

Informe – Caso 3

Infraestructura Computacional

Repositorio del caso: https://github.com/sofiaalcala/Caso3_Infracomp.git

Descripción de la organización del proyecto

CryptoUtils

Este archivo incluye métodos para verificar firmas digitales con RSA (verificarFirma), generar claves Diffie-Hellman (generarClavesDH), derivar claves de sesión simétricas para AES y HMAC (generarClavesSesion), y realizar operaciones de cifrado/descifrado con AES (cifrarAES, descifrarAES) y RSA (cifrarRSA, descifrarRSA). También ofrece funciones para generar y verificar HMACs (generarHMAC, verificarHMAC), firmar datos con RSA (firmarRSA), crear vectores de inicialización aleatorios (generarIV), y serializar objetos (serializarObjeto). Este archivo es el núcleo de la seguridad, asegurando que todas las comunicaciones estén cifradas y autenticadas según el protocolo especificado, utilizando AES-256 en modo CBC con PKCS5Padding, RSA-1024, SHA256withRSA para firmas, y HMACSHA256 para integridad.

InfoServicio

Esta clase define una estructura de datos serializable que representa la información de un servicio de la aerolínea, con atributos para el nombre del servicio, dirección IP y puerto. Su propósito es encapsular la información que el servidor envía al cliente tras una consulta, permitiendo su transmisión segura a través de la red. Al ser serializable, facilita el envío de objetos entre el cliente y el servidor, resolviendo la necesidad de compartir datos estructurados de manera eficiente.

ServidorPrincipal

El archivo implementa el servidor principal, que actúa como el punto de entrada para las conexiones de los clientes. Inicializa una tabla de servicios predefinida (inicializarTablaServicios) con identificadores y detalles de servicios como "Estado vuelo" o "Disponibilidad vuelos". Gestiona las claves RSA del servidor, cargándolas desde archivos o generándolas si no existen (cargarClaves, generarClaves), y escucha conexiones en un puerto específico usando sockets (iniciar). Por cada cliente conectado, crea un thread ServidorDelegado para manejar la comunicación de forma concurrente, cumpliendo con el requisito de delegados por conexión. Además, mide

tiempos de operaciones criptográficas (firma, cifrado de tabla, verificación de consultas) y presenta estadísticas al cerrar el servidor, abordando la tarea de evaluación de rendimiento.

ServidorDelegado

Este archivo define un thread que maneja la comunicación segura con un cliente específico. Establece claves de sesión usando Diffie-Hellman (establecerClavesSeguras), generando parámetros DH (p , g , l) y firmándolos con RSA para autenticación. Envía la tabla de servicios cifrada con AES y protegida con HMAC (enviarTablaServicios), y procesa la consulta del cliente (procesarConsulta), verificando su integridad con HMAC y respondiendo con la información del servicio solicitada, también cifrada y autenticada. Este diseño resuelve la necesidad de comunicaciones seguras y concurrentes, delegando tareas específicas por cliente y midiendo tiempos para análisis de rendimiento.

ClienteManager

Este archivo gestiona la ejecución de clientes en diferentes escenarios de prueba. Permite ejecutar un cliente único con múltiples consultas secuenciales (ejecutarClienteUnico) o múltiples clientes concurrentes (ejecutarClientesConcurrentes), cada uno con una sola consulta. Recopila estadísticas de tiempos de cifrado simétrico y asimétrico, mostrando promedios y relaciones entre ambos. Soluciona los requerimientos de evaluación de rendimiento en escenarios como 32 consultas secuenciales o 4-64 clientes concurrentes, facilitando la comparación de algoritmos criptográficos.

ClienteMain

Actúa como el punto de entrada para lanzar los clientes, configurando el host, puerto y clave pública del servidor, y utilizando ClienteManager para ejecutar los escenarios de prueba. Aunque su funcionalidad es simple, es esencial para iniciar el sistema del lado del cliente, permitiendo flexibilidad en la configuración de pruebas (un cliente iterativo o múltiples clientes concurrentes).

ClienteThread

Este archivo define un thread que representa un cliente concurrente en escenarios multi-cliente. Cada instancia se conecta al servidor, realiza una consulta y mide tiempos

Sofía Alcalá - 202321623

Juan Felipe Ochoa – 202320053

de cifrado, permitiendo simular cargas concurrentes. Resuelve la necesidad de evaluar el sistema bajo concurrencia, integrándose con ClienteManager para recopilar datos de rendimiento en escenarios como 64 clientes simultáneos.

