

# **Universidad de los Andes**

Departamento de Ingeniería Sistemas y Computación

*ISIS 2203 - Infraestructura computacional*

## **CASO 3**

# Índice

## [1. Organización de los Archivos](#)

### [Estructura del archivo](#)

### [Descripción de carpetas y archivos:](#)

#### [Algoritmos de criptografía](#)

##### [AES](#)

##### [DH](#)

##### [DIGEST](#)

##### [HMAC](#)

##### [RSA](#)

#### [Código fuente del cliente](#)

#### [Código fuente del servidor](#)

#### [Llave pública y llave privada \(generador de llaves\)](#)

## [2. Instrucciones para Ejecutar el Sistema](#)

### [3.1. Gráficas construidas](#)

### [3.5. Estimación de capacidad de procesamiento](#)

## 1. Organización de los Archivos

Estructura del archivo

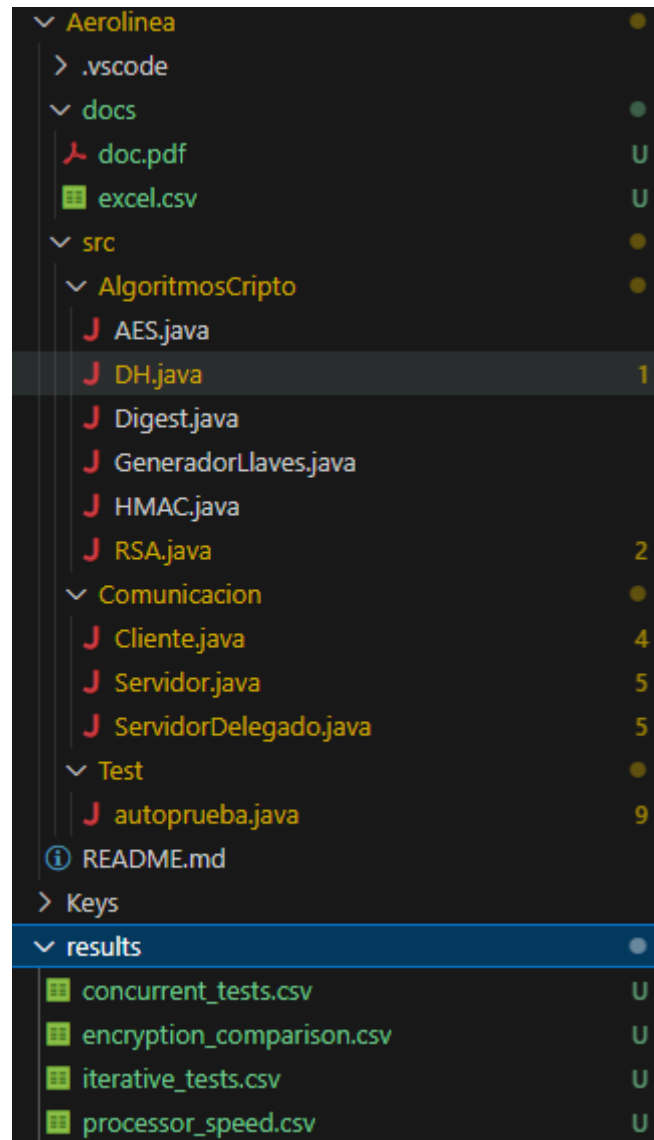


Figura 1: Organización de los archivos

Descripción de carpetas y archivos:

Algoritmos de criptografía

AES

```

public class AES {
    public static byte[] generarIV() {
        byte[] iv = new byte[16];
        SecureRandom aleatorio = new SecureRandom();
        aleatorio.nextBytes(iv);
        return iv;
    }

    public static byte[] encriptar(byte[] datos, byte[] clave, byte[] iv) throws Exception {
        Cipher cifrador = inicializarCifrador(Cipher.ENCRYPT_MODE, clave, iv);
        return cifrador.doFinal(datos);
    }

    public static byte[] desencriptar(byte[] datosEncriptados, byte[] clave, byte[] iv) throws Exception {
        Cipher cifrador = inicializarCifrador(Cipher.DECRYPT_MODE, clave, iv);
        return cifrador.doFinal(datosEncriptados);
    }

    public static Cipher inicializarCifrador(int modo, byte[] clave, byte[] iv) throws Exception {
        SecretKeySpec especificacionClave = new SecretKeySpec(clave, "AES");
        IvParameterSpec especificacionIV = new IvParameterSpec(iv);
        Cipher cifrador = Cipher.getInstance("AES/CBC/PKCS5Padding");
        cifrador.init(modo, especificacionClave, especificacionIV);
        return cifrador;
    }
}

```

Figura 1.1: AES

**GenerarIV():**

Generar un IV aleatorio de 16 bytes (128 bits) con SecureRandom. Se emplea en el paso 12a para preparar la encriptación AES, asegurando que cada encriptación sea única y previniendo ataques de repetición.

**encriptar(byte[] datos, byte[] clave, byte[] iv):**

Cifra datos con AES en modo CBC con relleno PKCS5Padding, y con la clave de 256 bits K\_AB1 y el IV. Se utiliza en los pasos 13 y 16 para cifrar la tabla de servicio y la respuesta del servidor a nosotros, garantizando la confidencialidad.

**desencriptar(byte[] datosEncriptados, byte[] clave, byte[] iv):**

Descifrar datos cifrados AES con la K\_AB1 y IV. El cliente lo usa en los pasos 13 y 16 después de obtener la tabla y la respuesta para que solo las partes con la clave puedan leer los datos.

**inicializarCifrador(int modo, byte[] clave, byte[] iv):**

Configura el cifrador AES con el modo, clave e IV. Es un "azúcar" interno para asegurarse de que se encripte y desencripte de la misma manera.

## DH

```

public class DH {
    private DHParameterSpec parametrosDH;

    public DH(BigInteger p, BigInteger g) {
        this.parametrosDH = new DHParameterSpec(p, g);
    }

    public static DHParameterSpec generarParametros() throws Exception {
        AlgorithmParameterGenerator generadorParametros = AlgorithmParameterGenerator.getInstance("DH");
        generadorParametros.init(size:1024);
        AlgorithmParameters parametros = generadorParametros.generateParameters();
        return parametros.getParameterSpec(paramSpec:DHParameterSpec.class);
    }

    public KeyPair generarParDeClaves() throws Exception {
        KeyPairGenerator generadorClaves = KeyPairGenerator.getInstance("DH");
        generadorClaves.initialize(parametrosDH);
        return generadorClaves.generateKeyPair();
    }

    public PublicKey decodificarClavePublica(DataInputStream entrada) throws Exception {
        int longitud = entrada.readInt();
        byte[] bytesClave = new byte[longitud];
        entrada.readFully(bytesClave);
        KeyFactory fabricaClaves = KeyFactory.getInstance("DH");
        return fabricaClaves.generatePublic(new X509EncodedKeySpec(bytesClave));
    }

    public void enviarClavePublica(DataOutputStream salida, PublicKey clavePublica) throws Exception {
        byte[] bytesClave = clavePublica.getEncoded();
        salida.writeInt(bytesClave.length);
        salida.write(bytesClave);
    }

    public byte[] calcularSecretoCompartido/PrivateKey clavePrivada, PublicKey clavePublica) throws Exception {
        KeyAgreement acuerdoClaves = KeyAgreement.getInstance("DH");
        acuerdoClaves.init(clavePrivada);
        acuerdoClaves.doPhase(clavePublica, lastPhase:true);
        return acuerdoClaves.generateSecret();
    }

    public static byte[][] generarClavesDeSesion(byte[] secretoCompartido) throws Exception {
        MessageDigest sha512 = MessageDigest.getInstance("SHA-512");
        byte[] digest = sha512.digest(secretoCompartido);
        byte[] claveAES = Arrays.copyOfRange(digest, from:0, to:32);
        byte[] claveHMAC = Arrays.copyOfRange(digest, from:32, to:64);
        return new byte[][]{claveAES, claveHMAC};
    }
}

```

Figura 1.2:DH

**generateKeyPair():**

Esta función calcula (numPairs) pares de claves de Diffie-Hellman (pública y privada) con valores de P y G. Se llama en el paso 7, donde el cliente y el servidor genera sus claves y comienzan un intercambio.

