Instituto Politécnico Nacional

Escuela Superior de Cómputo

“Control de flujo”

Asignatura:

Aplicaciones para comunicaciones en red

Alumnos:

León Flores Daniela Alejandra Ocaña

Castro Héctor Rigel

Profesor:

Moreno Cervantes Axel

Grupo: 6CM2

Introducción

El proyecto “Transmisión Musical” tiene como objetivo aplicar los conocimientos sobre comunicación cliente-servidor mediante el uso de sockets TCP en el lenguaje de programación Java. La práctica consiste en diseñar e implementar un sistema capaz de transmitir archivos de audio desde un servidor central hacia uno o varios clientes conectados a la red, simulando el comportamiento básico de un servicio de streaming musical.

A través de esta práctica, se busca comprender de forma más profunda el manejo de flujos de datos, la sincronización de conexiones y la importancia de la integridad en la transferencia de información binaria.

El estudio de los sockets TCP resulta fundamental en la ingeniería de software enfocada en redes, ya que son la base de la mayoría de los sistemas distribuidos modernos. Esta práctica demuestra cómo, a través de la programación en Java, es posible construir una arquitectura cliente-servidor robusta, confiable y escalable. Además, se fortalece la comprensión del modelo OSI y de los mecanismos de transporte que permiten una comunicación ordenada y libre de pérdidas entre procesos, elementos indispensables para el desarrollo de servicios de red profesionales.

Desarrollo

El sistema “Transmisión Musical” se compone de dos entidades principales: el Servidor y el Cliente.

El Servidor es el encargado de administrar las conexiones entrantes y distribuir los datos del archivo de audio. Se implementa utilizando la clase ServerSocket, que permite escuchar solicitudes en un puerto definido. Una vez aceptada la conexión, el servidor abre un flujo de salida (OutputStream) y comienza a enviar los bytes del archivo de audio seleccionado.

El manejo de flujos binarios en Java es especialmente importante para asegurar que el contenido se envíe en bloques ordenados y completos, evitando errores durante la transferencia y manteniendo la integridad del archivo.

Por su parte, el Cliente establece la conexión con el servidor mediante un objeto Socket, especificando la dirección IP y el puerto del servidor. Una vez conectados, el cliente recibe los datos a través de un flujo de entrada (InputStream) y los procesa en tiempo real. El diseño modular del cliente permite la posibilidad de reproducir el audio o guardarlo localmente para su posterior uso.

Además, se incluye el manejo de excepciones y validaciones para detectar cortes de conexión o transmisiones incompletas, lo que hace al sistema más robusto ante fallos de red o desconexiones inesperadas.

Durante el desarrollo del código, se emplearon conceptos clave como la lectura y escritura en buffers, el control de errores mediante excepciones, el uso de hilos (threads) para la gestión de múltiples clientes y el cierre seguro de sockets.

Se realizaron diversas pruebas controladas para observar la eficiencia del sistema, verificando la velocidad de transmisión y la sincronización entre los procesos de envío y recepción. Este enfoque permitió comprobar cómo los sockets TCP aseguran la fiabilidad en la comunicación incluso bajo condiciones de tráfico moderado o de latencia variable.

