INSTITUTO POLITÉCTICO NACIONAL

Escuela Superior de Cómputo

Carrera:

Ingeniería en Sistemas Computacionales

Unidad de aprendizaje:

Sistemas Distribuidos

Practica 3:

Modelo Multicliente - Servidor

Alumno:

Cardoso Osorio Atl Yosafat

Profesor:

Chadwick Carreto Arellano

Fecha:

09/03/2025

INSTITUTO POLITÉCNICO NACIONAL

ESCOM

Índice de contenido

Índice de figuras

Antecedentes

El modelo cliente-servidor es una arquitectura fundamental en la computación distribuida, y su diseño puede variar según la distribución de responsabilidades entre el cliente y el servidor, así como la cantidad de capas o niveles que lo componen.

Modelo Cliente/Servidor

El modelo Cliente/Servidor es una arquitectura clave en el desarrollo de sistemas informáticos y redes, permitiendo la comunicación eficiente entre dispositivos mediante la diferenciación de roles. En este esquema, el servidor es el encargado de gestionar y ofrecer recursos, mientras que el cliente es quien realiza peticiones para acceder a dichos servicios. Esta estructura facilita la administración centralizada, optimiza el acceso a la información y permite distribuir eficientemente las cargas de trabajo en entornos computacionales.

Desde sus inicios, el concepto Cliente/Servidor ha sido esencial en la evolución de la informática. Durante las primeras etapas de la computación, los sistemas centralizados dominaban el panorama con el uso de mainframes, donde múltiples terminales se conectaban a un único sistema de procesamiento. Con el avance de la tecnología y la aparición de computadoras personales, la arquitectura Cliente/Servidor se convirtió en la solución ideal para distribuir tareas entre dispositivos, mejorando la eficiencia y flexibilidad en los sistemas informáticos. Actualmente, esta arquitectura es la base de diversas aplicaciones, desde plataformas web hasta servicios en la nube y videojuegos en línea.

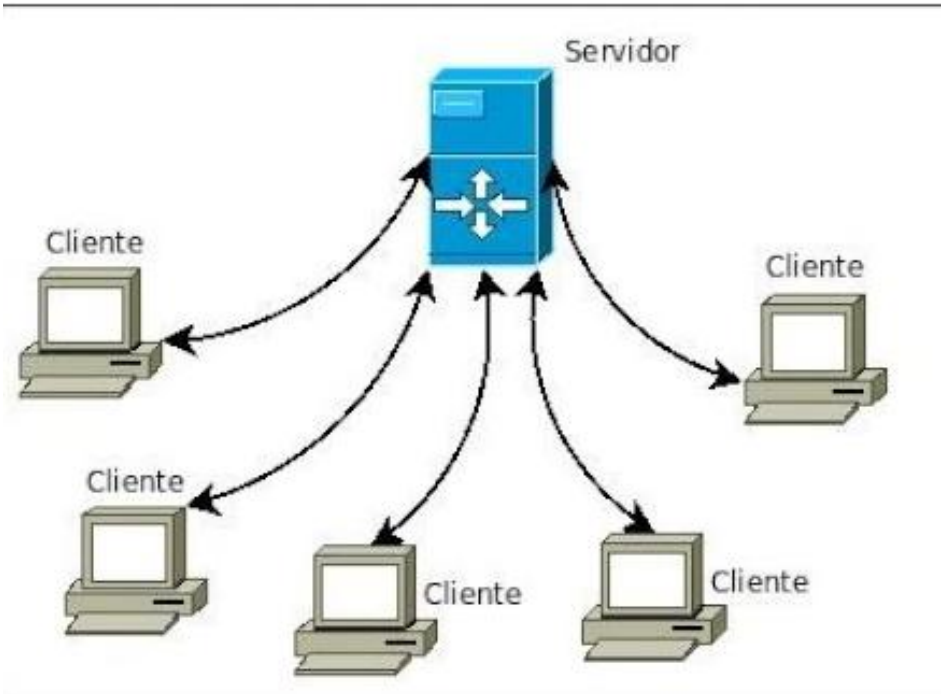


Figura 1. Diagrama de la arquitectura Cliente/Servidor

Tipos de modelos Cliente/Servidor

El modelo de dos capas (two-tier) es la forma más básica de esta arquitectura. En este enfoque, solo existen dos componentes principales: el cliente y el servidor. El cliente se encarga de la interfaz de usuario y la presentación de datos, mientras que el servidor gestiona la lógica de negocio y el almacenamiento de la información. Este modelo es sencillo de implementar y mantener, lo que lo hace ideal para aplicaciones pequeñas o medianas. Sin embargo, su escalabilidad es limitada, ya que el servidor puede sobrecargarse si hay una gran cantidad de clientes conectados. Un ejemplo común de este modelo es una aplicación de escritorio que se conecta a una base de datos centralizada.

En contraste, el modelo de tres capas (three-tier) introduce una capa intermedia, conocida como middleware, entre el cliente y el servidor. Esta capa adicional se encarga de procesar las solicitudes del cliente y aplicar las reglas de negocio, separando así las responsabilidades en tres niveles: la capa de presentación (cliente), la capa de lógica de negocio (middleware) y la capa de datos (servidor). Este enfoque ofrece una mayor escalabilidad y flexibilidad, ya que permite una separación clara de funciones y facilita el mantenimiento y las actualizaciones. No obstante, su implementación es más compleja y costosa debido a la infraestructura adicional requerida. Un ejemplo típico es una aplicación web en la que el navegador (cliente) interactúa con un servidor de aplicaciones (middleware) que, a su vez, se conecta a una base de datos.

El modelo de N capas (N-tier) es una extensión del modelo de tres capas, donde las responsabilidades se dividen en múltiples niveles para mejorar la modularidad y escalabilidad. En este enfoque, las capas pueden incluir, además de las tres mencionadas, una capa de servicios (como APIs o servicios web) y otras capas especializadas según las necesidades de la aplicación. Este modelo es altamente escalable y modular, lo que lo hace ideal para aplicaciones empresariales complejas, como sistemas ERP (Enterprise Resource Planning). Sin embargo, su implementación es más costosa y requiere un diseño cuidadoso para evitar cuellos de botella.

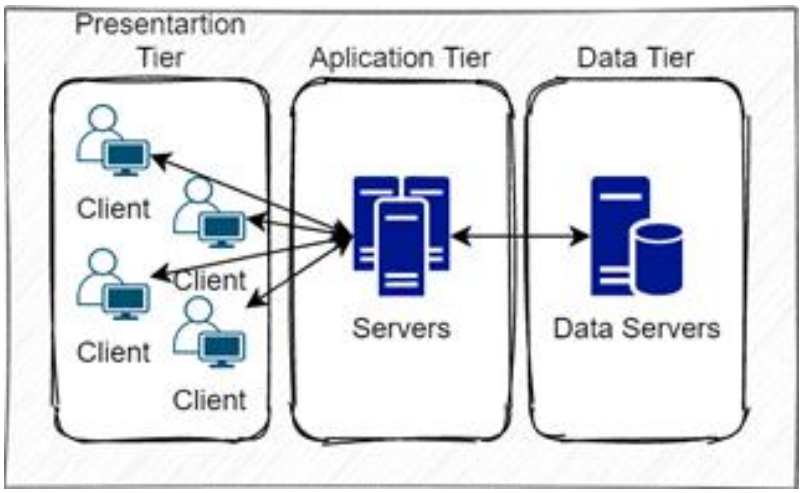


Figura 2. Modelo de N capas Cliente/Servidor

Protocolos de comunicación

Los sistemas cliente-servidor dependen en gran medida de los protocolos de comunicación para facilitar la transferencia de datos entre dispositivos. Estos protocolos establecen las reglas y los procedimientos para la transmisión de información, asegurando que la comunicación sea eficiente, confiable y segura. Algunos de los protocolos más importantes aplicados en estos sistemas:

TCP/IP (Transmission Control Protocol/Internet Protocol): Este protocolo es fundamental en Internet y en los sistemas cliente-servidor. TCP garantiza una conexión confiable al dividir los datos en paquetes y asegurarse de que se entreguen correctamente, mientras que IP se encarga de direccionar los paquetes a través de la red.

