**Caso 2**

**Infraestructura Computacional** 

**Nombre:** Mauricio Días Torres    **Código:** 201630971

**Nombre:** Luis Ernesto Viana Castillo **Código:** 201618266

1. **Definición Detallada de monitores:**

El monitor se estableció en ambos servidores, este monitor se define como una clase extra que se encarga de reportar el número de transacciones ejecutadas con éxito, el tiempo que dura cada una de las transacciones y el porcentaje de CPU utilizado por transacción.

Esta clase consiste en un monitor que contiene 3 variables:

* **cpuList**: Una lista que contiene el uso porcentual del CPU dadas las transacciones, por lo cual contiene elementos tipo *double.*
* **tiemposSolicitud:** Una lista que contiene el tiempo de ejecución de cada transacción, por lo que contiene elementos tipo *long*, ya que el valor esta dado en milisegundos.
* **transacciones:** representa el número de transacciones que se ejecutaron con éxito.

Dado todo lo anterior, utilizamos el método sugerido por el enunciado para medir el valor porcentual del CPU, teniendo esto en cuenta, realizamos 3 métodos más:

* **monitorearCPU**: Este método se encarga de generar el uso del CPU en el momento y luego lo agrega a la lista cpuList.
* **monitorearTiempo**: Este método recibe un parámetro tipo *long* que representa el tiempo de ejecución de una sola transacción, y lo guarda en la lista tiemposSolicitud.
* **reportarTrasacción**: Este método se encarga de cambiar la variable “transacciones” para que aumente, de modo que se pueda revisar después y saber si se perdió una transacción.

Además, para poder adaptar el servidor y usar este monitor se hicieron cambios en la clase C y D, los cambios en la clase C fueron:

* Creación de un *ExecutorService* que representa un pool de *threads* que ejecuta los *threads* delegados del servidor.
* Líneas de código que permitieran la entrada de información para definir el tamaño del pool de *threads* que se usa.
* Referencia estática al monitor, el cual se pasa por parámetro al crearse el thread.
* Usos de métodos del *ExecutorService* como *shutdown* y *awaitTermination* para que, al terminar todas las transacciones el servidor pueda continuar y reportar los resultados obtenidos.
* Finalmente, se usó una librería de Apache llamada POI para generar el archivo Excel con el reporte, de manera que se recorren las listas generadas por el monitor (cpuList y tiemposSolicitud) y las imprimen en un archivo xls para hacer el análisis con inmediatez.

Los cambios en la clase D, la cual representa un thread delegado que responde al cliente, fueron:

* En lugar de extender de la clase *Thread*, ahora implementa la interfaz *Runnable,* esto con la intención de facilitar la implementación del pool de threads.
* Una referencia al monitor, de manera que la clase C al crear este thread le pasa por parámetro el monitor estático y este delegado lo inicializa en el constructor.
* Debido a que el enunciado dice que el tiempo de transacción se cuenta desde la lectura de la llave simétrica, a partir de la fase 5 se utiliza el método del sistema: *currentTimeMillis* que se va a comparar con el tiempo después de terminar la transacción para obtener el tiempo de transacción.
* El monitor en esta clase tiene el rol de:
  + Realizar el reporte de que la transacción se realizó exitosamente
  + Realizar el reporte de porcentaje de CPU en este thread dada la transacción realizada anteriormente.
  + Dado el tiempo calculado por la comparación de tiempos desde el inicio hasta el final, se reporta el tiempo que tomó esta transacción en particular.

1. **Identificación de la plataforma**

Se utilizó el comando de Windows *dxdiag* a través de la aplicación del *Run* para encontrar los siguientes datos de la máquina en la que se corren las pruebas:

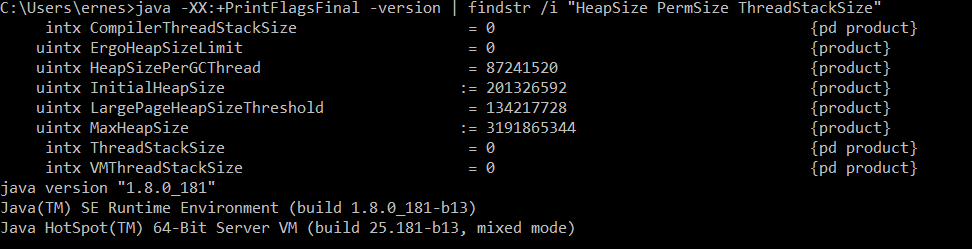
Arquitectura: 64bits

Numero de núcleos = 4

Velocidad Procesador = 2,5GHz

RAM = 12Gb

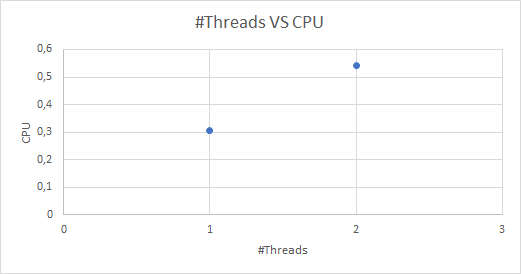
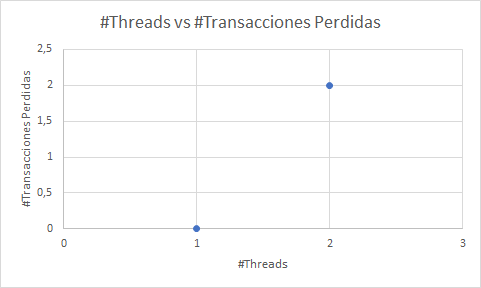
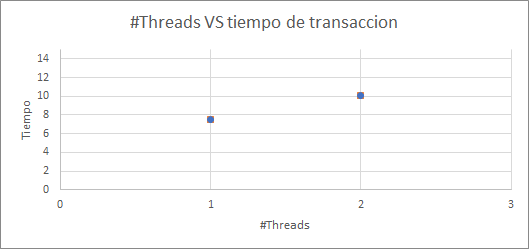
Por otro lado, para saber la cantidad de memoria alocada por Java para su máquina virtual (JVM) se utilizaron comandos de java a través de la consola de comando de Windows, de la siguiente manera:



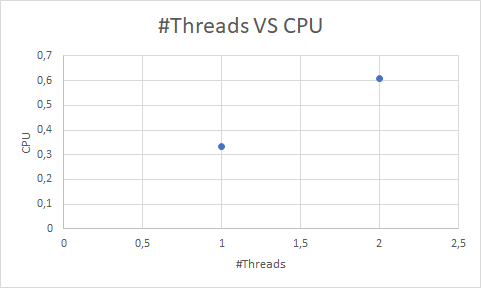
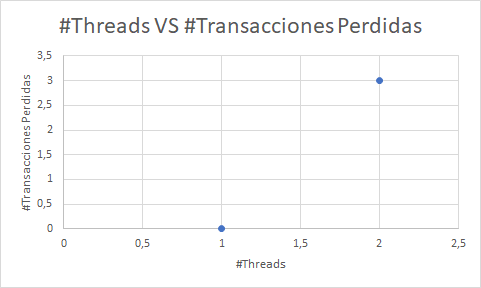
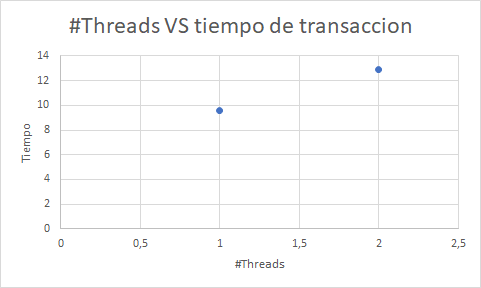
Por lo cual, podemos concluir, que la memoria *Heap* asignada a la máquina virtual de JAVA inicialmente es: 201.3 Mb, aunque se puede asignar hasta 3191.9 Mb. [1]