También se documentó el comportamiento del sistema ante escenarios con múltiples clientes simultáneos, lo que evidenció la necesidad de implementar mecanismos de concurrencia controlada en el servidor. Este hallazgo refuerza el valor de las estructuras de control y los bloqueos de sincronización en entornos de red.

```java
import java.io.File;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.util.Arrays;
import java.util.Map;
import java.util.Scanner;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

import javax.sound.sampled.AudioInputStream;
import javax.sound.sampled.AudioSystem;
import javax.sound.sampled.Clip;

public class Cliente {
    private DatagramSocket socket;
    private InetAddress multicastGroup;
    private Clip audioClip;
    private boolean isPlaying = false;

    public static class Mensaje {
        private int sequenceNumber;
        private int ackNumber;
        private String comando;
        private long timestamp;

        public Mensaje(int seq, int ack, String cmd) {
            this.sequenceNumber = seq;
```

```
            this.ackNumber = ack;
            this.comando = cmd;
            this.timestamp = System.currentTimeMillis();
        }

        public int getSequenceNumber() { return sequenceNumber; }
        public int getAckNumber() { return ackNumber; }
        public String getComando() { return comando; }
        public long getTimestamp() { return timestamp; }
        public void updateTimestamp() { this.timestamp =
System.currentTimeMillis(); }

        public byte[] toBytes() {
            String data = sequenceNumber + ":" + ackNumber + ":" +
comando;
            return data.getBytes();
        }

        public static Mensaje fromBytes(byte[] data) {
            try {
                String str = new String(data).trim();
                String[] parts = str.split(":", 3);
                if (parts.length == 3) {
                    return new Mensaje(
                        Integer.parseInt(parts[0]),
                        Integer.parseInt(parts[1]),
                        parts[2]
                    );
                }
            } catch (Exception e) {
                System.err.println("Error parseando mensaje: " + new
String(data));
            }
            return null;
        }
    }

    // Ventana deslizante
    private int nextSeqNumber = 0;
    private int windowSize = 6;
    private Map<Integer, Mensaje> sentMessages = new
ConcurrentHashMap<>();
    private ScheduledExecutorService scheduler =
Executors.newScheduledThreadPool(1);
```

```
    private int lastAckReceived = -1;
    private int reenvios = 0;

    // Buffer para respuestas del servidor
    private final Object responseLock = new Object();
    private String ultimaRespuesta = "";

    public Cliente() {
        try {
            socket = new DatagramSocket();
            multicastGroup = InetAddress.getByName("ff3e:40:2001::1");

            System.out.println("Cliente de Audio con Ventana Deslizante
iniciado...");
            System.out.println("Tamaño de ventana: " + windowSize);

            cargarAudioLocal();
            startClient();

        } catch (Exception e) {
            System.err.println("Error iniciando cliente: " +
e.getMessage());
        }
    }

    private void cargarAudioLocal() {
        try {
            File audioFile = new File("Prueba.wav");
            if (!audioFile.exists()) {
                System.err.println("Archivo de audio no encontrado
localmente");
                System.err.println("Coloca 'Prueba.wav' en la misma
carpeta que el cliente");
                return;
            }

            AudioInputStream audioStream =
AudioSystem.getAudioInputStream(audioFile);
            audioClip = AudioSystem.getClip();
            audioClip.open(audioStream);
            System.out.println("Audio cargado localmente");

        } catch (Exception e) {
            System.err.println("Error cargando audio local: " +
e.getMessage());
```

```
        }
    }

    private void startClient() {
        new Thread(this::recibirRespuestas).start();
        new Thread(this::verificarTimeouts).start();
        enviarComandos();
    }

    private void demostrarVentanaDeslizante() {
    System.out.println("\n=== DEMOSTRACION VENTANA DESLIZANTE ===");
    System.out.println("Enviando 6 comandos SIN esperar ACKs...");
    System.out.println("Tamaño de ventana: " + windowSize);

    String[] comandos = {"PLAY", "PAUSE", "STOP", "RESTART", "STATUS",
"PAUSE"};

    for (int i = 0; i < comandos.length; i++) {
        String comando = comandos[i];

        // Ejecutar localmente
        procesarComandoLocal(comando);

        // Pequeña pausa para ver los mensajes (100ms)
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {}
    }
}

    private void enviarComandoRapido(String comando) {
        // Versión simplificada que solo envía, no ejecuta localmente
        if (sentMessages.size() >= windowSize) {
            System.out.println("VENTANA LLENA - No se puede enviar: " +
comando);
            return;
        }

        int currentSeq = nextSeqNumber++;
        Mensaje mensaje = new Mensaje(currentSeq, lastAckReceived,
comando);

        try {
            byte[] datos = mensaje.toBytes();
```

```
            DatagramPacket packet = new DatagramPacket(datos,
datos.length, multicastGroup, 7777);
            socket.send(packet);

            sentMessages.put(currentSeq, mensaje);
            System.out.println("Enviado: " + comando + " [Seq:" +