HTTP (Hypertext Transfer Protocol): Ampliamente utilizado en la web, HTTP define cómo se comunican los navegadores web y los servidores. Establece un formato para las solicitudes y respuestas, permitiendo la transferencia de recursos como páginas web, imágenes y otros archivos.

FTP (File Transfer Protocol): Especializado en la transferencia de archivos, FTP facilita la carga y descarga de archivos entre un cliente y un servidor. Proporciona un método seguro y controlado para gestionar archivos remotos.

SMTP (Simple Mail Transfer Protocol): Esencial para el envío de correos electrónicos, SMTP define cómo se transmiten los mensajes de correo entre servidores. Asegura que los correos se entreguen de manera confiable y eficiente.

SSH (Secure Shell): Utilizado para acceder de forma segura a servidores remotos, SSH cifra la comunicación entre el cliente y el servidor, protegiendo los datos sensibles durante la transmisión.

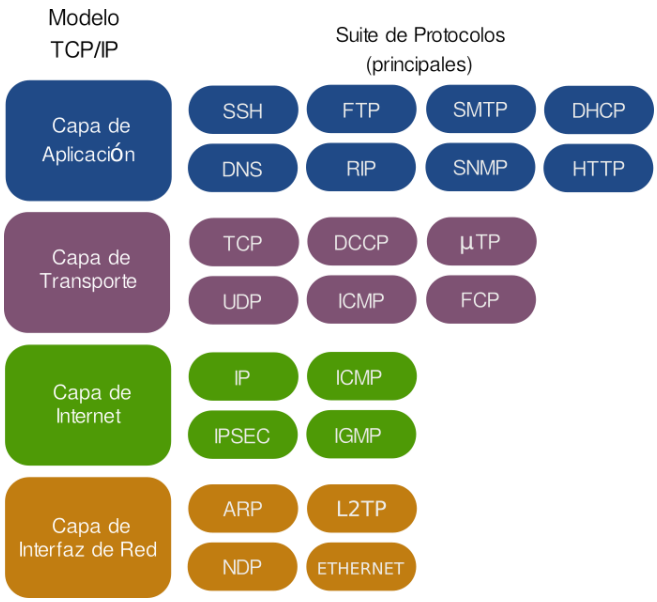


Figura 3. Diagrama de protocolos de comunicación

Estos protocolos son fundamentales para el funcionamiento efectivo de los sistemas cliente-servidor, ya que permiten la comunicación fluida y segura entre dispositivos. Desde la transferencia de archivos hasta el intercambio de correos electrónicos, estos protocolos juegan un papel crucial en numerosos aspectos de la interacción en línea. Su implementación adecuada garantiza una experiencia de usuario óptima y protege la integridad de los datos en todo momento. En general los protocolos de comunicación son el corazón de los sistemas cliente-servidor, proporcionando el marco necesario para la transmisión confiable de información en esta era, sin embargo, para el desarrollo de esta práctica únicamente nos enfocamos en el protocolo FTP y TCP.

Socket

Un socket, en el contexto de las redes informáticas, es un concepto fundamental que sirve como interfaz de comunicación entre dos programas que se ejecutan en la red. Esencialmente, actúa como un punto final de una conexión entre dos nodos de una red, permitiendo la transferencia de datos de manera bidireccional.

Este término proviene del mundo físico, donde un socket es el lugar donde se conectan los dispositivos eléctricos para recibir energía o comunicarse. En el ámbito de las redes informáticas, un socket es una abstracción que representa un extremo de una conexión de red, permitiendo que los programas se comuniquen entre sí, ya sea en la misma máquina (comunicación interna) o en diferentes máquinas (comunicación externa a través de la red).

Los sockets son una parte integral de la arquitectura de red de los sistemas informáticos modernos y son utilizados por una amplia gama de aplicaciones, desde navegadores web hasta aplicaciones de mensajería instantánea y servidores de archivos. Facilitan la comunicación entre procesos de manera eficiente y flexible, lo que permite la creación de aplicaciones y sistemas distribuidos complejos.

En términos de programación, los sockets se implementan mediante API (Interfaz de Programación de Aplicaciones) proporcionadas por el sistema operativo. Estas API permiten a los desarrolladores crear y gestionar sockets, establecer conexiones, enviar y recibir datos, y cerrar conexiones cuando ya no son necesarias.

Los sockets pueden ser de diferentes tipos, dependiendo del protocolo de transporte que utilicen. Los más comunes son los sockets de flujo (stream sockets) y los sockets de datagrama (datagram sockets). Los sockets de flujo utilizan el protocolo TCP (Protocolo de Control de Transmisión) para establecer una conexión orientada a la conexión y garantizar la entrega ordenada de datos, mientras que los sockets de datagrama utilizan el protocolo UDP (Protocolo de Datagrama de Usuario) y ofrecen una comunicación sin conexión y no garantizan la entrega ordenada de datos.

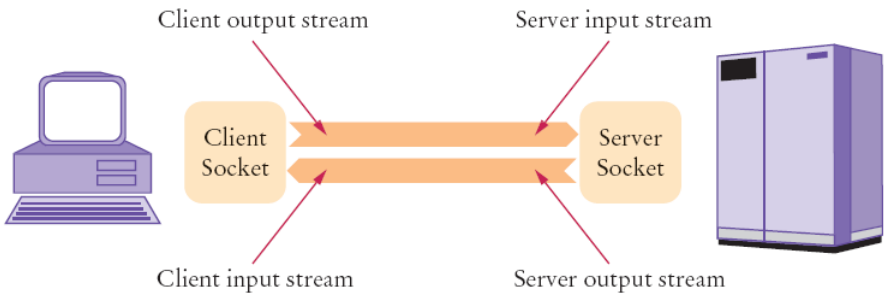


Figura 4. Diagrama de Sockets Cliente/Servidor

En pocas palabras un socket es una interfaz de comunicación que permite a los programas intercambiar datos a través de una red, ya sea en la misma máquina o en máquinas remotas. Es una abstracción fundamental en el desarrollo de aplicaciones de red y proporciona la base para la implementación de diversos servicios y protocolos de comunicación. Sin los sockets, la comunicación entre programas en una red sería extremadamente difícil, si no imposible, lo que subraya su importancia en el ámbito de la informática moderna.

Planteamiento problema

Esta práctica en contexto de los sistemas distribuidos, se propone el desarrollo de una aplicación multicliente para la gestión de boletos en un cine. Esta aplicación se basa en un servidor que maneja solicitudes concurrentes de múltiples clientes, permitiendo consultar la disponibilidad de boletos, realizar compras y mantener un control de estos. El sistema está compuesto por un servidor centralizado y múltiples clientes que interactúan con él de forma simultánea, simulando el comportamiento de un cine real que ofrece boletos a los clientes en un entorno distribuido.

El reto principal de esta práctica es diseñar un sistema capaz de manejar múltiples clientes simultáneamente, garantizando que los accesos a los recursos compartidos (boletos) se gestionen correctamente para evitar condiciones de carrera y asegurar que el número de boletos disponibles sea coherente en todo momento. Para ello, se hace uso de un modelo multihilo en el servidor, donde cada cliente es manejado por un hilo independiente, lo que permite que el sistema sea altamente concurrente.

El problema que se aborda en esta práctica está relacionado con la gestión de recursos compartidos en un entorno multicliente, en el que se debe garantizar la integridad de los datos (boletos) en presencia de múltiples solicitudes simultáneas, así como la correcta comunicación entre el cliente y el servidor.

