Caso 3

Integrantes (Grupo x, Sección 4)

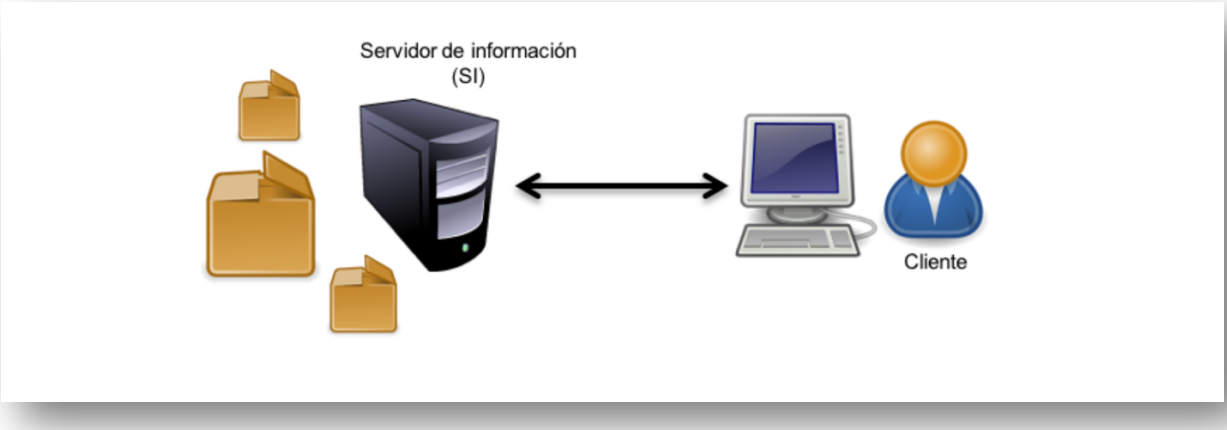
- Juan José Díaz Ortega - 202220657
- Paul Paffens - 202222496

1. Objetivos

• Comprender ventajas y limitaciones de los algoritmos para cifrar datos.
• Construir un prototipo a escala de una herramienta para soportar confidencialidad e integridad.

2. Problemática

Supondremos que una compañía de transportes ofrece a sus clientes el servicio de recogida y entrega de paquetes a domicilio. La compañía cuenta con un servidor para soportar la operación de manejo de paquetes y las consultas de los usuarios. Para el manejo de paquetes y unidades de distribución, tanto los puntos de atención al cliente como las unidades de distribución se comunican periódicamente con el servidor para informar su estado. El servidor almacena la información y atiende consultas relacionadas con estos datos vía internet.



Para este caso nos concentraremos en el proceso de atención a las consultas de los usuarios. Su tarea consiste en construir los programas servidor y cliente que reciben y responden las solicitudes de los clientes.

Servidor:

- Es un programa que guarda un identificador de cliente, un identificador de paquete y el estado de cada paquete recogido (los estados posibles son: ENOFICINA, RECOGIDO, ENCLASIFICACION, DESPACHADO, ENENTREGA, ENTREGADO, DESCONOCIDO) y responde las consultas de los clientes.
- Los estados deben representarse como constantes numéricas, excepto al presentarse mensajes a un usuario, allí deben presentarse en formato alfabético.
- Para simplificar el problema, tendremos un servidor con una tabla predefinida con: login de usuario, identificador de paquete y estado de 32 paquetes. Para consultar el estado de un paquete, un cliente envía al servidor su identificador y el identificador de un paquete, el servidor recibe los datos, consulta la información guardada y responde con el estado correspondiente. Si el identificador de usuario y paquete no corresponden a una entrada en la tabla, el servidor retorna el estado DESCONOCIDO.
- Es un servidor concurrente. El servidor principal crea los delegados por conexión al recibir cada cliente.

Cliente:

- Es un programa que envía una consulta al servidor, espera la respuesta y al recibirla la valida. Si la respuesta pasa el chequeo entonces despliega la repuesta en pantalla, si no pasa el chequeo entonces despliega el mensaje ˙Error en la consulta".
- Se sugiere manejarlo de forma concurrente.

Tanto servidor como cliente deben cumplir con las siguientes condiciones:

- Las comunicaciones entre cliente-servidor deben usar sockets y seguir el protocolo indicado.
- No use librerías especiales, solo las librerías estándar.
- Desarrolle en Java (java.security y javax.crypto).

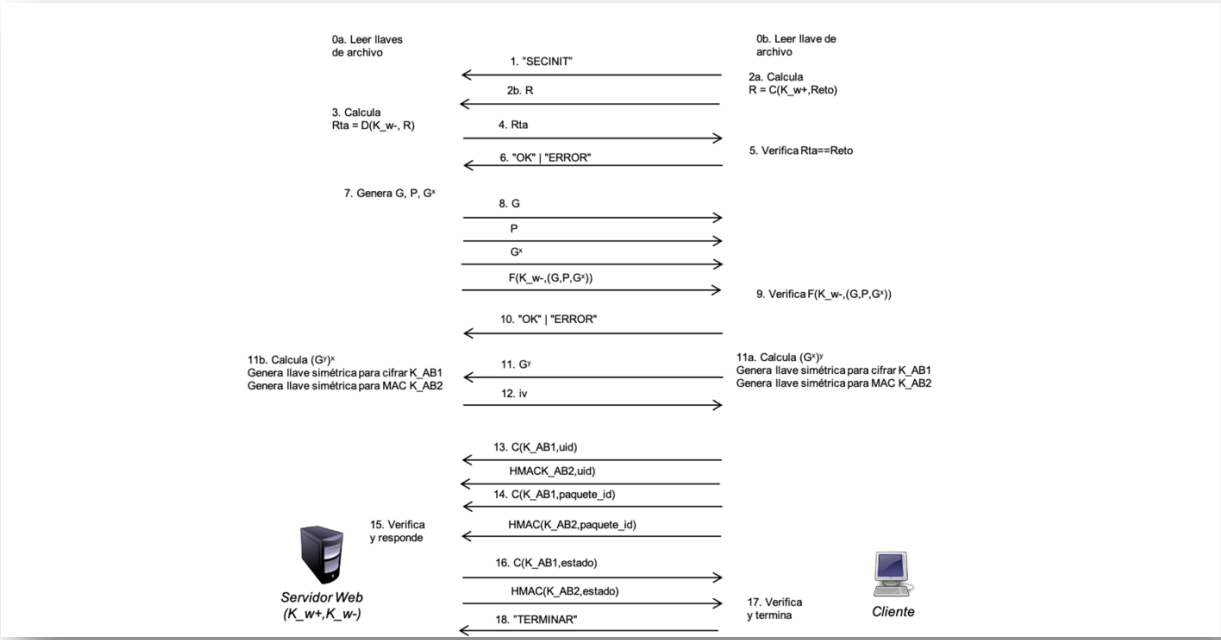


Ilustración 1

Opciones:

- En el código del servidor incluya un menú que tenga dos opciones:
  - o Opción 1: Generar la pareja de llaves asimétricas del servidor y almacenarlas en dos archivos. o Tenga en cuenta:
    - § En una instalación real los permisos asignados a estos archivos serían fundamentales; la llave pública puede ser leída por cualquiera, pero la llave privada solo debería estar al alcance del propietario.
    - § En este prototipo deberá copiar el archivo de la llave pública a un directorio donde los clientes la puedan leer.
  - o Opción 2: Ejecutar creando los delegados, cada delegado sigue el protocolo descrito en la figura 1.

