TALLER UDP CONNECTION

LUNA CATALINA MARTÍNEZ VÁSQUEZ - A00401964

RENZO FERNANDO MOSQUERA DAZA – A00401681

SIMON REYES RIVEROS – A00400880

PRESENTADO A

NICOLAS SALAZAR ECHEVERRY

ASIGNATURA

COMPUTACIÓN EN INTERNET I

UNIVERSIDAD ICESI

CALI - VALLE DEL CAUCA – COLOMBIA

2025-01

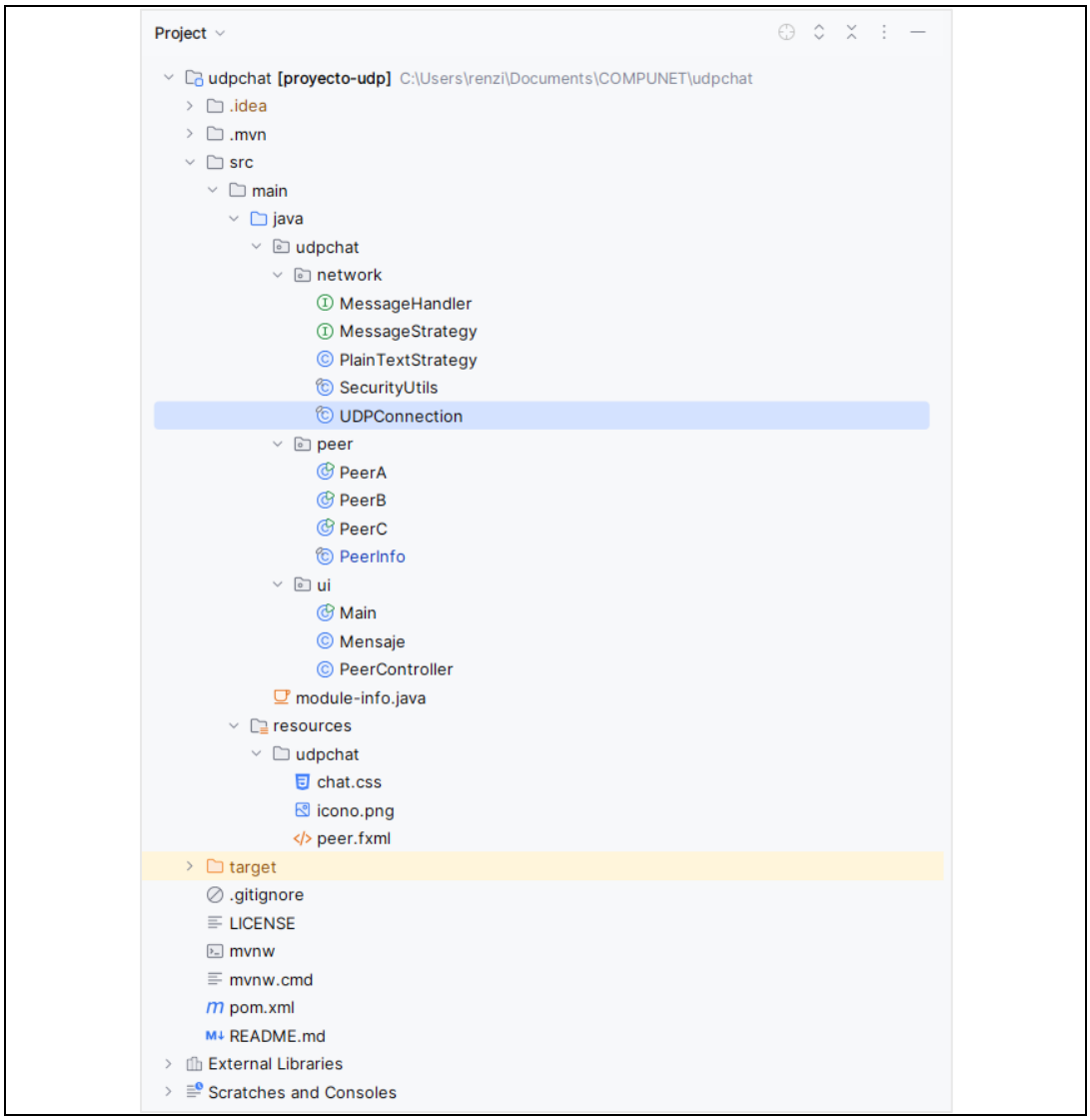
ENTREGA TALLER UDP CONNECTION

Repositorio guía: https://github.com/njse22/Computacion-I-2025-1/tree/master/05_udp_1