**sendPublicKey(DataOutputStream output, PublicKey publicKey):**

Envía la clave pública  $G^*$  al otro lado. Se utiliza en los pasos 7-8 para intercambiar claves públicas entre cliente y servidor.

**decodePublicKey(DataInputStream input):**

Acepta y decodifica la clave pública  $G^*$  del otro extremo. También se emplea en los pasos 7-8 para un intercambio final.

**calculateSharedSecret(PrivateKey privateKey, PublicKey publicKey):**

Calcula el secreto usando la propia clave privada y la clave pública del otro. Se utiliza en los pasos 11a y 11b para la generación del secreto compartido común.

**generateSessionKeys(byte[] sharedSecret):**

Calcula las claves  $K_{AB1}$  y  $K_{AB2}$  a partir del secreto compartido, usando SHA-512. Se invoca después de los pasos 11a y 11b para crear las claves AES y HMAC requeridas.

## DIGEST

```
public class Digest {  
    public static byte[][] derivarClaves(byte[] secretoCompartido) throws Exception {  
        MessageDigest sha512 = MessageDigest.getInstance("SHA-512");  
        byte[] digest = sha512.digest(secretoCompartido);  
        byte[] claveAES = Arrays.copyOfRange(digest, from:0, to:32);  
        byte[] claveHMAC = Arrays.copyOfRange(digest, from:32, to:64);  
        return new byte[][]{claveAES, claveHMAC};  
    }  
}
```

Figura 1.3: Digest

**derivarClaves(byte[] secretoCompartido):**

Genera un digest SHA-512 del secreto compartido y lo divide en  $K_{AB1}$  y  $K_{AB2}$ . Se usa en los pasos 11a y 11b para derivar las llaves simétricas, permitiendo el uso de AES y HMAC en los pasos posteriores.

## HMAC

```
public class HMAC {  
    public static byte[] generarHMAC(byte[] datos, byte[] clave) throws Exception {  
        Mac mac = Mac.getInstance("HmacSHA256");  
        SecretKeySpec especificacionClave = new SecretKeySpec(clave, "HmacSHA256");  
        mac.init(especificacionClave);  
        return mac.doFinal(datos);  
    }  
  
    public static boolean verificarHMAC(byte[] datos, byte[] esperado, byte[] clave) throws Exception {  
        return MessageDigest.isEqual(generarHMAC(datos, clave), esperado);  
    }  
}
```

Figura 1.4: HMAC

**generarHMAC(byte[] datos, byte[] clave):**

Crea un código HMAC-SHA256 de los datos usando la llave K\_AB2. Se usa en los pasos 13 y 16 para proteger la tabla de servicios y la respuesta

**verificarHMAC(byte[] datos, byte[] esperado, byte[] clave):**

Compara el HMAC generado con el recibido. Se usa en los pasos 15 y 17 para que el cliente confirme que los datos no fueron modificados

## RSA

```

public class RSA {
    public static byte[] firmar(byte[] datos, PrivateKey llavePrivada) throws Exception {
        Signature firma = Signature.getInstance("SHA256withRSA");
        firma.initSign(llavePrivada);
        firma.update(datos);
        return firma.sign();
    }

    public static boolean verificar(byte[] datos, byte[] firma, PublicKey llavePublica) throws Exception {
        Signature verificacion = Signature.getInstance("SHA256withRSA");
        verificacion.initVerify(llavePublica);
        verificacion.update(datos);
        return verificacion.verify(firma);
    }

    public static PublicKey cargarLlavePublica(String ruta) throws Exception {
        FileInputStream fis = new FileInputStream(ruta);
        ObjectInputStream ois = new ObjectInputStream(fis);
        PublicKey llavePublica = (PublicKey) ois.readObject();
        ois.close();
        return llavePublica;
    }

    public static PrivateKey cargarLlavePrivada(String ruta) throws Exception {
        FileInputStream fis = new FileInputStream(ruta);
        ObjectInputStream ois = new ObjectInputStream(fis);
        PrivateKey llavePrivada = (PrivateKey) ois.readObject();
        ois.close();
        return llavePrivada;
    }
}

```

Figura 1.5: RSA

### **firmar(byte[] datos, PrivateKey llavePrivada):**

Firma los datos con SHA256withRSA usando la llave privada del servidor. Se usa en el paso 9 para firmar los parámetros Diffie-Hellman

### **verificar(byte[] datos, byte[] firma, PublicKey llavePublica):**

Verifica la firma de los datos con la llave pública, usando SHA256withRSA. Se usa en el paso 9 para que el cliente valide los parámetros recibidos

### **cargarLlavePublica(String ruta) y cargarLlavePrivada(String ruta):**

Cargan las llaves RSA desde archivos. El cliente usa la pública para verificar firmas, y el servidor usa ambas para firmar y verificar, como se usa en el paso 9.



