INSTITUTO POLITÉCTICO NACIONAL

Escuela Superior de Cómputo

**Carrera:**

Ingeniería en Sistemas Computacionales

**Unidad de aprendizaje:**

Sistemas Distribuidos

**Practica 2:**

Modelo Cliente - Servidor


**Alumno:**

Cardoso Osorio Atl Yosafat

**Profesor:**

Chadwick Carreto Arellano


**Fecha:**

02/03/2025

INSTITUTO POLITÉCNICO NACIONAL

ESCOM

# Índice de contenido

# Índice de figuras

# Antecedentes

En el ámbito de los sistemas distribuidos, la arquitectura cliente-servidor es un modelo ampliamente utilizado para la distribución de tareas y servicios en una red. Este paradigma permite que múltiples clientes soliciten recursos o servicios a uno o varios servidores, los cuales procesan las peticiones y devuelven las respuestas correspondientes.

## 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.

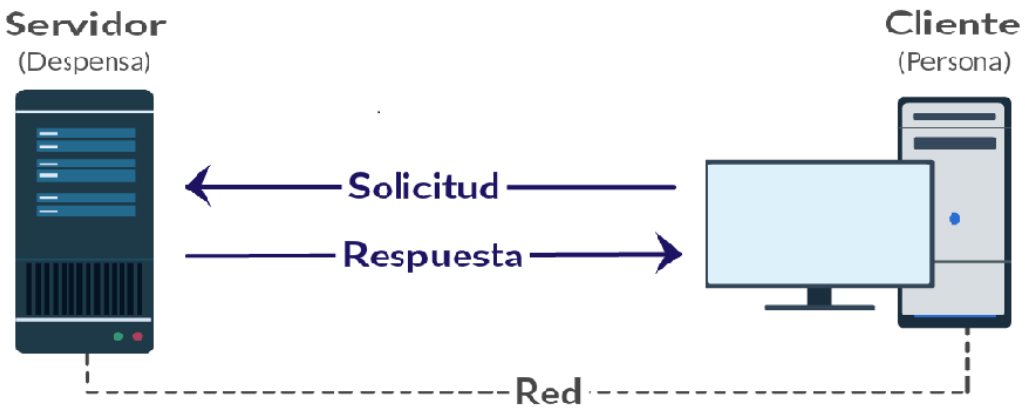


***Figura 1. Diagrama de la arquitectura Cliente/Servidor***

## 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.

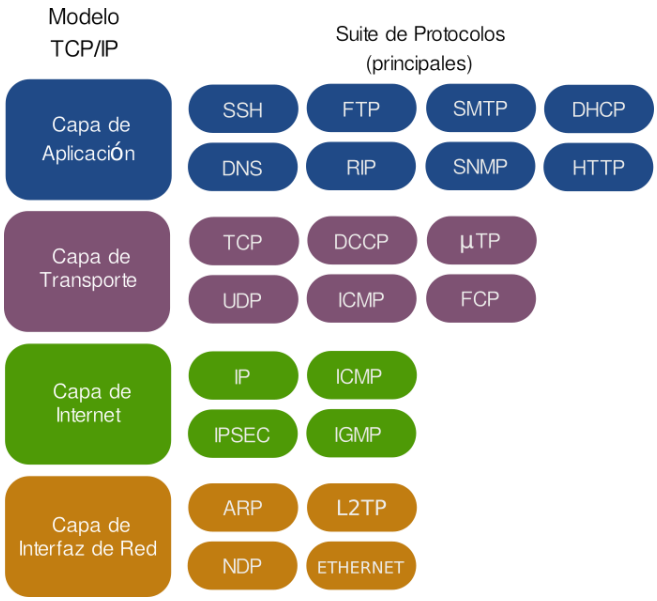


***Figura 2. Diagrama de protocolos de comunicación***

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.

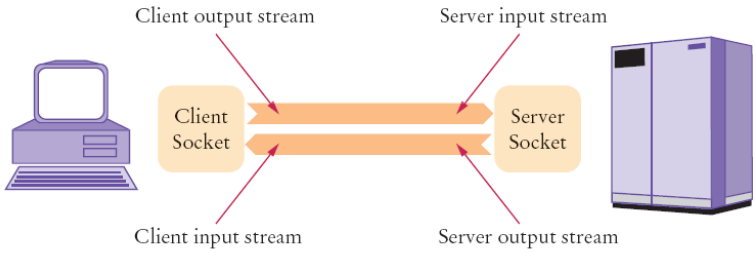


***Figura 3. Diagrama de Sockets Cliente/Servidor***

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

En los sistemas distribuidos, el modelo cliente-servidor es ampliamente utilizado para permitir la comunicación entre distintas partes de una aplicación, facilitando la interacción entre un servidor que proporciona servicios y múltiples clientes que los solicitan. En esta práctica, se presenta el diseño e implementación de un sistema distribuido basado en este modelo, que simula la compra de boletos en un cine, permitiendo a los clientes interactuar con el servidor para consultar boletos disponibles, comprar boletos y gestionar la reposición de boletos cuando se agotan.