3. Algoritmos usados

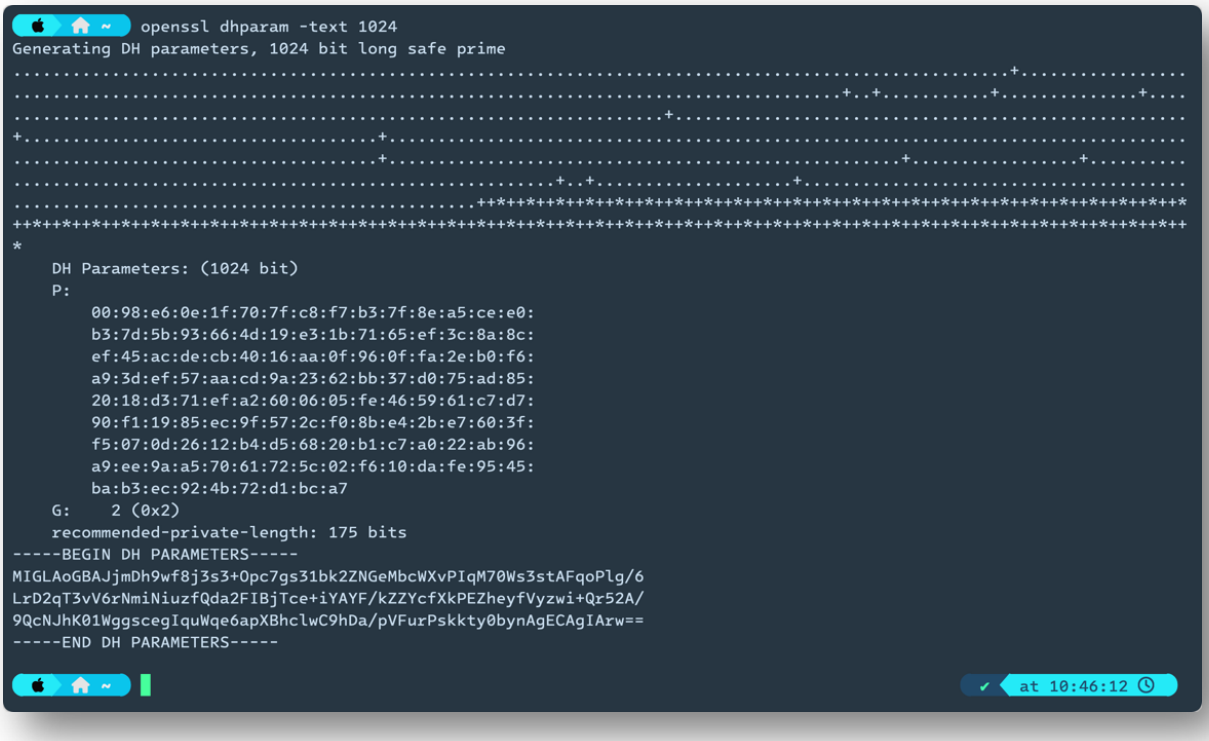
Algoritmos que usaremos:

- o AES. Modo CBC, esquema de relleno PKCS5, llave de 256 bits.
- o RSA. Llave de 1024.
- o El vector de inicialización para cifrado simétrico (iv) debe tener 16 bytes generados aleatoriamente o Firma: SHA1withRSA
- o HMACSHA384.

4. Llaves Simetricas

Para generar las llaves simétricas:

1. Primero calcule una llave maestra por medio de Diffie-Hellman.



P = 00:98:e6:0e:1f:70:7f:c8:f7:b3:7f:8e:a5:ce:e0:b3:7d
:5b:93:66:4d:19:e3:1b:71:65:ef:3c:8a:8c:ef:45:ac:de:cb:40:16:aa:0f:96:0f:fa:2e:b0:f6:
a9:3d:ef:57:aa:cd:9a:23:62:bb:37:d0:75:ad:85:20:18:d3:71:ef:a2:60:06:05:fe:46:59:61:c7:7
:90:f1:19:85:ec:9f:57:2c:f0:8b:e4:2b:e7:60:3f:f5:07:0d:26:12:b4:d5:68:20:b1:c7:a0:22:ab:96:
a9:ee:9a:a5:70:61:72:5c:02:f6:10:da:fe:95:45: ba:b3:ec:92:4b:72:d1:bc:a7

G = 2 (0x2)

2. Para manejar números grandes use la clase BigInteger de Java. Los números primos deben tener 1024 bits de longitud.

```
BigInteger sharedSecret = new BigInteger();
```

Esto lo hicimos con el respectivo valor que calculamos arriba y para los concurrentes se calcula cada vez que se inicia diferentes llaves.

3. Luego use la llave maestra para calcular un digest con SHA-512

Esto lo hicimos en una clase estática dentro del servidor llamada **DiffieHellman**

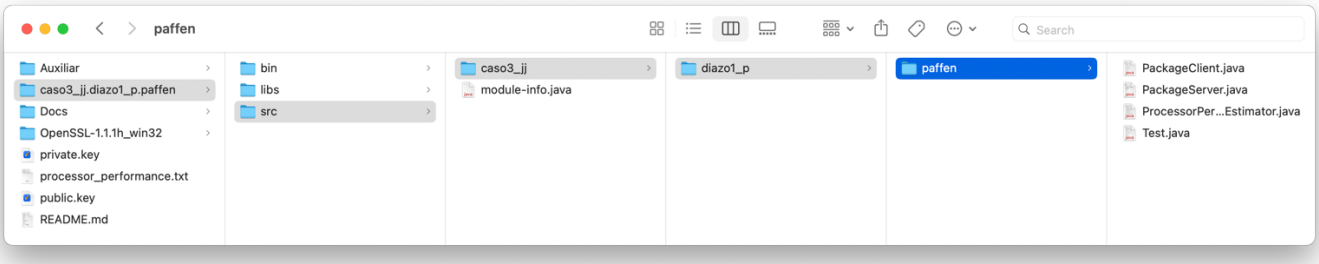
4. Parta el digest en dos mitades para generar la llave para cifrado y la llave para código de autenticación: usar los primeros 256 bits se deben usar para construir la llave para cifrar, y los últimos 256 bits para construir la llave para generar el código HMAC

```
byte[] encryptionKey = new byte[32];
System.arraycopy(digest, 0, encryptionKey, 0, 32);
byte[] hmacKey = new byte[32];
System.arraycopy(digest, 32, hmacKey, 0, 32);
```