Cliente

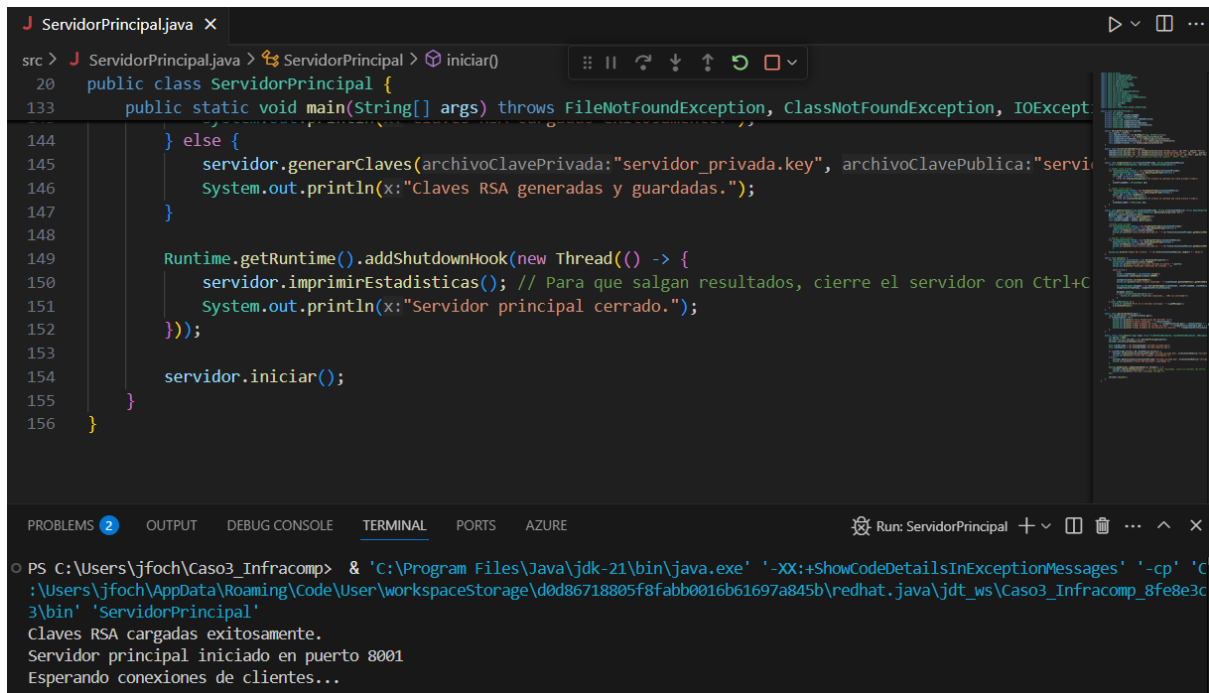
Implementa la lógica principal del cliente, estableciendo una conexión con el servidor (conectar), negociando claves seguras con Diffie-Hellman (establecerClavesSeguras), recibiendo y mostrando la tabla de servicios (recibirTablaServicios, mostrarServicios), seleccionando un servicio aleatoriamente (seleccionarServicioAleatorio), enviando la consulta (enviarConsulta), y procesando la respuesta (recibirRespuesta, mostrarResultado). Verifica la integridad de los datos con HMAC y mide tiempos de cifrado simétrico y asimétrico (medirTiempoCifradoAsimetrico), cumpliendo con los requisitos de comunicación segura, selección aleatoria de servicios, y comparación de algoritmos criptográficos.

El proyecto implementa un sistema bien estructurado que permite a los clientes de una aerolínea consultar información sobre vuelos de manera sencilla. Integra mecanismos sólidos para proteger los datos y garantizar su integridad, asegurando que solo usuarios autorizados puedan acceder a ellos. La solución está organizada en componentes claros y coordinados, lo que facilita su mantenimiento, refuerzo y escalabilidad en caso de ser necesario. Además, el sistema está optimizado para soportar múltiples usuarios de forma simultánea sin presentar bloqueos, y las pruebas realizadas confirman su rapidez y eficiencia.

Como correr el Proyecto

Sofía Alcalá - 202321623

Juan Felipe Ochoa – 202320053



The image shows a screenshot of an IDE with a dark theme. The top part displays the code for `ServidorPrincipal.java`. The code includes a `main` method that generates RSA keys, prints a message, and starts a server. The bottom part shows the terminal output, which confirms the successful execution of the program.

```
src > J ServidorPrincipal.java > ServidorPrincipal > iniciar()
20 public class ServidorPrincipal {
133 public static void main(String[] args) throws FileNotFoundException, ClassNotFoundException, IOException
144     } else {
145         servidor.generarClaves(archivoClavePrivada:"servidor_privada.key", archivoClavePublica:"servi
146         System.out.println(x:"Claves RSA generadas y guardadas.");
147     }
148
149     Runtime.getRuntime().addShutdownHook(new Thread(() -> {
150         servidor.imprimirEstadisticas(); // Para que salgan resultados, cierre el servidor con Ctrl+C
151         System.out.println(x:"Servidor principal cerrado.");
152     }));
153
154     servidor.iniciar();
155 }
156 }
```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS AZURE

