INSTITUTO POLITÉCTICO NACIONAL

Escuela Superior de Cómputo

Carrera:

Ingeniería en Sistemas Computacionales

Unidad de aprendizaje:

Sistemas Distribuidos

Practica 5:

Objetos Distribuidos

Alumno:

Cardoso Osorio Atl Yosafat

Profesor:

Chadwick Carreto Arellano

Fecha:

25/03/2025

INSTITUTO POLITÉCNICO NACIONAL

ESCOM

Índice de contenido

Índice de figuras

Antecedentes

Los sistemas distribuidos son un conjunto de computadoras interconectadas que colaboran para brindar un servicio coordinado. A diferencia de los sistemas centralizados, en los que toda la operación depende de un solo servidor, los sistemas distribuidos permiten la distribución de tareas y datos entre distintos nodos. Esta arquitectura proporciona beneficios como la escalabilidad, la tolerancia a fallos y una mayor eficiencia en el uso de recursos.

En un entorno distribuido, las aplicaciones requieren mecanismos eficientes para la comunicación entre componentes. Java RMI (Remote Method Invocation) es una solución que facilita esta comunicación mediante la invocación remota de métodos en objetos distribuidos. Este mecanismo permite que los programas interactúen como si estuvieran ejecutándose localmente, ocultando la complejidad de las operaciones de red.

Objetos Distribuidos

Los objetos distribuidos son una abstracción de la programación orientada a objetos aplicada en entornos distribuidos. En términos generales, un objeto distribuido es una instancia de una clase que reside en un entorno de red y puede ser accedido de manera remota por aplicaciones ubicadas en diferentes máquinas. Esto significa que los métodos de un objeto pueden ser invocados por clientes remotos como si fueran locales, lo que facilita la creación de sistemas complejos y escalables.

Uno de los principales beneficios de los objetos distribuidos es la capacidad de descomponer una aplicación en múltiples componentes, cada uno ejecutándose en diferentes nodos de una red. Esta distribución de componentes permite mejorar la disponibilidad, balancear la carga y aumentar la tolerancia a fallos. Además, los objetos distribuidos son fundamentales en arquitecturas como cliente-servidor o peer-to-peer, donde diferentes partes de una aplicación interactúan y colaboran a través de la red.

Para que un objeto distribuido funcione, es necesario contar con mecanismos que gestionen la comunicación entre el cliente y el objeto remoto. Esto se logra a través de tecnologías como Java RMI (Remote Method Invocation), CORBA (Common Object Request Broker Architecture) o gRPC. Estas tecnologías permiten la serialización de objetos, la transmisión de datos a través de la red y la invocación de métodos de manera transparente. En esencia, el cliente realiza una llamada a un método remoto sin preocuparse por los detalles de la red, ya que el middleware se encarga de gestionar la comunicación.

Además, los objetos distribuidos suelen incluir características adicionales como la gestión de concurrencia, la seguridad en las comunicaciones y el manejo de excepciones remotas. También pueden emplear registros de objetos o naming services para localizar y referenciar objetos distribuidos a través de la red. Esto facilita la escalabilidad y permite que los objetos se ubiquen dinámicamente según la demanda del sistema.

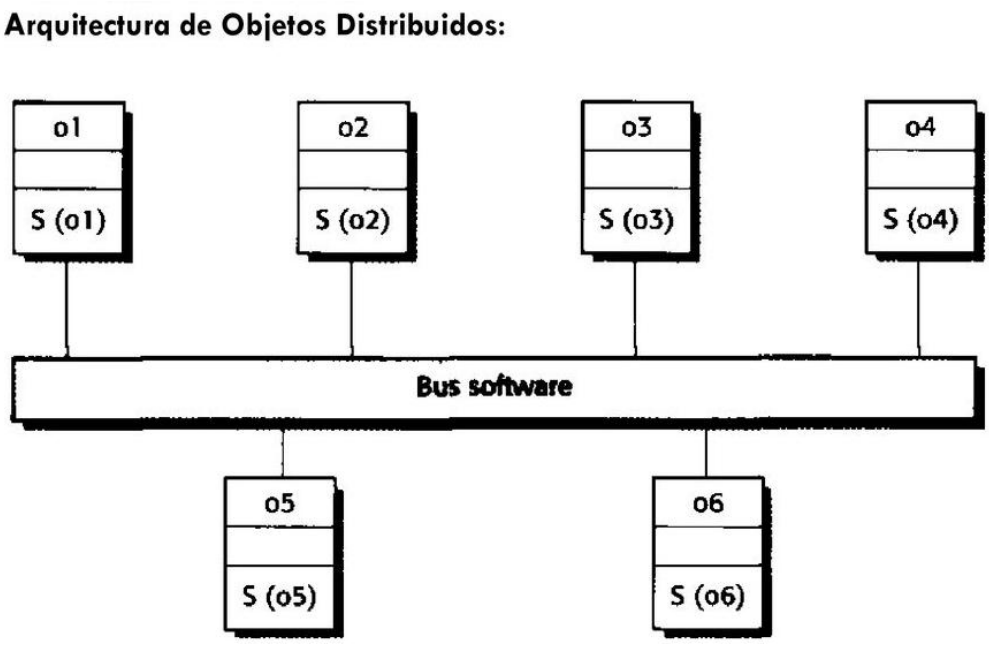

Figura 1. Diagrama de la arquitectura de Objetos Distribuidos

Java RMI (Remote Method Invocation)

Java RMI (Remote Method Invocation) es una tecnología que permite a los objetos en diferentes máquinas virtuales de Java (JVM) comunicarse entre sí mediante la invocación de métodos remotos. Esta herramienta forma parte del paquete java.rmi y está diseñada para facilitar la creación de aplicaciones distribuidas, donde los objetos pueden residir en servidores y ser utilizados por clientes de forma transparente, como si estuvieran en la misma máquina.

La arquitectura de RMI se basa en un modelo cliente-servidor. El servidor aloja los objetos remotos y los registra en un Registro RMI (rmiregistry), que actúa como un directorio de nombres para localizar y acceder a los objetos. Por otro lado, el cliente obtiene una referencia al objeto remoto utilizando el registro y puede invocar sus métodos sin conocer los detalles de su implementación. Esta abstracción proporciona una comunicación eficiente y sencilla para aplicaciones distribuidas.

Uno de los principales componentes de Java RMI es la Interfaz Remota (Remote Interface). Esta interfaz define los métodos que pueden ser invocados de forma remota y debe extender la interfaz java.rmi.Remote. Además, todos los métodos declarados deben especificar la excepción RemoteException, que maneja posibles fallos en la comunicación de red. La implementación del servidor se realiza extendiendo la interfaz remota y utilizando la clase UnicastRemoteObject para exportar los objetos y hacerlos accesibles para los clientes.

La comunicación en RMI se basa en la serialización de objetos, lo que permite transferir datos complejos a través de la red. Cuando un cliente invoca un método remoto, los

parámetros y el resultado son serializados y transmitidos mediante el protocolo RMI. Este protocolo utiliza TCP/IP para garantizar una comunicación confiable y segura. Además, RMI puede integrarse con SSL (Secure Sockets Layer) para proporcionar cifrado y autenticación en entornos que requieran seguridad adicional.

Para la ejecución de aplicaciones RMI, es necesario seguir varios pasos fundamentales. Primero, se debe definir la interfaz remota, implementarla en el servidor y crear un stub (representación local del objeto remoto) que actúa como intermediario. Luego, el servidor registra el objeto en el Registro RMI, permitiendo que los clientes lo localicen y lo utilicen. Finalmente, los clientes buscan el objeto remoto, obtienen una referencia y realizan invocaciones a sus métodos como si estuvieran ejecutándose localmente.

En términos de aplicaciones prácticas, Java RMI es ampliamente utilizado en sistemas distribuidos como aplicaciones de banca en línea, juegos en red, aplicaciones empresariales y sistemas de monitoreo remoto. Su capacidad para gestionar la comunicación de objetos remotos de manera eficiente lo convierte en una opción ideal para proyectos que requieren interacción entre múltiples componentes distribuidos.

