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

Se implementaron un método para hacer la medición de CPU, el método está en la clase D.java de cada servidor. Por otro lado, para el tiempo de transacción se hace un cálculo en la clase D.java de cada servidor. En la clase C.java del servidor se establece la comunicación a través del protocolo TCP en el puerto 5555 por medio de sockets. Una vez creada la conexión, se calcula el uso de CPU antes del inicio del protocolo y después de que se acaba el protocolo y se imprime por consola. A continuación se presenta la imagen del método que calcula el uso de CPU:

```
public double getSystemCpuLoad() throws Exception
{
    MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
    ObjectName name = ObjectName.getInstance("java.lang:type=OperatingSystem");
    System.out.println(name);
    AttributeList list = mbs.getAttributes(name, new String[]{ "SystemCpuLoad" });
    System.err.println(list);
    if (list.isEmpty()) return Double.NaN;
    Attribute att = (Attribute)list.get(0);
    Double value = (Double)att.getValue();
    // usually takes a couple of seconds before we get real values
    if (value == -1.0) return Double.NaN;
    // returns a percentage value with 1 decimal point precision
    return ((int)(value * 1000) / 10.0);
}
```

Paralelo a esto, para el cálculo del tiempo se crea una variable donde comienza el protocolo y a partir de esta se calcula la duración del protocolo con una resta:

```
long start = System.nanoTime();
```

```
double elapsedTimeInSec = (System.nanoTime() - start);

System.err.println("Elapsed Time: "+ elapsedTimeInSec);
System.err.println("Memory "+ getSystemCpuLoad());
```

A su vez, hay un método que imprime el tiempo y el porcentaje de memoria utilizados por cada transacción, se usa un script en python que analiza cada uno de los archivos generados por la transacción y se generan las gráficas con matplotlib.

2. **Identificación de la plataforma:**

   No se hizo uso de la máquina virtual puesto que la profesora indicó que no era obligatorio.

   Las características de la máquina en las que corre el servidor son:

   - Arquitectura de 64 bits
   - Número de núcleos: 8
   - Velocidad del procesador: 3.7 GHz
   - Tamaño de la memoria RAM 64GB
   - Espacio de memoria asignado a la JVM: El tamaño máximo es Xmx 512m y el tamaño mínimo es Xms 40m

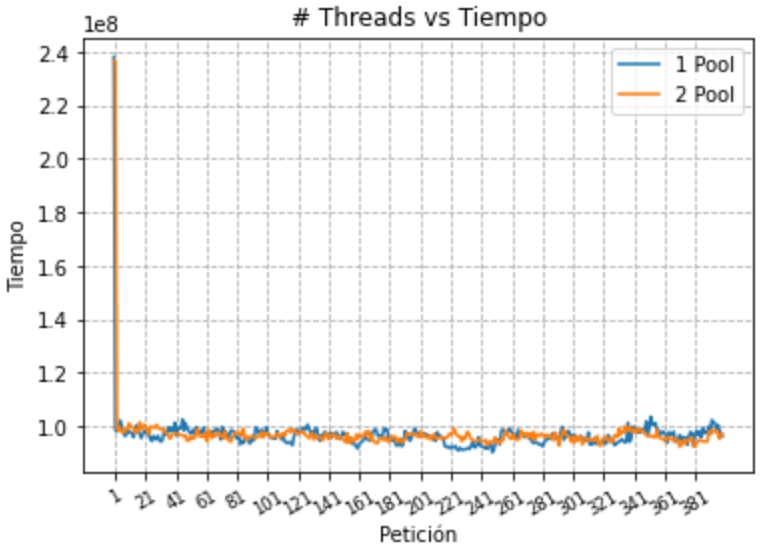
3. **Comportamiento de la aplicación con diferentes estructuras de administración de la concurrencia:**

   A continuaciń se muestra el comportamiento de la aplicación en diferentes escenarios. En el archivo *Datos Finales Con Seguridad.xlsx* se encuentran el análisis de los datos de los escenarios con distintas cargas y con distinto número de pools. Las gráficas a continuación se hicieron basadas en el promedio de sus tiempos y de su % de uso de CPU. Sin embargo, en el archivo de excel se encuentra el análisis sobre la desviación estándar de los datos, el máximo y el mínimo.