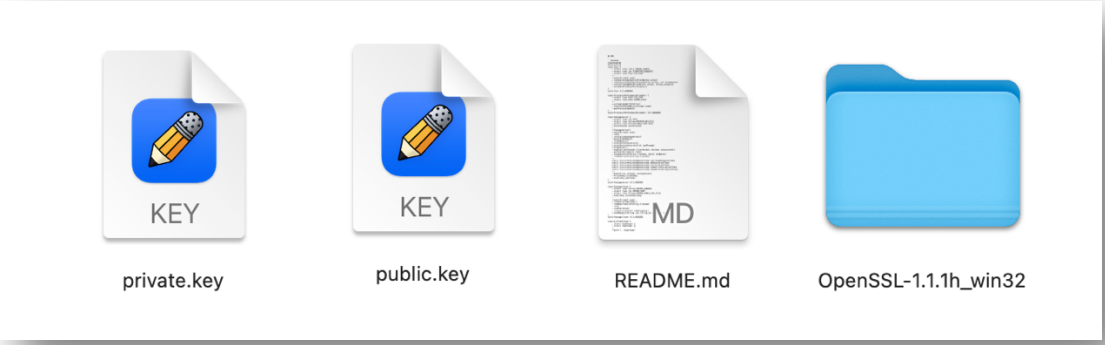
Esto lo hicimos dentro del metodo **handleClient**() en el servidor y en el **sendRequest**() en el lado del cliente

5. **Organización en el zip**

En el archivo zip se encuentra el código que se usa para generar el escenario (bajo la carpeta caso3_jj.diazo1_p.paffen) están las *clases PackageClient, PackageServer, ProcessorPerformanceEstimator* y *Test*, que se explicaran más adelante con un UML.

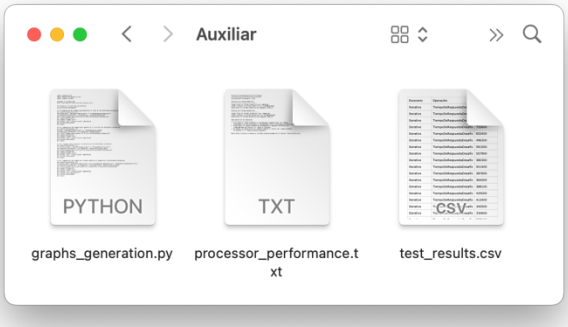


También se encuentran las llaves generadas, la carpeta *OpenSSL* que utilizamos para la generación de llaves y otras operaciones.

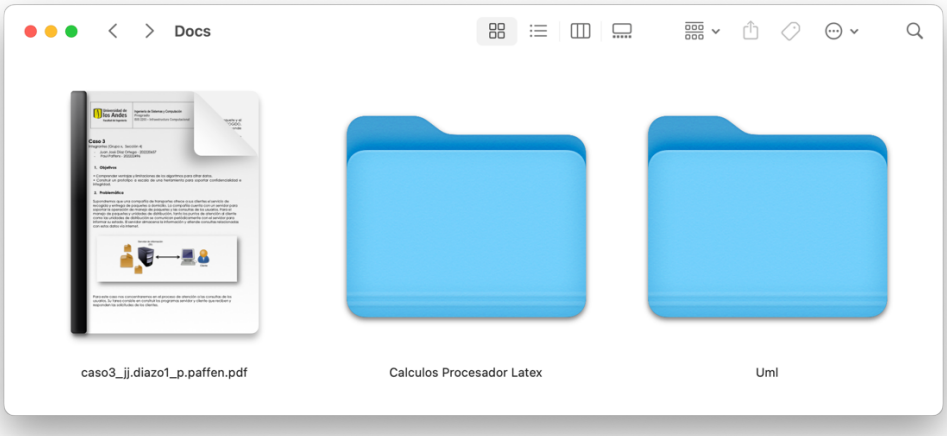


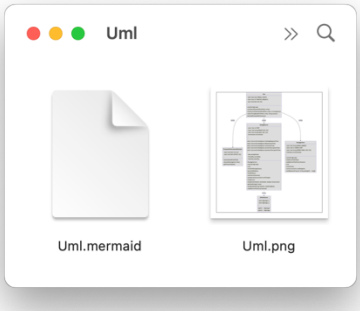
También está el README de GitHub pero lo pueden ignorar (todavía lo tenemos privado pero lo pondremos publico después de la entrega).

Junto con esto se encuentra la carpeta *auxiliar*, donde se genera el archivo CSV con los tiempos solicitados para los 4 escenarios distintos *test_results.csv* y el *processor_performance.txt* que contiene los cálculos del procesador (el output de la clase que hace los cálculos del procesador), además de estos archivos la carpeta también contiene un generador de graficas *graphs_generation.py* que analiza los datos del CSV y genera la gráfica correspondiente.

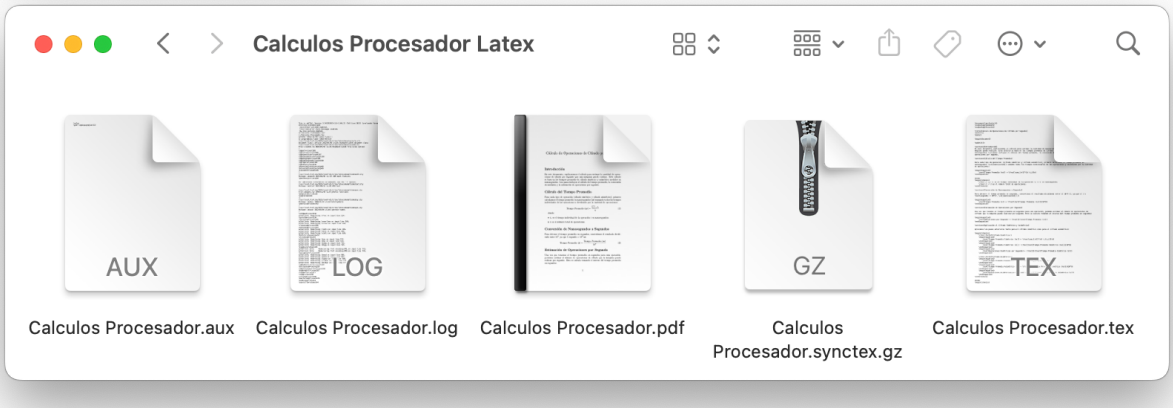


Por ultimo encontramos la carpeta *Docs* que si están leyendo este documento es desde la que accedieron a él, en esta tenemos además 2 subcarpetas, una para el código del UML que hicimos en mermaid (una herramienta de graficas) y los cálculos del procesador que teníamos en Latex antes de pasarlos a este documento.





