Instituto Politécnico Nacional

ESCUELA SUPERIOR DE CÓMPUTO

Mini Red Social

Materia: Aplicaciones para Comunicaciones en Red

Profesor: Axel Ernesto Moreno Cervantes

Grupo: 6CM1

Integrantes:

Jiménez Rodríguez Alejandro Martin
Núñez Ramírez Valery Aylin

Introducción

El objetivo principal de esta práctica fue desarrollar una aplicación de chat funcional basada en el modelo **Cliente-Servidor**. A diferencia de las prácticas convencionales que suelen usar TCP , en este proyecto se tuvo el reto de utilizar **UDP (User Datagram Protocol)** a través de DatagramSocket que no garantiza la conexión.

Para lograr que el chat funcionara correctamente y cumpliera con los requerimientos (salas de chat, mensajes privados y transferencia de archivos), se tuvieron que integrar varios conceptos clave de la programación en red y sistemas concurrentes:

- **Sockets de Datagrama (UDP):** A diferencia de TCP, UDP es un protocolo "sin conexión". Esto significa que no garantiza si el envío de paquetes (datagramas) a una dirección IP y puerto específicos llegarán en orden o incluso si llegaran.
- **Arquitectura Cliente-Servidor:** El sistema se divide en dos partes. El **Servidor**, que actúa como un nodo central que recibe todos los mensajes y los redistribuye a los usuarios correspondientes ; y el **Cliente**, que es la interfaz que utiliza el usuario final.
- **Hilos y Concurrencia:** Este fue un punto crítico. Un chat necesita ser asíncrono; es decir, que para que sea un chat como los que estamos acostumbrados, es necesario que el usuario sea capaz de enviar mensajes y al mismo tiempo recibirlos. Para esto, se utilizó la clase Thread y la interfaz Runnable de Java, permitiendo que el cliente tenga un proceso escuchando (receive) y otro enviando (send) simultáneamente sobre el mismo socket.
- **Serialización y Protocolo de Aplicación:** Como UDP solo envía bytes, se diseñó un pequeño "protocolo" simple basado en cadenas de texto separadas por dos puntos (COMMAND:ARG1:ARG2), de tal forma que el servidor entienda si le estoy pidiendo entrar a una sala (JOIN), enviar un mensaje (MSG) o iniciar una transferencia de archivo (FILE_INIT).
- **Manejo de Archivos Binarios:** Para enviar imágenes y audios, se implementó una lógica de fragmentación. Dado que un paquete UDP tiene un límite de tamaño, se tuvo que

leer los archivos en pequeños trozos de bytes, enviarlos secuencialmente y ensamblarlos en el servidor.

Desarrollo

El desarrollo de la práctica se dividió en varias etapas lógicas, comenzando por el diseño de los datos y terminando con la implementación de la transferencia de archivos.

Estructura de Datos del Cliente

Lo primero fue definir cómo el servidor iba a recordar quiénes están conectados, ya que UDP no mantiene el estado de la conexión. Por lo anterior, se creó una clase auxiliar llamada ClientInfo.

Lo importante aquí fue implementar los métodos equals y hashCode. Sin esto, el servidor no podría distinguir correctamente si un cliente ya existe en una lista o eliminarlo correctamente cuando sale.

```
// Fragmento de ClientInfo.java
@Override
public boolean equals(Object o) {
   if (this == o) return true;
   // ... validaciones ...
   // Identificamos al cliente por su IP, Puerto y Nombre
   return port == that.port &&
         Objects.equals(username, that.username) &&
         Objects.equals(address, that.address);
}
```

El Servidor y el Manejo de Salas

El servidor (ServidorChat) funciona con un bucle infinito while(true). Su única tarea es esperar a recibir un paquete. Se utilizó un Map para gestionar las salas de chat, donde la clave es el nombre de la sala y el valor es un Set de clientes.