Run: ServidorPrincipal

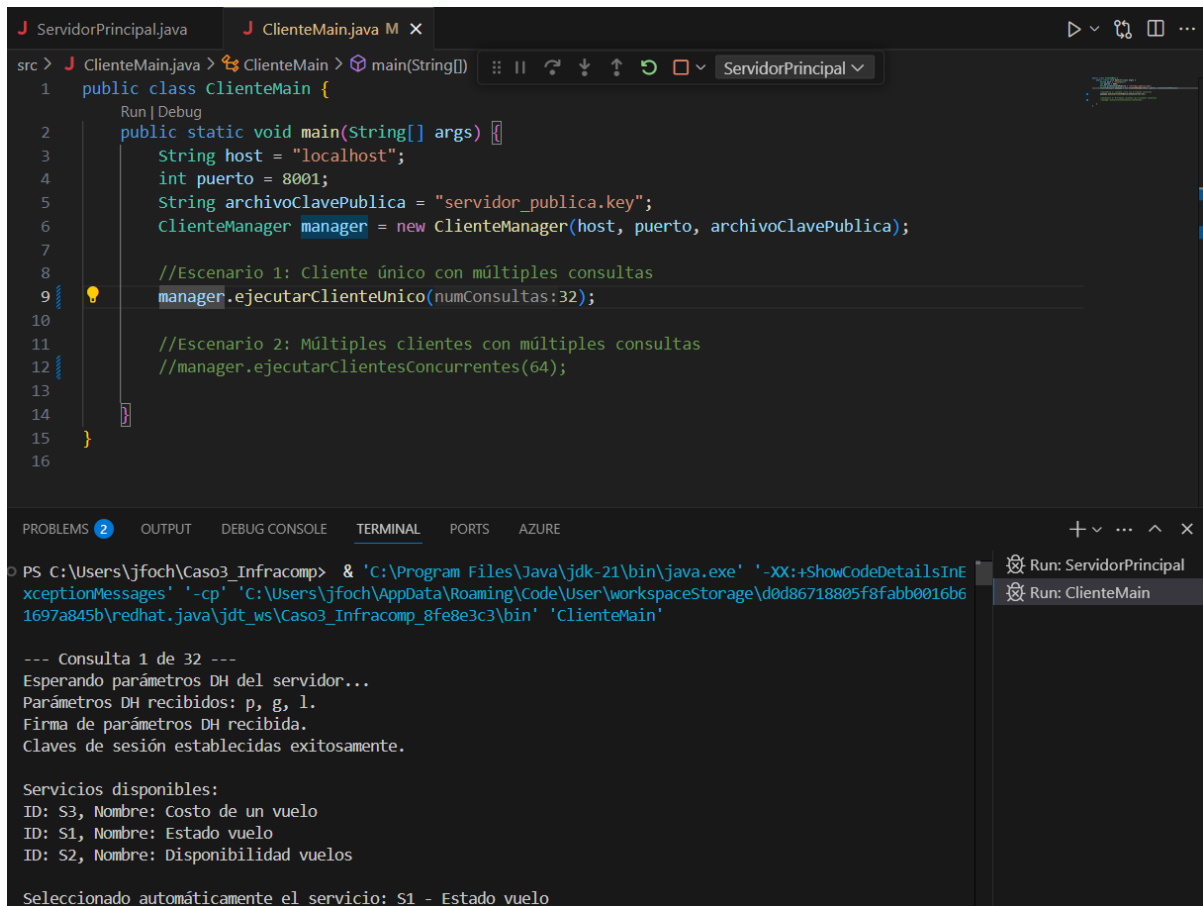
```
PS C:\Users\jfoch\Caso3_Infracomp> & 'C:\Program Files\Java\jdk-21\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'C
:\Users\jfoch\AppData\Roaming\Code\User\workspaceStorage\d0d86718805f8fabb0016b61697a845b\redhat.java\jdt_ws\Caso3_Infracomp_8fe8e3c
3\bin' 'ServidorPrincipal'
Claves RSA cargadas exitosamente.
Servidor principal iniciado en puerto 8001
Esperando conexiones de clientes...
```

Imagen 1

Para correr el servidor, necesitamos correr la clase `ServidorPrincipal.java`. Al correrlo debería salir en consola el mensaje que aparece en la anterior imagen.

Sofía Alcalá - 202321623

Juan Felipe Ochoa – 202320053



```
src > J ClienteMain.java > ClienteMain > main(String[])
1 public class ClienteMain {
2     public static void main(String[] args) {
3         String host = "localhost";
4         int puerto = 8001;
5         String archivoClavePublica = "servidor_publica.key";
6         ClienteManager manager = new ClienteManager(host, puerto, archivoClavePublica);
7
8         //Escenario 1: Cliente único con múltiples consultas
9         manager.ejecutarClienteUnico(numConsultas:32);
10
11         //Escenario 2: Múltiples clientes con múltiples consultas
12         //manager.ejecutarClientesConcurrentes(64);
13
14     }
15 }
16
```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS AZURE