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

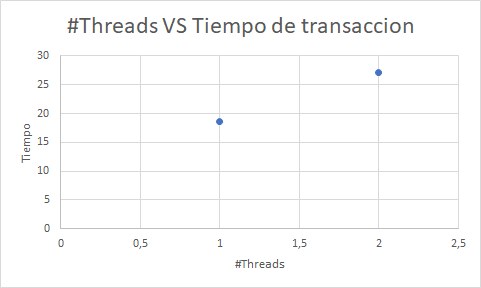
* **Carga = 400**

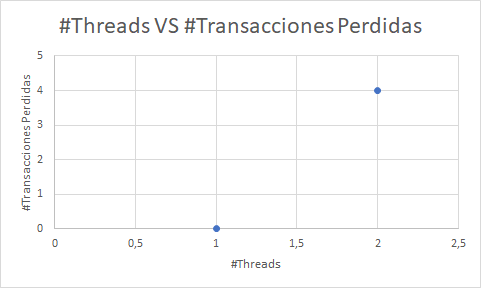


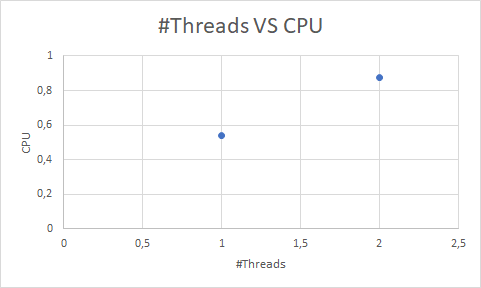
* **Carga = 200**



* **Carga = 80**







* **CONCLUSION**

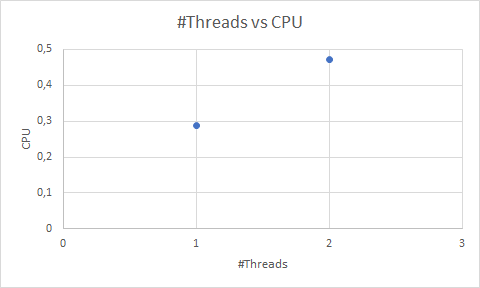
Al comparar las 9 gráficas se concluyó que el menor tiempo de transacción y menor uso de CPU fue al tener una carga de 400 con 1 thread, ya que al incrementar # de threads se incrementaba tanto el uso de CPU como el tiempo de transacción, también al tener menos carga se incrementaban estos dos indicadores notablemente, creemos que esto se debe a que la memoria asignada por la máquina virtual aumenta con la carga, por lo que a cargas menores hay menor memoria, y dado que la máquina real en la que se está corriendo la JVM tiene la capacidad para surtir las necesidades de la memoria que pide la JVM, no tiene muchos problemas al ejecutar cargas muy grandes.

Por otro lado, el aumento en el número de threads genera más uso de CPU y tiempos de transacción, concluimos entonces que esto se debe al uso de la concurrencia como solución al problema, es decir, debido a que se tiene la misma cantidad de recursos y se aumentan las tareas que se deben hacer al mismo tiempo, esto genera que individualmente utilicen más CPU y se demoren más, sin embargo, esto ayudaría el tiempo de todo el proceso (todas las transacciones), por lo tanto, es una solución que no dé debe ignorar por completo.

Lo anterior nos dice que no es suficiente con aumentar el número de threads que atienden solicitudes, por lo menos no el doble, y que, si queremos saber que tanto hay que aumentar el tamaño del pool, debemos experimentar.

Para el número de transacciones perdidas se observó que nunca se perdían transacciones con 1 thread sin importar la carga, pero cuando eran 2 threads el número de transacciones perdidas incrementó a medida que se disminuía la carga (pasó de 2 a 4 transacciones perdidas), esto se debe a que los clientes aceptados quedan esperando por la ejecución de su transacción, y al haber más de un thread aceptando solicitudes puede generar conflictos sobre el uso de los recursos, lo cual lleva a perdida de transacciones.

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



Se espera que una aplicación con seguridad debería tener mayor consumo de CPU y mayor tiempo de transacciones por lo que aparte de hacer las mismas funciones que una aplicación sin seguridad, una aplicación que si implementa funciones de seguridad tiene que hacer protocolos de seguridad donde: crea llaves públicas, privadas y funciones criptográficas, las cuales tienen algoritmos que consumen más CPU y que se demoran más en cálculos que llevan al cifrado y descifrado de información.

* **CONCLUSION**

Como era de esperarse, se pudo ver que la diferencia entre una aplicación con seguridad vs una sin seguridad es que en promedio consume 1.53% más CPU (1 thread) y 7.23% (2 threads), por lo dicho anteriormente que una con seguridad tiene más funciones y complejidad que una app sin seguridad.

**Bibliografía**

[1] https://www.mkyong.com/java/find-out-your-java-heap-memory-size/