Propuesta de solución

La propuesta de solución para la práctica se basa en la implementación de una arquitectura cliente-servidor en la que el servidor centraliza la gestión de los boletos de cine, mientras que múltiples clientes pueden interactuar con el servidor simultáneamente para consultar la

disponibilidad de boletos, realizar compras y finalizar sus sesiones. El sistema debe garantizar la correcta administración de los recursos compartidos, en este caso los boletos, de manera eficiente y segura.

En primer lugar, el servidor debe ser capaz de manejar múltiples clientes de forma concurrente, para lo cual se utiliza un modelo multihilo. Cada cliente que se conecta al servidor será atendido por un hilo independiente, lo que asegura que el servidor pueda procesar varias solicitudes al mismo tiempo sin bloquearse. El servidor debe implementar mecanismos de sincronización para asegurar que el número de boletos disponibles sea consistente y evitar condiciones de carrera, por lo que las operaciones sobre los boletos (consultar, comprar, reponer) se gestionan mediante métodos sincronizados.

El sistema de servidor y clientes también debe ser capaz de manejar posibles errores y cerrar las conexiones de manera segura. Si un cliente se desconecta o el servidor experimenta algún error, la conexión se cierra correctamente, y el cliente recibe la información adecuada. El servidor es capaz de seguir atendiendo a nuevos clientes, lo que asegura la continuidad del servicio sin interrupciones.

La solución propuesta se enfoca principalmente en aprovechar las técnicas de programación concurrente mediante el uso de hilos, lo que permite gestionar las conexiones simultáneas de múltiples clientes de manera eficiente. Al implementar una sincronización adecuada en el acceso a los recursos compartidos, el sistema asegura que los boletos no se vendan de manera incorrecta, y al mismo tiempo, ofrece una experiencia fluida y en tiempo real a los usuarios. Con esta implementación, se garantiza que el sistema pueda manejar un alto número de conexiones sin afectar el rendimiento, lo que demuestra la capacidad del sistema para ser escalable y eficiente en un entorno distribuido.

Materiales y métodos empleados

Para llevar a cabo la práctica y desarrollar la simulación de la venta de boletos con un modelo Cliente/Servidor se emplearon las siguientes herramientas y métodos:

Materiales

1. *Lenguaje de programación:* ***Java***

Se eligió Java debido a su robusto manejo de hilos y sus herramientas integradas para la concurrencia, como la palabra clave synchronized. Por otro lado, para la creación de la interfaz gráfica del cliente, se utilizó la librería Swing de Java. Swing permite crear interfaces gráficas interactivas con componentes como botones, cuadros de texto, áreas de texto, etc., facilitando la interacción del usuario con el sistema.

2. *Entorno de desarrollo:* ***Visual Studio Code (VS Code)***

Se empleo este entorno de desarrollo debido a que es bastante cómodo trabajar en él, así como también es muy personalizable y bastante útil para desarrollar código.

3. ***JDK (Java Development Kit):***

Se usó el JDK para compilar y ejecutar el programa.

4. *Sistema Operativo:* ***Windows***

La práctica se desarrolló y ejecutó en un sistema operativo Windows, debido a que es el que se emplea más a menudo y el más cómodo e intuitivo para trabajar.

Métodos

Manejo de concurrencia con hilos

Dado que el servidor necesita atender a múltiples clientes simultáneamente, se implementó el uso de hilos para manejar las conexiones concurrentes. El servidor emplea un ThreadPoolExecutor, que gestiona un conjunto de hilos para asegurar que cada cliente sea atendido de manera independiente, optimizando el uso de los recursos del sistema.

> ***Hilos para Clientes:*** Cada cliente es atendido por un hilo independiente, representado por la clase HiloCliente. Este hilo gestiona la interacción con un cliente específico, recibiendo y enviando mensajes, y procesando las solicitudes realizadas por el cliente.
>
> ***Hilos del Servidor:*** El servidor por su parte crea un pool de hilos (con ThreadPoolExecutor), lo que permite manejar múltiples clientes sin bloquear el sistema. Esto facilita una atención eficiente y concurrente a todos los usuarios.

Programación con Sockets TCP

Se utilizó la API de sockets de Java para establecer la comunicación entre el cliente y el servidor mediante el protocolo TCP. Este protocolo garantiza una transmisión confiable de datos, asegurando que los mensajes enviados entre el cliente y el servidor lleguen de manera íntegra y en orden.

> ***Servidor:*** El servidor utiliza un ServerSocket en el puerto 8888, que se mantiene escuchando constantemente las conexiones entrantes. Al recibir una solicitud de conexión de un cliente, el servidor acepta la conexión con serverSocket.accept() y crea un nuevo Socket para gestionar la comunicación con ese cliente.
>
> ***Cliente:*** El cliente se conecta al servidor utilizando la clase Socket y establece una conexión con la dirección IP y el puerto especificado. Para la comunicación, se usan los flujos de entrada (BufferedReader) y salida (PrintWriter), que permiten el intercambio de datos entre el cliente y el servidor de manera eficiente.

Sincronización de recursos compartidos

En el sistema, el stock de boletos disponibles es un recurso compartido entre los diferentes clientes. Para evitar problemas de concurrencia, como condiciones de carrera, se implementaron métodos sincronizados (synchronized) en la clase Cine para garantizar que el acceso a los boletos sea seguro y consistente.

Implementación de un productor (reposición de boletos) y consumidores (clientes)

Siguiendo el modelo Productor-Consumidor, se implementó un hilo adicional denominado "ReposicionBoletos", que simula el proceso de reposición de boletos cada cierto tiempo. Este hilo es responsable de aumentar el stock de boletos disponibles para los clientes.

Manejo de Entrada y Salida de Datos

El manejo adecuado de la entrada y salida de datos es fundamental tanto en el servidor como en el cliente para permitir una interacción efectiva.

> ***Servidor*:** El servidor utiliza BufferedReader para recibir datos de los clientes y PrintWriter para enviar respuestas. El servidor proporciona un menú interactivo y recibe las elecciones de los clientes, procesando cada opción de manera adecuada.
>
> ***Cliente:*** En el cliente, se ha diseñado una interfaz gráfica en Java Swing que facilita la interacción del usuario mediante botones y cuadros de texto, mejorando la experiencia de uso en comparación con una aplicación basada únicamente en línea de comandos.

Desarrollo

En esta práctica se implementó un sistema de venta de boletos para un cine utilizando el modelo cliente-servidor en Java, solamente que en este caso se utilizó el tipo de modelo Multicliente/Servidor. El sistema consta de un servidor que administra los boletos disponibles y varios clientes que pueden consultarlos, comprarlos y recibir confirmaciones en tiempo real. Además, se implementa una funcionalidad de reposición automática de boletos cada cierto tiempo. También este sistema emplea comunicación mediante sockets TCP, permitiendo a múltiples clientes conectarse simultáneamente a un servidor que gestiona las solicitudes de compra de boletos. Por otro lado, cuenta con una interfaz gráfica en Java Swing, lo que facilita la interacción del usuario con la plataforma.