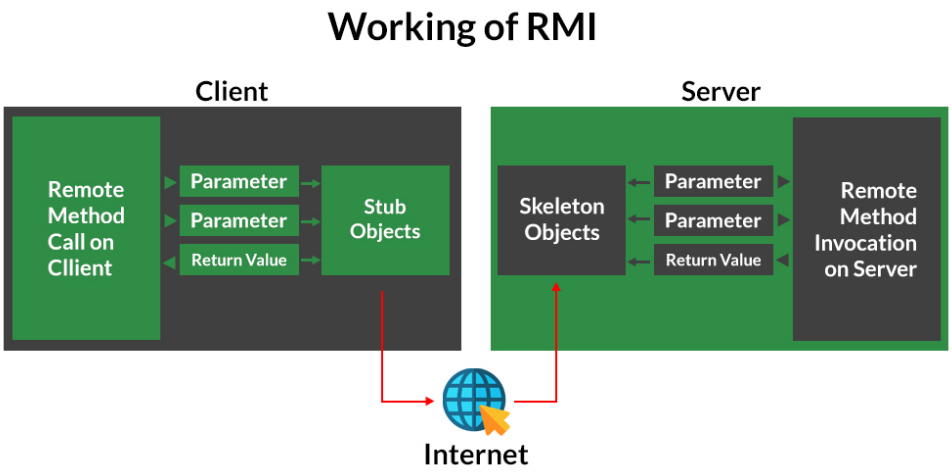

Figura 2. Método Java RMI (Remote Method Invocation)

Protocolo de comunicación en RMI

El protocolo de comunicación en RMI (Remote Method Invocation) es el mecanismo mediante el cual los objetos en diferentes máquinas virtuales de Java (JVM) pueden intercambiar información y ejecutar métodos de forma remota. Este protocolo establece las reglas y procedimientos necesarios para que un cliente envíe solicitudes a un objeto remoto y reciba respuestas a través de una red. RMI utiliza un modelo de comunicación basado en TCP/IP, garantizando una conexión confiable y estable entre los componentes distribuidos.

El protocolo de comunicación en RMI opera en tres niveles principales: el nivel de aplicación, el nivel de transporte y el nivel de referencia. El nivel de aplicación gestiona las solicitudes del cliente y la ejecución de los métodos en el servidor. El nivel de referencia se

encarga de la localización de objetos remotos y la creación de stubs y skeletons. Por último, el nivel de transporte utiliza sockets TCP para la transferencia de datos entre las JVMs, asegurando una comunicación eficiente y segura.

Cuando un cliente realiza una invocación remota, RMI utiliza un proceso llamado serialización de objetos. Este proceso convierte el estado de un objeto en una secuencia de bytes que puede ser enviada a través de la red. Tanto los parámetros del método como los valores de retorno son serializados en el cliente, transmitidos al servidor y deserializados para su ejecución. El resultado de la operación se serializa nuevamente en el servidor y se envía de regreso al cliente. Este mecanismo permite que los objetos complejos sean transferidos entre máquinas sin perder su estructura o estado.

El protocolo RMI también utiliza un sistema de referencia llamado RMI Registry para localizar y acceder a los objetos remotos. Cuando un servidor publica un objeto remoto, lo registra en el RMI Registry utilizando un nombre específico. Los clientes, a su vez, consultan el registro para obtener una referencia al objeto remoto utilizando la dirección IP y el puerto correspondiente. Esta referencia actúa como un stub o proxy local, que envía las solicitudes al objeto remoto a través de la red.

Para garantizar la seguridad y confiabilidad de la comunicación, RMI puede utilizar protocolos de transporte seguros como SSL (Secure Sockets Layer) o TLS (Transport Layer Security). Estas tecnologías permiten cifrar los datos transmitidos, protegiéndolos de posibles interceptaciones o ataques maliciosos. Además, RMI proporciona manejo de excepciones a través de la clase RemoteException, lo que facilita la detección y gestión de errores durante la comunicación.

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.

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.

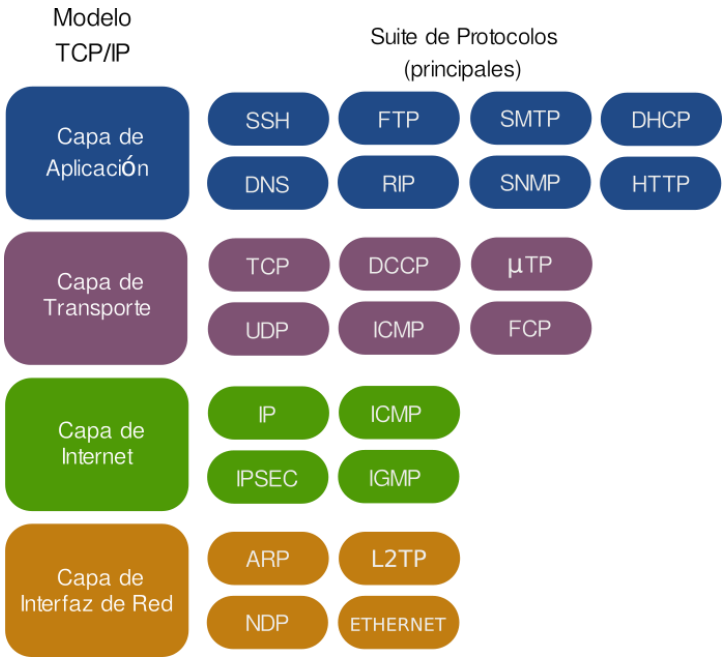

Figura 3. Diagrama de protocolos de comunicación

Modelos de Comunicación en objetos distribuidos

El desarrollo de sistemas distribuidos ha llevado a la creación de diferentes modelos de comunicación que permiten la interacción entre objetos ubicados en distintas máquinas. Estos modelos buscan abstraer la complejidad de la comunicación en red y ofrecer mecanismos que faciliten la integración entre aplicaciones sin importar su ubicación o plataforma de ejecución.

A continuación, se presentan los principales modelos utilizados en la comunicación entre objetos distribuidos, destacando sus características, ventajas y limitaciones.

Modelo de comunicación basado en mensajes

El modelo de comunicación basado en mensajes es uno de los enfoques más fundamentales en los sistemas distribuidos. En este paradigma, los procesos que se ejecutan en diferentes nodos de la red intercambian información a través del envío y recepción de mensajes. Para facilitar esta comunicación, se utilizan protocolos de transporte como TCP (Transmission Control Protocol) y UDP (User Datagram Protocol). TCP proporciona una comunicación confiable y orientada a la conexión, lo que garantiza que los datos lleguen en el orden correcto y sin pérdidas. Por otro lado, UDP es un protocolo más ligero y rápido, pero sin mecanismos de control de flujo ni confirmación de entrega.

Este enfoque es altamente flexible y permite un control detallado sobre la transmisión de datos. Sin embargo, presenta ciertas desventajas. En primer lugar, los desarrolladores deben manejar manualmente la estructura de los mensajes, lo que implica definir formatos específicos para la serialización y deserialización de datos. Además, la sincronización entre procesos puede ser compleja, especialmente cuando se deben coordinar múltiples actores en un sistema distribuido. Otra desventaja importante es la necesidad de gestionar explícitamente los errores y fallos en la red, lo que complica la implementación de aplicaciones robustas.

Dado que el modelo de comunicación basado en mensajes introduce una carga significativa de trabajo para los desarrolladores, con el tiempo han surgido modelos más avanzados que buscan abstraer estos problemas. Estos nuevos enfoques simplifican la comunicación en sistemas distribuidos al proporcionar mecanismos que gestionan la serialización, la detección de fallos y la sincronización de procesos de manera más eficiente.

Llamada a Procedimiento Remoto (RPC)

Para reducir la complejidad de la comunicación basada en mensajes, se desarrolló la Llamada a Procedimiento Remoto (Remote Procedure Call, RPC). RPC permite que un proceso ejecute una función en otra máquina remota de la misma manera en que invocaría una función local. Esta tecnología oculta los detalles de la comunicación en la red, permitiendo que los desarrolladores trabajen con un modelo más simple.

El funcionamiento de RPC se basa en el concepto de cliente-servidor. Un cliente envía una solicitud de ejecución a un servidor, el cual procesa la petición y devuelve una respuesta. Para hacer esto posible, se utiliza un mecanismo de serialización de datos conocido como "marshalling" y su correspondiente "unmarshalling" para deserializar los datos en el lado del receptor.

El stub es la pieza de código que le permite al servidor ejecutar la tarea que se le asignó. Se encarga de proveer la información necesaria para que el servidor convierta las direcciones de los parámetros que el cliente envió en direcciones de memoria válidos dentro del ambiente del servidor. La representación de datos en cliente y servidor (big-endian o little-endian) podría discrepar, el stub también provee la información necesaria para solucionar esta situación. De forma general es la pieza de código que se encarga de enmascarar toda discrepancia entre el cliente y servidor. Es necesario que las bibliotecas de stubsestén instaladas tanto en el cliente como en el servidor.