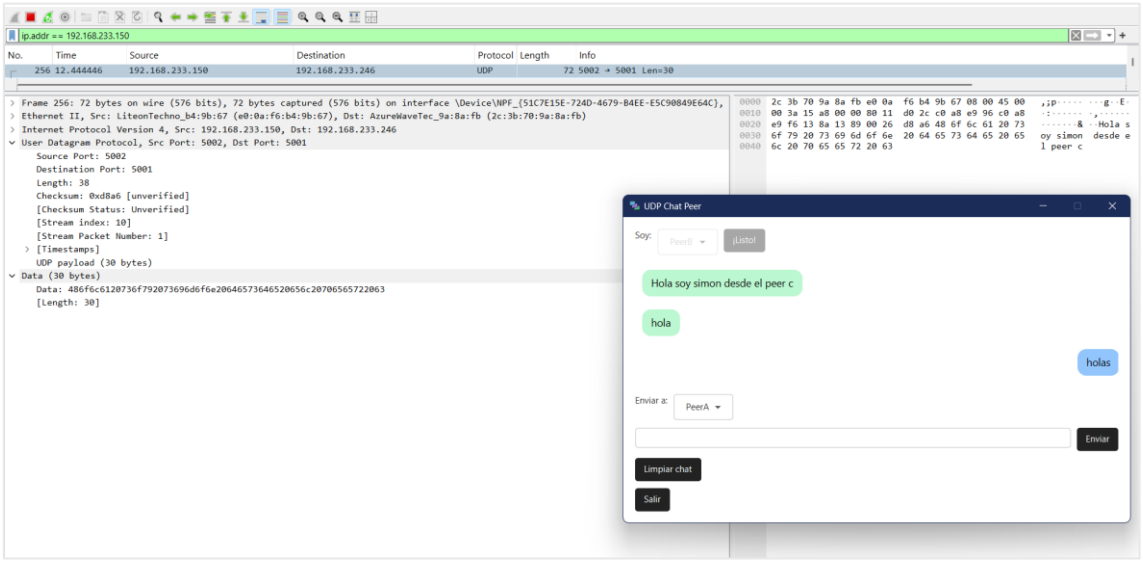
Video guía: https://youtu.be/sPIJm_dii1o?si=dp2zddqlIn6Uf2pJ

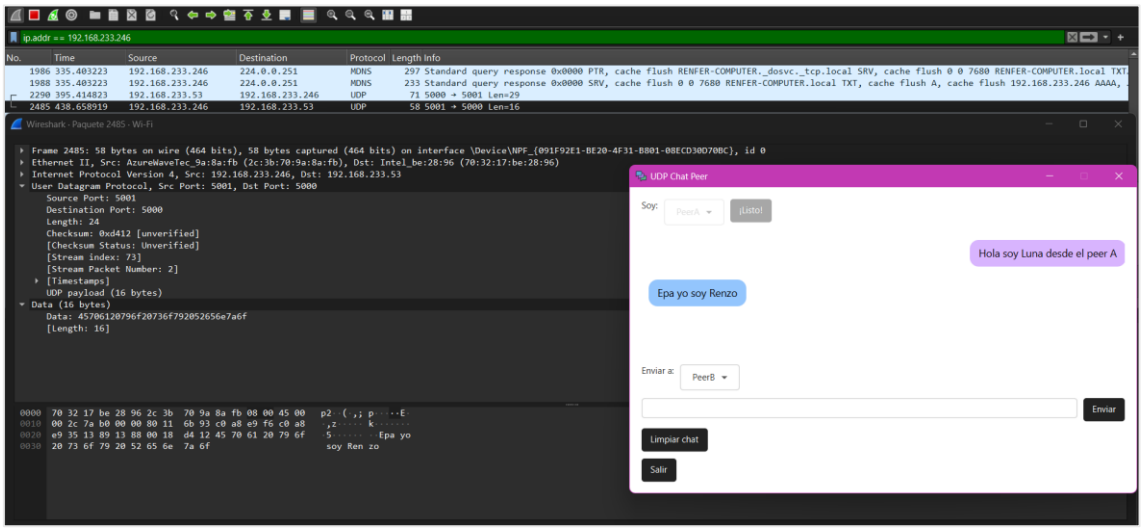
- **Código fuente del proyecto [25%]**
  Repositorio:
  https://github.com/RenzoFernando/udpchat