Primeramente, se realizó la clase cine, la cual representa el recurso compartido entre los clientes, es decir el servidor. Su principal función es administrar la cantidad de boletos disponibles y gestionar las compras de manera segura y sincronizada.

```
import java.io.*;
import java.net.*;
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadPoolExecutor;
```

Figura 5. Bibliotecas empleadas

El atributo “boletosDisponibles” almacena la cantidad de boletos en stock. La sincronización en el método “comprarBoleto” garantiza que solo un hilo pueda modificar la cantidad de boletos a la vez, evitando problemas de concurrencia. Adicionalmente, se proporciona un método para reponer boletos cada cierto tiempo.

```
class Cine {
    private int boletosDisponibles;

    public Cine(int boletosDisponibles) {
        this.boletosDisponibles = boletosDisponibles;
    }

    public synchronized String comprarBoleto(String cliente, int cantidad) {
        if (boletosDisponibles < cantidad) {
            return "No hay suficientes boletos. Espere a la próxima función.";
        }
        boletosDisponibles -= cantidad;
        System.out.println(cliente + " compró " + cantidad + " boletos. Boletos restantes: " + boletosDisponibles);
        return "Compra exitosa. Boletos restantes: " + boletosDisponibles;
    }

    public synchronized int getBoletosDisponibles() {
        return boletosDisponibles;
    }

    public synchronized void reponerBoletos(int cantidad) {
        boletosDisponibles += cantidad;
        notifyAll();
        System.out.println("Se han repuesto " + cantidad + " boletos. Boletos disponibles: " + boletosDisponibles);
    }
}
```

Figura 6. Class cine y metodo de boletos disponibles

Para evitar que los boletos se agoten permanentemente, se implementó un mecanismo de reposición automática mediante la clase “ReposicionBoletos”. Este hilo repone boletos cada 20 segundos para simular la llegada de nuevas funciones en el cine, asegurando que los clientes tengan nuevas oportunidades de compra. Aquí es donde se maneja el hilo para la reposición de boletos, que es importante para no quedarnos sin boletos que vender.

```
class ReposicionBoletos extends Thread { // Este metodo se encarga de reponer los boletos
    private final Cine cine; // Se crea un objeto de la clase Cine

    public ReposicionBoletos(Cine cine) {// Se crea un constructor que recibe un objeto de la clase Cine
        this.cine = cine;
    }

    @Override
    public void run() { // Se crea un metodo run
        while (true) {
            try { // Se utiliza un try-catch para manejar excepciones
                Thread.sleep(millis:20000); // Se duerme el hilo por 20 segundos    Thread.sleep called in loop
                cine.reponerBoletos(cantidad:10); // Se llama al metodo reponerBoletos de la clase Cine
                System.out.println("Se han repuesto 10 boletos. Boletos disponibles: " + cine.getBoletosDisponibles());
            } catch (InterruptedException e) {
                System.err.println("Error al reponer boletos: " + e.getMessage());
            }
        }
    }
}
```

Figura 7. Clase para manejar la reposición de boletos

La clase HiloCliente es un componente clave en el servidor, ya que permite gestionar la comunicación con múltiples clientes de manera simultánea mediante hilos. Su propósito principal es recibir solicitudes de los clientes, procesarlas y devolver respuestas adecuadas, garantizando una interacción fluida con el sistema de compra de boletos de un cine.

Al iniciar, el hilo establece la comunicación con el cliente a través de un socket, permitiendo el intercambio de mensajes. Primero, recibe el nombre del cliente y le envía un mensaje de bienvenida. Luego, entra en un bucle donde espera recibir opciones del cliente, procesando cada solicitud según corresponda.

```
class HiloCliente extends Thread { // Se crea una clase HiloCliente que extiende de Thread
    private final Socket socket;
    private final Cine cine;

    public HiloCliente(Socket socket, Cine cine) {
        this.socket = socket;
        this.cine = cine;
    }

    @Override
    public void run() { // Se crea un metodo run
        try (
            BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            PrintWriter out = new PrintWriter(socket.getOutputStream(), autoFlush:true)
        ) {
            String nombreCliente = in.readLine(); // Se lee el nombre del cliente
            System.out.println(nombreCliente + " se ha conectado desde " + socket.getInetAddress());
            out.println("Bienvenido al Cine, " + nombreCliente + "!");

            boolean seguir = true; // Se crea una variable booleana seguir
            while (seguir) {
                String opcionStr = in.readLine(); // Se lee la opcion del cliente
                if (opcionStr == null) break;

                int opcion; // Se crea una variable opcion
                try {
                    opcion = Integer.parseInt(opcionStr);
                } catch (NumberFormatException e) {
                    out.println(x:"Opción no válida.");
                    continue;
                }
```

Figura 8. Primera sección de la clase HiloCliente

El cliente puede elegir entre tres acciones principales: consultar la cantidad de boletos disponibles, comprar boletos o salir del sistema. Para la compra, se verifica que el usuario ingrese un número válido y se intenta completar la transacción. Si la cantidad de boletos solicitada es mayor a la disponible, se informa al cliente.

El código también maneja posibles errores, como entradas no válidas o problemas en la conexión, evitando que el servidor se vea afectado por fallas en la comunicación. Además, cuando el cliente decide salir, el hilo finaliza y libera los recursos utilizados.

Gracias a esta implementación, el servidor puede atender a varios clientes de forma independiente, asegurando que cada usuario tenga una experiencia adecuada al interactuar con el sistema de compra de boletos.

```
        switch (opcion) {
            case 1 -> out.println("Boletos disponibles: " + cine.getBoletosDisponibles());
            case 2 -> {
                String cantidadStr = in.readLine();
                int cantidad;
                try {
                    cantidad = Integer.parseInt(cantidadStr);
                } catch (NumberFormatException e) {
                    out.println(x:"Cantidad no válida.");
                    continue;
                }
                out.println(cine.comprarBoleto(nombreCliente, cantidad));
            }
            case 3 -> {
                out.println(x:"Gracias por su compra. ¡Hasta luego!");
                seguir = false;
                System.out.println(nombreCliente + " ha salido.");
            }
            default -> out.println(x:"Opción no válida.");
        }
    }
} catch (IOException e) { // Se utiliza un catch para manejar excepciones
    System.err.println("Error en la conexión con el cliente: " + e.getMessage());
} finally {
    try { // Se utiliza un try-catch para manejar excepciones
        socket.close(); // Se cierra el socket
    } catch (IOException e) {
        System.err.println("Error al cerrar el socket: " + e.getMessage());
    }
```

Figura 9. Segunda sección de la clase HiloCliente

Por último, de la parte del servidor tenemos a la clase ServidorCine, la cual se encarga de iniciar el servidor y administrar las conexiones de los clientes. Utiliza un ServerSocket que escucha en el puerto 8888, así como también se especifica la IP del servidor para que cualquier cliente en red pueda acceder a él, en este apartado también lo podemos cambiar por el localhost si es que solo deseamos similar el sistema en la misma computadora sin necesidad de conectar otra y, al recibir una conexión, asigna un hilo HiloCliente a cada nuevo cliente. Además, se utiliza un ThreadPoolExecutor para administrar la concurrencia de manera eficiente y limitar la cantidad de clientes atendidos simultáneamente.

```java
public class ServidorCine { // Se crea una clase ServidorCine
    Run | Debug | Run main | Debug main
    public static void main(String[] args) { // Se crea un metodo main
        String ipServidor = "192.168.1.74"; //localhost";   // Se crea una variable ipServidor
        int puerto = 8888; // Se crea una variable puerto
        Cine cine = new Cine(boletosDisponibles:30); // Se crea un objeto de la clase Cine
        ThreadPoolExecutor pool = (ThreadPoolExecutor) Executors.newFixedThreadPool(nThreads:4);

        new ReposicionBoletos(cine).start(); // Se crea un objeto de la clase ReposicionBoletos y

        try (ServerSocket serverSocket = new ServerSocket(puerto)) {
            System.out.println("Servidor de cine iniciado en " + ipServidor + ":" + puerto);

            while (true) { // Se crea un ciclo while
                Socket socketCliente = serverSocket.accept();
                pool.execute(new HiloCliente(socketCliente, cine));
            }
        } catch (IOException e) { // Se utiliza un catch para manejar excepciones
            System.err.println("Error en el servidor: " + e.getMessage());
```

Figura 10.Clase principal del servidor

Por otro lado, tenemos al cliente, representado por la clase ClienteCine, establece una conexión a un servidor para realizar acciones como consultar la disponibilidad de boletos, comprarlos o salir de la sesión. Utiliza sockets para la comunicación entre el cliente y el servidor, y una interfaz visual permite al usuario interactuar de manera sencilla.