   - Gráficas:

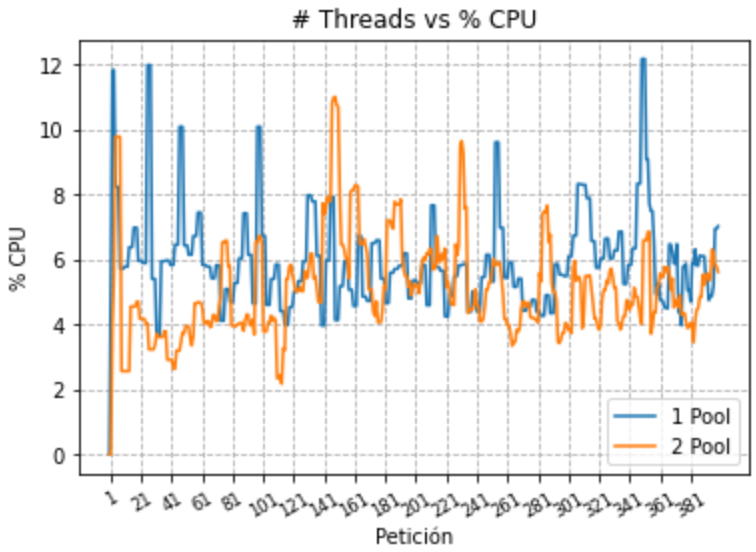
a. 400 transacciones con 20 ms de retraso:

- **# de Threads vs Tiempo de Transacción:**



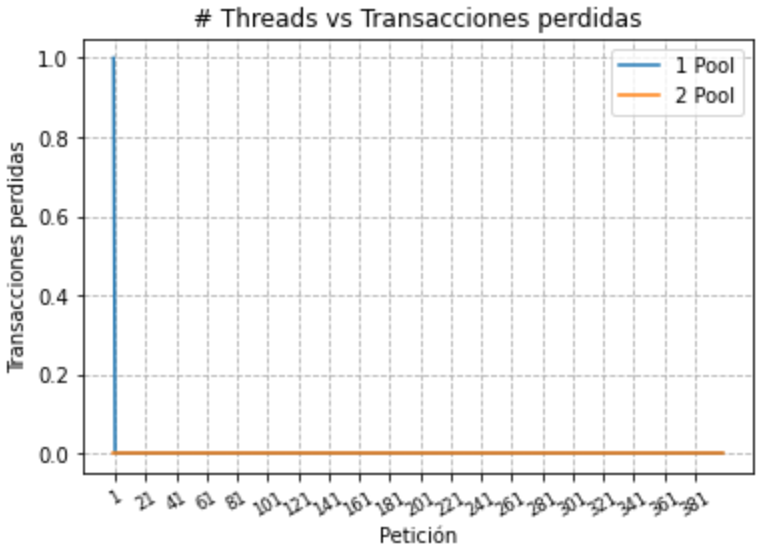
Se tomaron los datos para los 10 experimentos para un pool de tamaño 1 y 2 respectivamente. Luego se hacen las respectivas gráficas y se puede observar cómo los datos se comportan de manera similar. Esto se puede ver como si los threads estuvieran ocupados haciendo otra tarea.

- **# de Threads vs % uso de CPU:**



Aquí se puede observar cómo en la mayoría de las transacciones hubo menor uso de CPU cuando hay 2 threads encargados, puesto que entre los 2 se pueden turnar las transacciones y despachar las tareas mucho más rápido.

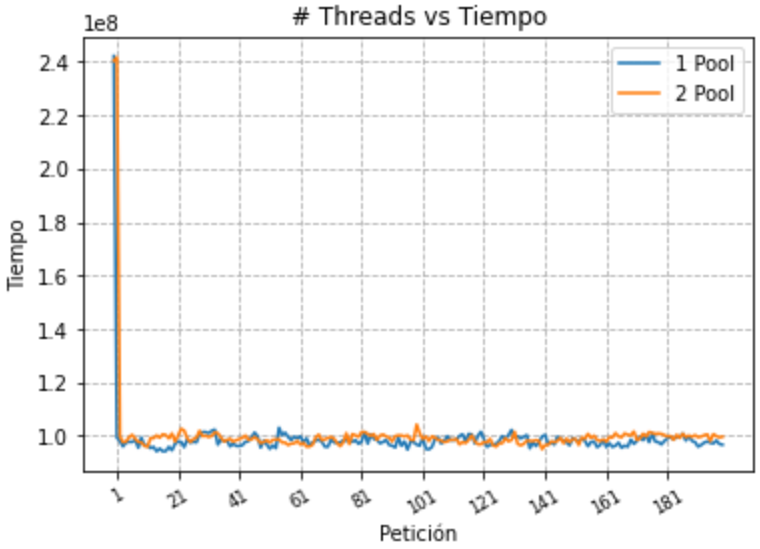
- **# de Thread vs Transacciones perdidas:**



Se revisa el log del proyecto y se observa que las pérdidas suceden al inicio de las transacciones solicitadas. Se puede entender lo anterior como la búsqueda de recursos dentro del pool de threads, lo que se traduce en retrasos muy cortos que generan pérdidas.