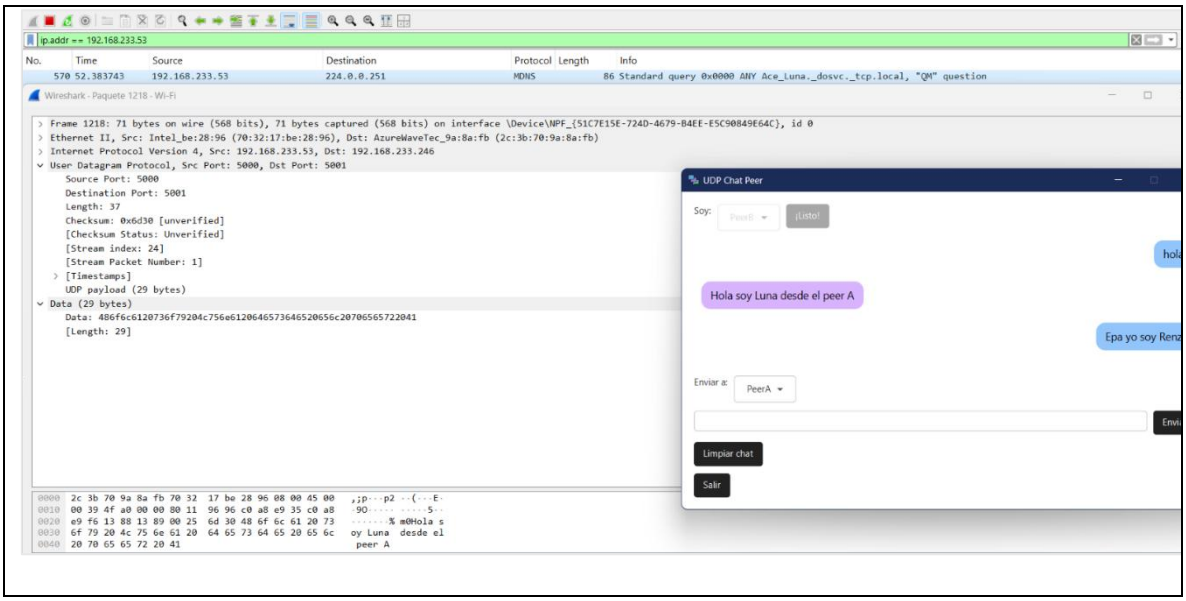
  Código:



- **Capturas de Wireshark de cada Peer el nombre de las capturas deberá tener el formato PeerX.pcapng, con base a estas capturas deben responder a estas preguntas:**
  - ***¿Es posible ver en la captura de Wireshark el contenido del mensaje enviado? [10%]***
  - ***¿Cuál es el checksum de la captura? Explique/investiguen porque este checksum? [10%]***







Después de una profunda investigación, encontramos que muchas tarjetas de red modernas no calculan el checksum UDP en el emomento de la captura, debido a que delegan este proceso al hardware o al sistema operativo. Por lo que, Wireshark captura el paquete antes de que se calcule el checksum real.

Además, Wireshark muestra el UDP payload en hexadecimal y ASCII (por ejemplo `48 6f 6c 61 …` → “Hola…”), así que sí, si se ve el contenido del mensaje. Se ve el campo Checksum marcado como unverified porque muchas NIC hacen checksum offload y lo calculan tras que Wireshark capture el paquete.