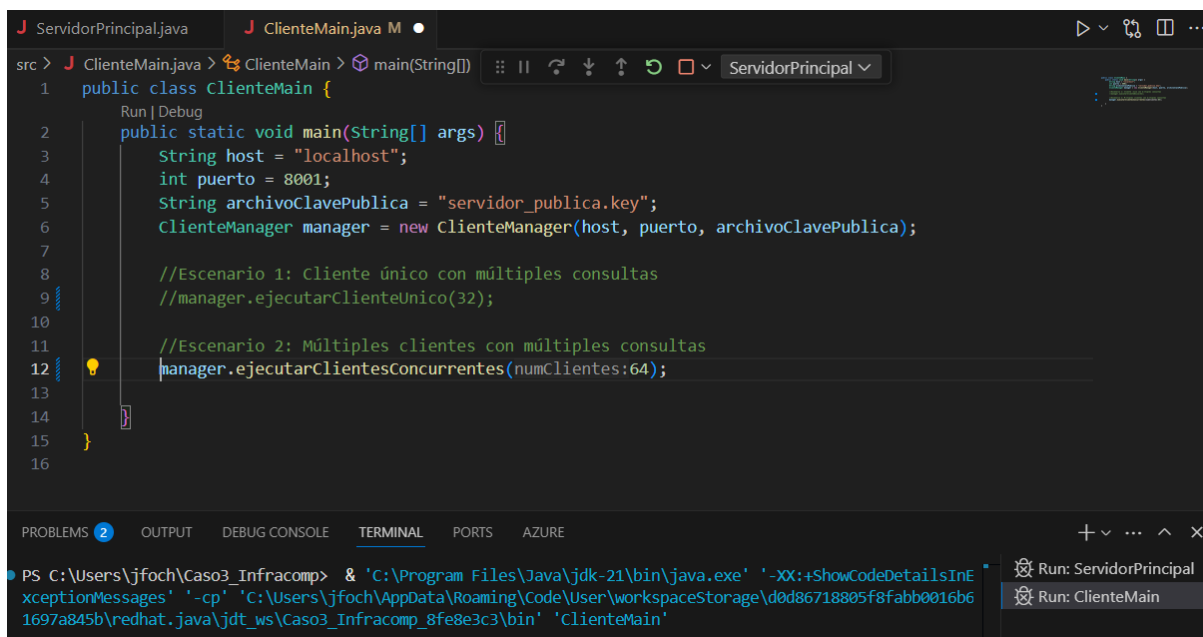
```
PS C:\Users\jfoch\Caso3_Infracomp> & 'C:\Program Files\Java\jdk-21\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\jfoch\AppData\Roaming\Code\User\workspaceStorage\d0d86718805f8fabb0016b61697a845b\redhat.java\jdt_ws\Caso3_Infracomp_8fe8e3c3\bin' 'ClienteMain'
```

--- Consulta 1 de 32 ---
Esperando parámetros DH del servidor...
Parámetros DH recibidos: p, g, l.
Firma de parámetros DH recibida.
Claves de sesión establecidas exitosamente.

Servicios disponibles:
ID: S3, Nombre: Costo de un vuelo
ID: S1, Nombre: Estado vuelo
ID: S2, Nombre: Disponibilidad vuelos

Seleccionado automáticamente el servicio: S1 - Estado vuelo

Imagen 2



```
src > J ClienteMain.java > ClienteMain > main(String[])
1 public class ClienteMain {
2     public static void main(String[] args) {
3         String host = "localhost";
4         int puerto = 8001;
5         String archivoClavePublica = "servidor_publica.key";
6         ClienteManager manager = new ClienteManager(host, puerto, archivoClavePublica);
7
8         //Escenario 1: Cliente único con múltiples consultas
9         //manager.ejecutarClienteUnico(32);
10
11         //Escenario 2: Múltiples clientes con múltiples consultas
12         manager.ejecutarClientesConcurrentes(numClientes:64);
13
14     }
15 }
16
```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS AZURE

```
PS C:\Users\jfoch\Caso3_Infracomp> & 'C:\Program Files\Java\jdk-21\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\jfoch\AppData\Roaming\Code\User\workspaceStorage\d0d86718805f8fabb0016b61697a845b\redhat.java\jdt_ws\Caso3_Infracomp_8fe8e3c3\bin' 'ClienteMain'
```

imagen 3

Después de haber corrido el ServidorPrincipal, se debe correr la clase ClienteMain (Sin dejar de correr el servidorPrincipal). En la imagen 2, se evidencia como correr las 32

Sofía Alcalá - 202321623

Juan Felipe Ochoa – 202320053

consultas secuenciales. En la imagen 3, se evidencia como correr los delegados concurrentes. Para configurar el número de clientes concurrentes solamente se debe cambiar el parámetro de la función ejecutarClientesConcurrentes(#), poniendo el número de clientes concurrentes que se requiere.