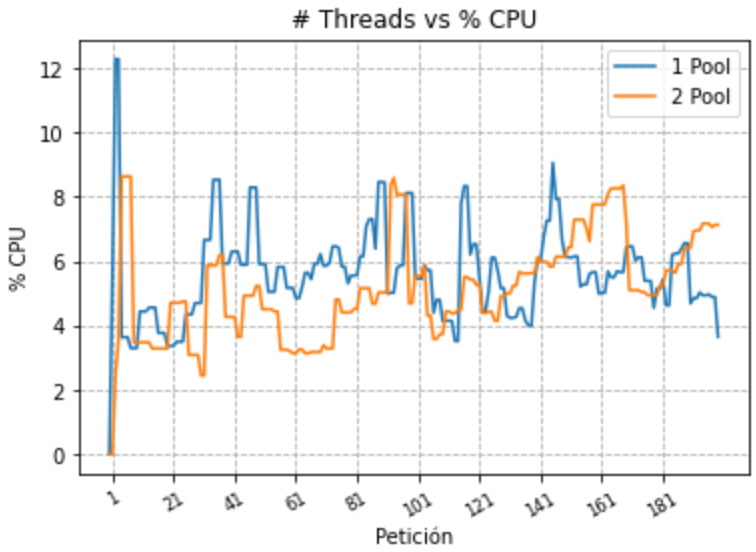
b. 200 transacciones con 40 ms de retraso:

i. # de Threads vs Tiempo de Transacción:



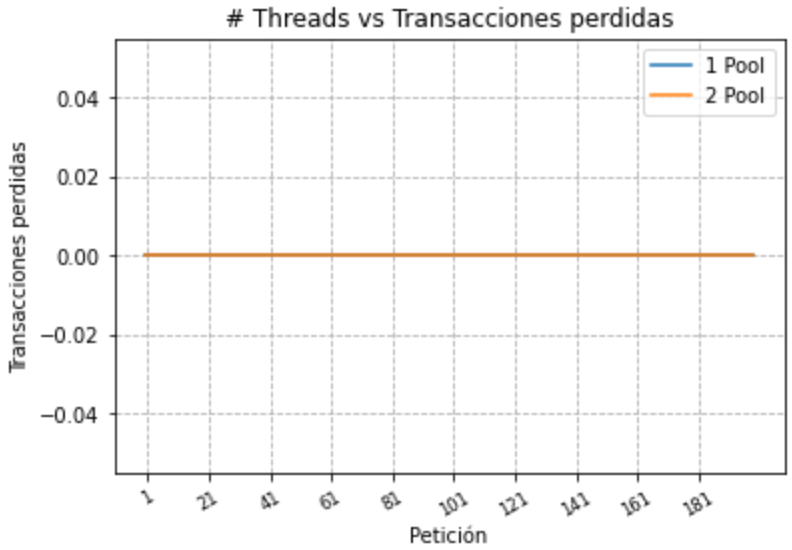
Cómo se puede observar, la gráfica se comporta muy similar a la que se obtuvo en la gráfica de las 400 transacciones. Lo anterior se debe a que, aunque la carga es menor las tareas al ser las mismas en los tres escenarios, gastan el mismo tiempo en realizarse.

ii. # de Threads vs % uso de CPU:



Se observa el mismo comportamiento que en la gráfica de 400 transacciones. Con la diferencia que al ser menos tareas a realizar es menos probable que se necesite más de un thread para realizarlas.