## Código fuente del cliente

```
public Cliente(String direccion, int puerto, BigInteger primo, BigInteger generador) throws Exception {
    socketCliente = new Socket(direccion, puerto);
    salida = new DataOutputStream(socketCliente.getOutputStream());
    entrada = new DataInputStream(socketCliente.getInputStream());

    // Generar la clave publica
    publicKey = RSA.cargarClavePublica(ruta:"Llaves/LlavePublica.txt");

    intercambioDH = new DH(primo, generador);
    realizarIntercambioClaves();
    this.servicios = recibirTablaServicios();
}

private void realizarIntercambioClaves() throws Exception {
    KeyPair parClaves = intercambioDH.generarParDeClaves();
    PrivateKey clavePrivada = parClaves.getPrivate();
    PublicKey clavePublica = parClaves.getPublic();

    intercambioDH.enviarClavePublica(salida, clavePublica);
    PublicKey clavePublicaServidor = intercambioDH.decodificarClavePublica(entrada);

    byte[] secretoCompartido = intercambioDH.calcularSecretoCompartido(clavePrivada, clavePublicaServidor);
    byte[] clavesSesion = DH.generarClavesDeSesion(secretoCompartido);
    claveAES = clavesSesion[0];
    claveHMAC = clavesSesion[1];
}

private Map<Integer, String> recibirTablaServicios() throws Exception {
    // Recibir IV
    byte[] iv = new byte[entrada.readInt()];
    entrada.readFully(iv);

    // Recibir mensaje cifrado
    byte[] mensajeCifrado = new byte[entrada.readInt()];
    entrada.readFully(mensajeCifrado);

    // Recibir HMAC
    byte[] hmacRecibido = new byte[entrada.readInt()];
    entrada.readFully(hmacRecibido);

    // Verificar HMAC
    boolean hmacValido = HMAC.verificarHMAC(mensajeCifrado, hmacRecibido, claveHMAC);
    if (!hmacValido) {
        throw new Exception("Error en la consulta: HMAC inválido en la tabla de servicios");
    }

    // Descriptar el mensaje
    byte[] mensaje = AES.desencriptar(mensajeCifrado, claveAES, iv);

    // Leer el mensaje: longitud de serviciosBytes + serviciosBytes + longitud de firma + firma
    ByteArrayInputStream bais = new ByteArrayInputStream(mensaje);
    DataInputStream dis = new DataInputStream(bais);
    int lenServicios = dis.readInt();
    byte[] serviciosBytes = new byte[lenServicios];
    dis.readFully(serviciosBytes);
    int lenFirma = dis.readInt();
    byte[] firma = new byte[lenFirma];
    dis.readFully(firma);

    // Verificar la firma
    boolean firmaValida = RSA.verificar(serviciosBytes, firma, publicKey);
    if (!firmaValida) {
        throw new Exception("Error en la consulta: firma inválida en la tabla de servicios");
    }
}
```

```
// Deserializar la tabla de servicios
bais = new ByteArrayInputStream(serviciosBytes);
ObjectInputStream ois = new ObjectInputStream(bais);
Map<Integer, String> servicios = (Map<Integer, String>) ois.readObject();

// Mostrar los servicios al usuario
System.out.println("Servicios disponibles:");
for (Map.Entry<Integer, String> entry : servicios.entrySet()) {
    System.out.println("ID: " + entry.getKey() + " -> " + entry.getValue());
}

return servicios;
}

public String solicitarServicio(int idServicio) throws Exception {
    byte[] vectorInicial = AES.generarIV();
    salida.writeInt(vectorInicial.length);
    salida.write(vectorInicial);

    String mensaje = String.valueOf(idServicio);
    byte[] datos = mensaje.getBytes();
    byte[] datosCifrados = AES.encryptar(datos, claveAES, vectorInicial);

    salida.writeInt(datosCifrados.length);
    salida.write(datosCifrados);

    byte[] hmac = HMAC.generarHMAC(datosCifrados, claveHMAC);
    salida.writeInt(hmac.length);
    salida.write(hmac);

    byte[] vectorInicialRespuesta = new byte[entrada.readInt()];
    entrada.readFully(vectorInicialRespuesta);

    byte[] datosCifradosRespuesta = new byte[entrada.readInt()];
    entrada.readFully(datosCifradosRespuesta);

    byte[] hmacRespuesta = new byte[entrada.readInt()];
    entrada.readFully(hmacRespuesta);

    boolean hmacValido = HMAC.verificarHMAC(datosCifradosRespuesta, hmacRespuesta, claveHMAC);
    if (!hmacValido) {
        throw new Exception("Error en la consulta: HMAC inválido en la respuesta del servidor");
    }

    byte[] datosDescifrados = AES.desencriptar(datosCifradosRespuesta, claveAES, vectorInicialRespuesta);
    return new String(datosDescifrados);
}

public void cerrar() throws Exception {
    salida.close();
    entrada.close();
    socketCliente.close();
}
```

```

public static void main(String[] args) throws Exception {
    BigInteger primo = new BigInteger("179733134862315907783315679774513178602908487568179644423684197280216158519368947833795864925415021805654858895636464485481992391080567928770033558163229531362390765887357599148225748625758074253020774471258958957937");
    BigInteger generador = new BigInteger("2");

    Scanner scanner = new Scanner(System.in);
    System.out.println("¿Cuántos clientes concurrentes desea ejecutar? (0 para modo interactivo): ");
    int numeroClientes = scanner.nextInt();

    if (numeroClientes < 0) {
        System.out.println("Por favor, ingrese un número mayor o igual a 0.");
        scanner.close();
        return;
    }

    if (numeroClientes == 0) {
        // Modo interactivo: un cliente permite al usuario seleccionar un servicio
        Cliente cliente = new Cliente(direccion:"localhost", puerto, primo, generador);
        System.out.println("Ingrese el ID del servicio: ");
        int idServicio = scanner.nextInt();
        try {
            String respuesta = cliente.solicitarServicio(idServicio);
            System.out.println("Respuesta para servicio " + idServicio + ": " + respuesta);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        } finally {
            cliente.cerrar();
        }
    } else {
        // Modo concurrente: múltiples clientes, cada uno selecciona un servicio aleatoriamente
        ExecutorService grupodillos = Executors.newFixedThreadPool(numeroClientes);
        List<Future<>> tareas = new ArrayList<>();

        System.out.println("Iniciando " + numeroClientes + " clientes concurrentes...");
        for (int i = 0; i < numeroClientes; i++) {
            tareas.add(grupodillos.submit(() -> {
                try {
                    Cliente cliente = new Cliente(direccion:"localhost", puerto, primo, generador);
                    Random random = new Random();
                    List<Integer> ids = new ArrayList<>(cliente.servicios.keySet());
                    int idServicio = ids.get(random.nextInt(ids.size()));
                    String respuesta = cliente.solicitarServicio(idServicio);
                    System.out.println("Cliente " + Thread.currentThread().getName() + " recibió respuesta para servicio " + idServicio + ": " + respuesta);
                    cliente.cerrar();
                } catch (Exception e) {
                    System.err.println("Error en cliente " + Thread.currentThread().getName() + ": " + e.getMessage());
                }
            }));
        }

        for (Future<> tarea : tareas) {
            try {
                tarea.get();
            } catch (Exception e) {
                System.err.println("Error esperando tarea: " + e.getMessage());
            }
        }

        grupodillos.shutdown();
    }

    scanner.close();
    System.out.println("Todos los clientes han terminado.");
}

```

El Cliente tiene como finalidad establecer una conexión segura entre un cliente y un servidor a través de sockets. Inicialmente, realiza un intercambio de claves usando el algoritmo Diffie-Hellman, que permite a ambas partes generar de forma segura una clave compartida sin necesidad de transmitirla directamente. Esta clave se divide en dos partes: una para cifrar los datos utilizando AES (Advanced Encryption Standard) y otra para verificar la integridad de los mensajes mediante HMAC. Una vez establecida esta comunicación segura, el cliente recibe una tabla de servicios disponible en el servidor, la cual viene cifrada para garantizar la confidencialidad y firmada digitalmente usando RSA para asegurar su autenticidad. Posteriormente, el cliente puede solicitar un servicio específico enviando un mensaje cifrado y protegido con HMAC. Además, el programa ofrece la posibilidad de ejecutar múltiples clientes de manera concurrente, probando así la robustez y seguridad del sistema en condiciones de alta demanda. En conjunto, la clase integra distintas técnicas criptográficas para asegurar que los datos no sean interceptados, modificados o falsificados durante la comunicación.

*La parte de que varios clientes puedan funcionar de manera concurrente fue diseñada de manera adicional con el propósito de facilitar a la hora de realizar las debidas pruebas.*

De manera más detallada lo primero que se hace es realizar la conexión con el servidor

202313509

202315338

```

socketCliente = new Socket(direccion, puerto);
salida = new DataOutputStream(socketCliente.getOutputStream());
entrada = new DataInputStream(socketCliente.getInputStream());

// Cargar la llave pública
PublicKey clavePublica = RSA.cargarLlavePublica(ruta:"Llaves/LlavePublica.txt");

intercambioDH = new DH(primo, generador);
realizarIntercambioClaves();
this.servicios = recibirTablaServicios();
}