En resumen:

1. Sí. Como estamos usando UDP sin cifrar, Wireshark muestra directamente el payload.
2. El checksum de la captura para cada uno es:

Peer	***Checksum (capturado)***	***Comentario***
Peer C	0xd8a6 (unverified)	Mensaje “Hola soy simon…”
Peer A	0xd6d0 (unverified)	Mensaje “Hola soy Luna desde el peer A…”
Peer B	0xd4d2 (unverified)	Mensaje “Epa yo soy Renzo”

- **¿Qué patrones de diseño/arquitectura aplicaría al desarrollo de un programa basado en red como este? [15%]**

En el desarrollo de un programa basado en red, se aplicaron dos patrones de diseño fundamentales que aportan claridad, escalabilidad y robustez a la arquitectura del sistema:

1. **Patrón Singleton**
   El patrón Singleton se aplica a la clase UDPConnection, que representa el núcleo de la comunicación UDP. Garantiza que exista una única instancia de conexión a lo largo de toda la aplicación. En contextos de red, esto es importante porque:

   - Se evita el conflicto de múltiples sockets abiertos sobre el mismo puerto.
   - Se centraliza la recepción de mensajes y el control de hilos.
   - Se facilita el manejo de recursos compartidos (como los listeners y el pool de envío).
   - Se asegura una interfaz global para acceder a la conexión, manteniendo la coherencia y evitando errores por instancias duplicadas.

```
public final class UDPConnection implements Runnable {  RenzoFernando

    /* Singleton */
    private static final UDPConnection INSTANCIA = new UDPConnection();  1 usage

    public static UDPConnection get(){ return INSTANCIA; }  4 usages  RenzoFernando

    private UDPConnection(){}                    // nadie más instancia  1 usage  RenzoFernando
```

2. **Patrón Strategy:**
   La interfaz MessageStrategy define operaciones comunes (prepare y parse), y su implementación concreta —como PlainTextStrategy— encapsula la lógica de serialización/deserialización. Gracias a este patrón:

   - Es posible intercambiar dinámicamente el comportamiento del formato de mensaje (texto plano, cifrado, JSON, etc.).
   - Se desacopla la lógica de red de la lógica de codificación de mensajes, haciendo el sistema más limpio y mantenible.
   - Se favorece la extensibilidad, ya que nuevas estrategias pueden incorporarse sin modificar la clase UDPConnection.

```java
public class PlainTextStrategy implements MessageStrategy {  RenzoFernando

    @Override public byte[] prepare(String p){  1 usage  RenzoFernando
        return p.getBytes();
    }

    @Override public String  parse(byte[] d,int len){  1 usage  RenzoFernando
        return new String(d, offset: 0,len).trim();
    }

}
```

- **Modifique el código provisto de tal forma que: el hilo de recepción no'muera' una vez recibido el mensaje [10%]**

Para asegurar que el hilo de recepción UDP no finalice tras recibir un único mensaje, se implementó una estructura de ciclo en la clase UDPConnection, Esta clase sigue el patrón Singleton y maneja tanto el socket como el hilo receptor.

El método run(), que representa el hilo de recepción, fue diseñado para mantenerse en ejecución mientras la bandera vivo sea verdadera:

```
@Override public void run(){  RenzoFernando
    byte[] buf = new byte[2048];
    while(vivo){
        try{
            DatagramPacket p = new DatagramPacket(buf,buf.length);
            socket.receive(p);                                  // ← bloquea
            String msg = estrategia.parse(p.getData(), p.getLength());
            for(MessageHandler h : oyentes){                    // notifica listeners
                h.onMessage(msg, p.getAddress(), p.getPort());
            }
        }catch(IOException e){
            if(vivo) e.printStackTrace();
        }
    }
}
```

Este ciclo while(vivo) permite que el hilo continúe escuchando nuevos mensajes de forma indefinida, evitando así que se termine tras una única recepción. La bandera vivo se establece en true mediante el método iniciar(), el cual lanza el hilo por primera vez:

```
public void iniciar(){  4 usages  RenzoFernando
    if(!vivo){
        vivo = true;
        new Thread( task: this, name: "Hilo-RX-UDP").start();
    }
}
```

De esta forma, el hilo queda en espera continua de nuevos mensajes UDP y solo se detiene de manera controlada a través del método cerrar(), que actualiza la bandera y cierra el socket:

```
public void cerrar(){  1 usage  RenzoFernando
    vivo=false;
    if(socket!=null) socket.close();
    poolTX.shutdownNow();
}
```

- **Modifique el código provisto de tal forma que la lógica de transmisión de paquetes quede en un hilo aparte [10%]**

Para cumplir con el requerimiento de que la transmisión de mensajes UDP se realice en un hilo separado del principal, se empleó un ExecutorService configurado como newCachedThreadPool(). Esta estructura permite lanzar tareas en segundo plano de forma no bloqueante.