currentSeq + ", Ventana:" + sentMessages.size() + "/" + windowSize + "]");

        } catch (Exception e) {
            System.err.println("Error enviando comando: " +
e.getMessage());
        }
    }

    private void recibirRespuestas() {
        try {
            byte[] buffer = new byte[1024];
            System.out.println("Escuchando ACKs del servidor...");

            while (true) {
                DatagramPacket packet = new DatagramPacket(buffer,
buffer.length);
                socket.receive(packet);

                Mensaje respuesta =
Mensaje.fromBytes(Arrays.copyOf(packet.getData(), packet.getLength()));
                if (respuesta != null) {
                    procesarACK(respuesta);
                }
            }
        } catch (Exception e) {
            System.err.println("Error recibiendo respuestas: " +
e.getMessage());
        }
    }

    private void procesarACK(Mensaje ack) {
        int ackNumber = ack.getAckNumber();

        synchronized (responseLock) {
            System.out.println("\n=== RESPUESTA DEL SERVIDOR ===");
            System.out.println("ACK recibido para secuencia: " +
ackNumber);

            if (ackNumber > lastAckReceived) {
```

```
                lastAckReceived = ackNumber;
            }

            int antes = sentMessages.size();
            sentMessages.entrySet().removeIf(entry -> entry.getKey() <=
ackNumber);
            int despues = sentMessages.size();

            if (antes != despues) {
                System.out.println("Ventana liberada: " + despues + "/" +
windowSize + " mensajes en vuelo");
            }

            if (!ack.getComando().startsWith("ACK:")) {
                String mensajeServidor = ack.getComando().replace("ACK:",
"");
                System.out.println("Servidor: " + mensajeServidor);
                ultimaRespuesta = mensajeServidor;
            }
            System.out.println("==============================\n");
        }

        // Mostrar prompt después de procesar la respuesta
        mostrarPrompt();
    }

    private void verificarTimeouts() {
        scheduler.scheduleAtFixedRate(() -> {
            long currentTime = System.currentTimeMillis();
            sentMessages.forEach((seq, msg) -> {
                if (currentTime - msg.getTimestamp() > 2000) {
                    synchronized (responseLock) {
                        System.out.println("\n=== TIMEOUT ===");
                        System.out.println("Timeout para secuencia " + seq
+ ", reenviando...");
                        System.out.println("================\n");
                    }
                    reenviarMensaje(msg);
                }
            });
        }, 0, 1, TimeUnit.SECONDS);
    }

    private void reenviarMensaje(Mensaje mensaje) {
```

```
        try {
            mensaje.updateTimestamp();

            byte[] datos = mensaje.toBytes();
            DatagramPacket packet = new DatagramPacket(datos,
datos.length, multicastGroup, 7777);
            socket.send(packet);

            reenvios++;
            synchronized (responseLock) {
                System.out.println("\n=== REENVIO ===");
                System.out.println("Reenviado: " + mensaje.getComando() +
" [Seq:" + mensaje.getSequenceNumber() + "] (Reenvio #" + reenvios + ")");
                System.out.println("================\n");
            }

        } catch (Exception e) {
            System.err.println("Error reenviando mensaje: " +
e.getMessage());
        }
    }

    private void enviarComandoConVentana(String comando) {
        if (sentMessages.size() >= windowSize) {
            System.out.println("Ventana llena (" + sentMessages.size() +
"/" + windowSize + "), esperando ACKs...");
            return;
        }

        int currentSeq = nextSeqNumber++;
        Mensaje mensaje = new Mensaje(currentSeq, lastAckReceived,
comando);

        try {
            byte[] datos = mensaje.toBytes();
            DatagramPacket packet = new DatagramPacket(datos,
datos.length, multicastGroup, 7777);
            socket.send(packet);

            sentMessages.put(currentSeq, mensaje);
            System.out.println("Enviado: " + comando + " [Seq:" +
currentSeq + ", Ventana:" + sentMessages.size() + "/" + windowSize + "]");

        } catch (Exception e) {
```

```
            System.err.println("Error enviando comando: " +
e.getMessage());
        }
    }

    private void mostrarPrompt() {
        synchronized (responseLock) {
            System.out.print("Ingresa comando: ");
        }
    }

    private void enviarComandos() {
        Scanner scanner = new Scanner(System.in);

        System.out.println("\n=== CONTROL DE AUDIO (VENTANA DESLIZANTE)
===");
        System.out.println("Comandos disponibles:");
        System.out.println("PLAY    - Reproducir audio LOCAL");
        System.out.println("PAUSE   - Pausar audio LOCAL");
        System.out.println("STOP    - Detener audio LOCAL");
        System.out.println("RESTART - Reiniciar audio LOCAL");
        System.out.println("STATUS  - Estado del audio LOCAL");
        System.out.println("TEST    - Probar ventana deslizante (6
comandos rapidos)");
        System.out.println("EXIT    - Salir del cliente");
        System.out.println("==========================================\n
");

        // Mostrar primer prompt
        System.out.print("Ingresa comando: ");

        while (true) {
            String comando = scanner.nextLine().trim();