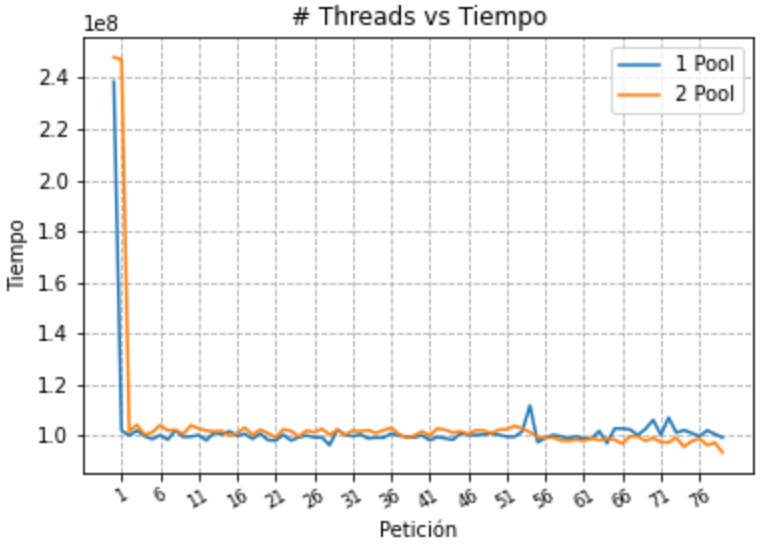
iii. # de Thread vs Transacciones perdidas:



En la anterior gŕafica se comprueba que, debido a que el tiempo de retraso es el mayor entre cada tarea, las órdenes no pelean por recursos ya que tienen suficiente tiempo entrar a la cola.

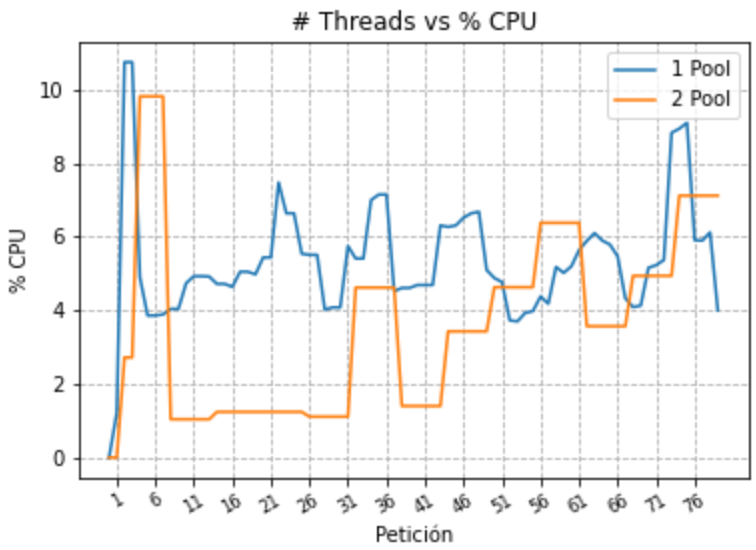
c. 80 transacciones con 40 ms de retraso:

i. # de Threads vs Tiempo de Transacción:



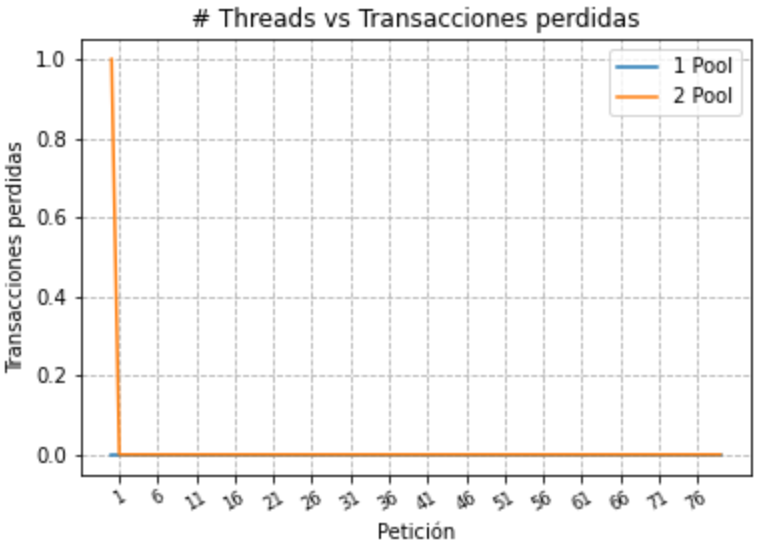
El número de threads vs el tiempo se comporta de manera similar a los demás, debido a que el tiempo en realizar una tarea, en este caso crear la llave simétrica y el código de autenticación es el mismo sin importar la carga que se tenga.

ii. # de Threads vs. % uso de CPU:



Se observa menos uso del procesador que en los anteriores casos, lo anterior se debe a que es menos carga y además el tiempo de retraso es mucho mayor. De la misma manera que las demás gráficas, el pool 2 emplea menos porcentaje del procesador.

iii. # de Threads vs. Transacciones perdidas:



Lo que se explicó en las anteriores gráficas, dado que el tiempo es mayor entre transacción, no hay necesidad de luchar por buscar recursos, sin embargo hay una pérdida al inicio de las transacciones.