Cabe aclarar que no tienen que entrar a estas 2 subcarpetas, solo las dejamos por si necesitábamos editar algo, pero todo está dentro de este mismo documento, tanto los cálculos del procesador, que se encuentran al final del documento como el UML que se presentara en corto (punto 7).



6. Instrucciones de uso

1. Correr el servidor y generar las llaves RSA

```
Seleccione una opción:
1. Generar llaves RSA
2. Iniciar el servidor
1
Llaves RSA generadas y guardadas.
```

2. Volver a correr el servidor, pero esta vez seleccionar la opción 2 correspondiente a iniciar el servidor y despues seleccionar el modo de operación que desea probar.

```
Seleccione una opción:
1. Generar llaves RSA
2. Iniciar el servidor
2
Llaves RSA leídas desde archivos.
Seleccione el modo de operación:
 1. Iterativo
 2. Concurrente
1
Servidor iterativo iniciado en el puerto 12345
```

3. Correr el cliente mientras el servidor aún está en operación y seleccionar el modo de operación.

```
Seleccione el modo de operación:
 1. Iterativo
 2. Concurrente
1
```

4. Mirar los resultados del lado de cliente

```
Estado del paquete: ENCLASIFICACION
Estado del paquete: ENENTREGA
Estado del paquete: ENTREGADO
Estado del paquete: ENENTREGA
Estado del paquete: ENTREGADO
Estado del paquete: ENTREGADO
Estado del paquete: ENCLASIFICACION
Estado del paquete: ENCLASIFICACION
Estado del paquete: ENTREGADO
```

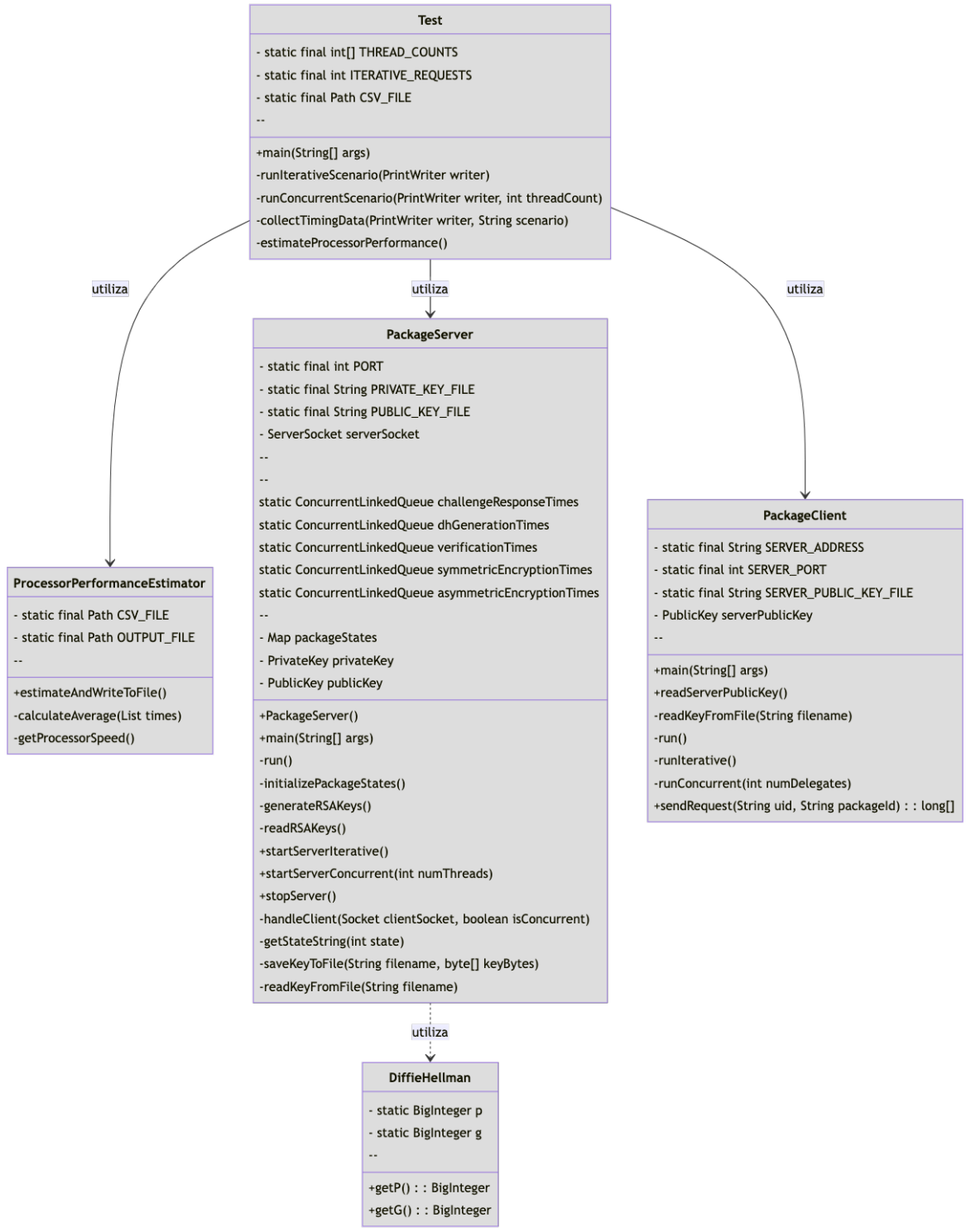
5. Ahora si nos devolvemos a la terminal correspondiente al servidor podremos ver lo datos requeridos

```
Servidor iterativo iniciado en el puerto 12345
Tiempo para descifrar el reto: 44899738 ns
Tiempo para generar G, P, G^x: 4418385 ns
Tiempo para verificar la consulta: 4207821 ns
Tiempo para cifrar el estado con cifrado asimétrico: 366669 ns
Tiempo para cifrar el estado con cifrado simétrico: 20900 ns
Tiempo para descifrar el reto: 1573324 ns
Tiempo para generar G, P, G^x: 5475654 ns
Tiempo para verificar la consulta: 17671 ns
Tiempo para cifrar el estado con cifrado asimétrico: 399918 ns
Tiempo para cifrar el estado con cifrado simétrico: 24288 ns
Tiempo para descifrar el reto: 2394807 ns
Tiempo para generar G, P, G^x: 5385604 ns
Tiempo para verificar la consulta: 22299 ns
Tiempo para cifrar el estado con cifrado asimétrico: 410324 ns
Tiempo para cifrar el estado con cifrado simétrico: 21337 ns
```

Nota: en el caso de los concurrentes también se verá junto con los cálculos el p y g usado, esto con el propósito de que sea más fácil verificar que se están usando diferentes.