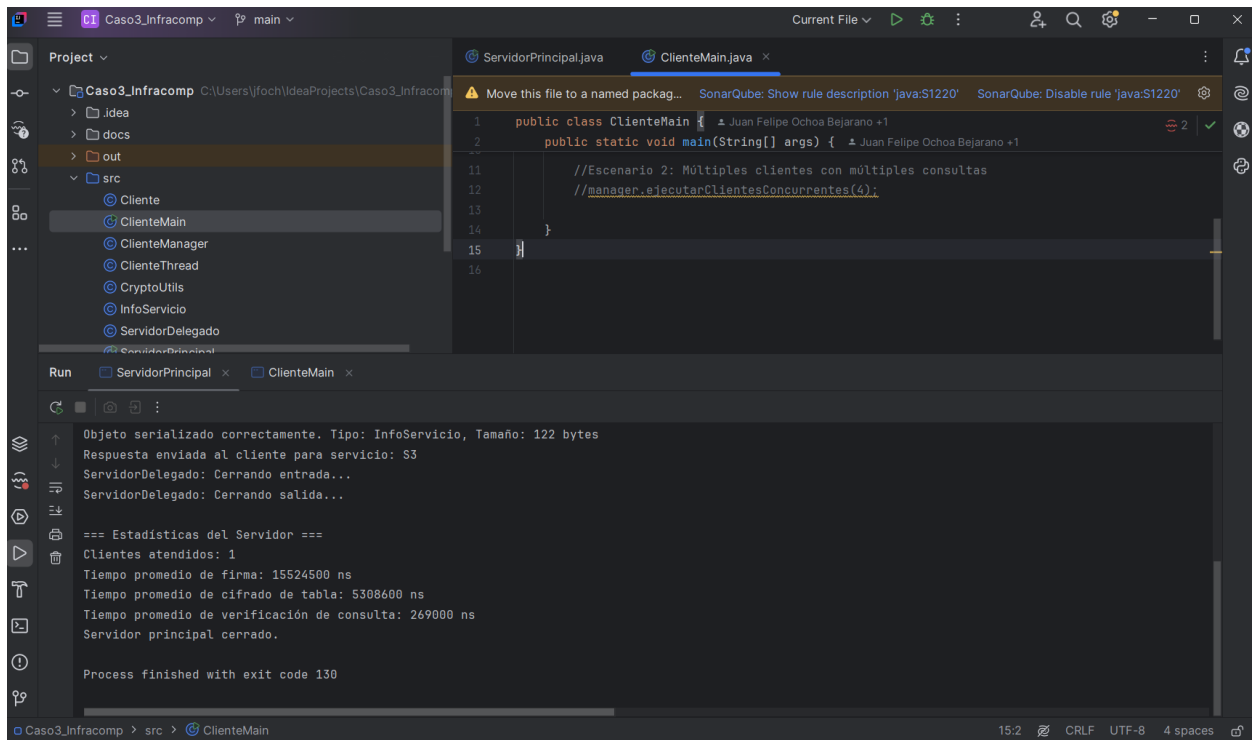


Imagen 4

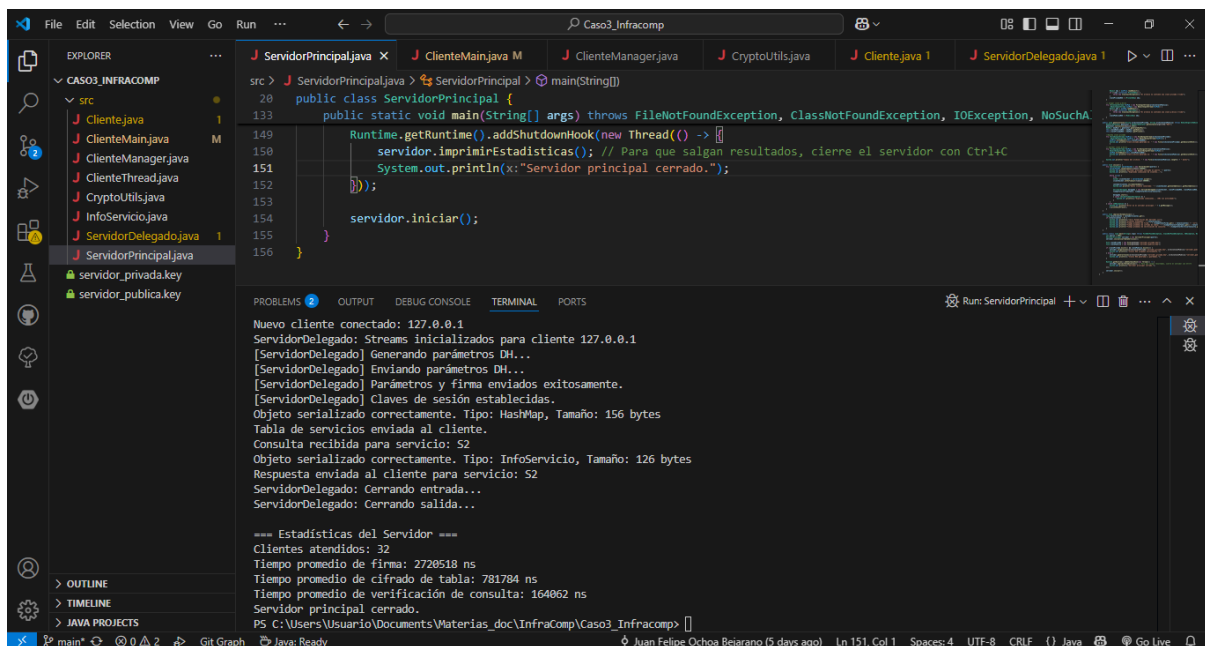


Imagen 5

Sofía Alcalá - 202321623

Juan Felipe Ochoa – 202320053

Al finalizar cada escenario, se debe parar el servidor para que aparezcan las estadísticas finales. En caso de usar Visual Studio Code, se debe parar el servidorPrincipal con el comando “Control + C”. Las estadísticas deberían aparecer como en la imagen 4 o en la imagen 5.