El principal desafío de este sistema radica en la gestión adecuada de la concurrencia, especialmente en el acceso a los recursos compartidos, como los boletos disponibles. Para resolver esto, se implementa un mecanismo de sincronización, asegurando que múltiples clientes no compren los mismos boletos simultáneamente. Además, el servidor debe ser capaz de manejar correctamente la reposición de boletos, lo que se realiza en intervalos regulares mediante un hilo que se ejecuta en segundo plano.

El propósito de este sistema es simular un cine en línea en el cual los clientes pueden interactuar con el servidor de manera eficiente y segura, garantizando que las transacciones de compra de boletos sean consistentes y que los recursos estén disponibles en todo momento, incluso cuando múltiples clientes estén realizando compras de manera concurrente.

Esta práctica busca principalmente el resolver un ejemplo práctico de cómo un sistema distribuido puede implementarse para manejar tareas simples, pero con desafíos en términos de concurrencia, comunicación de red y gestión de recursos compartidos.

## Propuesta de solución

Para la solución de este problema desarrollaremos un sistema de venta de boletos de cine utilizando el modelo cliente-servidor. El servidor se encarga de gestionar el estado de los boletos disponibles, procesar las solicitudes de compra y reponer los boletos de manera automática. El cliente por otro lado interactuara con el usuario, permitiéndole ver la disponibilidad de boletos, realizar compras y seleccionar opciones del menú.

La arquitectura del servidor se basa en el uso de hilos, donde cada cliente se maneja independientemente a través de un hilo dedicado, permitiendo la atención simultánea de múltiples usuarios. Es decir será el encargado de manejar las conexiones de los clientes

utilizando un pool de hilos, lo que permitirá la concurrencia sin afectar el rendimiento del sistema. Cada cliente será atendido por un hilo independiente, garantizando que las operaciones de compra y consulta de boletos se realicen de manera simultánea sin interferencias

Además, la clase Cine implementa métodos sincronizados para garantizar que las compras de boletos no se solapen, asegurando la consistencia de los datos, así como se incluye un hilo adicional, "ReposicionBoletos", que se encarga de reponer boletos cada 15 segundos, simulando la reposición periódica de boletos.

Este enfoque asegura eficiencia en el manejo de múltiples usuarios, actualizaciones en tiempo real sobre la disponibilidad de boletos y la reposición automática para mantener el inventario adecuado.

# 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.

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 mas 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:*** El cliente emplea un Scanner para leer las entradas del usuario. El cliente presenta un menú interactivo que permite al usuario elegir entre diferentes opciones como ver boletos disponibles, comprar boletos o salir del sistema.

## Desarrollo

En esta práctica se implementó un sistema de venta de boletos para un cine utilizando el modelo cliente-servidor en Java. 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.

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 4. 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.

```
// Clase que maneja los boletos disponibles
class Cine {
    private int boletosDisponibles;

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

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

    public synchronized int getBoletosDisponibles() {
        return boletosDisponibles;
    }

    public synchronized void reponerBoletos(int cantidad) {
        boletosDisponibles += cantidad;
        System.out.println("Se repusieron " + cantidad + " boletos. Nuevo stock: " + boletosDisponibles);
        notifyAll();
    }
}
```

***Figura 5. 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.


***Figura 6. Class para manejar la reposición de boletos***

Por otra parte, cada cliente es gestionado por una instancia de la clase HiloCliente, que maneja la comunicación entre el cliente y el servidor. Este hilo recibe las solicitudes del cliente, muestra el menú de opciones y procesa las respuestas adecuadamente.

El cliente puede:

1. Consultar la cantidad de boletos disponibles.
2. Comprar boletos.
3. Salir del sistema.

El uso de hilos permite que múltiples clientes interactúen simultáneamente con el servidor sin interferencias.

```java
// Hilo para manejar cada cliente
class HiloCliente extends Thread {
    private final Socket socket;
    private final Cine cine;

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

    @Override
    public void run() {
        try ( // Manejo de excepciones
            BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            PrintWriter out = new PrintWriter(socket.getOutputStream(), autoFlush:true)
        ) {
            String nombreCliente = in.readLine(); // Leer nombre del cliente
            System.out.println(nombreCliente + " se ha conectado."); // Mensaje de conexión
            out.println("Bienvenido al Cine, " + nombreCliente + "!"); // Mensaje de bienvenida

            while (true) { // Bucle para mostrar el menú
                out.println(x:"\n--- Menú del Cine ---");
                out.println(x:"1. Ver boletos disponibles");
                out.println(x:"2. Comprar boletos");
                out.println(x:"3. Salir");
                out.println(x:"Seleccione una opción:");

                String opcionStr = in.readLine(); // Leer opción del cliente
                if (opcionStr == null) break;

                int opcion; // Convertir opción a entero
                try {
                    opcion = Integer.parseInt(opcionStr);
                } catch (NumberFormatException e) {
                    out.println(x:"Opción no válida. Intente de nuevo.");
                    continue;
                }
                switch (opcion) { // Switch para manejar las opciones
                    case 1 -> out.println("Boletos disponibles: " + cine.getBoletosDisponibles());
                    case 2 -> { // Comprar boletos
                        out.println(x:"Ingrese la cantidad de boletos que desea comprar:");
                        String cantidadStr = in.readLine();
                        int cantidad;
                        try { // Convertir cantidad a entero
                            cantidad = Integer.parseInt(cantidadStr);
                        } catch (NumberFormatException e) {
                            out.println(x:"Cantidad no válida. Intente de nuevo.");
                            continue;
                        } // Comprar boletos
                        String respuesta = cine.comprarBoleto(nombreCliente, cantidad);
                        out.println(respuesta);
                    }
                    case 3 -> { // Salir
                        out.println(x:"Gracias por su compra. ¡Hasta luego!");
                        System.out.println(nombreCliente + " se ha desconectado.");
                    } // Salir del bucle si elige salir
                    default -> out.println(x:"Opción no válida. Intente de nuevo.");
                }
            }
        } catch (IOException e) { // Excepción de entrada/salida
            System.err.println("Error en la conexión con el cliente: " + e.getMessage());
        } finally {
            try {
                socket.close();
            } catch (IOException e) { // Excepción de entrada/salida
                System.err.println("Error al cerrar el socket: " + e.getMessage());
            }
        }
    }
```

*Figura 7. Class Hilo cliente en el servidor*

La clase ServidorCine se encarga de iniciar el servidor y administrar las conexiones de los clientes. Utiliza un ServerSocket que escucha en el puerto 8888 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
// Servidor principal
public class ServidorCine { // Clase principal
    Run | Debug | Run main | Debug main
    public static void main(String[] args) { // Método principal
        int puerto = 8888; // Puerto del servidor
        Cine cine = new Cine(boletosDisponibles:30); // Crear cine con 30 boletos disponibles
        ThreadPoolExecutor pool = (ThreadPoolExecutor) Executors.newFixedThreadPool(nThreads:4);

        new ReposicionBoletos(cine).start(); // Iniciar hilo de reposición de boletos

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

            while (true) { // Bucle para aceptar conexiones
                Socket socketCliente = serverSocket.accept();
                pool.execute(new HiloCliente(socketCliente, cine));
            }
        } catch (IOException e) { // Excepción de entrada/salida
            System.err.println("Error en el servidor: " + e.getMessage());
        }
    }
}
```

***Figura 8.Clase principal del servidor***

Por último, tenemos al cliente, representado por la clase ClienteCine, establece una conexión con el servidor a través de un Socket en la dirección 127.0.0.1 y el puerto 5000. Una vez conectado, el cliente ingresa su nombre y recibe un mensaje de bienvenida del servidor.

**Interacción Cliente-Servidor**

Después de conectarse, el cliente interactúa con el servidor mediante un menú de opciones:

***Consultar boletos:*** El cliente puede solicitar la cantidad de boletos disponibles y recibir la respuesta del servidor.

***Comprar boletos:*** Si el cliente elige comprar boletos, el servidor verifica la disponibilidad y responde si la compra fue exitosa o si no hay suficientes boletos.

***Salir:*** Si el cliente selecciona esta opción, la conexión se cierra y el cliente se desconecta del servidor.