La lógica se encuentra en el método enviarAsync(...) de la clase UDPConnection:

```
public void enviarAsync(String msg,String ipDest,int puertoDest){  5 usages  RenzoFernando
    poolTX.execute(() -> {
        try{
            byte[] data = estrategia.prepare(msg);
            DatagramPacket p=new DatagramPacket(
                    data, data.length,
                    InetAddress.getByName(ipDest),
                    puertoDest);
            socket.send(p);
        }catch(IOException e){ e.printStackTrace(); }
    });
}
```

Esto asegura que cada mensaje enviado se procese en un hilo aparte, evitando bloqueos y manteniendo la responsividad de la interfaz gráfica. Con esta implementación, tanto la recepción como el envío funcionan en paralelo, cumpliendo con un diseño concurrente robusto.

- **Investiguen que modificaciones son necesarias para implementar este mismo sistema, pero para la comunicación TCP en java [10%]**

Para migrar nuestro chat de UDP a TCP, en lugar de usar DatagramSocket y DatagramPacket utilizaremos ServerSocket en el servidor y Socket en el cliente. El servidor crea un ServerSocket en el puerto configurado y llama a accept() para esperar conexiones entrantes; el cliente, por su parte, inicializa un Socket(host, puerto) antes de enviar mensajes. La comunicación ya no se basa en paquetes independientes, sino en flujos bidireccionales (InputStream/OutputStream o DataInputStream/DataOutputStream), lo que garantiza entrega ordenada, sin pérdidas ni duplicados.

Como TCP es un stream continuo, es necesario acordar un protocolo de delimitación (por ejemplo, mensajes terminados en \n o prefijos de longitud) para que el receptor sepa cuándo termina cada mensaje. Tanto en el cliente como en el servidor se recomienda crear hilos dedicados a lectura y escritura, de modo que readLine() o readUTF() puedan bloquearse sin impedir la interacción de la interfaz. Al finalizar, hay que cerrar siempre socket.close(), así como los streams, y capturar IOException para manejar reconexiones o notificar errores.

Opcionalmente, para minimizar la latencia de paquetes pequeños podemos desactivar el algoritmo Nagle con socket.setTcpNoDelay(true). Con estos cambios nuestro sistema evoluciona de un modelo “sin estado” basado en UDP a uno “orientado a conexión” con TCP, obteniendo un canal de comunicaciones más fiable y con control de errores y orden.

- **¿Qué utilidades de codificación o seguridad agregaría al código? [10%]**

Para reforzar la confidencialidad e integridad de los mensajes, podríamos emplear directamente las APIs de Java:

- **javax.crypto.Cipher.getInstance("AES/CBC/PKCS5Padding")** para el cifrado y descifrado AES,
- **javax.crypto.Mac.getInstance("HmacSHA256")** para generar y verificar firmas HMAC,
- **java.util.zip.CRC32** para un checksum ligero de transporte.

Aun así, para agrupar estas utilidades, se creó la clase **SecurityUtils,** esta clase:

SecurityUtils.java

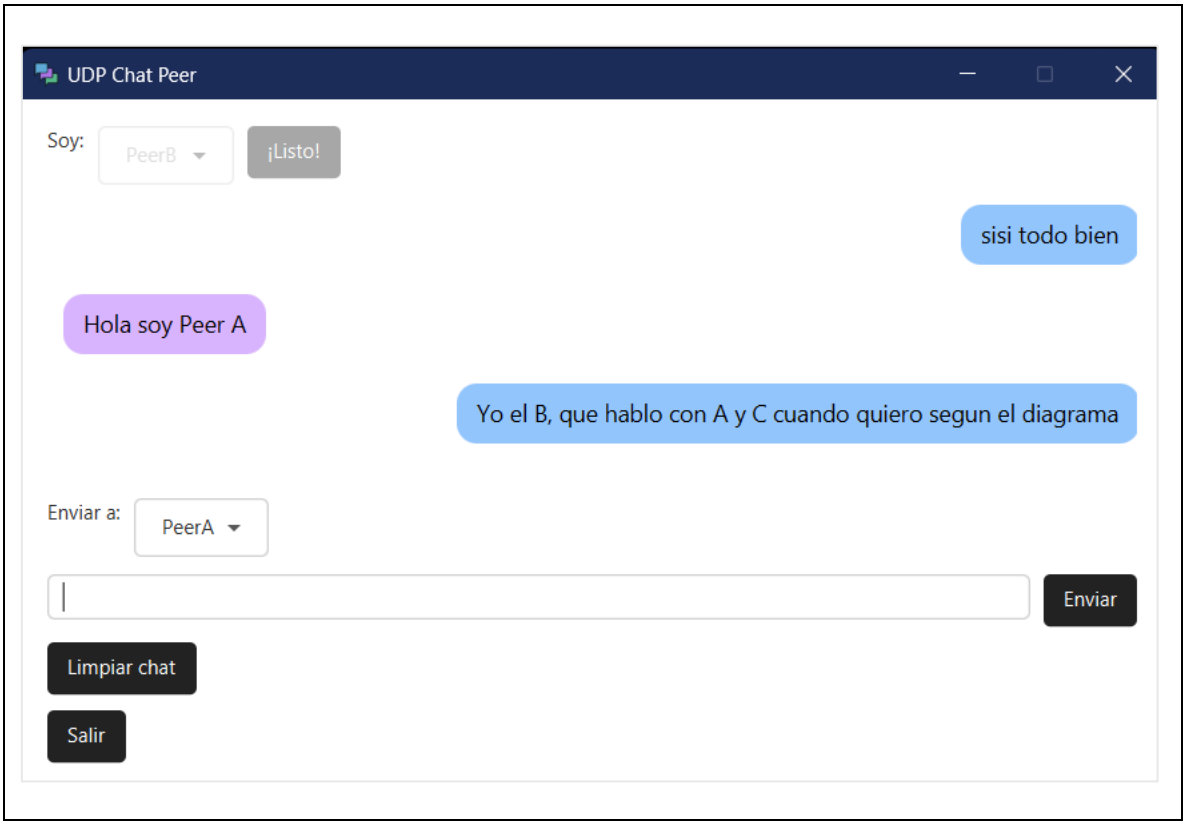
```java
package udpchat.network;

import javax.crypto.*;
import javax.crypto.spec.SecretKeySpec;
import java.util.Base64;
import java.util.zip.CRC32;

Edit | Explain | Test | Document | Fix
public final class SecurityUtils {  RenzoFernando

    private static final byte[] KEY = "1234567890123456".getBytes();  // 128 bit DEMO  3 usages

    private SecurityUtils(){}

    Edit | Explain | Test | Document | Fix
    /*  AES (simétrico)  */
    public static String encryptAES(String plain){...}
    Edit | Explain | Test | Document | Fix
    public static String decryptAES(String cipher){...}

    Edit | Explain | Test | Document | Fix
    /*  HMAC SHA-256  */
    public static String hmacSHA256(String msg){...}

    Edit | Explain | Test | Document | Fix
    /*  CRC-32 para verificar integridad  */
    public static long crc32(byte[] data){...}
}