```

Luego de saber del servidor realiza el intercambio de llaves para proceder a recibir las tablas de servicios, lo que ocurre dentro de este código en adelante.

```

private void realizarIntercambioClaves() throws Exception {
    KeyPair parClaves = intercambioDH.generarParDeClaves();
    PrivateKey clavePrivada = parClaves.getPrivate();
    PublicKey clavePublica = parClaves.getPublic();

    intercambioDH.enviarClavePublica(salida, clavePublica);
    PublicKey clavePublicaServidor = intercambioDH.decodificarClavePublica(entrada);

    byte[] secretoCompartido = intercambioDH.calcularSecretoCompartido(clavePrivada, clavePublicaServidor);
    byte[][] clavesSesion = DH.generarClavesDeSesion(secretoCompartido);
    claveAES = clavesSesion[0];
    claveHMAC = clavesSesion[1];
}

private Map<Integer, String> recibirTablaServicios() throws Exception {
    // Recibir IV
    byte[] iv = new byte[entrada.readInt()];
    entrada.readFully(iv);

    // Recibir mensaje cifrado
    byte[] mensajeCifrado = new byte[entrada.readInt()];
    entrada.readFully(mensajeCifrado);

    // Recibir HMAC
    byte[] hmacRecibido = new byte[entrada.readInt()];
    entrada.readFully(hmacRecibido);

    // Verificar HMAC
    boolean hmacValido = HMAC.verificarHMAC(mensajeCifrado, hmacRecibido, claveHMAC);
    if (!hmacValido) {
        throw new Exception("Error en la consulta: HMAC inválido en la tabla de servicios");
    }

    // Desencriptar el mensaje
    byte[] mensaje = AES.desencriptar(mensajeCifrado, claveAES, iv);
}

```

Dentro del main se agrega una funcionalidad de implementar varios clientes de manera concurrente que seleccionan un servicio aleatorio para las pruebas lo cual se ve presente dentro del siguiente fragmento de código, esta es la parte que se agregó para facilitar las pruebas sobre el sistema:

```

// Modo concurrente: multiples clientes, cada uno selecciona un servicio aleatoriamente
ExecutorService grupoHilos = Executors.newFixedThreadPool(numeroClientes);
List<Future<?>> tareas = new ArrayList<>();

System.out.println("Iniciando " + numeroClientes + " clientes concurrentes...");
for (int i = 0; i < numeroClientes; i++) {
    tareas.add(grupoHilos.submit(() -> {
        try {
            Cliente cliente = new Cliente(direccion:"localhost", puerto, primo, generador);
            Random random = new Random();
            List<Integer> ids = new ArrayList<>(cliente.servicios.keySet());
            int idServicio = ids.get(random.nextInt(ids.size()));
            String respuesta = cliente.solicitarServicio(idServicio);
            System.out.println("Cliente " + Thread.currentThread().getName() + " recibió respuesta para servicio " + idServicio + ": " + respuesta);
            cliente.cerrar();
        } catch (Exception e) {
            System.err.println("Error en cliente " + Thread.currentThread().getName() + ": " + e.getMessage());
        }
    }));
}

```

## Código fuente del servidor

```
public Servidor(int puerto, BigInteger primo, BigInteger generador) throws Exception {
    this.puerto = puerto;
    this.primo = primo;
    this.generador = generador;
    servicios = new HashMap<>();
    servicios.put(1, "192.168.1.1:9001");
    servicios.put(2, "192.168.1.2:9002");
    System.out.println("Servicios disponibles:");
    servicios.forEach((id, direccion) -> System.out.println("ID: " + id + " -> " + direccion));
    grupoHilos = Executors.newFixedThreadPool(100);
    corriendo = true;

    // Cargar las llaves RSA
    privateKey = RSA.cargarLlavePrivada(ruta:"Llaves/LlavePrivada.secret");
    publicKey = RSA.cargarLlavePublica(ruta:"Llaves/LlavePublica.txt");
    System.out.println("Llaves RSA cargadas exitosamente.");
}

public void iniciar() throws Exception {
    socketServidor = new ServerSocket(puerto, 100);
    try {
        while (corriendo) {
            Socket socketCliente = socketServidor.accept();
            System.out.println("Cliente conectado: " + socketCliente.getInetAddress());
            // Pasar las llaves al ServidorDelegado
            grupoHilos.submit(new ServidorDelegado(socketCliente, servicios, primo, generador, privateKey, publicKey));
        }
    } catch (SocketException e) {
        if (!corriendo) {
            System.out.println("Servidor detenido.");
        } else {
            throw e;
        }
    } finally {
        socketServidor.close();
    }
}

public void detener() throws Exception {
    corriendo = false;
    if (socketServidor != null && !socketServidor.isClosed()) {
        socketServidor.close();
    }
    grupoHilos.shutdown();
    try {
        grupoHilos.awaitTermination(20, TimeUnit.SECONDS);
    } catch (InterruptedException e) {
        System.err.println("Error al cerrar grupo de hilos: " + e.getMessage());
    }
}

Run[Debug]
public static void main(String[] args) throws Exception {
    int puerto = 5000;
    BigInteger primo = new BigInteger("17976931348623159077083915679378745319766029604875601170644423684197180216158519368947833795864925541502180565485980503646440548199239100050792877003355816639229553136");
    BigInteger generador = new BigInteger("2");

    Servidor servidor = new Servidor(puerto, primo, generador);
    System.out.println("Servidor iniciado en el puerto " + puerto + "...");
    servidor.iniciar();
}
```

El servidor lo primero que realiza es la carga de llaves de los archivos, tanto la pública como la privada, a partir de ese momento empieza a aceptar conexiones de clientes que va delegando en cada servidor donde se realiza la conexión con los clientes.

## Llave pública y llave privada (generador de llaves)

```
package AlgoritmosCripto;

import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.PublicKey;

public class GeneradorLlaves {
    private static final String ALGORITMO = "RSA";

    Run | Debug
    public static void main(String[] args) {
        try {
            System.out.println("Iniciando creación de llaves...");
            Thread.sleep(1000);

            KeyPairGenerator generador = KeyPairGenerator.getInstance(ALGORITMO);
            generador.initialize(1024);
            KeyPair parLlaves = generador.generateKeyPair();

            System.out.println("Llaves generadas de manera exitosa");
            Thread.sleep(1000);

            PublicKey llavePublica = parLlaves.getPublic();
            PrivateKey llavePrivada = parLlaves.getPrivate();

            File directorio = new File("Llaves");
            if (!directorio.exists()) {
                directorio.mkdirs();
            }

            System.out.println("Almacenando llaves en archivos...");
            try (FileOutputStream archivoPublico = new FileOutputStream("Llaves/LlavePublica.txt");
                ObjectOutputStream oos = new ObjectOutputStream(archivoPublico)) {
                oos.writeObject(llavePublica);
            }

            try (FileOutputStream archivoPrivado = new FileOutputStream("Llaves/LlavePrivada.secret");
                ObjectOutputStream oos1 = new ObjectOutputStream(archivoPrivado)) {
                oos1.writeObject(llavePrivada);
            }

            System.out.println("Almacenamiento exitoso");
        } catch (NoSuchAlgorithmException | IOException | InterruptedException e) {
            System.err.println("Error durante la generación o almacenamiento de llaves: " + e.getMessage());
        }
    }
}
```

Este archivo se ejecuta de primeras, es el encargado de generar la llave Pública y la llave Privada que se van a emplear para cifrar la conexión entre cliente y servidor. Para ello emplea la clase **KeyPairGenerator** con el algoritmo **RSA** y una longitud de llaves de 1024 bits para mantener una mayor seguridad.

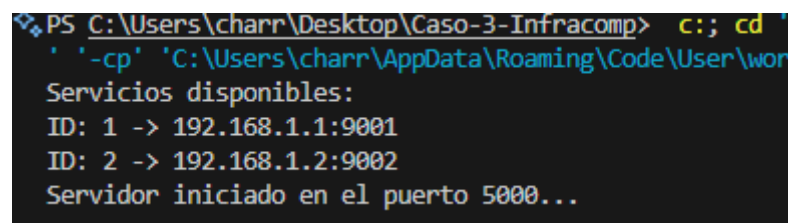
Se generan dos archivos que se almacenan dentro de la carpeta Llaves, como paso adicional se hace una verificación de que el directorio Llaves exista o en su defecto lo crea.