```java
public class ClienteCine { // Clase ClienteCine
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        final String SERVIDOR = "127.0.0.1"; // Dirección del servidor (localhost)
        final int PUERTO = 8888; // Mismo puerto que el servidor

        try (Socket socket = new Socket(SERVIDOR, PUERTO);
             BufferedReader entrada = new BufferedReader(new InputStreamReader(socket.getInputStream()));
             PrintWriter salida = new PrintWriter(socket.getOutputStream(), autoFlush:true);
             Scanner scanner = new Scanner(System.in)) {

            System.out.println(x:"Conectado al cine."); // Mensaje de conexión exitosa

            // Ingresar nombre del cliente
            System.out.print(s:"Ingrese su nombre: "); // Solicitar nombre al cliente
            String nombre = scanner.nextLine();
            salida.println(nombre); // Enviar nombre al servidor
            System.out.println(entrada.readLine()); // Mensaje de bienvenida del servidor

            while (true) {
                // Leer y mostrar el menú hasta que se reciba "Seleccione una opción:"
                String linea;
                while (!(linea = entrada.readLine()).contains(s:"Seleccione una opción")) {
                    System.out.println(linea);
                }
                System.out.println(linea); // Imprime "Seleccione una opción:"

                // Leer opción del usuario
                System.out.print(s:"Opción: "); // Solicitar opción al cliente
                String opcionStr = scanner.nextLine();
                salida.println(opcionStr); // Enviar opción al servidor
```

```java
        switch (opcionStr) {
            case "1", "3" -> {
                // Si es ver funciones o salir, recibir respuesta del servidor
                System.out.println("Servidor: " + entrada.readLine());
                if (opcionStr.equals(anObject:"3")) {
                    System.out.println(x:"Saliendo del cine... ¡Nos vemos Cinefilo!");
                    return; // Salir del método si elige salir
                }
            }
            case "2" -> {
                // Comprar boletos
                System.out.println(entrada.readLine()); // Esperar mensaje del servidor "Ingrese la cantidad..."
                String cantidad = scanner.nextLine();
                salida.println(cantidad); // Enviar cantidad de boletos al servidor
                String respuesta = entrada.readLine(); // Recibir respuesta de compra
                System.out.println("Servidor: " + respuesta);
            }
            default -> System.out.println(x:"Opción no válida.");
```

***Figura 9. Código del lado del Cliente y las acciones que puede realizar***

Por último, tenemos el manejo de errores ya que se ha considerado el manejo de errores para garantizar una experiencia fluida y evitar caídas inesperadas. Por ejemplo: Si un cliente introduce una opción inválida, el servidor responde con un mensaje de error. Si un cliente intenta comprar más boletos de los disponibles, el servidor informa que no hay suficientes boletos. En caso de interrupciones en la conexión, el servidor maneja el cierre adecuado de los sockets.

```java
        } catch (IOException e) { // Excepción de entrada/salida
            System.err.println("Error en el cliente: " + e.getMessage());
        }
```

***Figura 10. Manejo de errores***

## Resultados

La implementación del sistema de venta de boletos para el cine utilizando el modelo cliente-servidor permitió comprobar la comunicación eficiente entre múltiples clientes y un servidor central. Se logró manejar la concurrencia mediante el uso de hilos, asegurando que cada cliente pudiera interactuar con el servidor sin interferencias en las operaciones de compra de boletos.

Uno de los aspectos más destacados de la práctica fue la implementación de sincronización en la clase Cine, la cual garantizó que las compras de boletos se procesaran correctamente, evitando problemas de concurrencia o condiciones de carrera. La función de reposición automática de boletos añadió una capa de realismo al sistema, simulando la disponibilidad periódica de nuevas funciones en el cine.

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

- Los clientes pueden conectarse simultáneamente al servidor y operar de manera independiente.
- 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 cada 15 segundos permite nuevas compras sin necesidad de reiniciar el servidor.
- El servidor puede manejar múltiples clientes de manera eficiente gracias al uso de ThreadPoolExecutor.

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

```
PS C:\Users\atl1c>  & 'C:\Program Files\Java\jdk-23\bin\java.exe' '-
Temp\vscodesws_530a7\jdt_ws\jdt.ls-java-project\bin' 'ServidorCine'
Servidor de cine iniciado en el puerto 8888
Se repusieron 10 boletos. Nuevo stock: 40
```

***Figura 11. Inicialización del servidor y stock***

```
Ingrese su nombre: Atl Cardoso
Bienvenido al Cine, Atl Cardoso!