```

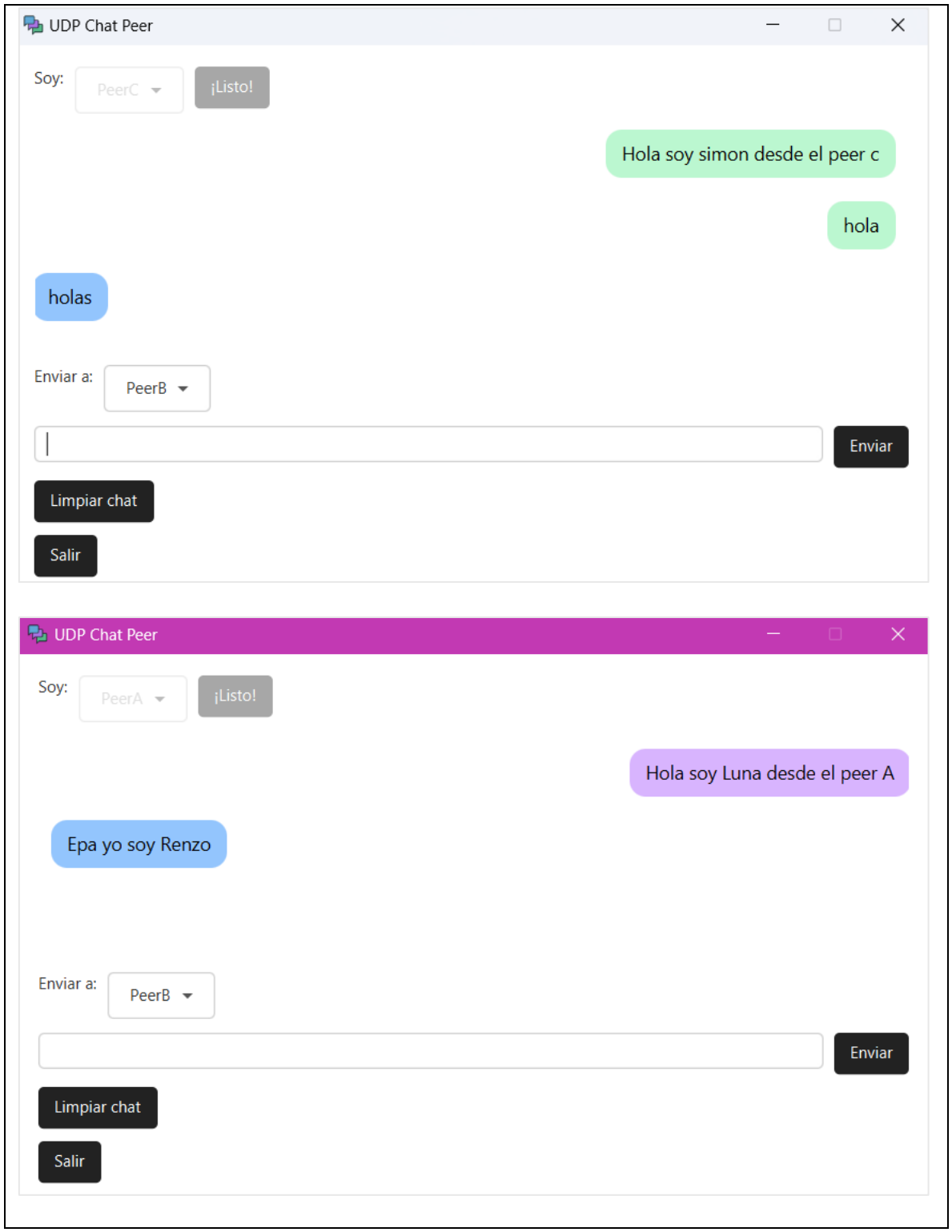
Que define métodos como encryptAES(), decryptAES(), hmacSHA256() y crc32(). Sin embargo, esta clase no llegó a enlazarse con la aplicación final, quedando pendiente su integración.

- **BONUS: desarrollen una interfaz de usuario en JavaFX para este programa [15%]**

Para mejorar la experiencia de usuario, se implementó una interfaz gráfica en JavaFX. Utilizando la clase Stage y Scene, se diseñó un layout con VBox y HBox.

La lógica de red (UDP) corre en hilos separados para no bloquear la UI, y las actualizaciones al ListView se realizan con Platform.runLater(). Además, se aplicaron estilos personalizados mediante un archivo CSS de JavaFX.




UDP Chat Peer
Soy:
PeerC
¡Listo!
Hola soy simon desde el peer c
hola
holas
Enviar a:
PeerB
Enviar
Limpiar chat
Salir
UDP Chat Peer
Soy:
PeerA
¡Listo!
Hola soy Luna desde el peer A
Epa yo soy Renzo
Enviar a:
PeerB
Enviar
Limpiar chat
Salir