            if ("EXIT".equalsIgnoreCase(comando)) {
                break;
            }

            if (!comando.isEmpty()) {
                procesarComandoLocal(comando);
            }

            // No mostrar prompt aquí - se mostrará después de cada
respuesta
```

```
        }

        scanner.close();
        scheduler.shutdown();
        try {
            if (!scheduler.awaitTermination(2, TimeUnit.SECONDS)) {
                scheduler.shutdownNow();
            }
        } catch (InterruptedException e) {
            scheduler.shutdownNow();
        }
        if (audioClip != null) audioClip.close();
        socket.close();
        System.out.println("Cliente terminado. Total reenvios: " +
reenvios);
    }

        private void procesarComandoLocal(String comando) {
        String comandoUpper = comando.toUpperCase();

        // Caso especial para TEST
        if ("TEST".equals(comandoUpper)) {
            demostrarVentanaDeslizante();
            return;
        }

        // Ejecutar localmente inmediatamente
        switch (comandoUpper) {
            case "PLAY":
                reproducirAudioLocal();
                break;
            case "PAUSE":
                pausarAudioLocal();
                break;
            case "STOP":
                detenerAudioLocal();
                break;
            case "RESTART":
                reiniciarAudioLocal();
                break;
            case "STATUS":
                mostrarEstadoLocal();
                break;
            case "MUTE":
                System.out.println("Comando MUTE ejecutado localmente");
```

```
                break;
            default:
                System.out.println("Comando no reconocido: " + comando);
                System.out.print("Ingresa comando: ");
                return;
        }

        // Enviar comando al servidor con ventana deslizante
        enviarComandoConVentana(comandoUpper);
    }

    private void reproducirAudioLocal() {
        if (audioClip == null) {
            System.out.println("Error: Audio no disponible localmente");
            return;
        }
        if (!isPlaying) {
            audioClip.start();
            isPlaying = true;
            System.out.println("Reproduciendo audio LOCAL");
        } else {
            System.out.println("Audio ya se esta reproduciendo
LOCALMENTE");
        }
    }

    private void pausarAudioLocal() {
        if (audioClip == null) {
            System.out.println("Error: Audio no disponible localmente");
            return;
        }
        if (isPlaying) {
            audioClip.stop();
            isPlaying = false;
            System.out.println("Audio LOCAL pausado");
        } else {
            System.out.println("Audio LOCAL ya esta pausado");
        }
    }

    private void detenerAudioLocal() {
        if (audioClip == null) {
            System.out.println("Error: Audio no disponible localmente");
            return;
```

```
        }
        audioClip.stop();
        audioClip.setFramePosition(0);
        isPlaying = false;
        System.out.println("Audio LOCAL detenido y reiniciado");
    }

    private void reiniciarAudioLocal() {
        if (audioClip == null) {
            System.out.println("Error: Audio no disponible localmente");
            return;
        }
        audioClip.setFramePosition(0);
        if (!isPlaying) {
            audioClip.start();
            isPlaying = true;
            System.out.println("Audio LOCAL reiniciado y reproduciendo");
        } else {
            System.out.println("Audio LOCAL reiniciado (continuando
reproduccion)");
        }
    }

    private void mostrarEstadoLocal() {
        if (audioClip == null) {
            System.out.println("Estado: Audio no cargado localmente");
        } else {
            String estado = isPlaying ? "Reproduciendo LOCALMENTE" :
"Pausado LOCALMENTE";
            long posicion = audioClip.getMicrosecondPosition() / 1000000;
            long duracion = audioClip.getMicrosecondLength() / 1000000;
            System.out.println(estado + " | Tiempo: " + posicion + "/" +
duracion + "s");
        }
    }

    public static void main(String[] args) {
        new Cliente();
    }
}
```

```
import java.net.DatagramPacket;
import java.net.InetAddress;
import java.net.MulticastSocket;
```

```
import java.util.Arrays;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

public class Servidor {
    private MulticastSocket serverSocket;
    private InetAddress multicastGroup;
    private int nextExpectedSeq = 0;
    private Map<Integer, String> receivedCommands = new
ConcurrentHashMap<>();

    public Servidor() {
        try {
            multicastGroup = InetAddress.getByName("ff3e:40:2001::1");
            serverSocket = new MulticastSocket(7777);
            serverSocket.joinGroup(multicastGroup);
            serverSocket.setReuseAddress(true);
            serverSocket.setTimeToLive(255);

            System.out.println("Servidor de Audio con Ventana Deslizante
iniciado...");
            System.out.println("Grupo: " + multicastGroup + ", Puerto:
7777");
            System.out.println("Esperando comandos con ventana
deslizante...");

            listenForCommands();

        } catch (Exception e) {
            System.err.println("Error iniciando servidor: " +
e.getMessage());
            e.printStackTrace();
        }
    }

    public static class Mensaje {
        private int sequenceNumber;
        private int ackNumber;
        private String comando;