--- Menú del Cine ---
1. Ver boletos disponibles
2. Comprar boletos
3. Salir
Seleccione una opción:
Opción: 1
Servidor: Boletos disponibles: 50
```

***Figura 12. Inicialización del cliente y su menú***

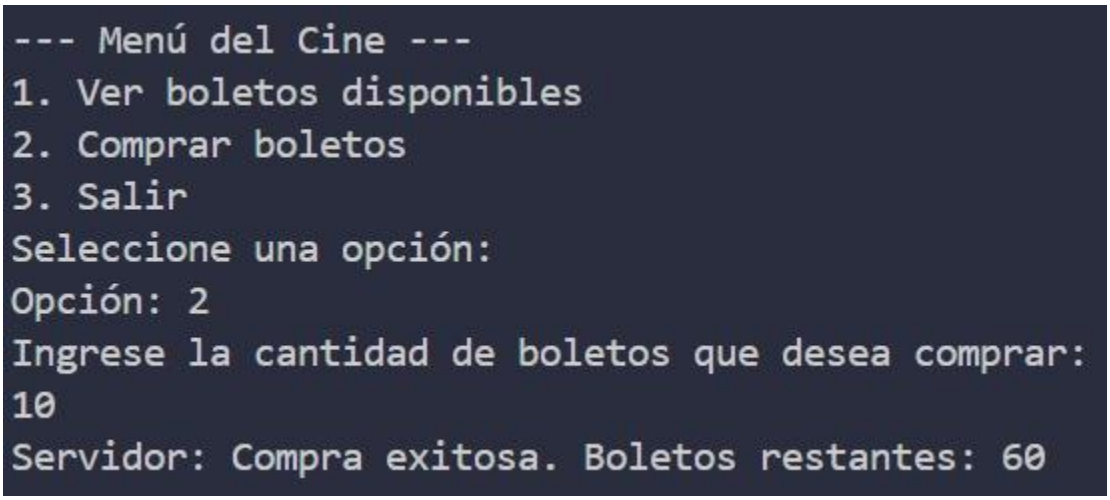


*Figura 13. Opción 2 compra de boletos del cliente*

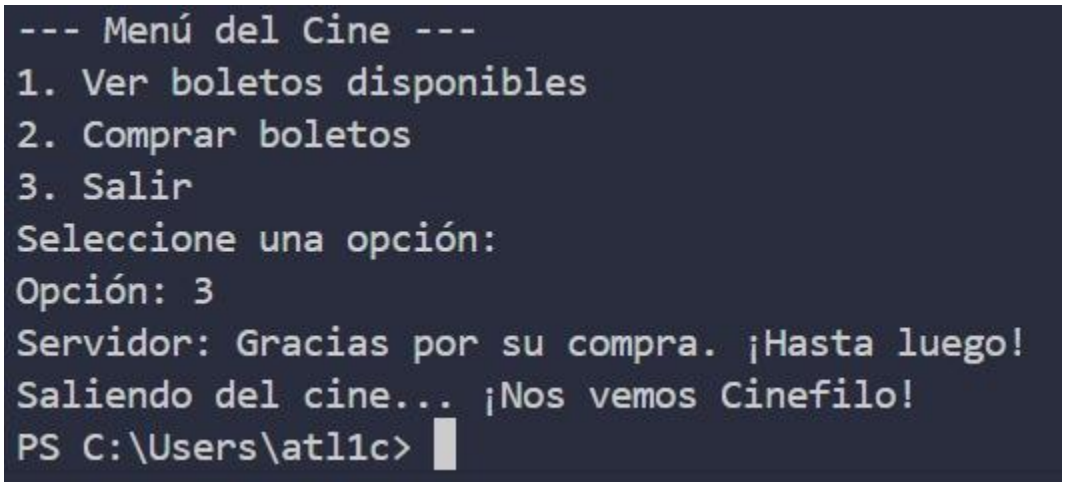


*Figura 14. Opción 3 salir del sistema*

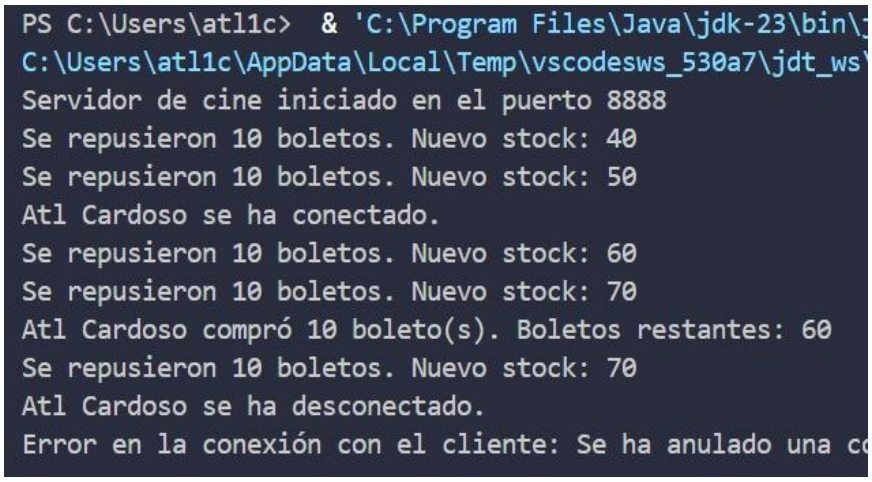


*Figura 15. Vista desde el servidor*

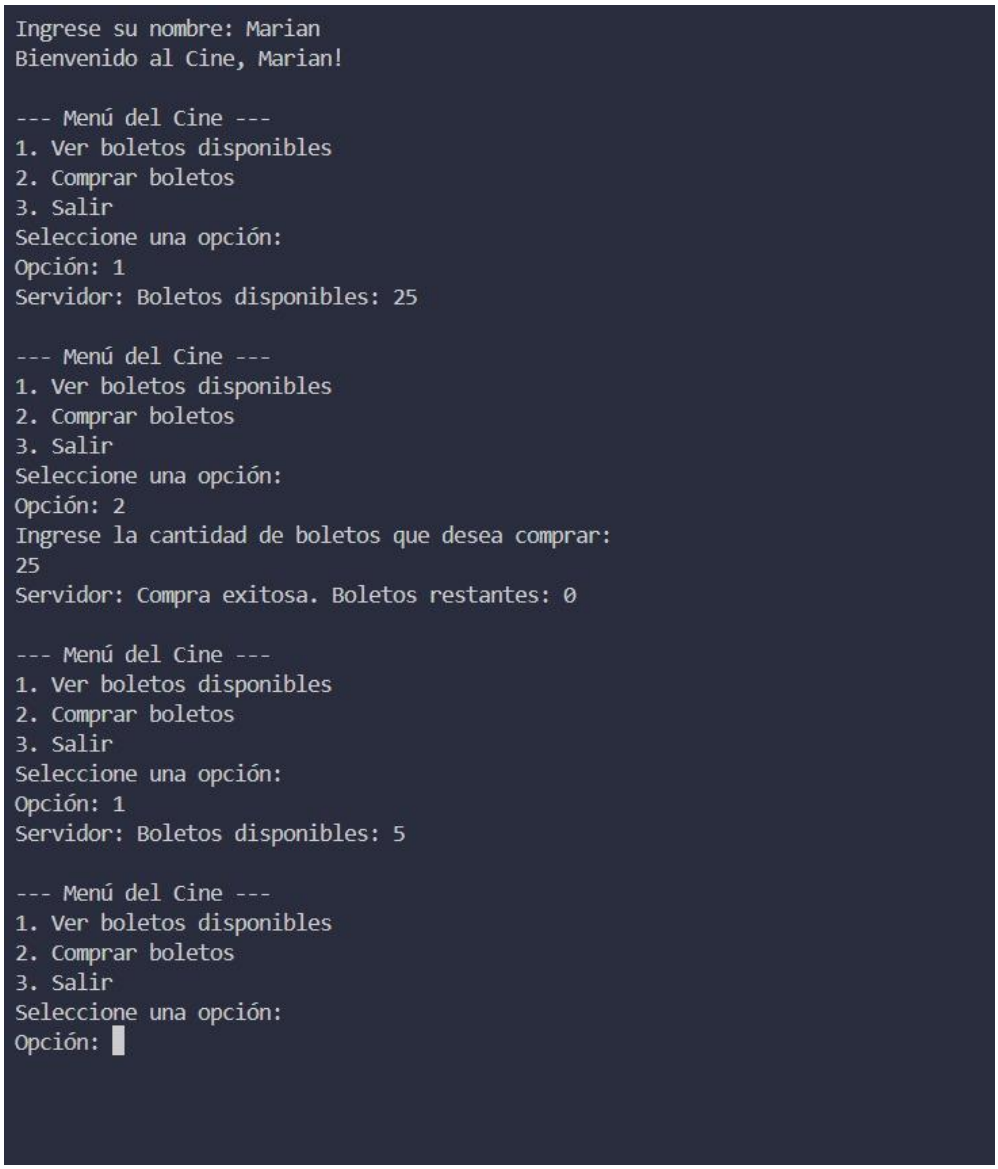


***Figura 16. Ingreso y compra de boletos desde otro usuario***

## Conclusión

Durante esta práctica, reforcé mis conocimientos sobre el modelo cliente-servidor y la comunicación a través de sockets en Java. Aunque ya tenía nociones previas, llevarlo a la práctica con un sistema de venta de boletos para un cine fue un verdadero reto. Uno de los aspectos que más me costó fue manejar múltiples conexiones simultáneas sin que se generaran problemas de concurrencia, especialmente al acceder y modificar el número de boletos disponibles.

Inicialmente, tuve dificultades al implementar la sincronización en la clase Cine, ya que, sin los métodos synchronized, el acceso concurrente a los boletos podía ocasionar inconsistencias, como vender más boletos de los disponibles o mostrar información incorrecta a los clientes. Además, la gestión de hilos en el servidor me resultó un desafío, especialmente al decidir entre crear un hilo por cliente o utilizar un ThreadPoolExecutor. Al principio, mi implementación creaba un nuevo hilo por cada cliente, lo que generaba un consumo innecesario de recursos; al cambiar a un pool de hilos, la eficiencia mejoró considerablemente.