A pesar de sus ventajas, RPC tiene ciertas limitaciones. En primer lugar, está diseñado principalmente para entornos procedimentales, lo que lo hace menos adecuado para sistemas orientados a objetos. Además, dado que se basa en llamadas síncronas, puede generar problemas de latencia y bloquear procesos en espera de una respuesta. Esto ha llevado a la evolución de modelos más sofisticados, como RMI y CORBA, que buscan mejorar la interoperabilidad y la eficiencia en entornos distribuidos. En la siguiente figura podemos observar de forma resumida cómo funciona una llamada a procedimiento remoto:

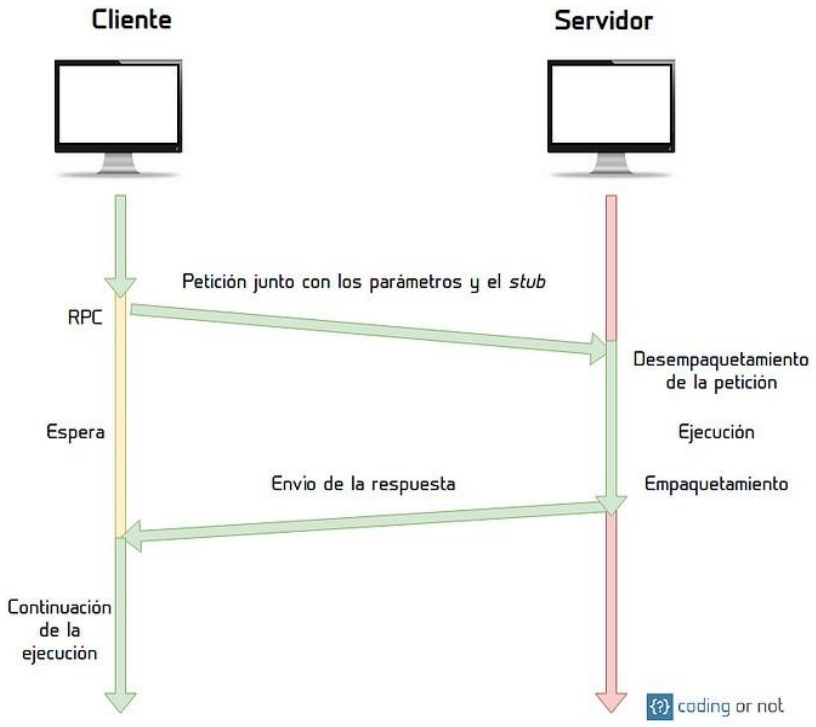

Figura 4. Diagrama del funcionamiento de RPC

RMI (Remote Method Invocation)

Permite a un programa invocar métodos en un objeto ubicado en otra JVM (Máquina de Java). Abstrae la complejidad deVirtual la comunicación en red, haciendo que las llamadas remotas parezcan locales. RMI es una tecnología específica de Java que permite a los objetos de diferentes máquinas comunicarse a través de una red y ejecutar métodos de manera remota. Permite que los objetos en diferentes máquinas JVM (Java Virtual Machines) se comuniquen y ejecuten métodos en un entorno distribuido. RMI utiliza servidores remotos para registrar objetos y clientes que invocan métodos de esos objetos de manera remota. Esta tecnología es parte del estándar de Java, lo cual hace que su uso sea sumamente sencillo.

DCOM (Distributed Component Object Model)

Protocolo de comunicación desarrollado por Microsoft que permite que los componentes de software interactúen entre sí en una red distribuida. Es una extensión del modelo COM (Component Object Model), diseñado para facilitar la comunicación y la interacción entre aplicaciones y componentes de software distribuidos en diferentes máquinas o procesos, incluso en redes de área amplia (WAN) o internet. Actualmente, ha sido en gran medida reemplazado por tecnologías más modernas, como .NET y Web Services en muchos entornos debido a sus limitaciones y la complejidad en la configuración y mantenimiento.

CORBA (Common Object Request Broker Architecture)

Es un estándar abierto para la comunicación entre objetos distribuidos en una red, desarrollado por OMG (Object Management Group). Al igual que DCOM, CORBA permite que los componentes de software interactúen y se comuniquen de manera transparente, independientemente del lenguaje de programación o del sistema operativo en el que estén ejecutándose. Permite que los objetos de diferentes sistemas, escritos en diferentes lenguajes de programación, se comuniquen entre sí. Utiliza el Object Request Broker (ORB) para manejar las solicitudes entre clientes y servidores, permitiendo la invocación de métodos remotos. Sin embargo, presenta complejidad en la implementación y gestión. Puede ser más pesado y menos eficiente en términos de rendimiento que otros modelos más ligeros como RMI.

Comparación entre Modelos de Objetos Distribuidos

A continuación, se presenta una tabla comparativa entre tres modelos de objetos distribuidos: RMI (Remote Method Invocation), CORBA (Common Object Request Broker Architecture) y DCOM (Distributed Component Object Model). Cada uno de estos modelos tiene características distintas en cuanto a lenguajes soportados, plataformas, interoperabilidad, escalabilidad y seguridad, lo que los hace adecuados para diferentes tipos de aplicaciones distribuidas.

Característica	*RMI*	*CORBA*	*DCOM*
Lenguaje principal	Java	Multilenguaje (C++, java, etc)	C++ (principalmente Windows)
Plataforma	Java	Multiplataforma	Principalmente Windows
Protocolo de comunicación	RMI sobre Java	IIOP	DCOM
Interoperabilidad	Solo Java	Alta (multiplataforma)	Limitado a Windows
Escalabilidad	Dentro de Java	Alta, adecuado para grandes sistemas distribuidos	Limitada a Windows
Seguridad	Java Security Manager	SSL/TLS	Proporcionada por Windows

Planteamiento del problema

En el contexto de los sistemas distribuidos, la gestión eficiente de recursos compartidos y la coordinación entre múltiples clientes representan desafíos clave. Uno de estos escenarios es la venta de boletos para un cine, donde la demanda de los clientes puede generar concurrencia y potenciales conflictos al intentar acceder al mismo recurso de manera simultánea.

En un entorno centralizado, la administración del inventario de boletos y el manejo de las solicitudes de compra podrían resultar ineficientes y propensos a errores, especialmente cuando existe una alta concurrencia. Además, la falta de mecanismos adecuados para la sincronización y la gestión de las transacciones podría conducir a problemas como la sobreventa de boletos o la pérdida de datos.

Para abordar estos desafíos, se plantea la implementación de un sistema distribuido utilizando la tecnología Java RMI (Remote Method Invocation). Java RMI permite la comunicación entre objetos ubicados en diferentes máquinas, lo que facilita la distribución de tareas y mejora la escalabilidad del sistema.

Este proyecto propone el desarrollo de un sistema de venta de boletos de cine distribuido, conformado por un servidor RMI y múltiples clientes. El servidor se encargará de administrar el inventario de boletos, procesar las solicitudes de compra, y reponer boletos automáticamente mediante un hilo dedicado. Los clientes podrán conectarse de forma remota

al servidor, consultar la disponibilidad de boletos, realizar compras y recibir confirmaciones en tiempo real. Asimismo, se implementarán mecanismos de sincronización y manejo de transacciones para asegurar la integridad de los datos y evitar inconsistencias.

Adicionalmente, se llevará a cabo un registro de conexiones y desconexiones de los clientes, lo que permitirá un monitoreo efectivo de las actividades del sistema. Esta implementación permitirá evaluar las ventajas del uso de RMI en comparación con otros modelos de comunicación distribuida, destacando aspectos como la facilidad de implementación, la seguridad y la eficiencia en la gestión de recursos.

Propuesta de solución

La propuesta de solución para la gestión eficiente de boletos de cine en un entorno distribuido, se propone el desarrollo de un sistema basado en la tecnología Java RMI (Remote Method Invocation). Este sistema permitirá la comunicación entre objetos distribuidos a través de una red, facilitando la coordinación entre clientes y servidores.

El objetivo principal es implementar un servidor RMI encargado de administrar el inventario de boletos de cine. Este servidor permitirá la conexión remota de múltiples clientes, quienes podrán consultar la disponibilidad de boletos, realizar compras y recibir confirmaciones en tiempo real. Además, se registrarán las conexiones y desconexiones de los clientes para un monitoreo efectivo.

Para garantizar la consistencia de los datos, se utilizarán transacciones y bloqueo de registros mediante SQL. Este enfoque evitará problemas como la sobreventa de boletos o la corrupción de la información. Asimismo, se implementará un sistema de reposición automática que añadirá boletos al inventario en intervalos regulares, asegurando una disponibilidad continua.

La solución estará estructurada en tres componentes principales. Primero, una interfaz remota CineRMI que definirá los métodos disponibles para los clientes, como la consulta de boletos, la compra y el registro de actividades. Segundo, el servidor RMI, que administrará las solicitudes de los clientes, ejecutará las transacciones y garantizará la integridad de los datos. Por último, un cliente RMI que permitirá a los usuarios interactuar con el sistema a través de una interfaz de consola.

