**Documento de Informe, Análisis y Entendimiento del Problema**

**Luis Miguel Gómez Londoño – 201729597**

**Infraestructura Computacional**

**Universidad de los Andes**

**2019-20**

**Caso #3**

**Profesora: Sandra Rueda**

**Domingo 17 de Noviembre de 2019**

1. **Descripción detallada de la implementación de los monitores.**

\*Se recomienda la lectura del Readme antes de la ejecución del programa.

* 1. **Cambios generales**

Para este caso se hicieron diversas modificaciones en el código para poder monitorear el desempeño de una aplicación servidor/cliente utilizando alta demanda de clientes y manejo de respuestas de servidores a través de un pool de threads… Las principales modificaciones que se hicieron a partir del caso #2 fueron la implementación del pool de threads en el servidor, así como el *GLoad* para la carga masiva de clientes. Como había que implementar un servidor inseguro además del que ya teníamos (con protocolo seguro), se modificó la clase del servidor para que cada protocolo tuviera un método y con esto se da la opción al usuario de elegir el tipo de servidor que se iba a utilizar. El servidor es quien decide que archivo se va a crear. Se hizo una adecuación para tener mas agilidad en la serie de pruebas en donde el usuario corre el servidor y luego elige el punto del trabajo que se va a realizar (P3 seguro, P4 Inseguro), el numero de escenario que va a realizar, y el numero de prueba de este que se va a realizar. Dados estos parámetros, por cada ejecución del servidor se crean dos archivos, el [log de la ejecución](../data/logs/) en:

*data/logs/<P3 ó P4>/Escenario <Numero del escenario ejecutado>/ Escenario<Numero del escenario ejecutado>R<Numero de prueba de ese escenario>-resultados.txt*

y consecuentemente se crea el [archivo de datos de análisis](../data/datos) que sigue el formato:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Delegado* | *Tiempo* | *CPU Load Inicial* | *CPU Load Mitad* | *CPU Load Final* | *Estado Final* |

Que están ubicados en:

*data/datos/<P3 ó P4>/Escenario <Numero del escenario ejecutado>/ Escenario<Numero del escenario ejecutado>R<Numero de prueba de ese escenario>-datos.csv*

Luego, para que los datos fueran mas fáciles de recopilar en Excel, se utilizo en la terminal el comando: *cat \*.csv >Escenario<# del escenario>Merged.csv* , en el respectivo punto de la [carpeta padre](../data/datos) y de aquí se procedía a importar y vincular estos datos a Excel. Revísese el Excel con los nombres de archivos generados [aquí](Nombres%20de%20Archivos.xlsx).

A screenshot of a cell phone

Description automatically generated

Vale la pena resaltar que para que los mensajes asociados con un delegado quedaran escritos en bloque se implementaron dos bloques *synchronized* para la escritura en los logs y data en donde se sincronizaban los archivos tipo *File* del log *D.java, Líneas 238 y 401* y del data *D.java, Líneas 243 y 406* para todo el tema de la concurrencia.

La misma personalización y practicidad se hizo para el cliente, esta vez, el cliente no escribía datos, únicamente debía coincidir con el protocolo del servidor… Como ya no se tenia que usar el *main* del cliente porque el *generator* usa es el constructor (objeto), entonces se dejo el *main* por si se quería dejar alguna opción de correr un solo cliente (Como solía ser) y se implemento una personalización de datos para la creación de clientes en el *generator*, y, al igual que el servidor, poder elegir si se va a utilizar un cliente con protocolo seguro o no. Todo esto de la personalización precisamente para que las pruebas se llevaran a cabo mas eficientemente sin tener que cambiar directamente el código.

**1.2 Cambios para la implementación de monitores**

La implementación de los monitores se llevo a cabo siguiendo el formato de los parámetros de la tabla mencionada en la sección anterior. Los datos de tiempo se recolectaron a partir de un tiempo inicial *D.java, Lineas 171 y 319* para servidor inseguro y seguro respectivamente, justo después de recibir la llave simétrica (Inseguro) y antes de la creación de la llave simétrica (Seguro), y el fin de recolección de tiempo *D.java, Lineas 217 y 378* para servidor inseguro y seguro respectivamente después del envío del hash (Inseguro) y justo después del envió del HMAC cifrado (Seguro). En cuanto a los datos del CPU, para el servidor inseguro *Data.java, Lineas 117, 174 y 234* para CPU Load Inicial, CPU Load Mitad, CPU Load Final respectivamente, y *Data.java, Lineas 264, 322, 395* para el servidor Seguro. Dado que el método *getSystemCpuLoad()* retorna en algunos casos valores negativos, estos se cambian adecuadamente para la impresión de los datos en los archivos de datos.

1. **Identificación de la plataforma.**

**2.1 Características de la plataforma**

**Arquitectura:** *64 bits*

**Número de núcleos:** *Dual-Core Intel Core i5*

**Velocidad del procesador:** *2,9 GHz*

**Tamaño de memoria RAM:** *8 GB*

**Espacio de memoria asignado a la JVM:**

|  |  |
| --- | --- |
| *Real Memory Size:* | *55.3 MB* |
| *Virtual Memory Size:* | *7.50 GB* |
| *Shared Memory Size:* | *15.1 MB* |
| *Private Memory Size:* | *41.4 MB* |

\*Tomado del Activity Monitor de MacOS.