En primer lugar, el programa importa las bibliotecas necesarias para la interfaz gráfica (javax.swing.* y java.awt.*), la entrada y salida de datos (java.io.*), y la gestión de conexiones de red (java.net.*). Estas bibliotecas permiten manejar tanto la comunicación con el servidor como la presentación visual del cliente.

```java
import java.awt.*;
import java.io.*;
import java.net.*;
import javax.swing.*;
```

Figura 11. Librerías utilizadas en el cliente

El constructor principal de la clase MultiClienteCine establece la conexión con el servidor, cuyo IP y puerto están definidos como constantes (SERVIDOR = 192.168.1.74 y PUERTO = 8888). Si la conexión falla, el programa muestra un mensaje de error y se cierra. Posteriormente, se configura la ventana principal de la interfaz gráfica (JFrame), que contiene varios componentes como un campo de texto para ingresar el nombre del cliente, un área de texto para mostrar los mensajes del servidor y botones para interactuar con el sistema. Los botones permiten al usuario realizar tres acciones principales: ver la cantidad de boletos disponibles, comprar boletos y salir del sistema.

```
public class MultiClienteCine { // Clase MultiClienteCine
    private final String SERVIDOR = "192.168.1.74"; // Se declara una constante
    private final int PUERTO = 8888; // Se declara una constante de tipo int co
    private Socket socket; // Se declara un objeto de tipo Socket
    private BufferedReader entrada; // Se declara un objeto de tipo BufferedRea
    private PrintWriter salida;
    private final JFrame frame;
    private final JTextField nombreField;
    private final JTextArea displayArea;
    private final JButton verBoletosBtn, comprarBtn, salirBtn;

    public MultiClienteCine() { // Constructor de la clase MultiClienteCine
        try {
            socket = new Socket(SERVIDOR, PUERTO); // Se crea un socket con la
            entrada = new BufferedReader(new InputStreamReader(socket.getInputS
            salida = new PrintWriter(socket.getOutputStream(), autoFlush:true);
        } catch (IOException e) {// Se utiliza un try-catch para manejar excepc
            JOptionPane.showMessageDialog(parentComponent:null, message:"No se
            System.exit(status:1);
        }

        frame = new JFrame(title:"Cliente Cine");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(width:400, height:300);
        frame.setLayout(new BorderLayout());
```

```
JPanel topPanel = new JPanel();
topPanel.setLayout(new BorderLayout());
nombreField = new JTextField();
topPanel.add(new JLabel(text:"Ingrese su nombre: "), BorderLayout.WEST);
topPanel.add(nombreField, BorderLayout.CENTER);

JButton conectarBtn = new JButton(text:"Conectar");
conectarBtn.addActionListener(e -> conectar());    The value of the lambd
topPanel.add(conectarBtn, BorderLayout.EAST);

frame.add(topPanel, BorderLayout.NORTH);

displayArea = new JTextArea();
displayArea.setEditable(b:false);
frame.add(new JScrollPane(displayArea), BorderLayout.CENTER);

JPanel sidePanel = new JPanel();
sidePanel.setLayout(new GridLayout(rows:3, cols:1));

verBoletosBtn = new JButton(text:"Ver Boletos");
verBoletosBtn.addActionListener(e -> verBoletos());    The value of the l
sidePanel.add(verBoletosBtn);

comprarBtn = new JButton(text:"Comprar");
comprarBtn.addActionListener(e -> comprarBoletos());    The value of the
sidePanel.add(comprarBtn);

salirBtn = new JButton(text:"Salir");
salirBtn.addActionListener(e -> salir());    The value of the lambda para
sidePanel.add(salirBtn);

frame.add(sidePanel, BorderLayout.EAST);
frame.setVisible(b:true);
```

Figura 12. Class MultiClienteCine en el código del cliente

Por otro lado el método conectar() se encarga de enviar el nombre del cliente al servidor y mostrar el mensaje de bienvenida recibido. Si el nombre ingresado está vacío, muestra un error. El método verBoletos() envía una solicitud al servidor para obtener la cantidad de boletos disponibles, y luego muestra la respuesta en el área de texto. El método comprarBoletos() solicita al usuario la cantidad de boletos que desea comprar y envía esa información al servidor, mostrando la respuesta del servidor (ya sea confirmando la compra o indicando un error). Finalmente, el método salir() cierra la conexión con el servidor y la interfaz gráfica, despidiéndose del usuario.

```
private void conectar() {
    String nombre = nombreField.getText().trim();
    if (nombre.isEmpty()) {
        JOptionPane.showMessageDialog(frame, message:"Por favor, ingrese su nombre.", title:"Error", JOptionPane.ERROR_MESSAGE);
        return;
    }
    salida.println(nombre);
    try {
        displayArea.append(entrada.readLine() + "\n");
    } catch (IOException e) {
        displayArea.append(str:"Error al recibir mensaje de bienvenida.\n");
    }
}
```

Figura 13. Método conectar ()

```
private void verBoletos() { // Metodo verBoletos
    salida.println(x:"1"); // Se envia un 1 al servidor
    try {
        displayArea.append("Servidor: " + entrada.readLine() + "\n");
    } catch (IOException e) { // Se utiliza un catch para manejar excepciones
        displayArea.append(str:"Error al recibir información.\n");
    }
}
```

Figura 14. Método verBoletos ()

```
private void comprarBoletos() { // Metodo comprarBoletos
    String cantidadStr = JOptionPane.showInputDialog(frame, message:"Ingrese la cantidad de boletos:");
    if (cantidadStr == null) return; // Se verifica si la cantidad es nula
    salida.println(x:"2"); // Se envia un 2 al servidor
    salida.println(cantidadStr);
    try {
        displayArea.append("Servidor: " + entrada.readLine() + "\n");
    } catch (IOException e) {
        displayArea.append(str:"Error al recibir respuesta.\n");
    }
```

Figura 15. Método comprarBoletos()

```
private void salir() { //  Metodo salir
    salida.println(x:"3"); // Se envia un 3 al servidor
    try {
        displayArea.append("Servidor: " + entrada.readLine() + "\n");
        socket.close(); // Se cierra el socket
    } catch (IOException e) {
        displayArea.append(str:"Error al cerrar la conexión.\n");
    }
    frame.dispose();
```

Figura 16. Método salir()

Por último, el método main() garantiza que la interfaz gráfica se ejecute en el hilo correcto utilizando SwingUtilities.invokeLater(MultiClienteCine::new), lo que asegura que la aplicación funcione correctamente en plataformas gráficas. Este enfoque es necesario para mantener la UI interactiva y responsiva.

```
Run | Debug | Run main | Debug main
public static void main(String[] args) { // Metodo main
    SwingUtilities.invokeLater(MultiClienteCine::new);
}
```

Figura 17. Método main

Resultados

La implementación del sistema cliente-servidor para la compra de boletos de cine permitió evaluar la comunicación eficiente entre múltiples clientes y un servidor central. Se comprobó que la concurrencia fue gestionada correctamente mediante el uso de hilos y un pool de conexiones, asegurando que cada cliente pudiera interactuar sin interferencias.

Uno de los aspectos más relevantes fue la sincronización en la gestión de boletos, lo que evitó sobreventas y garantizó una actualización precisa del inventario. Además, la reposición automática de boletos cada 20 segundos permitió que los clientes tuvieran nuevas oportunidades de compra sin necesidad de reiniciar el servidor.

Durante la ejecución de la práctica, se pudo observar que:

- Los clientes pueden conectarse simultáneamente al servidor sin conflictos.
- La cantidad de boletos disponibles se actualiza correctamente en tiempo real.
- Si un cliente intenta comprar más boletos de los disponibles, el sistema lo notifica adecuadamente.

- La reposición automática de boletos mantiene la disponibilidad sin intervención manual.
- El servidor maneja múltiples clientes de manera eficiente gracias al uso de ThreadPoolExecutor.

En términos de robustez, se verificó que el sistema maneja adecuadamente entradas incorrectas, desconexiones controladas y cierres de conexión sin pérdida de recursos. Esto permitió garantizar la estabilidad del servidor y la experiencia fluida para los clientes conectados.