Otro punto que me complicó fue la funcionalidad de reposición automática de boletos con la clase ReposicionBoletos. Al principio, tenía problemas con la actualización de los boletos en tiempo real, ya que los clientes no se enteraban cuando se reponían. Aunque finalmente logré hacerlo funcionar correctamente, me hizo reflexionar sobre la importancia de la comunicación en tiempo real y cómo un sistema más avanzado podría implementar WebSockets o algún mecanismo de notificación para mejorar la experiencia del usuario.

Comparado con otras prácticas en las que había trabajado con hilos dentro de un solo proceso, aquí pude ver cómo las tareas se distribuyen en un entorno de red real, donde múltiples clientes interactúan con un servidor central. Este ejercicio me permitió conectar la teoría de sistemas distribuidos con una aplicación práctica, como la compra de boletos en línea.

Por último, esta práctica me ayudó a consolidar mis conocimientos sobre concurrencia y sincronización en sistemas distribuidos. Aunque enfrenté varios retos, también me dejó con la inquietud de seguir explorando cómo mejorar la comunicación entre componentes y optimizar el rendimiento en aplicaciones con múltiples clientes.

# 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: 2 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: 2 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:3 Marzo 2025].*

*Daemon4, "Arquitectura Cliente-Servidor," 2024. [En línea]. Available: https://www.daemon4.com/empresa/noticias/arquitectura-cliente-servidor/. [Último acceso: 1 Marzo 2025].*

# Anexos

## Código Fuente ServidorCine.java