6. Ahora que vio cómo funciona individualmente, pruebe a usar la clase Test que simula estas conexiones para los casos descritos y vea como se mira en el CSV y las gráficas generadas.

7. Diseño (Diagrama de clases)



8. **Corra su programa en diferentes escenarios y mida el tiempo que el servidor requiere para hacer lo siguiente:**

   1. Responder el reto
   2. Generar G, P y Gx
   3. Verificar la consulta

   Los escenarios son:
   - (i) Un servidor iterativo y un cliente iterativo. El cliente genera 32 consultas.
   - (ii) Un servidor y un cliente que implementen delegados. El número de delegados, tanto servidores como clientes, debe variar entre 4, 8, y 32 delegados concurrentes. Cada cliente genera una sola solicitud

   Para correr los escenarios diseñamos la clase Test.java en la que se corren todos a la vez.

9. **Construya una tabla con los datos recopilados.**

   Esto lo hicimos en un archivo test en el que generamos los resultados en CSV.

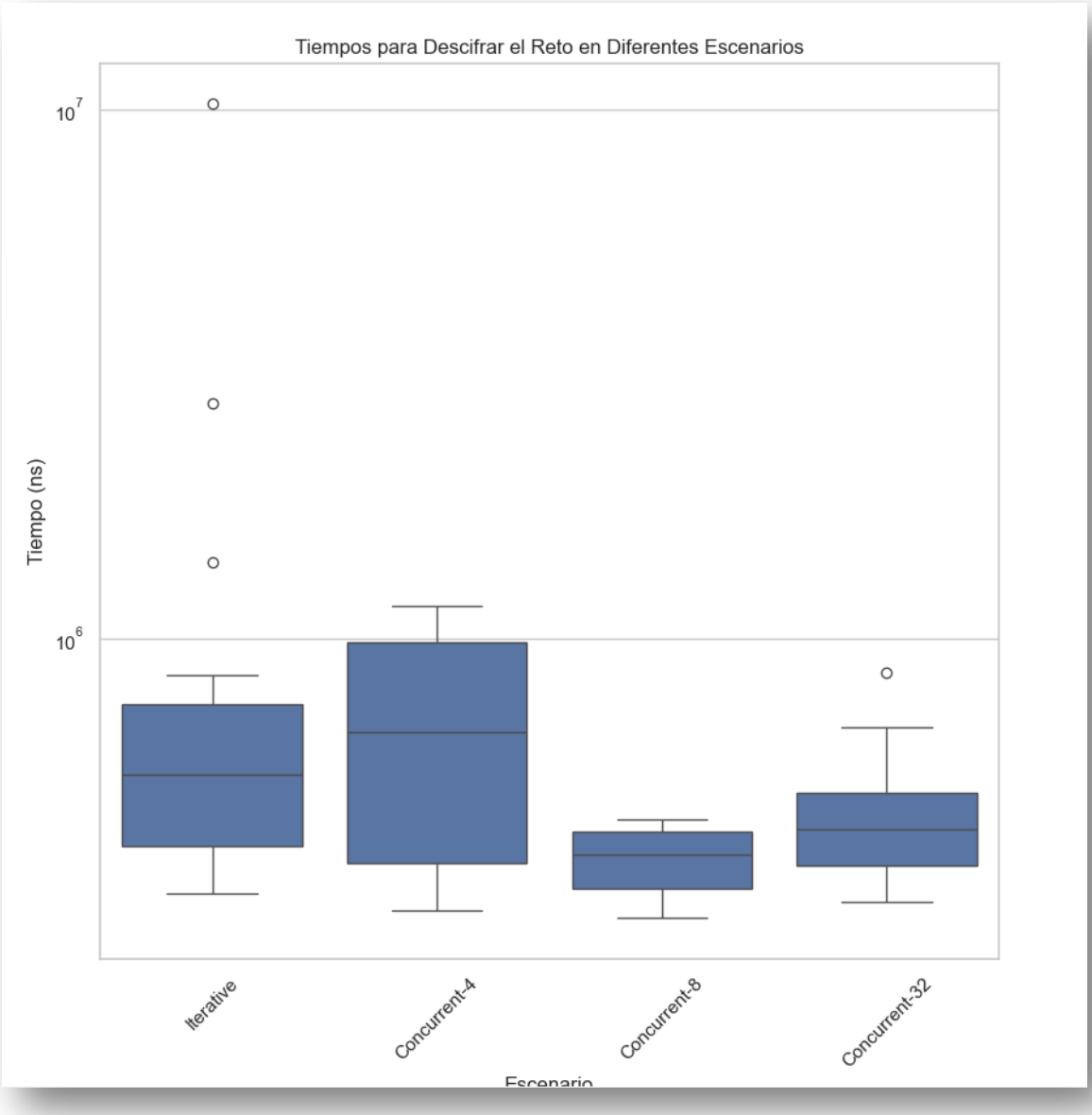
Escenario	Operación	Tiempo(ns)
Iterative	TiempoDeRespuestaDesafío	10216600
Iterative	TiempoDeRespuestaDesafío	2781400
Iterative	TiempoDeRespuestaDesafío	665600
Iterative	TiempoDeRespuestaDesafío	755600
Iterative	TiempoDeRespuestaDesafío	852400
Iterative	TiempoDeRespuestaDesafío	496300
Iterative	TiempoDeRespuestaDesafío	591500

10. **Mida el tiempo que el servidor requiere para cifrar el estado del paquete con cifrado simétrico y con cifrado asimétrico.**

    Para esto implementamos dentro de los métodos del servidor la toma del tiempo.