Es por eso que podemos decir que la implementación cumplió con los principios de los sistemas distribuidos, logrando una comunicación efectiva, sincronización de datos y escalabilidad. La correcta gestión de la concurrencia y la automatización en la reposición de boletos demostraron ser estrategias clave para mantener un sistema funcional y eficiente.

A continuación, se muestran diversas capturas de pantalla en donde observamos la ejecución de nuestro servidor y de nuestro cliente:

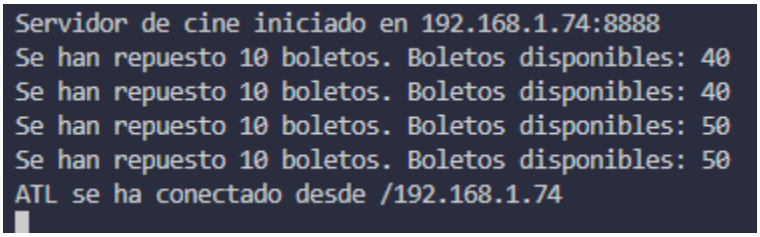


Figura 18. Inicialización del servidor y stock

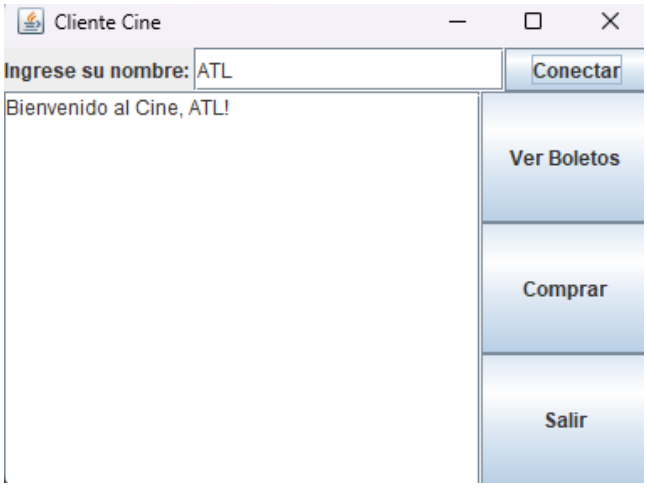


Figura 19. Inicialización del cliente y su menú

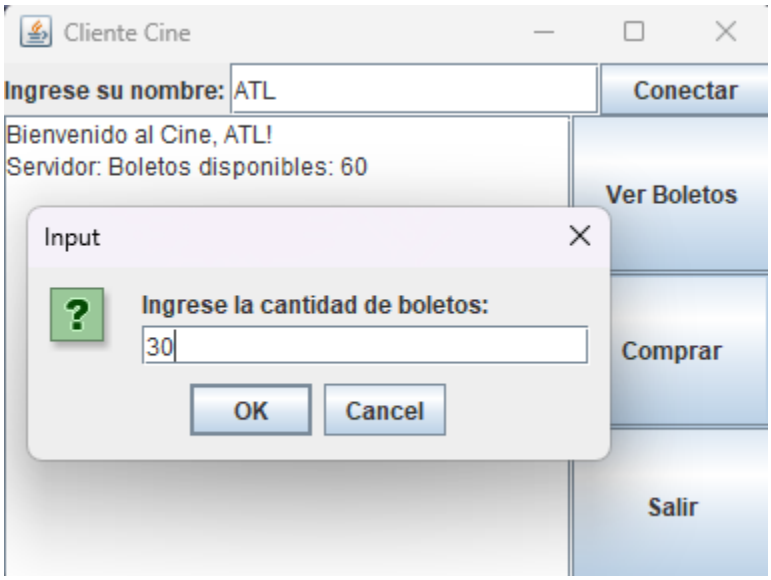


Figura 20. Opción 1 y 2, ver y compra de boletos del cliente

```
Servidor de cine iniciado en 192.168.1.74:8888
Se han repuesto 10 boletos. Boletos disponibles: 40
Se han repuesto 10 boletos. Boletos disponibles: 40
Se han repuesto 10 boletos. Boletos disponibles: 50
Se han repuesto 10 boletos. Boletos disponibles: 50
ATL se ha conectado desde /192.168.1.74
Se han repuesto 10 boletos. Boletos disponibles: 60
Se han repuesto 10 boletos. Boletos disponibles: 60
Se han repuesto 10 boletos. Boletos disponibles: 70
Se han repuesto 10 boletos. Boletos disponibles: 70
ATL compró 30 boletos. Boletos restantes: 40
```

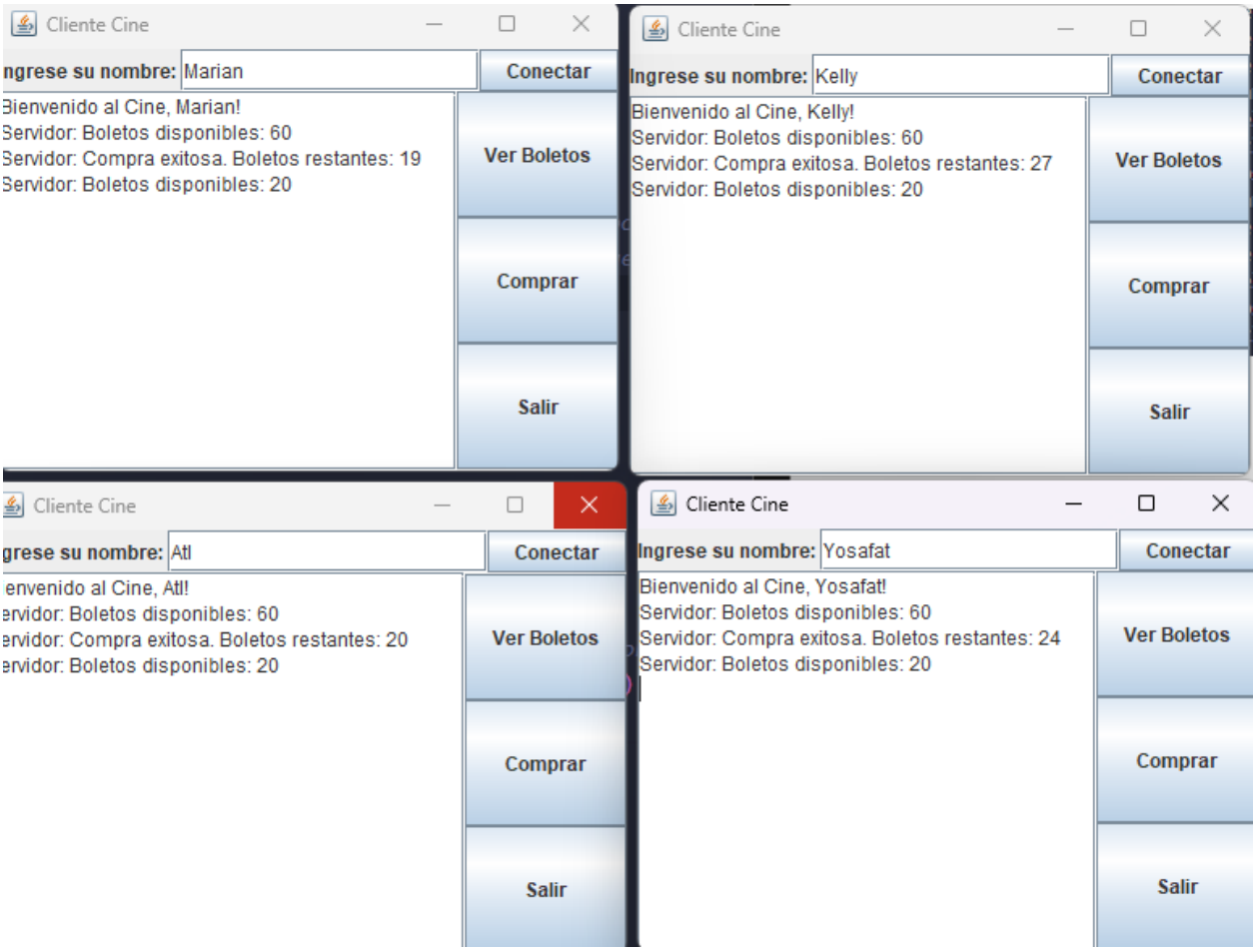
Figura 21. Vista del servidor al realizar una compra

```
ATL compró 30 boletos. Boletos restantes: 40
Se han repuesto 10 boletos. Boletos disponibles: 50
Se han repuesto 10 boletos. Boletos disponibles: 50
ATL ha salido.
```