Cuando llega un paquete, solo se convierte a String y se analiza el comando. Aquí, se implementó un switch para manejar la lógica:

```
// Fragmento de ServidorChat.java
private void handleMessage(...) {
   // Se separa el mensaje usando ":" como delimitador
   String[] parts = message.split(":", 3);
   String command = parts[0];

   switch (command) {
      case "JOIN":
         handleJoin(parts[1], parts[2], address, port, socket);
         break;
      case "MSG":
         // Lógica para enviar mensaje a todos en la sala
         handlePublicMessage(...);
         break;
      // otros casos como PRIV, LEAVE, etc.
   }
}
```

El Cliente y la Concurrencia (Hilos)

En el lado del cliente (ClienteChat), el reto principal fue no bloquear la terminal. Si solo se usaba un hilo, cuando el programa ejecutaba socket.receive(), la consola se hubiese congelado esperando un mensaje y no hubiese dejado escribir.

Para solucionar esto, separamos las responsabilidades:

1. **Hilo Principal (Main):** Se encarga de leer lo que se escribe por el teclado y enviarlo al servidor.
2. **Hilo Receptor (Receiver):** Es una clase interna que implementa Runnable. Su trabajo es quedarse escuchando siempre.

```
// Fragmento del Hilo Receptor en ClienteChat
class Receiver implements Runnable {
   public void run() {
      while (true) {
         // El socket se bloquea aquí esperando mensajes del servidor
         socket.receive(packet);
         String msg = new String(packet.getData(), ...);
         System.out.println(msg); // Muestra el mensaje en cuanto llega
      }
   }
}
```

Implementación de Transferencia de Archivos

Esta fue la parte más compleja. Como UDP no garantiza el orden ni la llegada, y además tiene un límite de tamaño por paquete (MTU), no podíamos enviar una imagen de golpe.

Por lo tanto, se implementó lo siguiente:

1. **Protocolo de Inicio:** El cliente envía un comando FILE_INIT con un identificador único y el tamaño del archivo. El servidor prepara un flujo de salida (FileOutputStream) para guardar los bytes que llegarán.
2. **Fragmentación (Chunking):** El cliente lee el archivo en bloques de 974 bytes (para dejar espacio a las cabeceras UDP) y los envía uno por uno.

```
// Fragmento de envío de archivos en ClienteChat
try (FileInputStream fis = new FileInputStream(file)) {
   byte[] buffer = new byte[MAX_UDP_PAYLOAD];
   while ((bytesRead = fis.read(buffer)) != -1) {
      // Se crea un encabezado con el ID de transferencia
      String chunkHeader = transferId + ":CHUNK_DATA:";
      // Se concatena el encabezado con los bytes del archivo y envío
      sendPacket(fullChunkData);
   }
}
```

3. **Reensamblaje:** El servidor recibe estos paquetes. Detecta que son datos binarios, extrae los bytes "limpios" y los escribe en el archivo correspondiente en el disco duro.

Conclusión

Valery Aylin Nuñez Ramirez

Realizar este proyecto fue un reto bastante interesante porque me obligó a entender cómo funcionan las redes "a bajo nivel". Diría que fue un proyecto que involucró bastantes conceptos y partes, por lo que mantener un orden fue bastante importante.

Lo más difícil fue darme cuenta de que, al usar DatagramSocket, el servidor realmente no "conoce" a los clientes; por eso tuvimos que implementar manualmente toda la lógica para guardar las IPs y puertos en un Map para simular que estaban conectados a una sala.

Al final, me quedo con la idea de que, aunque UDP es muy rápido y útil para cosas simples, intentar hacer una transferencia de archivos fiable sobre él es complicado porque te toca programar a mano todo lo que TCP ya hace automáticamente (como el orden y la integridad de los paquetes). Fue una gran práctica para valorar lo que ocurre detrás de las aplicaciones que usamos a diario.

Alejandro Martin Jiménez Rodríguez

Lo que más destaco de esta práctica es haber aprendido a implementar correctamente el multihilo (Threads) en una aplicación real.

También me pareció muy interesante cómo tuvimos que crear nuestro propio "protocolo" usando cadenas de texto separadas por dos puntos (como MSG:sala:usuario) para que el cliente y el servidor se entendieran. Esto demostró que la comunicación en red se basa en cómo estructurar los datos que se envían.

En resumen, logramos crear una arquitectura Cliente-Servidor funcional donde múltiples usuarios pueden interactuar en tiempo real. Ver cómo el servidor gestiona los mensajes y los archivos y los reparte a las salas correctas me ayudó a consolidar mis conocimientos sobre Sockets y concurrencia en Java.