## 2. Instrucciones para Ejecutar el Sistema

El sistema está construido para que se pueda ejecutar de manera simple, existe una autopruueba que genera archivos csv con lo cual nos ayudamos para la construcción de la recolección de datos, de todas maneras cada clase de comunicación cuenta con su método main para poder ser ejecutada de manera independiente, todos asumen que ya generaste la llave pública y privada explicada arriba

### Cómo ejecutar el servidor:

Para ejecutar el servidor, tienes que pararte sobre la clase servidor y le das al boton de Run Java arriba en la esquina derecha. Eso te saldrá una terminal donde se te indicarán los servicios disponibles y el puerto donde se está corriendo el servidor

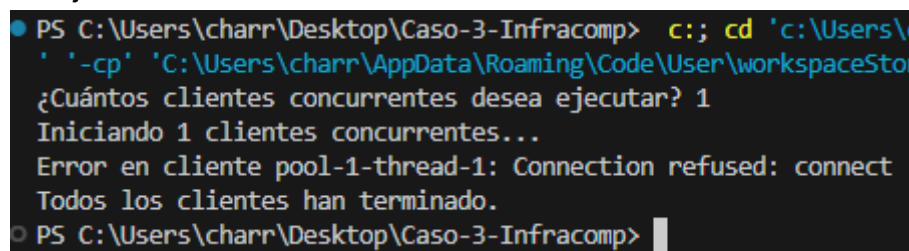


```
PS C:\Users\charr\Desktop\Caso-3-Infracomp> c:; cd 'C:\Users\charr\AppData\Roaming\Code\User\workspaceStorage'
'-cp' 'C:\Users\charr\AppData\Roaming\Code\User\workspaceStorage'
Servicios disponibles:
ID: 1 -> 192.168.1.1:9001
ID: 2 -> 192.168.1.2:9002
Servidor iniciado en el puerto 5000...
```

Figura 2.1: Ejecución de la terminal

### Cómo ejecutar el cliente y cómo configurar el número de clientes concurrentes:

Para ejecutar el o los clientes tienes que ejecutar primero el servidor, de no ser así, te saldrá que no fue posible ejecutar el/los clientes.



```
PS C:\Users\charr\Desktop\Caso-3-Infracomp> c:; cd 'C:\Users\charr\AppData\Roaming\Code\User\workspaceStorage'
'-cp' 'C:\Users\charr\AppData\Roaming\Code\User\workspaceStorage'
¿Cuántos clientes concurrentes desea ejecutar? 1
Iniciando 1 clientes concurrentes...
Error en cliente pool-1-thread-1: Connection refused: connect
Todos los clientes han terminado.
PS C:\Users\charr\Desktop\Caso-3-Infracomp>
```

Figura 2.2: Ejecución del cliente sin ejecutar el servicio

Una vez ejecutado el servidor, te pedirá por consola que indiques el número de clientes que quieres ejecutar de manera concurrente, pueden uno o muchos, y luego el sistema continuará con su operación

202313509

202315338

```

C:\Users\charr\Desktop\Caso-3-Infracomp> cd "C:\Users\charr\Desktop\Caso-3-Infracomp" & & "C:\Program Files\Eclipse Adoptium\jdk-21.0.3-hotspot\bin\java.exe" -XX:+ShowCodeDetailsInExceptionMessages -cp "C:\Users\charr\AppData\Roaming\Code\User\workspaceStorage\4b666d9cd3a61438724542c483851e75\redhat_java\jdt_ws\Caso-3-Infracomp_af8cfdf7\bin" "Comunicacion.Cliente"
¿Cuántos clientes concurrentes desea ejecutar? 1
Iniciando 1 clientes concurrentes...
Error en cliente pool-1-thread-1: Connection refused: connect
Todos los clientes han terminado.
PS C:\Users\charr\Desktop\Caso-3-Infracomp> cd "C:\Users\charr\Desktop\Caso-3-Infracomp" & & "C:\Program Files\Eclipse Adoptium\jdk-21.0.3-hotspot\bin\java.exe" -XX:+ShowCodeDetailsInExceptionMessages -cp "C:\Users\charr\AppData\Roaming\Code\User\workspaceStorage\4b666d9cd3a61438724542c483851e75\redhat_java\jdt_ws\Caso-3-Infracomp_af8cfdf7\bin" "Comunicacion.Cliente"
¿Cuántos clientes concurrentes desea ejecutar? 1
Iniciando 1 clientes concurrentes...
Cliente pool-1-thread-1 recibió respuesta: 192.168.1.1:9001
Todos los clientes han terminado.
PS C:\Users\charr\Desktop\Caso-3-Infracomp>

```

Figura 2.3: Ejecución del cliente singular

```

TERMINAL PROBLEMS DEBUG CONSOLE OUTPUT PORTS GITLENS COMMENTS SQL HISTORY TASK MONITOR
¿Cuántos clientes concurrentes desea ejecutar? 1
Iniciando 1 clientes concurrentes...
Error en cliente pool-1-thread-1: Connection refused: connect
Todos los clientes han terminado.
PS C:\Users\charr\Desktop\Caso-3-Infracomp> cd "C:\Users\charr\Desktop\Caso-3-Infracomp" & & "C:\Program Files\Eclipse Adoptium\jdk-21.0.3-hotspot\bin\java.exe" -XX:+ShowCodeDetailsInExceptionMessages -cp "C:\Users\charr\AppData\Roaming\Code\User\workspaceStorage\4b666d9cd3a61438724542c483851e75\redhat_java\jdt_ws\Caso-3-Infracomp_af8cfdf7\bin" "Comunicacion.Cliente"
¿Cuántos clientes concurrentes desea ejecutar? 1
Iniciando 1 clientes concurrentes...
Cliente pool-1-thread-1 recibió respuesta: 192.168.1.1:9001
Todos los clientes han terminado.
PS C:\Users\charr\Desktop\Caso-3-Infracomp> cd "C:\Users\charr\Desktop\Caso-3-Infracomp" & & "C:\Program Files\Eclipse Adoptium\jdk-21.0.3-hotspot\bin\java.exe" -XX:+ShowCodeDetailsInExceptionMessages -cp "C:\Users\charr\AppData\Roaming\Code\User\workspaceStorage\4b666d9cd3a61438724542c483851e75\redhat_java\jdt_ws\Caso-3-Infracomp_af8cfdf7\bin" "Comunicacion.Cliente"
¿Cuántos clientes concurrentes desea ejecutar? 12
Iniciando 12 clientes concurrentes...
Cliente pool-1-thread-9 recibió respuesta: 192.168.1.1:9001
Cliente pool-1-thread-5 recibió respuesta: 192.168.1.1:9001
Cliente pool-1-thread-3 recibió respuesta: 192.168.1.1:9001
Cliente pool-1-thread-7 recibió respuesta: 192.168.1.1:9001
Cliente pool-1-thread-12 recibió respuesta: 192.168.1.2:9002
Cliente pool-1-thread-11 recibió respuesta: 192.168.1.1:9001
Cliente pool-1-thread-10 recibió respuesta: 192.168.1.2:9002
Cliente pool-1-thread-8 recibió respuesta: 192.168.1.2:9002
Cliente pool-1-thread-1 recibió respuesta: 192.168.1.1:9001
Cliente pool-1-thread-6 recibió respuesta: 192.168.1.2:9002
Cliente pool-1-thread-4 recibió respuesta: 192.168.1.2:9002
Cliente pool-1-thread-2 recibió respuesta: 192.168.1.2:9002
Todos los clientes han terminado.
PS C:\Users\charr\Desktop\Caso-3-Infracomp>

```

Figura 2.4: Ejecución de muchos clientes

## Cómo correr la autopruueba:

La clase `autopruueba.java` ejecuta cuatro tipos de solicitudes secuenciales de un cliente, múltiples clientes simultáneos, comparación de tiempos entre cifrado simétrico y asimétrico, y estimación de la velocidad del procesador. Los resultados se guardan en archivos CSV en `results/`, lo que permite analizarlos posteriormente con Excel para generar gráficos.