        private long timestamp;

        public Mensaje(int seq, int ack, String cmd) {
            this.sequenceNumber = seq;
            this.ackNumber = ack;
```

```
            this.comando = cmd;
            this.timestamp = System.currentTimeMillis();
        }

        public int getSequenceNumber() { return sequenceNumber; }
        public int getAckNumber() { return ackNumber; }
        public String getComando() { return comando; }
        public long getTimestamp() { return timestamp; }

        public byte[] toBytes() {
            String data = sequenceNumber + ":" + ackNumber + ":" +
comando;
            return data.getBytes();
        }

        public static Mensaje fromBytes(byte[] data) {
            try {
                String str = new String(data).trim();
                String[] parts = str.split(":", 3);
                if (parts.length == 3) {
                    return new Mensaje(
                        Integer.parseInt(parts[0]),
                        Integer.parseInt(parts[1]),
                        parts[2]
                    );
                }
            } catch (Exception e) {
                System.err.println("Error parseando mensaje: " + new
String(data));
            }
            return null;
        }
    }

    private void listenForCommands() {
        try {
            byte[] buffer = new byte[1024];

            while (true) {
                DatagramPacket packet = new DatagramPacket(buffer,
buffer.length);
                serverSocket.receive(packet);

                Mensaje mensaje =
Mensaje.fromBytes(Arrays.copyOf(packet.getData(), packet.getLength()));
```

```
                if (mensaje != null) {
                    procesarMensaje(mensaje, packet.getAddress(),
packet.getPort());
                } else {
                    System.err.println("Mensaje no valido recibido");
                }
            }
        } catch (Exception e) {
            System.err.println("Error recibiendo comandos: " +
e.getMessage());
        }
    }

    private void procesarMensaje(Mensaje mensaje, InetAddress
clientAddress, int clientPort) {
        int seqNumber = mensaje.getSequenceNumber();
        String comando = mensaje.getComando();

        System.out.println("Mensaje recibido: " + comando + " [Seq:" +
seqNumber + "]");

        if (seqNumber == nextExpectedSeq) {
            System.out.println("Secuencia esperada, procesando
inmediatamente...");
            procesarYResponderComando(comando, clientAddress, clientPort,
seqNumber);
            nextExpectedSeq++;

            procesarBufferComandos(clientAddress, clientPort);

        } else if (seqNumber > nextExpectedSeq) {
            receivedCommands.put(seqNumber, comando);
            System.out.println("Comando almacenado en buffer [Seq:" +
seqNumber + "]");
            System.out.println("Buffer size: " + receivedCommands.size());

            enviarACK(nextExpectedSeq - 1, "BUFFERED", clientAddress,
clientPort);

        } else {
            System.out.println("Comando duplicado [Seq:" + seqNumber + "],
reenviando ACK");
            enviarACK(seqNumber, "DUPLICADO", clientAddress, clientPort);
        }
```

```
    }

    private void procesarBufferComandos(InetAddress clientAddress, int
clientPort) {
        while (receivedCommands.containsKey(nextExpectedSeq)) {
            String comando = receivedCommands.remove(nextExpectedSeq);
            System.out.println("Procesando comando del buffer [Seq:" +
nextExpectedSeq + "]");
            procesarYResponderComando(comando, clientAddress, clientPort,
nextExpectedSeq);
            nextExpectedSeq++;
        }
    }

    private void procesarYResponderComando(String comando, InetAddress
clientAddress, int clientPort, int seqNumber) {
        String respuesta = "";

        switch (comando.toUpperCase()) {
            case "CONNECT":
                respuesta = "Conectado - Servidor listo";
                break;
            case "PLAY":
                respuesta = "Comando PLAY recibido y procesado";
                break;
            case "PAUSE":
                respuesta = "Comando PAUSE recibido y procesado";
                break;
            case "STOP":
                respuesta = "Comando STOP recibido y procesado";
                break;
            case "RESTART":
                respuesta = "Comando RESTART recibido y procesado";
                break;
            case "STATUS":
                respuesta = "Servidor funcionando - Esperando comandos";
                break;
            default:
                respuesta = "Comando no reconocido: " + comando;
        }

        enviarACK(seqNumber, respuesta, clientAddress, clientPort);
        System.out.println("Procesado: " + comando + " [Seq:" + seqNumber
+ "] -> " + respuesta);
    }
```

```
    private void enviarACK(int ackNumber, String mensaje, InetAddress
clientAddress, int clientPort) {
        try {
            Mensaje ack = new Mensaje(0, ackNumber, "ACK:" + mensaje);
            byte[] ackData = ack.toBytes();
            DatagramPacket ackPacket = new DatagramPacket(ackData,
ackData.length, clientAddress, clientPort);
            serverSocket.send(ackPacket);
            System.out.println("ACK enviado: " + ackNumber + " -> " +
mensaje);
        } catch (Exception e) {
            System.err.println("Error enviando ACK: " + e.getMessage());
        }
    }

    public static void main(String[] args) {
        new Servidor();
    }
}
```