1. **Comportamiento de la aplicación con diferentes estructuras de administración de la concurrencia.**

El objetivo de estos escenarios era evaluar los cambios en el comportamiento del servidor (seguro) ante diferencias en la carga. En este caso se hicieron en total 60 casos: 10 repeticiones por cada caso (6 en total).

Los casos 1 y 4 evaluaban escenarios de cuatrocientas transacciones con retardos de veinte milisegundos en un pool de threads de tamaño 1 y 2 respectivamente (ver Grafica 1). Los casos 2 y 5 evaluaban escenarios de doscientas transacciones con retardos de cuarenta milisegundos en un pool de threads de tamaño 1 y 2 respectivamente (ver Grafica 2). Los casos 3 y 6 evaluaban escenarios de ochenta transacciones con retardos de cien milisegundos en un pool de threads de tamaño 1 y 2 respectivamente (ver Grafica 3). Se analizaron las siguientes graficas y se llegaron a diversas conclusiones respecto a el comportamiento ante diversas cargas:

**A screenshot of a cell phone

Description automatically generated**

*Grafica 1.* [Punto 3 Escenario 1 vs. 4](PUNTO%203.xlsx)

*A screenshot of a cell phone

Description automatically generated*

*Grafica 2.* [Punto 3 Escenario 2 vs. 5](PUNTO%203.xlsx)

A screenshot of a cell phone

Description automatically generated

*Grafica 3.* [Punto 3 Escenario 3 vs. 6](PUNTO%203.xlsx)

En las graficas de tiempo de transacción podemos ver casi el mismo comportamiento en todos los escenarios… Se tiene en los threads iniciales un mayor tiempo de transaccionalidad y luego este se queda por debajo de los 50 milisegundos a mas tardar el delegado ≈ 10. Esto, en mi opinión, se podría tratar debido a la inicialización de estos hilos con algo de secuencialidad mientras se establece el pool de threads. Esto lo podríamos ver cuando los threads intentan acceder a los archivos que están dentro del bloque *synchronized*, solo el thread que lo use en ese instante tendrá acceso inmediato al archivo, cualquier otro que intente accederlo tendrá que esperar… Una vez se distribuyen estas tareas el tiempo se reduce en cuanto se coordine tal “secuencialidad”. Los escenarios 4, 5, y 6 tienden a un tiempo inicial menos que los tres primeros precisamente por que es un hilo mas para trabajar y llegarle a dichos archivos mas rápido. En cuanto a las graficas del uso del CPU, se puede ver muy precisamente lo que pasa cuando se crece el pool de threads. Quizá en los escenarios 3 y 6 no se vea muy bien ya que son menos datos y no se es muy notable el cambio, sin embargo, en los casos 1,6 y 2,5 si podemos efectuar algunas conclusiones. Puede que aumentar el tamaño del pool traiga mas velocidad, sin embargo, se paga el precio adicional de procesamiento extra por mas hilos ejecutándose “a la vez”.

1. **Comportamiento de la aplicación ante diferentes niveles de seguridad.**

Este punto solo resalta una misma carga de datos en el mismo intervalo y cambio de tamaño en el pool, pero en un servidor con un protocolo de seguridad no tan seguro. Se encargaba de hacer una carga de cuatrocientas transacciones con retardos de veinte milisegundos. Una vez se tomaron estos datos pudimos ver algunas diferencias de desempeño y tiempos cuando se usa un servidor con protocolo inseguro:

A screenshot of a cell phone

Description automatically generated *Grafica 4.* [Punto 4 Escenario 1 vs. 2](PUNTO%204.xlsx)

Únicamente con esta grafica podemos llegar a algunas conclusiones parecidas a las del punto pasado, sin embargo, ¿Cómo se compara la aplicación que implementa funciones de seguridad vs. la aplicación que no implementa funciones de seguridad? Para responder esta pregunta podríamos comparar la Grafica 4 con las demás graficas del punto pasado y fijarnos en las del uso de CPU. Nótese que el uso del CPU en un servidor con protocolo inseguro toma menos procesamiento a comparación de las tres ultimas graficas… Viéndolo de forma muy general fíjese que en el servidor inseguro sin tener en cuenta los primeros delegados, el procesamiento del CPU no supera el 28% mientras que en los otros casos (sobretodo en las Graficas 2 y 3) se puede ver mas “de cerca” que el uso de protocolos mas robustos y seguros toman mas procesamiento de la maquina. Esto debido a que estos protocolos necesitan la generación de llaves y certificados de mucha mas seguridad y complejidad que en uno que no los use. En todos los escenarios del servidor seguro se utilizaron los siguientes algoritmos: para el cifrado simétrico se utilizó AES, modo ECB, esquema de lleno PKCS5 y llave de 128 bits; para el cifrado del HMAC se uso un SHA 256. El procesamiento de cifrado y descifrado de estos algoritmos elevan el procesamiento y se puede notar en las graficas. Además, esto coincide con la forma en la que fueron “instalados” los monitores ya que estos fueron montados antes y después de la creación y generación de estas llaves con estos algoritmos, lo que marca sus efectos en los datos recolectados.