`autopruueba.java` está configurado para llevar a cabo pruebas concurrentes con 4, 16, 32 y 64 clientes. Para ejecutar las pruebas con estos valores por defecto primero parate en el archivo `autopruueba.java` y oprime el boton de ejecutar

El programa iniciará el servidor y ejecutará las cuatro pruebas:

202313509

202315338

prueba	descripción	se guarda en
Pruebas iterativas	32 solicitudes secuenciales, alternando entre los servicios 1 y 2.	iterative_tests.csv
Pruebas concurrentes	Ejecutará 4, 16, 32 y 64 clientes concurrentes, midiendo tiempos promedio.	concurrent_tests.csv
Comparación de cifrado	Comparará los tiempos de cifrado AES y RSA.	encryption_comparison.csv
Estimación de velocidad	Medirá operaciones por segundo para AES y RSA.	processor_speed.csv

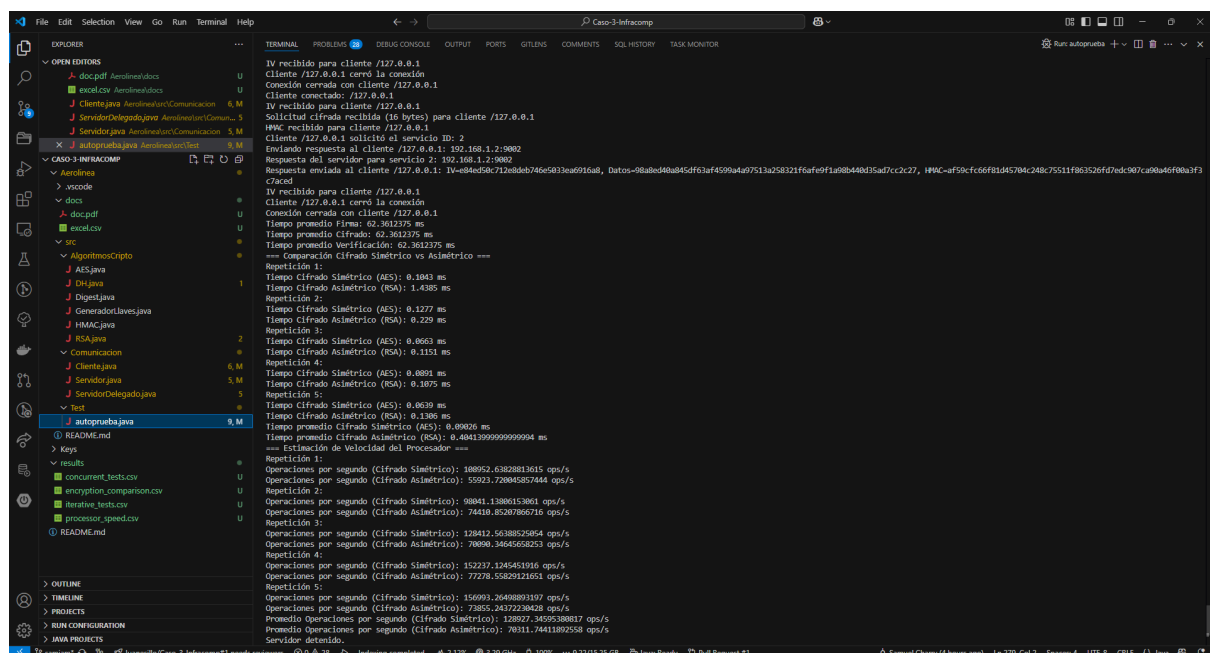


Figura 2.4: Ejecución de autoprueba

### 3. Respuestas a las Tareas y Preguntas



### 3.1. Gráficas construidas

- Tiempos de firma por escenario-Tiempos de cifrado de la tabla por escenario-Tiempos de verificación de consulta por escenario

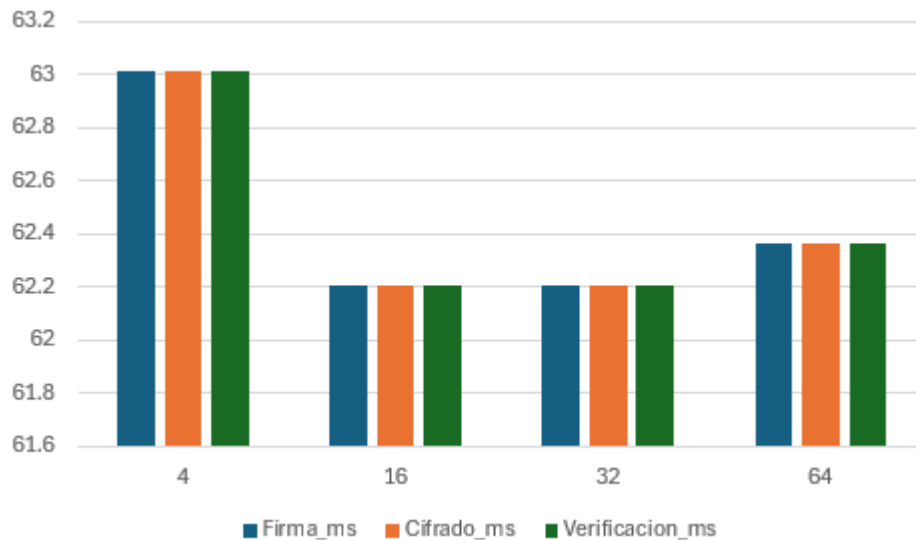


Figura 3.1.1: tablas tiempos

#### Gráfica de Consultas Iterativas

##### Escenario (i): 1 cliente iterativo, 32 consultas secuenciales:

El programa `autopruueba.java` fue ejecutado, el cual realiza 32 peticiones consecutivas de un cliente al servidor. Las peticiones se emitieron de manera alterna a los servicios asociados con los IDs 1 y 2.

Los resultados de la primera ronda indicaron un tiempo de respuesta de 12 ms.

Para las iteraciones 2 a 6, 10, 16 y 32 veces fueron 2 milisegundos cada una. Otras ejecuciones (7-9, 11-15, 17-31) reportaron 1 ms por petición.

La duración promedio de las peticiones fue de 1.59375 milisegundos. Esto corresponde a un tiempo total aproximado de 51 milisegundos para ejecutar las 32 consultas ( $32 \times 1.59375 = 51$  ms).

El pequeño costo promedio por petición indica un buen rendimiento al iterar, en su mayoría con 25/32 peticiones satisfechas en 1 milisegundo.

La diferencia en latencia de la primera petición (12 ms) se debe al intercambio de claves Diffie-Hellman inicial, que solo ocurre una vez cuando la sesión segura se establece. Después del intercambio de claves, todas las peticiones posteriores pueden beneficiarse de la reutilización de la clave de sesión, resultando en reducciones extensas del tiempo de procesamiento.