Para manejar la concurrencia, el sistema empleará la instrucción FOR UPDATE en SQL, bloqueando los registros afectados durante las transacciones. Además, el método comprarBoleto estará sincronizado para garantizar que las solicitudes de compra se procesen de manera ordenada.

El servidor también mantendrá un registro detallado de las actividades, incluyendo las conexiones y desconexiones de los clientes, así como las transacciones realizadas. Esta información facilitará el análisis del comportamiento del sistema y permitirá identificar posibles mejoras.

Entre los principales beneficios esperados se encuentra la reducción de inconsistencias en la venta de boletos, el incremento en la eficiencia de la gestión del inventario y una mejor experiencia del usuario al recibir confirmaciones en tiempo real. Además, la solución será escalable y confiable, permitiendo la conexión de múltiples clientes simultáneamente.

Materiales y métodos empleados

Para llevar a cabo la práctica y desarrollar esta práctica con Objetos Distribuidos para resolver el problema de la venta de boletos se emplearon las siguientes herramientas y métodos:

Materiales

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

El lenguaje de programación Java fue elegido debido a su robustez y amplia compatibilidad con aplicaciones distribuidas. Java proporciona una API nativa para la implementación de RMI (Remote Method Invocation), lo que facilita la creación de aplicaciones cliente-servidor que pueden comunicarse a través de una red. Además, las características de Java para el manejo de hilos y concurrencia fueron fundamentales para gestionar múltiples solicitudes de clientes de manera eficiente.

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

Fue seleccionado como entorno de desarrollo por su flexibilidad, amplia disponibilidad de extensiones y facilidad de uso. Además, cuenta con herramientas integradas para la depuración y la administración de proyectos en Java. Así como también es bastante cómodo trabajar en él.

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

El JDK fue utilizado para compilar y ejecutar el código Java. Su compatibilidad con las bibliotecas y la implementación de concurrencia lo convierten en una herramienta esencial para esta práctica.

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

La práctica se realizó en un sistema operativo Windows debido a su amplia compatibilidad con Visual Studio Code y el JDK.

5. *Base de Datos:* ***MySQL***

MySQL se usó para gestionar el inventario de la panadería y almacenar las transacciones de compra, asegurando la persistencia y consistencia de los datos.

6. *Biblioteca para la Base de Datos:* ***JDBC***

Se empleó JDBC para conectar el servidor Java con la base de datos MySQL y realizar operaciones de lectura y escritura de datos.

Métodos

Para la implementación del sistema distribuido basado en RMI, se emplearon diversos métodos y técnicas que garantizaron la correcta comunicación entre los clientes y el servidor, la concurrencia en la ejecución de métodos remotos y la sincronización de objetos compartidos en el entorno distribuido. A continuación, se describen los principales métodos utilizados:

Arquitectura Cliente-Servidor Distribuida:

Se implementó un sistema cliente-servidor distribuido mediante RMI, donde el servidor gestiona el inventario de boletos y los clientes interactúan de manera remota. Esta arquitectura permitió que los clientes pudieran realizar operaciones de compra y consulta de boletos a través de la red, invocando métodos remotos en el servidor para obtener información y realizar cambios en el estado de los boletos.

Concurrencia en Base de Datos:

Se utilizó JDBC con transacciones SQL para manejar compras simultáneas de boletos, asegurando la coherencia de los datos y evitando inconsistencias en el inventario. Gracias a las transacciones, se logró que múltiples clientes pudieran realizar compras sin generar conflictos o cambios erróneos en el stock, incluso si las solicitudes eran concurrentes.

Sincronización de Datos:

A través de RMI, los clientes consultan y actualizan el inventario de boletos en tiempo real, garantizando que todos los usuarios tengan acceso a datos actualizados y consistentes. Esta sincronización es clave para evitar que los clientes compren boletos de manera incorrecta debido a un estado desactualizado del inventario, manteniendo la integridad del sistema en todo momento.

Manejo de Conexiones Concurrentes:

El servidor maneja múltiples clientes de manera concurrente mediante hilos (threads), optimizando el procesamiento de solicitudes y permitiendo que varios usuarios puedan interactuar con el sistema de manera simultánea sin interferir entre sí. El uso de hilos también permitió que el servidor fuera capaz de añadir boletos al inventario de forma periódica, sin bloquear el acceso de los clientes al sistema.

Con estas herramientas y métodos, se desarrolló un sistema eficiente para la simulación de un cine distribuido con objetos distribuidos y RMI. La implementación de RMI y técnicas de concurrencia y sincronización permitió garantizar un sistema robusto y escalable, apto para gestionar múltiples solicitudes de clientes sin perder consistencia ni rendimiento.

Desarrollo

En esta práctica se implementó un sistema distribuido utilizando Java RMI (Remote Method Invocation) para gestionar una aplicación de cine. Este sistema incluye funcionalidades como la gestión de boletos, compra de boletos y registro de conexiones de clientes a través de un servidor centralizado. También cabe aclarar que en esta práctica se diseño un sistema de Cine que permite a los clientes consultar y comprar boletos de cine de manera remota, a través de un servidor RMI que gestiona la disponibilidad de boletos y la conexión de clientes. El servidor también mantiene un stock de boletos, que se repone automáticamente utilizando un hilo en segundo plano.

Primeramente, se realizó la clase CineRMI, la cual representa La interfaz remota define los métodos que pueden ser invocados por el cliente de forma remota. En nuestro caso, la interfaz tiene métodos para obtener boletos disponibles, comprar boletos, y registrar la conexión y desconexión de clientes.

```java
import java.rmi.Remote;
import java.rmi.RemoteException;
// Métodos remotos
// Interfaz remota
public interface CineRMI extends Remote { // Interfaz remota
    int obtenerBoletosDisponibles() throws RemoteException; // Métodos remotos
    boolean comprarBoleto(String nombreCliente, int cantidad) throws RemoteException;
    void registrarConexion(String nombreCliente, String ipCliente) throws RemoteException;
    void registrarDesconexion(String nombreCliente) throws RemoteException;
}
```

Figura 5. Interfaz remota CineRMI

Por otro lado, tenemos al Servidor CineServidorRMI, este código implementa un servidor RMI (Remote Method Invocation) para la gestión de boletos en un cine. El servidor expone varios métodos remotos que permiten a los clientes interactuar de manera remota con el sistema. La clase principal, CineServidorRMI, extiende UnicastRemoteObject e implementa la interfaz CineRMI, lo que le permite ser invocada remotamente por los clientes. Además, el servidor se conecta a una base de datos MySQL donde almacena el stock de boletos, lo que permite consultar y actualizar la disponibilidad de boletos en tiempo real.

```java
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.sql.*;
import java.time.LocalDateTime;

public class CineServidorRMI extends UnicastRemoteObject implements CineRMI {// Clase que implementa la i
    private static final String URL = "jdbc:mysql://localhost:3306/cine?useSSL=false&serverTimezone=UTC";
    private static final String USER = "root"; // Usuario de la base de datos
    private static final String PASSWORD = "Atl1God$"; // Contraseña de la base de datos

    protected CineServidorRMI() throws RemoteException { // Constructor
        super(); // Llama al constructor de la clase padre
        // Iniciar el hilo que generará boletos en intervalos
        Thread repositorBoletos = new Thread(new RepositorBoletos(this));
        repositorBoletos.start(); // Inicia el hilo    Starting new Thread in constructor
    }
```

Figura 6. CineServidorRMI y conexión a la base de datos

Uno de los métodos más importantes del servidor es obtenerBoletosDisponibles(), que consulta la base de datos para obtener la cantidad de boletos disponibles. Utiliza una conexión SQL para ejecutar una consulta y obtener el valor del campo stock de la tabla boletos. Por otro lado, el método comprarBoleto(String nombreCliente, int cantidad) permite a los clientes realizar compras de boletos. Este método verifica si hay suficiente stock disponible y, si es así, actualiza la base de datos, restando los boletos comprados. Si no hay suficientes boletos, la transacción se revierte, asegurando que el sistema no permita compras cuando no hay disponibilidad. Este método está sincronizado para evitar problemas de concurrencia entre varios clientes intentando realizar compras al mismo tiempo.

```java
@Override // Implementación de los métodos remotos
public int obtenerBoletosDisponibles() throws RemoteException {
    int stock = 0; // Inicializar el stock
    try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);
         Statement stmt = conn.createStatement(); // Crear una sentencia
         ResultSet rs = stmt.executeQuery(sql:"SELECT stock FROM boletos WHERE producto='Boleto'")
    ) { // Ejecutar una consulta
        if (rs.next()) { /// Si hay un resultado
            stock = rs.getInt(columnLabel:"stock");
        }
    } catch (SQLException e) {
        e.printStackTrace();    Print Stack Trace
    }
    return stock;
}
```