Tareas y preguntas planteadas

Datos recopilados

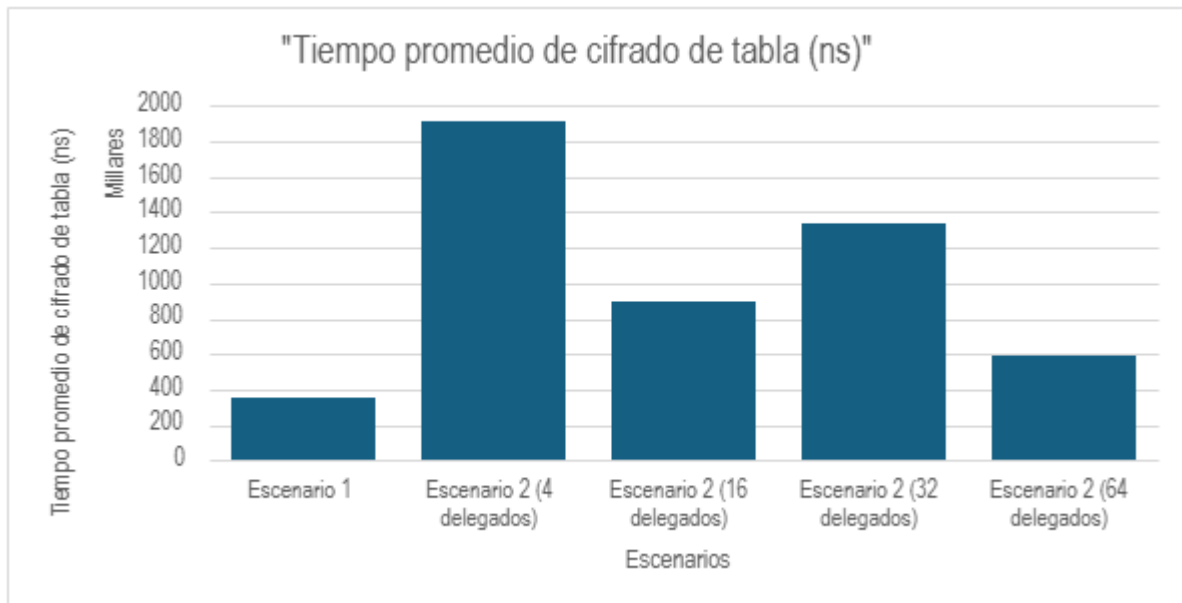
	Escenario 1	Escenario 2 (4 delegados)	Escenario 2 (16 delegados)	Escenario 2 (32 delegados)	Escenario 2 (64 delegados)
Tiempo promedio de firma (ns)	779418	4839825	3487450	15002759	75406005
Tiempo promedio de cifrado de tabla (ns)	353662	1915075	901981	1340728	595209
Tiempo promedio de verificación de consulta (ns)	110393	263550	137831	227740	281992
Tiempo total cifrado simétrico (ns)	217500	418025	293318	325750	419578
Tiempo total cifrado asimétrico (ns)	294309	1048050	459606	412731	393040
Relación promedio RSA/AES	1.35	2.51	1.57	1.28	0.94