- Comparación entre cifrado simétrico y asimétrico

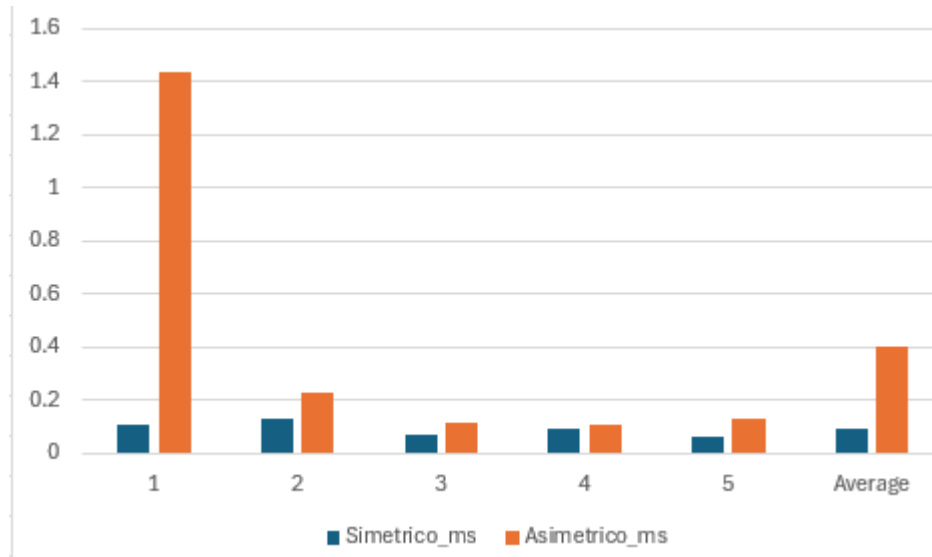


Figura 3.3.2: simétrico vs asimétrica

Repeticion	Simetrico_ms	Asimetrico_ms
1	1.043	1.4385
2	1.277	229
3	663	1.151
4	891	1.075
5	639	1.306

Se utilizó el método `compararCifrado()` de `autopueba.java` para medir los tiempos de cifrado simétrico (AES) y asimétrico (RSA).

```
private static void compararCifrado() throws Exception {
    System.out.println(x:"=== Comparación Cifrado Simétrico vs Asimétrico ===");
    byte[] datos = "Mensaje de prueba".getBytes();
    byte[] claveAES = new byte[32];
    byte[] vectorInicial = AES.generarIV();

    PublicKey llavePublica = RSA.cargarLlavePublica(ruta:"Keys/PublicKey.txt");
    PrivateKey llavePrivada = RSA.cargarLlavePrivada(ruta:"Keys/PrivateKey.secret");
}
```

Figura 3.2.4: compara cifrado RSA

Se cifró un mensaje de prueba con AES y con RSA

```
for (int i = 1; i <= 5; i++) {
    System.out.println("Repetición " + i + ":");

    long inicioSimetrico = System.nanoTime();
    AES.encryptar(datos, claveAES, vectorInicial);
    long finSimetrico = System.nanoTime();
    double tiempoSimetrico = (finSimetrico - inicioSimetrico) / 1_000_000.0;
    tiempoTotalSimetrico += tiempoSimetrico;
    System.out.println("Tiempo Cifrado Simétrico (AES): " + tiempoSimetrico + " ms");

    long inicioAsimetrico = System.nanoTime();
    Cipher cifradorRSA = Cipher.getInstance(transformation:"RSA");
    cifradorRSA.init(Cipher.ENCRYPT_MODE, llavePublica);
    byte[] datosCifradosRSA = cifradorRSA.doFinal(datos);
    long finAsimetrico = System.nanoTime();
    double tiempoAsimetrico = (finAsimetrico - inicioAsimetrico) / 1_000_000.0;
    tiempoTotalAsimetrico += tiempoAsimetrico;
    System.out.println("Tiempo Cifrado Asimétrico (RSA): " + tiempoAsimetrico + " ms");

    writer.write(String.format(format:"%d,%f,%f\n", i, tiempoSimetrico, tiempoAsimetrico));
}

double avgSimetrico = tiempoTotalSimetrico / 5.0;
double avgAsimetrico = tiempoTotalAsimetrico / 5.0;

System.out.println("Tiempo promedio Cifrado Simétrico (AES): " + avgSimetrico + " ms");
System.out.println("Tiempo promedio Cifrado Asimétrico (RSA): " + avgAsimetrico + " ms");
```

*Figura 3.3.5: compara cifrado RSA*

Se realizaron 5 repeticiones, midiendo los tiempos en milisegundos, y se calcularon los promedios.

AES es 4.48 veces más rápido que RSA en promedio ( $0.40414 / 0.09026 \approx 4.48$ ), lo que refleja la mayor eficiencia de los algoritmos simétricos para cifrado de datos.

Los tiempos de AES son estables (0.0639 ms a 0.1277 ms), mientras que RSA muestra mayor variabilidad (0.1075 ms a 1.4385 ms), probablemente debido a la inicialización del cifrado.

Concluimos que AES es mejor para cifrar las comunicaciones en este sistema, mientras que RSA es más mejor para el intercambio de claves o firmas digitales.

```

=== Comparación Cifrado Simétrico vs Asimétrico ===
Repetición 1:
Tiempo Cifrado Simétrico (AES): 0.1043 ms
Tiempo Cifrado Asimétrico (RSA): 1.4385 ms
Repetición 2:
Tiempo Cifrado Simétrico (AES): 0.1277 ms
Tiempo Cifrado Asimétrico (RSA): 0.229 ms
Repetición 3:
Tiempo Cifrado Simétrico (AES): 0.0663 ms
Tiempo Cifrado Asimétrico (RSA): 0.1151 ms
Repetición 4:
Tiempo Cifrado Simétrico (AES): 0.0891 ms
Tiempo Cifrado Asimétrico (RSA): 0.1075 ms
Repetición 5:
Tiempo Cifrado Simétrico (AES): 0.0639 ms
Tiempo Cifrado Asimétrico (RSA): 0.1306 ms
Tiempo promedio Cifrado Simétrico (AES): 0.09026 ms
Tiempo promedio Cifrado Asimétrico (RSA): 0.40413999999999994 ms

```

Figura 3.3.6: Simétrico vs Asimétrica

- Prueba iterativa

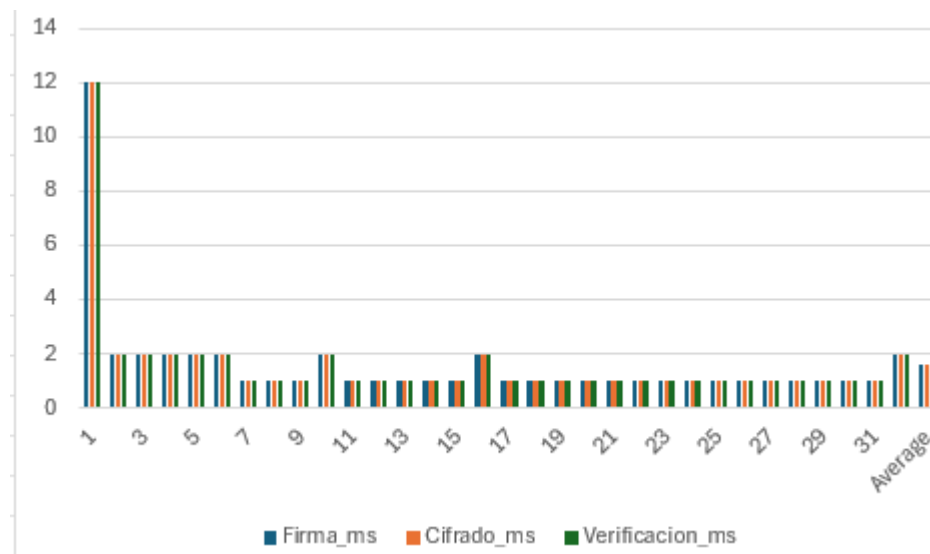


Figura 3.3.4: Firma, cifrado, Verificación

### Escenario (ii): Clientes y servidores concurrentes (4, 16, 32, 64 delegados)

El programa `autopruueba.java` se ejecutó con delegados predeterminados (4, 16, 32, 64). El programa generó varios clientes que enviaron solicitudes al servidor y registraron los tiempos promedio a través de cada cliente para firmar, cifrar y verificar. Los números se recuperaron del archivo `concurrent_tests.csv`.