Figura 7. Clase para manejar la obtención de boletos disponibles

```java
@Override // Implementación de los métodos remotos
public synchronized boolean comprarBoleto(String nombreCliente, int cantidad) throws RemoteException {
    try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD)) { // Crear una conexión
        conn.setAutoCommit(autoCommit:false); // Deshabilitar el modo de autocommit

        try (Statement stmt = conn.createStatement(); // Crear una sentencia
             ResultSet rs = stmt.executeQuery(sql:"SELECT stock FROM boletos WHERE producto='Boleto' FOR UPDATE")) {
            if (rs.next()) { // Si hay un resultado
                int stockActual = rs.getInt(columnLabel:"stock"); // Obtener el stock actual
                if (stockActual >= cantidad) { // Si hay suficientes boletos
                    try (PreparedStatement pstmt = conn.prepareStatement( // Crear una sentencia preparada
                            sql:"UPDATE boletos SET stock = stock - ? WHERE producto='Boleto'")) { // Actualizar el st
                        pstmt.setInt(parameterIndex:1, cantidad); // Establecer el valor del parámetro
                        pstmt.executeUpdate(); // Ejecutar la sentencia
                    }
                    conn.commit(); // Confirmar la transacción
                    System.out.println("[" + LocalDateTime.now() + "] Cliente '" + nombreCliente + "' compró " + canti
                    return true; // Compra exitosa
                }
            }
        }
        conn.rollback();
    } catch (SQLException e) {
        e.printStackTrace();    Print Stack Trace
    }
    return false;
}
```

Figura 8. Clase para comprar boletos disponibles

El método anadirStock(int cantidad) permite agregar boletos al stock del cine. Cada vez que se ejecuta, el sistema actualiza la base de datos y luego imprime el nuevo stock disponible. Para mejorar la experiencia del usuario, el servidor también tiene métodos para registrar las conexiones y desconexiones de los clientes, imprimiendo mensajes en la consola con el nombre y la dirección IP de los clientes cuando se conectan, y el nombre cuando se desconectan. Esto facilita el monitoreo de las interacciones del sistema.

```java
public void anadirStock(int cantidad) throws RemoteException { // Método para añadir boletos al stock
    try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);
         PreparedStatement pstmt = conn.prepareStatement(sql:"UPDATE boletos SET stock = stock + ? WHERE producto='Boleto'")) {
        pstmt.setInt(parameterIndex:1, cantidad); // Establecer el valor del parámetro
        pstmt.executeUpdate();
        int nuevoStock = obtenerBoletosDisponibles(); // Obtener el stock actualizado
        System.out.println("Se añadieron " + cantidad + " boletos. Nuevo stock: " + nuevoStock);
    } catch (SQLException e) { // Capturar excepciones
        e.printStackTrace(); // Imprimir la traza de la pila    Print Stack Trace
    }
```

Figura 9. Clase para añadir nuevo stock o para manejar la reposición de boletos

Una parte interesante del sistema es el hilo RepositorBoletos, que se ejecuta de manera continua en segundo plano. Este hilo se encarga de reponer automáticamente los boletos cada 10 segundos. Cada vez que pasa este intervalo de tiempo, el hilo invoca el método

anadirStock(10), lo que agrega 10 boletos al stock de la base de datos. De esta manera, se asegura que el stock esté siempre disponible para las compras.

```
// Hilo que repone boletos automáticamente
private static class RepositorBoletos implements Runnable {
    private final CineServidorRMI servidor;

    public RepositorBoletos(CineServidorRMI servidor) {
        this.servidor = servidor; // Inicializar el servidor
    }

    @Override
    public void run() { // Método run
        try { // Capturar excepciones
            while (true) { // Bucle infinito
                Thread.sleep(millis:10000); // Espera 10 segu
                servidor.anadirStock(cantidad:10); // Añadir
            }
        } catch (InterruptedException | RemoteException e) {
            e.printStackTrace();    Print Stack Trace
        }
```

Figura 10. Repositor de boletos

Finalmente, el servidor maneja las conexiones y transacciones de manera segura. Al deshabilitar el autocommit y usar transacciones SQL explícitas, se garantiza que las actualizaciones al stock se realicen correctamente. Si ocurre un error durante el proceso de compra, como la falta de boletos suficientes, la transacción se revierte para evitar que se realicen compras inconsistentes. Con esto, el servidor RMI proporciona un sistema robusto y confiable para gestionar la venta de boletos en el cine.

Posteriormente se realizó el código para el ServidorRMI, encargado de iniciar y registrar un servidor RMI para la gestión de boletos en un cine. Su función principal es crear una instancia del servidor CineServidorRMI y publicarla en un registro RMI, permitiendo que los clientes se conecten y realicen solicitudes remotas. Para ello, utiliza el puerto 1099, que es el puerto predeterminado para RMI.

```
import java.rmi.Naming;
import java.rmi.registry.LocateRegistry;

ublic class ServidorRMI { // Clase principal
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {// Método principa
        try {
            // Crear el servidor RMI
            CineServidorRMI servidor = new CineServidorRMI();
            int puerto = 1099; // Puerto por defecto
```

Figura 11. Código ServidorRMI y creacion de una instancia del servidor CineServidorRMI

En primer lugar, el código crear un registro RMI mediante LocateRegistry.createRegistry(). Si el puerto 1099 ya está en uso, el sistema captura la excepción y automáticamente intenta utilizar el puerto 1098 como alternativa. Esto garantiza que el servidor pueda iniciar incluso si el puerto predeterminado no está disponible. El manejo de excepciones brinda robustez al sistema, permitiendo que el servidor se adapte a diferentes escenarios de red.

```
try {
    LocateRegistry.createRegistry(puerto); // Crear un registro RMI en el puerto especificado
    System.out.println("Registro RMI creado en el puerto " + puerto); // Mensaje de éxito
} catch (Exception e) {    Can be replaced with multicatch or several catch clauses catching speci
    System.out.println("El puerto " + puerto + " ya está en uso, intentando con el puerto 1098.");
    LocateRegistry.createRegistry(port:1098); // Crear un registro RMI en el puerto 1098
    puerto = 1098; // Actualizar el puerto
}
```

Figura 12. Creación de un registro RMI

Una vez que se establece el registro RMI, el servidor CineServidorRMI se vincula al registro utilizando Naming.rebind(), asignándole el nombre lógico "Cine". Esto facilita que los clientes lo localicen e invoquen métodos remotos a través de la URL rmi://localhost:<puerto>/Cine. Finalmente, el servidor imprime mensajes de confirmación para indicar que el registro RMI se ha creado con éxito y que el servidor de cine está activo.

```
    // Registrar el servidor con el nombre "Cine"
    Naming.rebind("rmi://localhost:" + puerto + "/Cine", servidor); // Registrar
    System.out.println("Servidor RMI de cine iniciado en el puerto " + puerto);
} catch (Exception e) {    Can be replaced with multicatch or several catch clau
    e.printStackTrace();    Print Stack Trace
}
```

Figura 13. Establecimiento del registro RMI y asignación de nombre "Cine"

Por último, tenemos la implementación del cliente RMI que permite a los usuarios conectarse a un servidor remoto para gestionar la compra de boletos de cine. La clase ClienteRMI utiliza la interfaz CineRMI para invocar métodos remotos, facilitando la interacción entre el cliente y el servidor mediante el protocolo RMI.

```
import java.net.InetAddress;
import java.rmi.Naming;
import java.util.Scanner;

public class ClienteRMI {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        try {
            // Conectar con el servidor RMI
            CineRMI cine = (CineRMI) Naming.lookup(name:"rmi://localhost/Cine");
            System.out.println(x:"Conexión exitosa con el servidor RMI");
            Scanner scanner = new Scanner(System.in);    Resource leak: 'scanner
```

Figura 14. Clase ClienteRMI para invocar métodos remotos

Al iniciar, el cliente establece una conexión con el servidor RMI utilizando Naming.lookup(), accediendo a la URL rmi://localhost/Cine. Si la conexión es exitosa, se muestra un mensaje de confirmación. Luego, el cliente solicita al usuario que ingrese su nombre y obtiene automáticamente la dirección IP del cliente utilizando InetAddress.getLocalHost().getHostAddress(). Esta información se envía al servidor mediante el método remoto registrarConexion(), registrando la conexión del cliente.