Gráficas



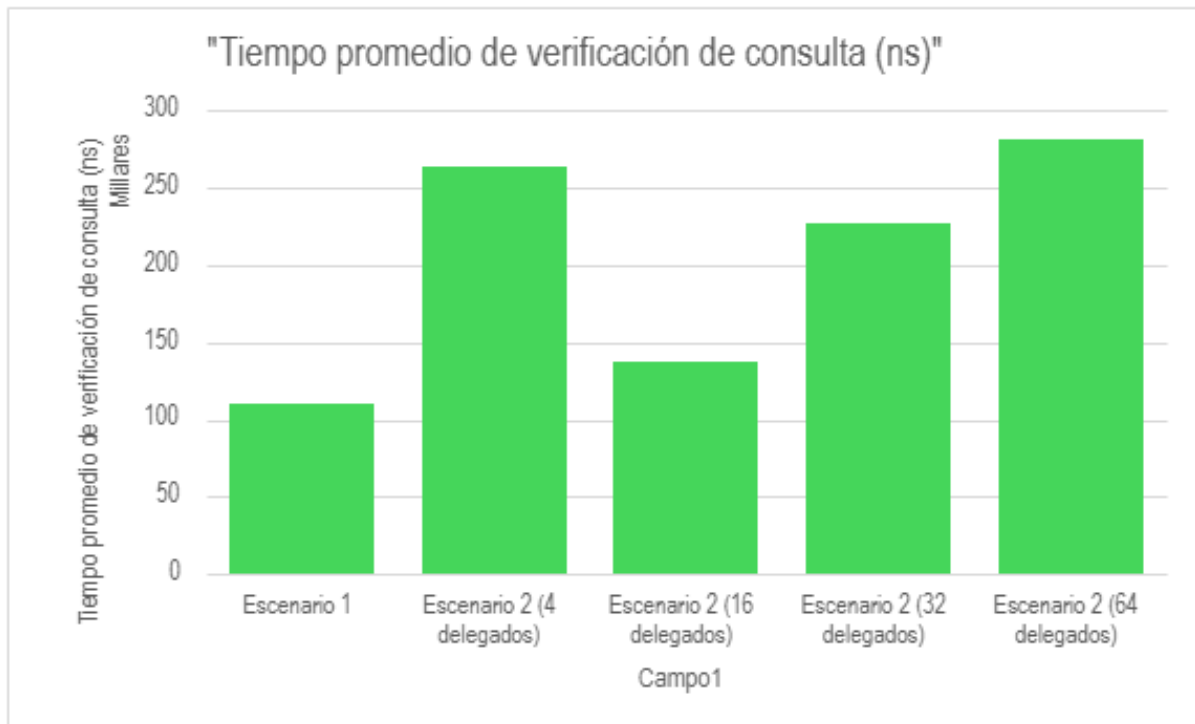
Tiempo promedio de firma

En el Escenario 1, donde se tiene un único servidor de consulta y un cliente iterativo, el tiempo promedio de firma es relativamente bajo, alrededor de 780 mil nanosegundos. Sin embargo, en el Escenario 2, cuando se introduce concurrencia con múltiples servidores delegados, el tiempo promedio de firma aumenta de forma significativa. Con 4 delegados, el tiempo sube a aproximadamente 4,83 millones, baja ligeramente con 16 delegados a 3,48 millones, pero incrementa de manera drástica a 15 millones con 32 delegados y alcanza hasta 75 millones con 64 delegados. Este comportamiento evidencia que, conforme aumenta el número de delegados concurrentes, el servidor sufre saturación, afectando de manera notable el desempeño de la operación de firma, que es computacionalmente costosa.



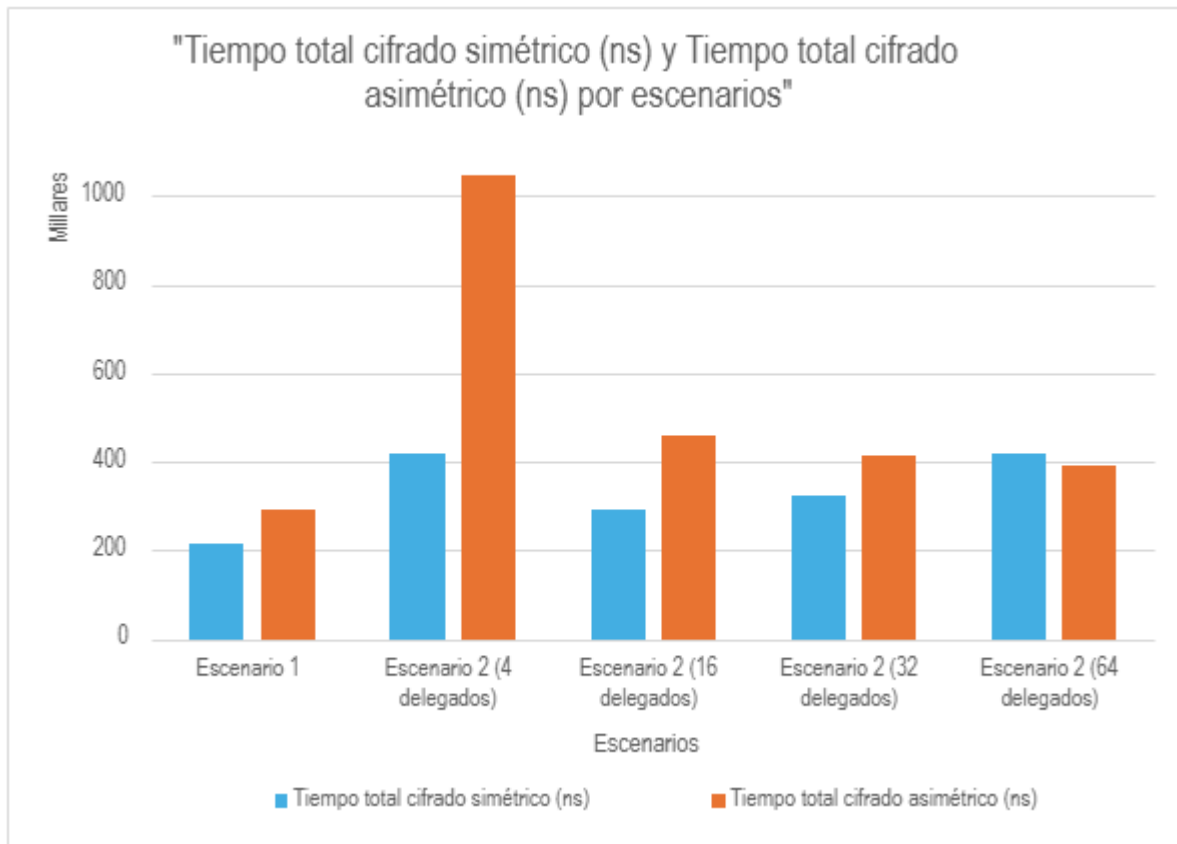
Tiempo promedio de cifrado de tabla

El tiempo de cifrado de la tabla muestra un comportamiento diferente. En el Escenario 1, es bajo, alrededor de 353 mil nanosegundos. En Escenario 2, se incrementa con 4 delegados hasta aproximadamente 1,91 millones, disminuye a 902 mil con 16 delegados, vuelve a subir a 1,34 millones con 32 delegados y baja nuevamente a 595 mil con 64 delegados. Esto refleja que, aunque el cifrado simétrico (AES) es más eficiente que el cifrado asimétrico, el efecto de la concurrencia varía según cómo se distribuya la carga entre los procesos. En escenarios de alta concurrencia (64 delegados), la distribución de carga parece haber sido más eficiente, logrando menores tiempos de cifrado.