Conclusiones del análisis:

Se puede observar como el uso de un pool de threads con tamaño fijo controla los recursos. En consecuencia, es más eficiente crear al inicio del proceso los threads que se necesita que en el momento de realizar la petición. Los datos obtenidos reflejan cómo un pool con tamaño fijo 2, en caso de necesitar otro por la finalización inoportuna de un thread, no debe ir a saturar al procesador dado que tiene otro thread disponible.

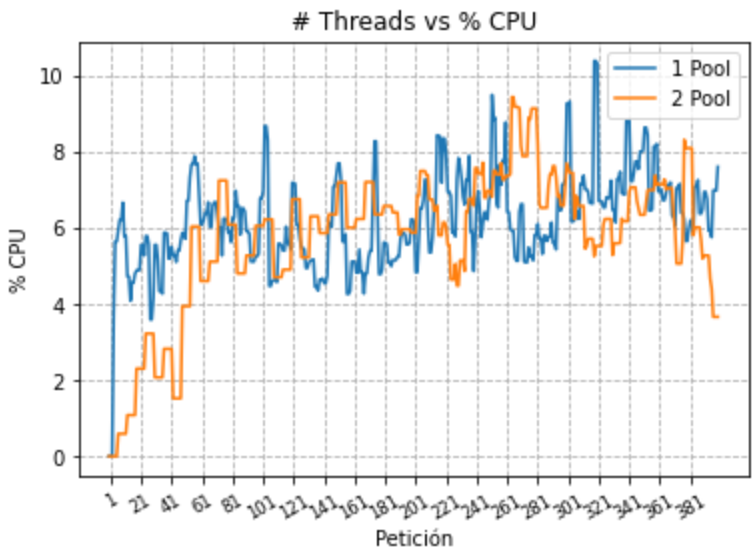
Por otro lado, el tamaño de los datos influye. Al ser menos datos y más tiempo en creación de las tareas hay menos probabilidad de pérdidas ya que no hay problemas en los recursos existentes.

Calidad de datos

1. Se midió la diferencia entre el consumo de CPU y tiempo entre una transacción no segura y segura.
2. Se tomaron todos los datos en computadores con las mismas especificaciones y siempre en esos dos.

4. **Comportamiento de la aplicación ante diferentes niveles de seguridad.**
    - Ahora se muestra el comportamiento de la aplicación sin algún manejo de seguridad. Los datos recopilados se encuentran en el archivo: *Datos Finales Sin Seguridad.xlsx*

    a. # Threads vs % uso de CPU:



Se observa en comparación a la gráfica obtenida en el protocolo de seguridad, que es más rápido debido a que no debe verificar la misma cantidad de algoritmos. Por lo tanto, es menos probable que una tarea no finalice conduciendo a que no se deben crean más threads de lo que ya se encuentran en el pool de threads.

5. **Logros y observaciones:**

- Se modificó el código servidor para manejar pool de threads.
- Se logró implementar el protocolo de seguridad entre el cliente y el servidor de manera exitosa para el literal 4.
- Se implementaron monitores de desempeño para medir indicadores, como: el porcentaje de CPU
- Se generaron los seis escenarios para los cuales se hicieron diez experimentos de cada uno. Por otro lado, se generaron las tablas correspondientes.
- Se logró analizar los resultados obtenidos.

6. Conclusiones:

- El proyecto “Análisis de desempeño” cubre de manera satisfactoria la evaluación de características de desempeño de la infraestructura computacional
- La realización del proyecto implicó la simulación de carga y configuraciones de software con el fin de analizar el comportamiento de la aplicación bajo el paradigma arquitectónico de cliente-servidor.
- Las competencias adquiridas fueron: la evaluación del efecto que tienen los mecanismos de seguridad sobre el desempeño de una aplicación y las consecuencias en el uso de threads fijo en el consumo de CPU.