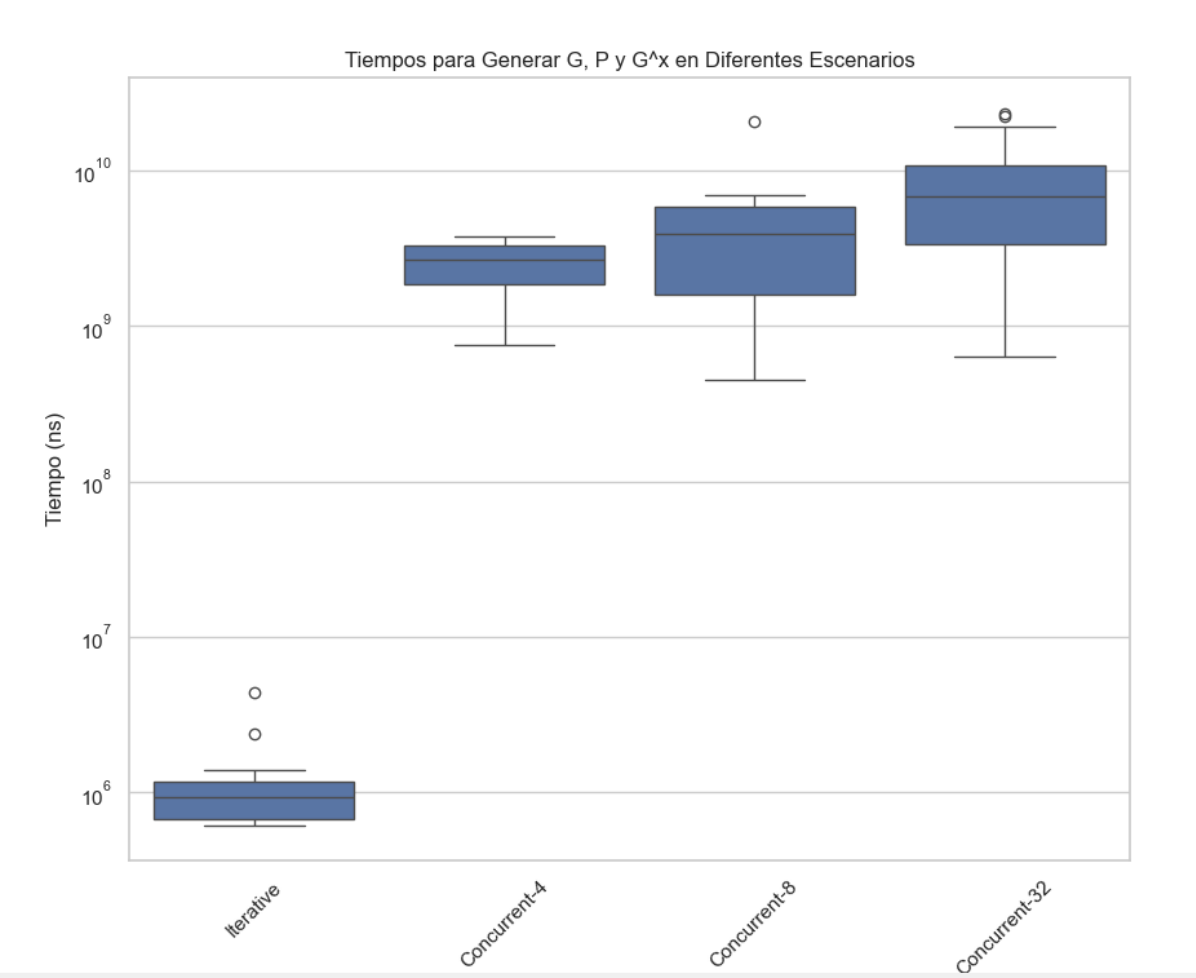
11. **Construya las siguientes gráficas:**

(i) Una que compare los tiempos para descifrar el reto en los diferentes escenarios



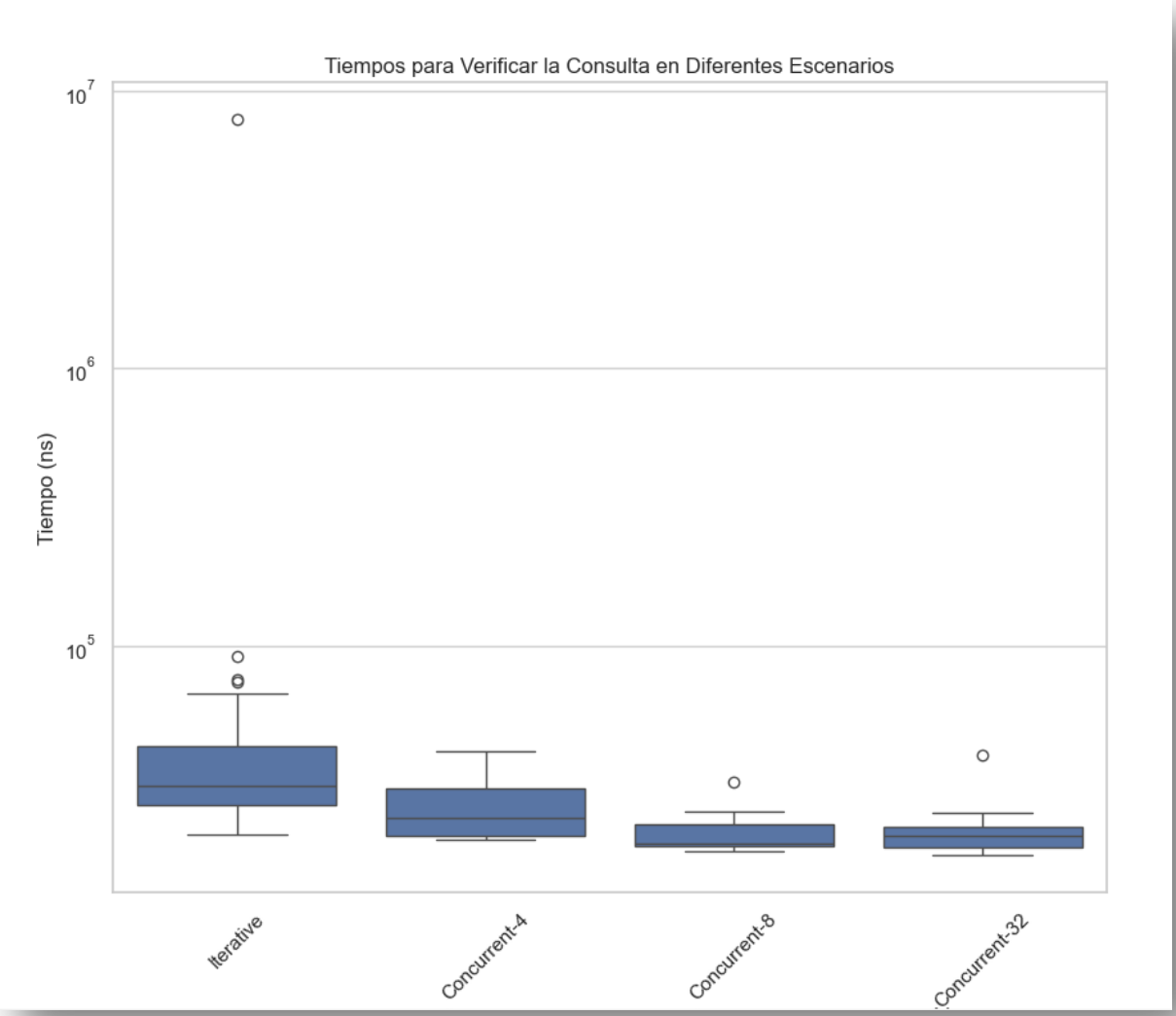
En la gráfica, se observa los tiempos para descifrar el reto en diferentes escenarios, en estos podemos destacar especialmente los escenarios en "Concurrent-8" y "Concurrent-32", ya que son consistentemente menores que en el escenario "Iterative". Aunque en teoría el escenario iterativo debería mostrar tiempos más bajos, en este caso los modos concurrentes parecen aprovechar la capacidad de procesamiento paralelo del ordenador. Esto permite que múltiples descifrados se ejecuten en paralelo, reduciendo la espera secuencial y el tiempo total de procesamiento. Así, en un sistema con suficiente capacidad, la concurrencia puede superar la eficiencia del procesamiento iterativo al distribuir la carga entre varios núcleos.