Tiempo promedio de verificación de consulta

En el Escenario 1, el tiempo promedio de verificación de consultas es bajo, cercano a 110 mil nanosegundos. Al introducir concurrencia en el Escenario 2, este tiempo se incrementa: alcanza 263 mil nanosegundos con 4 delegados, disminuye a 137 mil con 16 delegados, y vuelve a aumentar a 227 mil y 281 mil para 32 y 64 delegados respectivamente. Esta variabilidad indica que, aunque la verificación de firmas es menos exigente que la firma en sí, el desempeño del sistema se ve afectado conforme la carga de trabajo y la cantidad de procesos concurrentes aumentan, mostrando ineficiencias a medida que el número de delegados es mayor.



Tiempo total de cifrado simétrico y asimétrico

Comparando los tiempos totales de cifrado, el cifrado asimétrico (RSA) es consistentemente más lento que el cifrado simétrico (AES). En el Escenario 1, la relación entre los tiempos RSA y AES es de 1,35, es decir, RSA toma un 35% más tiempo que AES. En Escenario 2 con 4 delegados, la relación sube a 2,51, indicando que RSA tarda más del doble que AES en esas condiciones. Posteriormente, con 16, 32 y 64 delegados, esta relación disminuye progresivamente a 1,57, 1,28 y finalmente a 0,94. Esta tendencia muestra que, en escenarios de alta concurrencia, la diferencia de tiempos entre RSA y AES se reduce, probablemente debido a la saturación de recursos del servidor que impacta a ambos tipos de cifrado de manera similar.

Estimación de velocidad del procesador

Escenario para estimación

Para estimar la velocidad de mi procesador en operaciones criptográficas, tomé los tiempos medidos en el experimento del proyecto, específicamente:

- El tiempo total de cifrado simétrico (AES)
- El tiempo total de cifrado asimétrico (RSA)

Estos tiempos se encuentran en la tabla anteriormente insertada. Para este cálculo se usa el Escenario 1, en el que un único cliente realiza 32 consultas secuenciales.

Datos utilizados del Escenario 1:

- Número de operaciones (consultas): 32
- Tiempo total cifrado simétrico (AES): 217,500 ns
- Tiempo total cifrado asimétrico (RSA): 294,309 ns

Cálculos

1. Tiempo promedio por operación

- AES promedio = $217,500 \text{ ns} / 32 = 6,797 \text{ ns}$
- RSA promedio = $294,309 \text{ ns} / 32 = 9,197 \text{ ns}$

2. Operaciones por segundo

1 segundo = 1,000,000,000 nanosegundos

AES operaciones/segundo = $1,000,000,000 / 6,797 \approx 147,077$

RSA operaciones/segundo = $1,000,000,000 / 9,197 \approx 108,716$

Resultados finales

Algoritmo	Tiempo promedio por operación	Operaciones por segundo
AES (simétrico)	6,797 ns	~147,077 ops/s
RSA (asimétrico)	9,197 ns	~108,716 ops/s

Comentarios

A pesar de que los tiempos de cifrado RSA suelen ser mucho más altos que AES, en este escenario particular con claves pequeñas (RSA -1024) y cargas livianas, la diferencia no fue tan marcada como en otros escenarios. Aun así, AES es más eficiente y por eso se

usa para cifrado de datos, mientras RSA queda reservado para el establecimiento de claves o firma digital.

Referencias

- OpenAI. (2023). ChatGPT (versión del 26 de abril) [Modelo de lenguaje de gran tamaño]. <https://chatgpt.com/share/680dc0bc-08d4-800d-916f-c3d4eb32e4a4>
- GeeksforGeeks. (2023, 6 mayo). Java Implementation of DiffieHellman Algorithm between Client and Server. GeeksforGeeks. <https://www.geeksforgeeks.org/java-implementation-of-diffie-hellman-algorithm-between-client-and-server/>
- TutorialsPoint. (2023, 17 abril). Client-Server Diffie-Hellman Algorithm Implementation in Java. <https://www.tutorialspoint.com/client-server-diffie-hellman-algorithm-implementation-in-java>
- Java Cryptography Architecture (JCA) Reference Guide. (s. f.). <https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html>
- Sharifi, H., & Martin, G. (2025, 26 marzo). HMAC in Java. Baeldung. <https://www.baeldung.com/java-hmac>
- Ray, D. (2024, 15 marzo). AES Encryption and Decryption in Java (CBC Mode). Java Code Geeks. <https://www.javacodegeeks.com/2018/03/aes-encryption-and-decryption-in-javacbc-mode.html>
- GeeksforGeeks. (2025, 6 enero). RSA Algorithm in Cryptography. GeeksforGeeks. <https://www.geeksforgeeks.org/rsa-algorithm-cryptography/>
- Kampschmidt, J. (s. f.). Examples of creating base64 hashes using HMAC SHA256 in different languages - Joe Kampschmidt's Code.

Sofía Alcalá - 202321623

Juan Felipe Ochoa – 202320053

<https://www.jokecamp.com/blog/examples-of-creating-base64-hashes-using-hmac-sha256-in-different-languages/>