```
/*
 * Nombre: Cardoso Osorio Atl Yosafat
 * Grupo: 7CM1
 * Materia: Sistemas Distribuidos
 * Practica 2: Modelo Cliente/Servidor
 * Fecha: 01 de Marzo del 2025
 */
import java.io.*;
import java.net.*;
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadPoolExecutor;

// Clase que maneja los boletos disponibles
class Cine {
    private int boletosDisponibles;

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

```
    }

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

    public synchronized int getBoletosDisponibles() {
        return boletosDisponibles;
    }

    public synchronized void reponerBoletos(int cantidad) {
        boletosDisponibles += cantidad;
        System.out.println("Se repusieron " + cantidad + "
boletos. Nuevo stock: " + boletosDisponibles);
        notifyAll();
    }
}

// Hilo que maneja la reposición de boletos
class ReposicionBoletos extends Thread {
    private final Cine cine;

    public ReposicionBoletos(Cine cine) { // Constructor
        this.cine = cine;
    }

    @Override
    public void run() { // Método run
```

```
        while (true) {
            try { // Manejo de excepciones
                Thread.sleep(20000); // Reposición cada 20
segundos
                cine.reponerBoletos(10); // Reposición de 10
boletos
            } catch (InterruptedException e) { // Excepción de
interrupción
                System.err.println("Error al cerrar el socket: " +
e.getMessage());
            }
        }
    }
}

// Hilo para manejar cada cliente
class HiloCliente extends Thread {
    private final Socket socket;
    private final Cine cine;

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

    @Override
    public void run() {
        try ( // Manejo de excepciones
            BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            PrintWriter out = new
PrintWriter(socket.getOutputStream(), true)
        ) {
            String nombreCliente = in.readLine(); // Leer nombre
del cliente
            System.out.println(nombreCliente + " se ha
conectado."); // Mensaje de conexión
            out.println("Bienvenido al Cine, " + nombreCliente +
"!"); // Mensaje de bienvenida
```

```
            while (true) { // Bucle para mostrar el menú
                out.println("\n--- Menú del Cine ---");
                out.println("1. Ver boletos disponibles");
                out.println("2. Comprar boletos");
                out.println("3. Salir");
                out.println("Seleccione una opción:");

                String opcionStr = in.readLine(); // Leer opción
del cliente
                if (opcionStr == null) break;

                int opcion; // Convertir opción a entero
                try {
                    opcion = Integer.parseInt(opcionStr);
                } catch (NumberFormatException e) {
                    out.println("Opción no válida. Intente de
nuevo.");
                    continue;
                }

                switch (opcion) { // Switch para manejar las
opciones
                    case 1 -> out.println("Boletos disponibles: "
+ cine.getBoletosDisponibles());
                    case 2 -> { // Comprar boletos
                        out.println("Ingrese la cantidad de
boletos que desea comprar:");
                        String cantidadStr = in.readLine();
                        int cantidad;
                        try { // Convertir cantidad a entero
                            cantidad =
Integer.parseInt(cantidadStr);
                        } catch (NumberFormatException e) {
                            out.println("Cantidad no válida.
Intente de nuevo.");
                            continue;
                        } // Comprar boletos
```

```
                        String respuesta =
cine.comprarBoleto(nombreCliente, cantidad);
                        out.println(respuesta);
                    }
                    case 3 -> { // Salir
                        out.println("Gracias por su compra. ¡Hasta
luego!");
                        System.out.println(nombreCliente + " se ha
desconectado.");
                    } // Salir del bucle si elige salir
                    default -> out.println("Opción no válida.
Intente de nuevo.");
                }
            }
        } catch (IOException e) { // Excepción de entrada/salida
            System.err.println("Error en la conexión con el
cliente: " + e.getMessage());
        } finally {
            try {
                socket.close();
            } catch (IOException e) { // Excepción de
entrada/salida
                System.err.println("Error al cerrar el socket: " +
e.getMessage());
            }
        }
    }
}

// Servidor principal
public class ServidorCine { // Clase principal
    public static void main(String[] args) { // Método principal
        int puerto = 8888; // Puerto del servidor
        Cine cine = new Cine(30); // Crear cine con 30 boletos
disponibles
        ThreadPoolExecutor pool = (ThreadPoolExecutor)
Executors.newFixedThreadPool(4); // Pool de hilos
```

```
        new ReposicionBoletos(cine).start(); // Iniciar hilo de
reposición de boletos

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

            while (true) { // Bucle para aceptar conexiones
                Socket socketCliente = serverSocket.accept();
                pool.execute(new HiloCliente(socketCliente,
cine));
            }
        } catch (IOException e) { // Excepción de entrada/salida
            System.err.println("Error en el servidor: " +
e.getMessage());
        }
    }
}
```