Después de eso el sistema presenta un menú interactivo que ofrece tres opciones principales: consultar la cantidad de boletos disponibles, realizar una compra de boletos o salir del sistema. Si el usuario selecciona la opción de ver boletos, el cliente ejecuta el método obtenerBoletosDisponibles() y muestra la cantidad actual de boletos disponibles. Si el usuario elige comprar boletos, se solicita la cantidad deseada y se invoca el método comprarBoleto(). Dependiendo del resultado de la compra, se muestra un mensaje indicando si la transacción fue exitosa o si no hay suficientes boletos disponibles.

```
// Solicitar el nombre del cliente
System.out.print(s:"Por favor, ingrese su nombre: ");
String nombreCliente = scanner.nextLine();
String ipCliente = InetAddress.getLocalHost().getHostAddress();

// Notificar la conexión al servidor
cine.registrarConexion(nombreCliente, ipCliente);
System.out.println("¡Bienvenido, " + nombreCliente + "!");

while (true) {
    System.out.println(x:"\n--- Menú Cine ---");
    System.out.println(x:"1. Ver boletos disponibles");
    System.out.println(x:"2. Comprar boletos");
    System.out.println(x:"3. Salir");
    System.out.print(s:"Seleccione una opción: ");

    int opcion = scanner.nextInt();

    switch (opcion) {
        case 1 -> System.out.println("Boletos disponibles: " + cine.obtenerBoletosDisponibles());
        case 2 -> {
            System.out.print(s:"Ingrese la cantidad de boletos que desea comprar: ");
            int cantidad = scanner.nextInt();
            boolean compraExitosa = cine.comprarBoleto(nombreCliente, cantidad);
            if (compraExitosa) {
                System.out.println("Compra exitosa, " + nombreCliente + ". Boletos restantes: " +
            } else {
                System.out.println(x:"Lo sentimos, no hay suficientes boletos disponibles.");
            }
        }
```

Figura 15. Menú de usuario

En caso de que el usuario decida salir, el cliente ejecuta el método registrarDesconexion(), notificando al servidor que el usuario se ha desconectado. Finalmente, el programa termina y muestra un mensaje de despedida.

```
            case 3 -> {
                System.out.println("Gracias por visitar el cine, " + nombreCliente + ". ¡Hasta luego!");
                cine.registrarDesconexion(nombreCliente);
                return;
            }
            default -> System.out.println(x:"Opción no válida. Intente de nuevo.");
        }
    }
catch (Exception e) {    Can be replaced with multicatch or several catch clauses catching specific ex
    System.out.println(x:"Error al conectar con el servidor RMI:");
    e.printStackTrace();    Print Stack Trace
```

Figura 16. Desconexión del usuario

Este diseño basado en RMI permite que las operaciones realizadas por el cliente sean procesadas de forma remota en el servidor, garantizando la consistencia de los datos. Además, la interacción en tiempo real y el manejo de conexiones mediante RMI hacen que el sistema sea eficiente y adecuado para la gestión de boletos en un entorno distribuido.

Diseño de la Base de Datos

La base de datos se compone de una tabla principal llamada boletos, que contiene la siguiente estructura:

```
mysql> CREATE TABLE boletos (
    ->     producto VARCHAR(50) NOT NULL,  -- Nombre del producto (por ejemplo, Boleto de Cine)
    ->     stock INT NOT NULL,             -- Cantidad de boletos disponibles
    ->     PRIMARY KEY (producto)          -- La columna 'producto' será la clave primaria
    -> );
```

```
mysql> DESCRIBE boletos;
+----------+-------------+------+-----+---------+-------+
| Field    | Type        | Null | Key | Default | Extra |
+----------+-------------+------+-----+---------+-------+
| producto | varchar(50) | NO   | PRI | NULL    |       |
| stock    | int         | NO   |     | NULL    |       |
+----------+-------------+------+-----+---------+-------+
2 rows in set (0.00 sec)
```

Figura 17. Estructura de la base de datos

- Producto: un campo de texto (máximo 50 caracteres) que no puede ser nulo.
- Stock: un campo numérico (entero) que representa la cantidad de boletos disponibles, también no nulo.
- La columna producto se establece como clave primaria, lo que significa que cada nombre de producto debe ser único.

Resultados

En la implementación del sistema de cine con Java RMI, se verificó exitosamente el registro y conexión de los clientes al servidor RMI. Cada cliente se conecta al servidor a través del Naming.lookup(), lo que permite localizar el objeto remoto a partir del nombre registrado en el servicio RMI Registry. Durante las pruebas, se observó que el servidor captura y muestra correctamente la dirección IP del cliente y su nombre mediante el método registrarConexion(), lo que facilita el monitoreo de las conexiones.

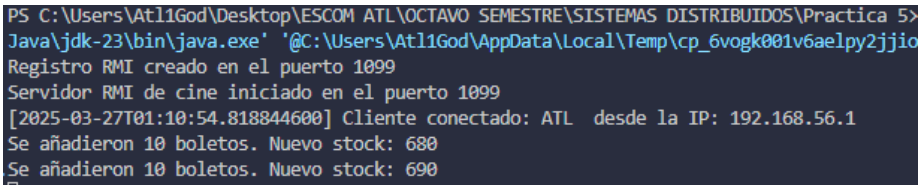

Figura 18. Monitoreo de usuarios

El método obtenerBoletosDisponibles() consulta la base de datos para obtener el stock actualizado de boletos. Las pruebas demostraron que los resultados devueltos fueron consistentes con la información almacenada en la base de datos. Además, al realizar compras, el stock se actualiza adecuadamente mediante transacciones seguras con conn.setAutoCommit(false) y conn.commit(), lo que garantiza la consistencia de los datos.

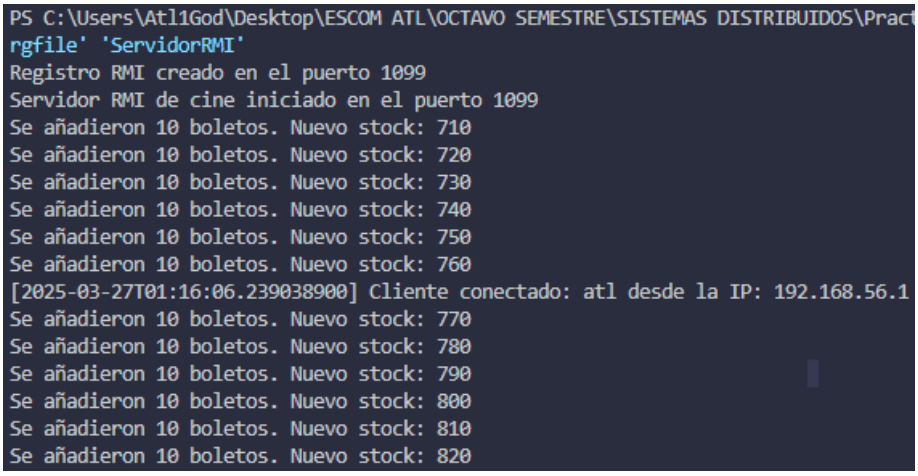

Figura 19. Boletos disponibles visto desde el servidor

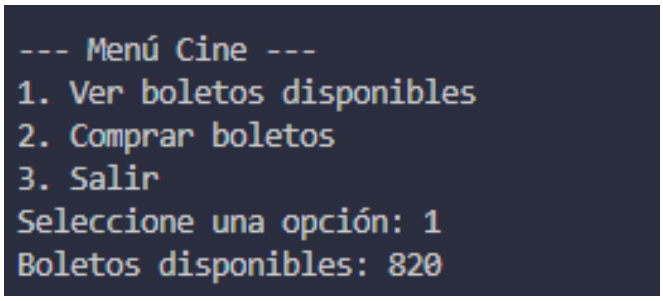

Figura 20. Boletos disponibles visto desde el cliente

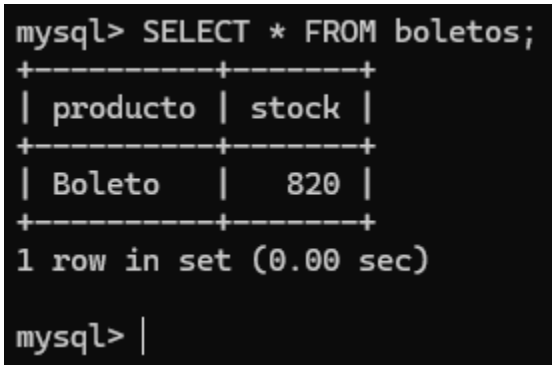

Figura 21. Boletos disponibles visto desde la base de datos

Un aspecto relevante es el correcto manejo de la concurrencia mediante la sincronización de los métodos remotos con la palabra clave synchronized. Esto aseguró que no ocurrieran problemas de condiciones de carrera durante las pruebas con múltiples clientes realizando compras simultáneas.