Conclusión

La práctica “Transmisión Musical” permitió reforzar los conocimientos teóricos sobre comunicación entre procesos en red, aplicándolos en un contexto práctico mediante la programación en Java.

El desarrollo de un sistema cliente-servidor funcional demostró la importancia del protocolo TCP para garantizar una transmisión confiable de datos binarios, y permitió comprender a profundidad el uso de las clases de red en Java (Socket, ServerSocket, InputStream, OutputStream), junto con la correcta gestión de los flujos de entrada y salida.

A nivel formativo, la práctica fortaleció las habilidades de razonamiento lógico, diseño modular, programación concurrente y depuración de red.

También destacó la relevancia de los mecanismos de sincronización entre procesos y del manejo adecuado de excepciones para asegurar un funcionamiento estable y predecible.

En conjunto, la experiencia consolidó la comprensión del modelo Cliente-Servidor y sentó las bases para el desarrollo de aplicaciones distribuidas más complejas, tales como servicios de streaming multimedia o sistemas de transferencia segura de archivos en tiempo real.

Esta práctica demostró que la teoría de redes y los fundamentos de los protocolos de transporte pueden materializarse de forma tangible en un entorno de programación moderno, abriendo el camino hacia aplicaciones escalables y confiables en el ámbito de las comunicaciones digitales.

Referencias

- Oracle. (2024). *Java Platform, Standard Edition – Networking*. Documentación oficial.
- González, J. D. M. (2021). *Sockets (Cliente-Servidor) en Java*.
- Universidad de Cantabria. (s.f.). *Manual de Sockets en C*.
- Panzarasa, N. (2021). *Socket Library in C* [Repositorio GitHub].
- Programarya. (2024). *Comunicación en red con sockets en Java*.