## Código Fuente ClienteCine.java

```
/*
 * Nombre: Cardoso Osorio Atl Yosafat
 * Grupo: 7CM1
 * Materia: Sistemas Distribuidos
 * Practica 2: Modelo Cliente/Servidor
 * Fecha: 01 de Marzo del 2025
 */
import java.io.*;
import java.net.*;
import java.util.Scanner;

public class ClienteCine { // Clase ClienteCine
    public static void main(String[] args) {
        final String SERVIDOR = "127.0.0.1"; // Dirección del
servidor (localhost)
        final int PUERTO = 8888; // Mismo puerto que el servidor
```

```
        try (Socket socket = new Socket(SERVIDOR, PUERTO);
             BufferedReader entrada = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
             PrintWriter salida = new
PrintWriter(socket.getOutputStream(), true);
             Scanner scanner = new Scanner(System.in)) {

            System.out.println("Conectado al cine."); // Mensaje
de conexión exitosa

            // Ingresar nombre del cliente
            System.out.print("Ingrese su nombre: "); // Solicitar
nombre al cliente
            String nombre = scanner.nextLine();
            salida.println(nombre); // Enviar nombre al servidor
            System.out.println(entrada.readLine()); // Mensaje de
bienvenida del servidor

            while (true) {
                // Leer y mostrar el menú hasta que se reciba
"Seleccione una opción:"
                String linea;
                while (!(linea =
entrada.readLine()).contains("Seleccione una opción")) {
                    System.out.println(linea);
                }
                System.out.println(linea); // Imprime "Seleccione
una opción:"

                // Leer opción del usuario
                System.out.print("Opción: "); // Solicitar opción
al cliente
                String opcionStr = scanner.nextLine();
                salida.println(opcionStr); // Enviar opción al
servidor

                switch (opcionStr) {
                    case "1", "3" -> {
```

```
                        // Si es ver funciones o salir, recibir
respuesta del servidor
                        System.out.println("Servidor: " +
entrada.readLine());
                        if (opcionStr.equals("3")) {
                            System.out.println("Saliendo del
cine... ¡Nos vemos Cinefilo!");
                            return; // Salir del método si elige
salir
                        }
                    }
                    case "2" -> {
                        // Comprar boletos
                        System.out.println(entrada.readLine()); //
Esperar mensaje del servidor "Ingrese la cantidad..."
                        String cantidad = scanner.nextLine();
                        salida.println(cantidad); // Enviar
cantidad de boletos al servidor
                        String respuesta = entrada.readLine(); //
Recibir respuesta de compra
                        System.out.println("Servidor: " +
respuesta);
                    }
                    default -> System.out.println("Opción no
válida.");
                }
            }
        } catch (IOException e) { // Excepción de entrada/salida
            System.err.println("Error en el cliente: " +
e.getMessage());
        }
    }
}
```