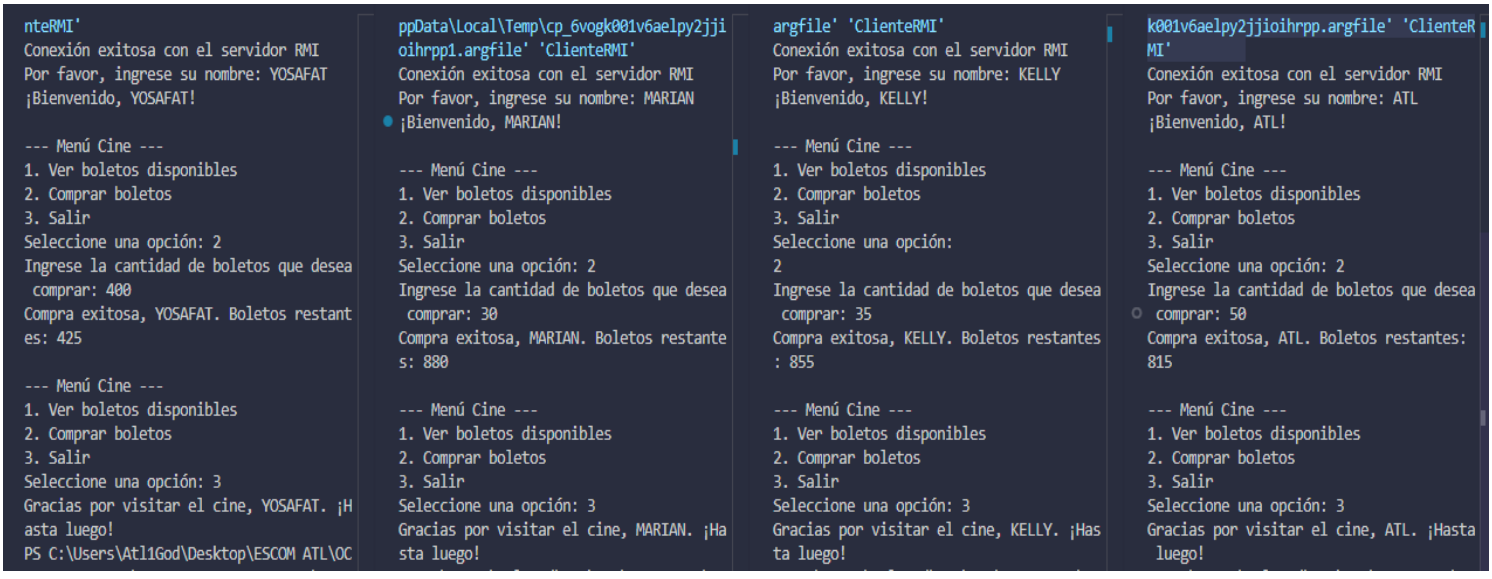

Figura 22. Conexión, compra y desconexión de múltiples usuarios a la vez

La funcionalidad de reposición automática de boletos a través del hilo RepositorBoletos también fue verificada. Durante las pruebas, se observó que cada 10 segundos se agregaban 10 boletos adicionales al stock, y el servidor registraba esta actualización en la consola. Esto permite asegurar la disponibilidad de boletos sin intervención manual.

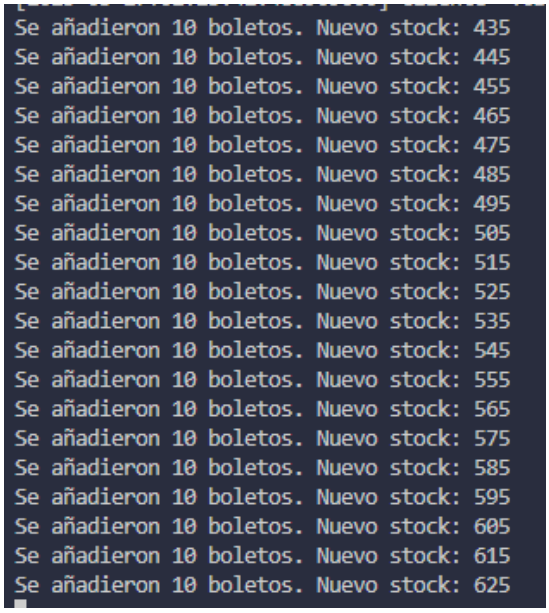

Figura 23. Reposición de boletos vista desde el servidor

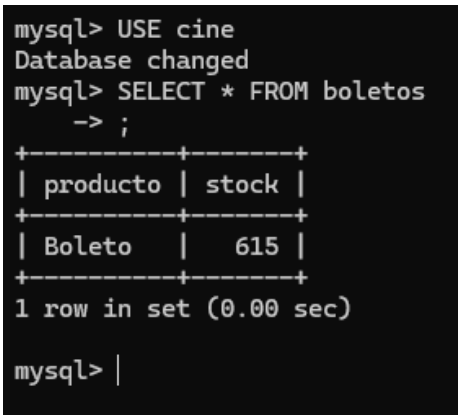

Figura 24. Actualización de la reposición de boletos vista desde la base de datos

Por otro lado la funcionalidad comprarBoleto() fue evaluada con diferentes escenarios. Cuando la cantidad solicitada estaba disponible, la compra se completó con éxito, actualizando el stock y registrando la transacción en la consola del servidor. En situaciones donde la cantidad solicitada superaba el stock disponible, el sistema devolvía un mensaje apropiado sin afectar la base de datos.

```
PS C:\Users\Atl1God\Desktop\ESCOM ATL\OCTAVO SEMESTRE\SISTEMAS DISTRIBUIDOS\Practica 5>  &
rgfile' 'ServidorRMI'
Registro RMI creado en el puerto 1099
Servidor RMI de cine iniciado en el puerto 1099
Se añadieron 10 boletos. Nuevo stock: 890
[2025-03-27T01:22:54.232859900] Cliente conectado: YOSAFAT desde la IP: 192.168.56.1
[2025-03-27T01:22:58.796080600] Cliente conectado: ATL desde la IP: 192.168.56.1
Se añadieron 10 boletos. Nuevo stock: 900
[2025-03-27T01:23:03.887301200] Cliente conectado: KELLY desde la IP: 192.168.56.1
[2025-03-27T01:23:07.421812] Cliente conectado: MARIAN desde la IP: 192.168.56.1
Se añadieron 10 boletos. Nuevo stock: 910
[2025-03-27T01:23:16.001342] Cliente 'MARIAN' compró 30 boletos. Stock restante: 880
Se añadieron 10 boletos. Nuevo stock: 890
[2025-03-27T01:23:25.122835400] Cliente 'KELLY' compró 35 boletos. Stock restante: 855
Se añadieron 10 boletos. Nuevo stock: 865
[2025-03-27T01:23:32.043055900] Cliente 'ATL' compró 50 boletos. Stock restante: 815
Se añadieron 10 boletos. Nuevo stock: 825
[2025-03-27T01:23:42.488909600] Cliente 'YOSAFAT' compró 400 boletos. Stock restante: 425
```

Figura 25. Compras realizadas desde diferentes clientes vista desde el servidor

```
¡Bienvenido, YOSAFAT!

--- Menú Cine ---
1. Ver boletos disponibles
2. Comprar boletos
3. Salir
Seleccione una opción: 2
Ingrese la cantidad de boletos que desea comprar: 400
Compra exitosa, YOSAFAT. Boletos restantes: 425
```

Figura 26. Compra de boletos vista desde el usuario

La función registrarDesconexion() permitió registrar adecuadamente las desconexiones de los clientes. Durante las pruebas, cada vez que un cliente seleccionaba la opción de salir, el servidor mostraba un mensaje de confirmación en la consola. Este registro facilita la administración de los usuarios conectados y brinda un historial básico de actividad.

```
Se añadieron 10 boletos. Nuevo stock: 695
Cliente desconectado: YOSAFAT
Cliente desconectado: ATL
Cliente desconectado: KELLY
Se añadieron 10 boletos. Nuevo stock: 705
Cliente desconectado: MARIAN
```

Figura 27. Desconexión de usuarios visto desde el servidor

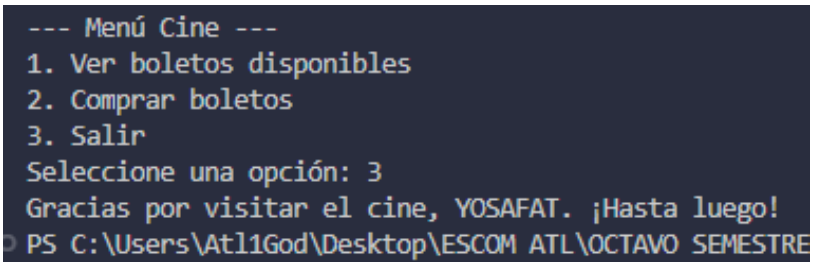

Figura 28. Desconexión y despedida al usuario

El uso de RMI demostró ser eficaz en un entorno de múltiples clientes. Incluso con varias conexiones simultáneas, el sistema mantuvo una respuesta adecuada y las transacciones se realizaron de manera eficiente. Esto resalta la capacidad de RMI para manejar aplicaciones distribuidas con un número moderado de clientes.