(ii) Una que compare los tiempos para generar G, P y Gx en los diferentes escenarios



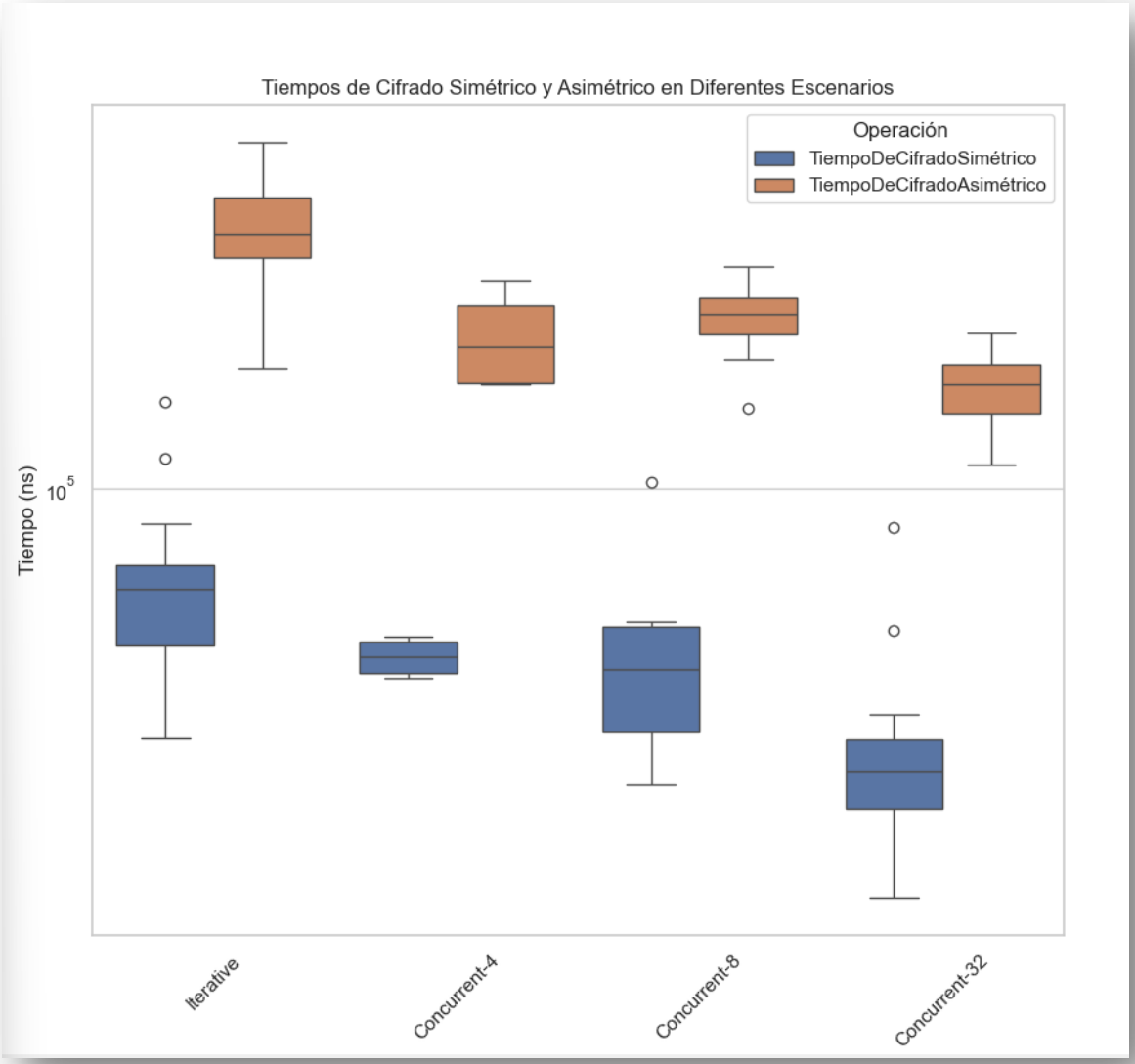
En la gráfica, se observa que la generación de p y g es considerablemente más alta en los tres escenarios concurrentes. Esto se debe a que, a diferencia del escenario iterativo, estos deben generar nuevos valores de p y g para cada conexión, lo cual implica un alto consumo de recursos. Este consumo provoca que los diferentes hilos compitan por el uso de la memoria, derivando en tiempos más altos. Como resultado, el escenario más lento es "Concurrent-32", seguido por "Concurrent-8" y, finalmente, "Concurrent-4".

(iii) Una que muestre los tiempos para verificar la consulta en los diferentes escenarios



En la gráfica, se observa que los tiempos para verificar la consulta son mayores en el escenario "Iterative" en comparación con los escenarios concurrentes, especialmente en "Concurrent-8" y "Concurrent-32", que presentan tiempos de verificación más bajos y consistentes. Esto sugiere que, en los escenarios concurrentes, la distribución de la carga de trabajo entre múltiples hilos permite un procesamiento más eficiente, reduciendo el tiempo promedio de verificación por consulta. Además, los tiempos en "Concurrent-4" son ligeramente superiores a los de "Concurrent-8" y "Concurrent-32", lo cual podría indicar que, a medida que se incrementa el nivel de concurrencia, el sistema gestiona mejor la verificación de consultas hasta cierto punto.

(iv) Una que muestre los tiempos para el caso simétrico y el caso asimétrico en los diferentes escenarios



En la gráfica se observa que los tiempos de cifrado asimétrico son considerablemente más altos en comparación con los tiempos de cifrado simétrico en todos los escenarios, lo cual es consistente con la naturaleza de ambos tipos de cifrado. El cifrado asimétrico, que generalmente es más intensivo en términos de recursos a cambio de proporcionar mayor seguridad, por lo que muestra tiempos más elevados, especialmente en el escenario "Iterative".

En los escenarios concurrentes, el tiempo para el cifrado simétrico disminuye a medida que aumenta el número de hilos, siendo "Concurrent-32" el más rápido, lo cual indica que el procesamiento paralelo beneficia este tipo de operación. El cifrado asimétrico también se optimiza en los escenarios concurrentes, aunque la reducción de tiempo no es tan significativa como en el cifrado simétrico. Esto puede deberse a la carga computacional inherentemente mayor del cifrado asimétrico, que limita la eficiencia de la concurrencia.

