuarios.

Para eso dos puntos se realizó el servidor en Node.js y se hizo uso del módulo socket.io con el fin de crear sockets y conexiones TCP. La ventaja de esta implementación es que, al ser basada en eventos, no tendrá problemas con el número de conexiones creadas ya que solo se utilizan recursos en el momento en que se realiza una acción simultánea en varias las conexiones.

Ese es el manejo de concurrencia que tienen estos puntos.

Por otro lado, para impedir los ataques de denegación de servicio se utilizó otro modulo llamado ‘ddos’ el cual realiza un conteo de las conexiones pendientes que tiene un mismo usuario y al llegar a un tope definido por el desarrollador envía un 429 al usuario indicándole que no se recibirán más peticiones hasta procesar las que tienen pendientes.

# UDP Streaming

El servidor UDP se implemento usando Java, y rehaciendo lo hecho en el punto anterior. Se uso MulticastSocket lo que permitió un fácil manejo de las direcciones de multicas para cada puerto, la concurrencia se manejó con java Threads inicializando un Threads para la transmisión de cada video. Debido a que no se comparten recursos el único punto de demora es en la creación de estos.

Se uso ScheduledExecutorService para la ejecución de la lectura y subida de los fotogramas usando como tasa la misma tasa de reproducción de video. Esto brinda la ventaja de que en caso de recursos en el host es encuentren muy exigidos, las peticiones se almacenaran en la cola de hilos de manera programada evitando así retrasos. SI embargo puede que por lo mismo fragmentos de video se aceleren y ajusten. Sin embargo la mayor ventaja es que los recursos al finalizar el Threads son liberados para otro de algún otro video.

Debido a que independientemente del numero de usuarios el servidor sube sus canales a un multicast, el servidor se ve poco afectado por el numero de usuarios concurrentes. Pero al tener un lanzador de tareas programadas por cada frame. el aumento de los canales trasmitidos afecta a sobre manera el servidor.

# Pregunta 2

Existen varias herramientas en el mercado para realizar pruebas de carga, pero para este el análisis de este laboratorio se eligieron las herramientas que permitían pruebas de carga sobre conexiones con sockets. Entre las cuales están: Artillery.io y nGrinder. La primera de ellas fue la elegida para realizar las pruebas de los puntos 4.1 y 4.2.   
Artillery.io: Se definen los Scripts de pruebas los cuales tienen dos secciones, config y scenarios. En la sección de config se indica el número de usuarios a crear por cada segundo durante un periodo de tiempo definido. i.e. 20 usuarios cada segundo durante 10 minutos. Y el escenario solo define lo que debe ejecutar cada usuario creado.

nGrinder: Es usado normalmente para testear aplicaciones web pero pueden ser utilizadas para testear aplicaciones en tiempo real que hacen uso de sockets y por ende conexiones TPC. Sus pruebas se escriben en Jython (Python corriendo en la JVM). nGrinder tiene su implementación basada en Grinder pero implementa algunas otras características para permitir la ejecución de múltiples test de forma concurrente.

Posee un Manager de consolas el cual se encarga de gestionar las consolas que estarán ejecutando los scripts de pruebas.

# Pregunta 4

Debido al descuido del equipo la medición de los parámetros no fue implementada en el servidor original para el punto uno. Así que se uso el servidor UDP que tiene implementadas las mismas funcionalidades pero con un log agregado que exporta las medidas a un csv al finalizar la ejecución.

|  |  |  |
| --- | --- | --- |
| **Escenario** | **Tiempos** | **Cola y respuesta** |
| 50 peticiones  /segundo | tiempo promedio en cola: 16.653846153846153 ms  tiempo promedio en procesamiento: 59.09615384615384ms |  |
| 150peticiones  /segundo | tiempo promedio en cola: 65.41447368421052 ms  tiempo promedio en procesamiento: 64.01315789473684ms |  |
| 250 | tiempo promedio en cola: 164.81349206349208 ms  tiempo promedio en procesamiento: 127.67063492063492ms |  |

\*aclaración importante debido a las implementaciones lazy es necesario hacer una primera petición fuera del test para no afectar los resultados reales.

\*\* eje x es tiempo eje y es el numero de usuarios en esa acción.

Se puede ver como en cuanto aumenta la demanda de los servicios empieza a prevalecer el tiempo de encolamiento y frente al de procesamiento. Debido a esto cuando se alcanzan extremos el buffer sencillamente empieza a rechazar todas las solicitudes entrantes para evitar llenarse fuera de la memoria permitida. Por otro lado el tiempo de procesamiento también sube debido a que se crean mas hilos compitiendo por recursos en el host. Anqué en este caso se encuentra limitado a 300 hilos activos.

# Pregunta 5

Permite monitorear todas las conexiones existentes en una máquina, puede ser ejecutado desde la consola de comandos o desde un archivo de scripts con el fin de poder realizarlo de forma periódica. Permite identificar qué puertos se encuentran abiertos y para qué protocolo lo están haciendo, además, permite identificar qué archivo es el que inició la conexión. Lo anterior es útil para identificar posibles malwares en el sistema que esté creando alguna conexión.

# TCP Streaming

Para esta parte realizamos la implementación de un servidor TCP para emisión de archivos de video, su implementación fue realizada en java y debido a una situación de fuerzas mayores para la fecha en la que fue escrito este informe, no fue posible implementarlo desde una máquina virtual de azure.  
Por este motivo el servidor es localhost y el puerto por el que se transmite es el 8080.

En el servidor se crea una instancia de la clase Transmisor.java la cual es la que ejecuta el método PrintStream(cliente.getOutputStream(), true) el cual es el core de la transmisión

Del mismo modo creamos un cliente que escucha lo que se transmite desde el puerto 8080 y desde un IDE como eclipse es posible correrlo y ver via consola cual es el proceso de transmisión de datos.

No obstante, la forma adecuada de probar la emisión del archivo de video es usar un cliente como es el de VLC Player el cual permite ejecutar las acciones básicas de un reproductor como conectar Streaming a canal, play y stop.

En este caso es necesario simplemente correr el servidor desde un IDE como eclipse oxigen como se especificó anteriormente. Abrir el programa VLC y en la pestaña medio se da clic a la opción “Abrir ubicación de red”. En el campo de introducir una URL se agrega

tcp://<IP SERVIDOR>: <PUERTO>

En el caso actual ya que no ha sido posible implementar el servidor desde una máquina virtual de azure o Google nuestra URL será   
tcp://localhost:8080