Figura 22. Vista desde el servidor con la salida de un usuario

```
Se han repuesto 10 boletos. Boletos disponibles: 50
Se han repuesto 10 boletos. Boletos disponibles: 60
Se han repuesto 10 boletos. Boletos disponibles: 60
Marian se ha conectado desde /192.168.1.74
Kelly se ha conectado desde /192.168.1.74
Atl se ha conectado desde /192.168.1.74
Yosafat se ha conectado desde /192.168.1.74
Marian compró 41 boletos. Boletos restantes: 19
Se han repuesto 10 boletos. Boletos disponibles: 29
Se han repuesto 10 boletos. Boletos disponibles: 29
Kelly compró 2 boletos. Boletos restantes: 27
Yosafat compró 3 boletos. Boletos restantes: 24
Atl compró 4 boletos. Boletos restantes: 20
```

Figura 23. Conexión de múltiples clientes al servidor



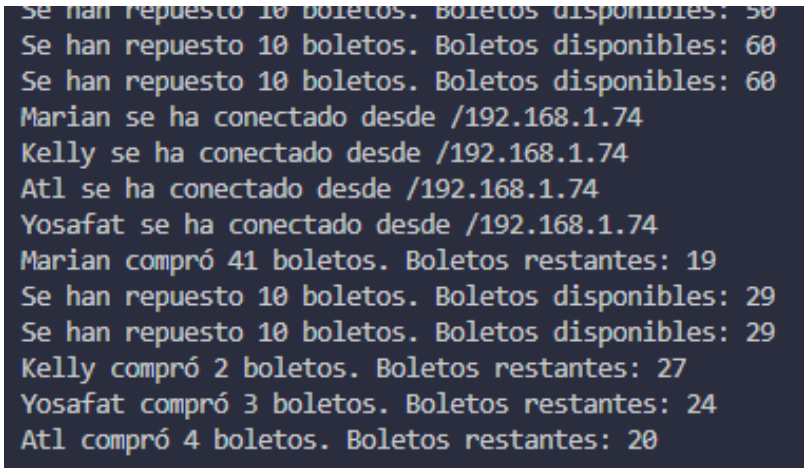


Figura 24. Múltiples clientes conectados al mismo tiempo al servidor

Conclusión

La realización de esta práctica me pareció bastante interesante pero también un poco complicada, especialmente al manejar la comunicación entre múltiples clientes y el servidor. Implementar el modelo cliente-servidor con Java y manejar múltiples conexiones al mismo tiempo utilizando hilos y un pool de threads fue un reto, ya que había que asegurarse de que los recursos compartidos, como la cantidad de boletos, estuvieran bien sincronizados para evitar inconsistencias.

Uno de los aspectos más difíciles que me toco enfrentar fue la gestión de la interfaz gráfica en el cliente. Integrar Swing para mostrar la información en una ventana y permitir la interacción con botones hizo que la aplicación fuera más intuitiva, pero también introdujo problemas con la actualización de la interfaz desde diferentes hilos. Además, manejar las excepciones correctamente para evitar que la aplicación se cerrara abruptamente requirió especial atención.

Por otro lado, en el servidor, la sincronización de los boletos con métodos synchronized fue clave para evitar problemas de concurrencia. También fue interesante implementar la reposición automática de boletos con un hilo separado, lo que le dio más dinamismo a la aplicación y simuló un comportamiento más realista.

Por último, cabe recalcar que esta práctica fue muy parecida a la anterior y me gusto mucho como la lo largo de esta pude darme cuenta de la evolución que tuvo a comparación de la otra, ya que a pesar de que solo cambian mínimos detalles, estos hacen que el sistema se vuelva cada vez más robusto y escalable, es por eso que me ayudó a entender mejor cómo funcionan los sistemas distribuidos en un entorno cliente-servidor y la importancia de la concurrencia en la programación. Aunque fue un reto, fue muy satisfactorio ver cómo los clientes podían conectarse, interactuar con el servidor y realizar operaciones de compra de boletos en tiempo real.

Referencias

J. Geeks, "Introducción a la programación con sockets en Java," Geeks for Geeks, 2023. [En línea]. Available: https://www.geeksforgeeks.org/socket-programming-in-java. [Último acceso: 9 Marzo 2025].

A. Silberschatz, P. Galvin y G. Gagne, Operating System Concepts, 10a ed., Wiley, 2018.

Code & Coke, "Sockets," 2024. [En línea]. Available: https://psp.codeandcoke.com/apuntes:sockets. [Último acceso: 9 Marzo 2025].

IBM, "Socket programming," IBM Knowledge Center, 2024. [En línea]. Available: https://www.ibm.com/docs/es/i/7.5?topic=communications-socket-programming. [Último acceso:8 Marzo 2025].

Daemon4, "Arquitectura Cliente-Servidor," 2024. [En línea]. Available: https://www.daemon4.com/empresa/noticias/arquitectura-cliente-servidor/. [Último acceso: 6 Marzo 2025].