12. Escriba sus comentarios sobre las gráficas, explicando los comportamientos observados.

En general, las gráficas muestran que la concurrencia mejora los tiempos de procesamiento en ciertas operaciones, especialmente en aquellas menos intensivas en recursos, como el cifrado simétrico y la verificación de consultas. Sin embargo, en operaciones de alta demanda, como la generación de parámetros y el cifrado asimétrico, la concurrencia elevada puede generar competencia por recursos y aumentar los tiempos de ejecución. Esto sugiere que la concurrencia es beneficiosa solo hasta cierto punto, donde el sistema aún puede manejar eficientemente el paralelismo sin que los hilos compitan excesivamente.

13. Identifique la velocidad de su procesador, y estime cuántas operaciones de cifrado puede realizar su máquina por segundo, en el caso evaluado de cifrado simétrico y cifrado asimétrico. Escriba todos sus cálculos.

Antes de mostrar los cálculos finales se debe explicar los cálculos para estimar la cantidad de operaciones de cifrado por segundo que una máquina puede realizar. Este cálculo se basa en los tiempos promedio de cifrado simétrico y asimétrico medidos en nanosegundos. Los pasos incluyen el cálculo del tiempo promedio, la conversión de unidades y la estimación de operaciones por segundo.

Cálculo del Tiempo Promedio

Para cada tipo de operación (cifrado simétrico y cifrado asimétrico), primero calculamos el tiempo promedio en nanosegundos (ns) sumando todos los tiempos individuales de las operaciones y dividiendo por la cantidad de operaciones:

Tiempo Promedio (ns) = $\frac{\sum_{i=1}^{n} Ti}{n}$ (1)

Donde:

- ti es el tiempo individual de la operación i en nanosegundos.
- n es el número total de operaciones.

Conversión de Nanosegundos a Segundos

Para obtener el tiempo promedio en segundos, convertimos el resultado dividiendo entre 10^9, ya que 1 segundo = 10^9 ns:

Tiempo Promedio (s) = $\frac{Tiempo\ \mathrm{Pr}omedio\,(ns)}{10^9}$ (2)

Estimación de Operaciones por Segundo

Una vez que tenemos el tiempo promedio en segundos para una operación, podemos estimar el número de operaciones de cifrado que la máquina puede realizar por segundo. Esto se calcula tomando el inverso del tiempo promedio en segundos:

Operación por Segundo = $\frac{1}{Tiempo\ \Pr omedio\ (s)}$ (3)

Aplicación al Cifrado Simétrico y Asimétrico

Aplicamos los pasos anteriores tanto para el cifrado simétrico como para el cifrado asimétrico:

- ***Cifrado Simétrico:***

  Tiempo Promedio Simétrico (ns) = $\frac{\sum_{I=1}^{n} Ts,i}{n}$ (4)
  Operaciones Simétricas por Segundo = $\frac{10^9}{Tiempo\ \Pr omedio\ Simétrico\ (ns)}$ (5)

- ***Cifrado Asimétrico:***

  Tiempo Promedio Asimétrico (ns) = $\frac{\sum_{i=1}^{m} Ta,i}{m}$ (6)
  Operaciones Asimétricas por Segundo = $\frac{10^9}{Tiempo\ \Pr omedio\ Asimétrico\ (ns)}$ (7)

donde:
• $t_{s,i}$ es el tiempo individual de una operación de cifrado simétrico i.
• $t_{a,i}$ es el tiempo individual de una operación de cifrado asimétrico i.
• n y m son las cantidades de muestras para el cifrado simétrico y asimétrico, respectivamente.

Lo cálculos respectivos según para la maquina son:

Estimación de Rendimiento del Procesador
==========
Velocidad del Procesador: 3 GHz

Cálculos de Cifrado Simétrico:

Tiempo Total de Cifrado Simétrico (ns): 44032,89
Tiempo Promedio de Cifrado Simétrico (s): 0,000044032895
Operaciones de Cifrado Simétrico por Segundo: 22710,29

Cálculos de Cifrado Asimétrico:

Tiempo Total de Cifrado Asimétrico (ns): 258907,89
Tiempo Promedio de Cifrado Asimétrico (s): 0,000258907895
Operaciones de Cifrado Asimétrico por Segundo: 3862,38

Explicación de los Cálculos:

---

1. Calculamos el tiempo promedio en nanosegundos sumando todos los tiempos individuales de las operaciones y dividiéndolo entre la cantidad de operaciones.
2. Convertimos el tiempo promedio de nanosegundos a segundos dividiendo por 1,000,000,000 para obtener el tiempo en segundos.
3. Estimamos las operaciones por segundo tomando el inverso del tiempo promedio en segundos (1 / tiempo_promedio_segundos).

Código utilizado:

Adjuntamos el código de la clase en la que calculamos esto:

```
public class ProcessorPerformanceEstimator {
    private static final Path CSV_FILE = Paths.get("Auxiliar",
"test_results.csv");
    private static final Path OUTPUT_FILE = Paths.get("Auxiliar",
"processor_performance.txt");

    public static void estimateAndWriteToFile() {
        try {
            // Leer los datos de tiempos desde test_results.csv
            Map<String, List<Long>> dataMap = new HashMap<>();
            BufferedReader reader = new BufferedReader(new
FileReader(CSV_FILE.toFile()));
            String line;
            reader.readLine(); // Saltar la cabecera

            while ((line = reader.readLine()) != null) {
                String[] parts = line.split(",");
                String operation = parts[1];
                long time = Long.parseLong(parts[2]);

                dataMap.computeIfAbsent(operation, k -> new
ArrayList<>()).add(time);
            }
            reader.close();

            // Calcular los tiempos promedio para cifrado simétrico y asimétrico
            double avgSymmetricTimeNs =
calculateAverage(dataMap.get("TiempoDeCifradoSimétrico"));
            double avgAsymmetricTimeNs =
calculateAverage(dataMap.get("TiempoDeCifradoAsimétrico"));

            // Convertir de nanosegundos a segundos para los cálculos de
operaciones por segundo
            double avgSymmetricTimeSec = avgSymmetricTimeNs / 1_000_000_000.0;
            double avgAsymmetricTimeSec = avgAsymmetricTimeNs / 1_000_000_000.0;

            // Estimar operaciones por segundo
            double symmetricOpsPerSec = 1 / avgSymmetricTimeSec;
            double asymmetricOpsPerSec = 1 / avgAsymmetricTimeSec;

            // Obtener velocidad del procesador por input del usuario
            String processorSpeed = getProcessorSpeed();
```