202313509

202315338

La información almacenada en `concurrent_tests.csv` proporciona los siguientes tiempos promedio por cliente:

Delegados	Firma_ms	Cifrado_ms	Verificacion_ms
4	63.010.325	63.010.325	63.010.325
16	62.203.438	62.203.438	62.203.438
32	62.204.966	62.204.966	62.204.966
64	62.361.238	62.361.238	62.361.238

Los tiempos medios por cliente son bastante estables, manteniéndose entre 62.20 ms y 63.01 ms al agregar delegados. Esto muestra que el servidor escala bien en términos de concurrencia, ya que la sobrecarga sigue siendo baja cuando el número de concurrencias aumenta de 4 a 64.

Observe también que con 4 y 16 delegados, el tiempo promedio se reduce en solo 0.8 milisegundos. Esta disminución indica que el servidor asigna sus recursos de manera más eficiente al atender más clientes, y la carga de trabajo se distribuye mejor.

Luego, para el escenario alternativo, con 16, 32 y 64 delegados, el tiempo promedio se vuelve ligeramente mayor (62.20 ms - 62.36 ms). Este pequeño aumento es probablemente causado por la gestión de hilos adicionales en un grado más alto de concurrencia.

En conjunto, los resultados muestran la escalabilidad del sistema. El tiempo promedio por cliente ha aumentado muy ligeramente (menos de 1 milisegundo). Esto demuestra una buena sensibilidad y estabilidad durante condiciones de carga.

- velocidad del procesador

202313509

202315338

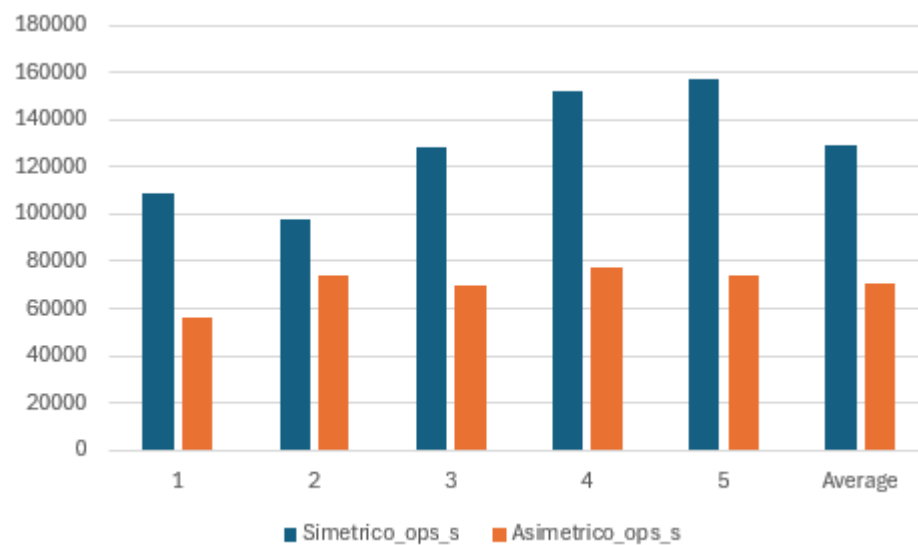


Figura 3.3.7: Simétrico vs Asimétrica

### Gráfica de Estimación de Velocidad del Procesador:

Aquí vemos que AES (azul) realiza muchas más operaciones por segundo, oscilando entre 98k y 157k ops/seg.

Por otro lado, RSA (línea naranja) es más rápido en 55,000 a 77,000 operaciones por segundo.

En general, la señal desea mostrarlo así también, ya que AES tiene una velocidad promedio de 128,927 ops por segundo, mientras que RSA es de 70,311 ops por segundo, lo cual es una prueba de que AES puede ser calculado más rápidamente.

### 3.2. Estimación de capacidad de procesamiento

Repeticion	Simetrico_ops_s	Asimetrico_ops_s
1	108.952.638	55923.72
2	980.411.381	744.108.521
3	128.412.564	700.903.465
4	152.237.125	772.785.583
5	156.993.265	738.552.437
Average	128.927.346	703.117.441

- Descripción del escenario de medición

El escenario de medición para estimar la capacidad de procesamiento se implementó mediante el método `estimarVelocidadProcesador()` en la clase `autopruueba.java`.

```
private static void estimarVelocidadProcesador() throws Exception {
    System.out.println(x:"=== Estimación de Velocidad del Procesador ===");
    byte[] datos = "Mensaje de prueba".getBytes();
    byte[] claveAES = new byte[32];
    byte[] vectorInicial = AES.generarIV();

    PublicKey llavePublica = RSA.cargarLlavePublica(ruta:"Keys/PublicKey.txt");
    PrivateKey llavePrivada = RSA.cargarLlavePrivada(ruta:"Keys/PrivateKey.secret");

    double opsTotalSimetrico = 0;
    double opsTotalAsimetrico = 0;

    String csvFile = RESULTS_DIR + "processor_speed.csv";
```

Este método evalúa la velocidad del procesador al medir cuántas operaciones de cifrado por segundo puede realizar utilizando los algoritmos AES (simétrico) y RSA (asimétrico)

```
try (FileWriter writer = new FileWriter(csvFile)) {
    writer.write(str:"Repeticion,Simetrico_ops_s,Asimetrico_ops_s\n");

    for (int i = 1; i <= 5; i++) {
        System.out.println("Repetición " + i + ":");

        long inicioSimetrico = System.nanoTime();
        for (int j = 0; j < 1000; j++) {
            AES.encryptar(datos, claveAES, vectorInicial);
        }
        long finSimetrico = System.nanoTime();
        double tiempoSimetrico = (finSimetrico - inicioSimetrico) / 1_000_000_000.0;
        double opsSimetrico = 1000 / tiempoSimetrico;
        opsTotalSimetrico += opsSimetrico;
        System.out.println("Operaciones por segundo (Cifrado Simétrico): " + opsSimetrico + " ops/s");

        long inicioAsimetrico = System.nanoTime();
        Cipher cifradorRSA = Cipher.getInstance(transformation:"RSA");
        cifradorRSA.init(Cipher.ENCRYPT_MODE, llavePublica);
        for (int j = 0; j < 1000; j++) {
            cifradorRSA.doFinal(datos);
        }
        long finAsimetrico = System.nanoTime();
        double tiempoAsimetrico = (finAsimetrico - inicioAsimetrico) / 1_000_000_000.0;
        double opsAsimetrico = 1000 / tiempoAsimetrico;
        opsTotalAsimetrico += opsAsimetrico;
        System.out.println("Operaciones por segundo (Cifrado Asimétrico): " + opsAsimetrico + " ops/s");

        writer.write(String.format(format:"%d,%f,%f\n", i, opsSimetrico, opsAsimetrico));
    }

    double avgOpsSimetrico = opsTotalSimetrico / 5.0;
    double avgOpsAsimetrico = opsTotalAsimetrico / 5.0;
```

- Estimación de operaciones de cifrado por segundo

Iteration	Firma_ms	Cifrado_ms	Verificacion_ms
1	12	12	12
2	2	2	2
3	2	2	2
4	2	2	2
5	2	2	2
6	2	2	2
7	1	1	1
8	1	1	1
9	1	1	1
10	2	2	2
11	1	1	1
12	1	1	1
13	1	1	1
14	1	1	1
15	1	1	1
16	2	2	2
17	1	1	1
18	1	1	1
19	1	1	1
20	1	1	1
21	1	1	1
22	1	1	1
23	1	1	1



202313509

202315338

24	1	1	1
25	1	1	1
26	1	1	1
27	1	1	1
28	1	1	1
29	1	1	1
30	1	1	1
31	1	1	1
32	2	2	2
Average	159.375	159.375	159.375