Anexos

Código Fuente ServidorCine.java

```
/*
 * Nombre: Cardoso Osorio Atl Yosafat
 * Grupo: 7CM1
 * Materia: Sistemas Distribuidos
 * Practica 3: Modelo Multicliente/Servidor
 * Fecha: 09 de Marzo del 2025
 */

import java.io.*;
import java.net.*;
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadPoolExecutor;

class Cine {
    private int boletosDisponibles;

    public Cine(int boletosDisponibles) {
        this.boletosDisponibles = boletosDisponibles;
```

```
    }

    public synchronized String comprarBoleto(String cliente, int
cantidad) {
        if (boletosDisponibles < cantidad) {
            return "No hay suficientes boletos. Espere a la
próxima función.";
        }
        boletosDisponibles -= cantidad;
        System.out.println(cliente + " compró " + cantidad + "
boletos. Boletos restantes: " + boletosDisponibles);
        return "Compra exitosa. Boletos restantes: " +
boletosDisponibles;
    }

    public synchronized int getBoletosDisponibles() {
        return boletosDisponibles;
    }

    public synchronized void reponerBoletos(int cantidad) {
        boletosDisponibles += cantidad;
        notifyAll();
        System.out.println("Se han repuesto " + cantidad + "
boletos. Boletos disponibles: " + boletosDisponibles);
    }
}

class ReposicionBoletos extends Thread { // Este metodo se encarga
de reponer los boletos
    private final Cine cine; // Se crea un objeto de la clase Cine

    public ReposicionBoletos(Cine cine) {// Se crea un constructor
que recibe un objeto de la clase Cine
        this.cine = cine;
    }

    @Override
    public void run() { // Se crea un metodo run
        while (true) {
```

```
            try { // Se utiliza un try-catch para manejar
excepciones
                Thread.sleep(20000); // Se duerme el hilo por 20
segundos
                cine.reponerBoletos(10); // Se llama al metodo
reponerBoletos de la clase Cine
                System.out.println("Se han repuesto 10 boletos.
Boletos disponibles: " + cine.getBoletosDisponibles());
            } catch (InterruptedException e) {
                System.err.println("Error al reponer boletos: " +
e.getMessage());
            }
        }
    }
}

class HiloCliente extends Thread { // Se crea una clase
HiloCliente que extiende de Thread
    private final Socket socket;
    private final Cine cine;

    public HiloCliente(Socket socket, Cine cine) {
        this.socket = socket;
        this.cine = cine;
    }

    @Override
    public void run() { // Se crea un metodo run
        try (
            BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            PrintWriter out = new
PrintWriter(socket.getOutputStream(), true)
        ) {
            String nombreCliente = in.readLine(); // Se lee el
nombre del cliente
            System.out.println(nombreCliente + " se ha conectado
desde " + socket.getInetAddress());
```

```
            out.println("Bienvenido al Cine, " + nombreCliente +
"!");

            boolean seguir = true; // Se crea una variable
booleana seguir
            while (seguir) {
                String opcionStr = in.readLine(); // Se lee la
opcion del cliente
                if (opcionStr == null) break;

                int opcion; // Se crea una variable opcion
                try {
                    opcion = Integer.parseInt(opcionStr);
                } catch (NumberFormatException e) {
                    out.println("Opción no válida.");
                    continue;
                }

                switch (opcion) {
                    case 1 -> out.println("Boletos disponibles: "
+ cine.getBoletosDisponibles());
                    case 2 -> {
                        String cantidadStr = in.readLine();
                        int cantidad;
                        try {
                            cantidad =
Integer.parseInt(cantidadStr);
                        } catch (NumberFormatException e) {
                            out.println("Cantidad no válida.");
                            continue;
                        }
                        out.println(cine.comprarBoleto(nombreClien
te, cantidad));
                    }
                    case 3 -> {
                        out.println("Gracias por su compra. ¡Hasta
luego!");
                        seguir = false;
```

```
                        System.out.println(nombreCliente + " ha
salido.");
                    }
                    default -> out.println("Opción no válida.");
                }
            }
        } catch (IOException e) { // Se utiliza un catch para
manejar excepciones
            System.err.println("Error en la conexión con el
cliente: " + e.getMessage());
        } finally {
            try { // Se utiliza un try-catch para manejar
excepciones
                socket.close(); // Se cierra el socket
            } catch (IOException e) {
                System.err.println("Error al cerrar el socket: " +
e.getMessage());
            }
        }
    }
}

public class ServidorCine { // Se crea una clase ServidorCine
    public static void main(String[] args) { // Se crea un metodo
main
        String ipServidor = "192.168.1.74"; //localhost";   // Se
crea una variable ipServidor
        int puerto = 8888; // Se crea una variable puerto
        Cine cine = new Cine(30); // Se crea un objeto de la clase
Cine
        ThreadPoolExecutor pool = (ThreadPoolExecutor)
Executors.newFixedThreadPool(4);

        new ReposicionBoletos(cine).start(); // Se crea un objeto
de la clase ReposicionBoletos y se inicia

        try (ServerSocket serverSocket = new ServerSocket(puerto))
{
```

```
            System.out.println("Servidor de cine iniciado en " +
ipServidor + ":" + puerto);

            while (true) { // Se crea un ciclo while
                Socket socketCliente = serverSocket.accept();
                pool.execute(new HiloCliente(socketCliente,
cine));
            }
        } catch (IOException e) { // Se utiliza un catch para
manejar excepciones
            System.err.println("Error en el servidor: " +
e.getMessage());
        }
    }
}
```

Código Fuente MultiClienteCine.java

```
/*
 * Nombre: Cardoso Osorio Atl Yosafat
 * Grupo: 7CM1
 * Materia: Sistemas Distribuidos
 * Practica 3: Modelo Multicliente/Servidor
 * Fecha: 09 de Marzo del 2025
 */

import java.awt.*;
import java.io.*;
import java.net.*;
import javax.swing.*;

public class MultiClienteCine { // Clase MultiClienteCine
    private final String SERVIDOR = "192.168.1.74"; // Se declara
una constante de tipo String con la direccion IP del servidor
    private final int PUERTO = 8888; // Se declara una constante
de tipo int con el puerto del servidor
```

```
    private Socket socket; // Se declara un objeto de tipo Socket
    private BufferedReader entrada; // Se declara un objeto de
tipo BufferedReader
    private PrintWriter salida;
    private final JFrame frame;
    private final JTextField nombreField;
    private final JTextArea displayArea;
    private final JButton verBoletosBtn, comprarBtn, salirBtn;

    public MultiClienteCine() { // Constructor de la clase
MultiClienteCine
        try {
            socket = new Socket(SERVIDOR, PUERTO); // Se crea un
socket con la direccion IP y puerto del servidor
            entrada = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            salida = new PrintWriter(socket.getOutputStream(),
true);
        } catch (IOException e) {// Se utiliza un try-catch para
manejar excepciones
            JOptionPane.showMessageDialog(null, "No se pudo
conectar al servidor.", "Error", JOptionPane.ERROR_MESSAGE);
            System.exit(1);
        }

        frame = new JFrame("Cliente Cine");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 300);
        frame.setLayout(new BorderLayout());

        JPanel topPanel = new JPanel();
        topPanel.setLayout(new BorderLayout());
        nombreField = new JTextField();
        topPanel.add(new JLabel("Ingrese su nombre: "),
BorderLayout.WEST);
        topPanel.add(nombreField, BorderLayout.CENTER);

        JButton conectarBtn = new JButton("Conectar");
        conectarBtn.addActionListener(e -> conectar());
```

```
        topPanel.add(conectarBtn, BorderLayout.EAST);

        frame.add(topPanel, BorderLayout.NORTH);

        displayArea = new JTextArea();
        displayArea.setEditable(false);
        frame.add(new JScrollPane(displayArea),
BorderLayout.CENTER);

        JPanel sidePanel = new JPanel();
        sidePanel.setLayout(new GridLayout(3, 1));

        verBoletosBtn = new JButton("Ver Boletos");
        verBoletosBtn.addActionListener(e -> verBoletos());
        sidePanel.add(verBoletosBtn);

        comprarBtn = new JButton("Comprar");
        comprarBtn.addActionListener(e -> comprarBoletos());
        sidePanel.add(comprarBtn);

        salirBtn = new JButton("Salir");
        salirBtn.addActionListener(e -> salir());
        sidePanel.add(salirBtn);

        frame.add(sidePanel, BorderLayout.EAST);
        frame.setVisible(true);
    }

    private void conectar() { // Metodo conectar
        String nombre = nombreField.getText().trim();
        if (nombre.isEmpty()) { // Se verifica si el nombre esta
vacio
            JOptionPane.showMessageDialog(frame, "Por favor,
ingrese su nombre.", "Error", JOptionPane.ERROR_MESSAGE);
            return;
        }
        salida.println(nombre); // Se envia el nombre al servidor
        try {
            displayArea.append(entrada.readLine() + "\n");
```

```
        } catch (IOException e) {
            displayArea.append("Error al recibir mensaje de
bienvenida.\n");
        }
    }

    private void verBoletos() { // Metodo verBoletos
        salida.println("1"); // Se envia un 1 al servidor
        try {
            displayArea.append("Servidor: " + entrada.readLine() +
"\n");
        } catch (IOException e) { // Se utiliza un catch para
manejar excepciones
            displayArea.append("Error al recibir información.\n");
        }
    }

    private void comprarBoletos() { // Metodo comprarBoletos
        String cantidadStr = JOptionPane.showInputDialog(frame,
"Ingrese la cantidad de boletos:");
        if (cantidadStr == null) return; // Se verifica si la
cantidad es nula
        salida.println("2"); // Se envia un 2 al servidor
        salida.println(cantidadStr);
        try {
            displayArea.append("Servidor: " + entrada.readLine() +
"\n");
        } catch (IOException e) {
            displayArea.append("Error al recibir respuesta.\n");
        }
    }

    private void salir() { //  Metodo salir
        salida.println("3"); // Se envia un 3 al servidor
        try {
            displayArea.append("Servidor: " + entrada.readLine() +
"\n");
            socket.close(); // Se cierra el socket
        } catch (IOException e) {
```

```
            displayArea.append("Error al cerrar la conexión.\n");
        }
        frame.dispose();
    }

    public static void main(String[] args) { // Metodo main
        SwingUtilities.invokeLater(MultiClienteCine::new);
    }
}
```