Conclusión

Durante la realización de esta práctica sobre objetos distribuidos utilizando Java RMI (Remote Method Invocation), tuve la oportunidad de profundizar en los conceptos de comunicación remota y la implementación de sistemas distribuidos. La práctica consistió en desarrollar un sistema de gestión de boletos para un cine, donde tanto los clientes como el servidor interactúan mediante métodos remotos. Esta experiencia me permitió comprender mejor el funcionamiento de las interfaces remotas, la serialización de objetos y la gestión de conexiones mediante RMI.

Uno de los principales desafíos que enfrenté fue la configuración de la base de datos MySQL y su integración con el servidor RMI. Aunque la creación de la base de datos y la tabla de boletos fue relativamente sencilla, la conexión con la base de datos a través de JDBC (Java Database Connectivity) presentó varios problemas. La correcta carga del driver JDBC y la configuración de las credenciales de acceso fueron puntos críticos. Además, resolver el error "No suitable driver found" fue una de las dificultades más notables. Aprendí la importancia de verificar el classpath y asegurarme de que el conector de MySQL estuviera correctamente referenciado.

Otro aspecto complicado fue la implementación de la concurrencia al realizar operaciones sobre la base de datos. Manejar las transacciones mediante conn.setAutoCommit(false) y asegurar la consistencia de los datos utilizando FOR UPDATE en las consultas SQL me permitió evitar condiciones de carrera. Esta parte fue especialmente interesante, ya que pude ver cómo los mecanismos de bloqueo y control de concurrencia aseguran la integridad de la información en un entorno distribuido.

Por otro lado, resultó muy satisfactorio observar el correcto funcionamiento del sistema después de superar estos retos. La interacción entre el cliente y el servidor, el registro de conexiones y desconexiones, así como la compra y reposición de boletos, reflejaron de manera clara el propósito de RMI. Además, implementar el hilo RepositorBoletos que añade

boletos de manera automática fue una tarea enriquecedora que me permitió reforzar mis conocimientos sobre hilos y multitarea en Java.

Por último, me gustaría mencionar que esta práctica fue una experiencia valiosa para entender el desarrollo de aplicaciones distribuidas utilizando Java RMI. La combinación de comunicación remota, manejo de bases de datos y control de concurrencia fue fundamental para consolidar los conocimientos adquiridos. Sin duda, los desafíos encontrados me permitieron mejorar mis habilidades de depuración y resolución de problemas en sistemas distribuidos, lo que considero un aprendizaje significativo para futuros proyectos.

Referencias

- *M. Akhitoccori, "Qué es RPC (Llamada a Procedimiento Remoto)," Medium, 2024. [En línea]. Available:https://medium.com/@maniakhitoccori/qu%C3%A9-es-rpc-llamada-a- procedimiento-remoto-7cbcbe45d8e. [Último acceso: 25 Marzo 2025].*

- *Gutiérrez, "Introducción a Java RMI," Universidad Técnica Federico Santa María, 2011. [En línea]. Available: http://profesores.elo.utfsm.cl/~agv/elo330/2s11/lectures/RMI/RMI.html. [Último acceso: 24 Marzo 2025].*

- *R. Millán, "Introducción a CORBA (Common Object Request Broker Architecture)," 2022. [En línea]. Available: https://www.ramonmillan.com/tutoriales/corba.php. [Último acceso: 25 Marzo 2025].*

- *Geeks for Geeks, "Distributed Component Object Model (DCOM)," 2023. [En línea]. Available: https://www.geeksforgeeks.org/distributed-component-object-model-dcom/. [Último acceso: 24 Marzo 2025].*

- *Oracle, "Remote Method Invocation (RMI)," 2023. [En línea]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/index.html. [Último acceso: 25 Marzo 2025].*

- *Microsoft Learn, "Conceptos de Sistemas Distribuidos," 2022. [En línea]. Available: https://learn.microsoft.com/es-es/azure/architecture/patterns/. [Último acceso: 26 Marzo 2025].*

- *M. J. LaMar, Thread Synchronization in Java, 2021. [En línea]. Available: https://www.example.com/thread-synchronization-java. [Último acceso: 16 Febrero 2025].*

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

- *MySQL, "MySQL Connector/J," MySQL, 2024. [En línea]. Available: https://dev.mysql.com/downloads/connector/j/. [Último acceso: 15 Marzo 2025].*

- *"Fundamentos de los sistemas distribuidos," Universidad de La Rioja, 2024. [En línea]. Available: https://www.unirioja.es/fundamentos-sistemas-distribuidos/ . [Último acceso: 16 Marzo 2025].*

Anexos

Código Fuente CineRMI.java

```
/*
 * Nombre: Cardoso Osorio Atl Yosafat
 * Grupo: 7CM1
 * Materia: Sistemas Distribuidos
 * Practica 5: Objetos Distribuidos
 * Fecha: 26 de Marzo del 2025
 */
import java.rmi.Remote;
import java.rmi.RemoteException;
// Métodos remotos
// Interfaz remota
public interface CineRMI extends Remote { // Interfaz remota
    int obtenerBoletosDisponibles() throws RemoteException; //
Métodos remotos
    boolean comprarBoleto(String nombreCliente, int cantidad)
throws RemoteException;
    void registrarConexion(String nombreCliente, String ipCliente)
throws RemoteException;
    void registrarDesconexion(String nombreCliente) throws
RemoteException;
}
```

Código Fuente ClienteRMI.java

```
/*
 * Nombre: Cardoso Osorio Atl Yosafat
 * Grupo: 7CM1
 * Materia: Sistemas Distribuidos
 * Practica 5: Objetos Distribuidos
 * Fecha: 26 de Marzo del 2025
 */
import java.net.InetAddress;
```

```
import java.rmi.Naming;
import java.util.Scanner;

public class ClienteRMI {
    public static void main(String[] args) {
        try {
            // Conectar con el servidor RMI
            CineRMI cine = (CineRMI)
Naming.lookup("rmi://localhost/Cine");
            System.out.println("Conexión exitosa con el servidor
RMI");
            Scanner scanner = new Scanner(System.in);

            // Solicitar el nombre del cliente
            System.out.print("Por favor, ingrese su nombre: ");
            String nombreCliente = scanner.nextLine();
            String ipCliente =
InetAddress.getLocalHost().getHostAddress();

            // Notificar la conexión al servidor
            cine.registrarConexion(nombreCliente, ipCliente);
            System.out.println("¡Bienvenido, " + nombreCliente +
"!");

            while (true) {
                System.out.println("\n--- Menú Cine ---");
                System.out.println("1. Ver boletos disponibles");
                System.out.println("2. Comprar boletos");
                System.out.println("3. Salir");
                System.out.print("Seleccione una opción: ");

                int opcion = scanner.nextInt();

                switch (opcion) {
                    case 1 -> System.out.println("Boletos
disponibles: " + cine.obtenerBoletosDisponibles());
                    case 2 -> {
                        System.out.print("Ingrese la cantidad de
boletos que desea comprar: ");
```

```
                        int cantidad = scanner.nextInt();
                        boolean compraExitosa =
cine.comprarBoleto(nombreCliente, cantidad);
                        if (compraExitosa) {
                            System.out.println("Compra exitosa, "
+ nombreCliente + ". Boletos restantes: " +
cine.obtenerBoletosDisponibles());
                        } else {
                            System.out.println("Lo sentimos, no
hay suficientes boletos disponibles.");
                        }
                    }
                    case 3 -> {
                        System.out.println("Gracias por visitar el
cine, " + nombreCliente + ". ¡Hasta luego!");
                        cine.registrarDesconexion(nombreCliente);
                        return;
                    }
                    default -> System.out.println("Opción no
válida. Intente de nuevo.");
                }
            }
        } catch (Exception e) {
            System.out.println("Error al conectar con el servidor
RMI:");
            e.printStackTrace();
        }
    }
}
```

Código Fuente ServidorRMI.java

```
/*
 * Nombre: Cardoso Osorio Atl Yosafat
 * Grupo: 7CM1
 * Materia: Sistemas Distribuidos
```

```java
 * Practica 5: Objetos Distribuidos
 * Fecha: 26 de Marzo del 2025
 */
import java.rmi.Naming;
import java.rmi.registry.LocateRegistry;

public class ServidorRMI { // Clase principal
    public static void main(String[] args) {// Método principal
        try {
            // Crear el servidor RMI
            CineServidorRMI servidor = new CineServidorRMI(); //
Crear una instancia del servidor
            int puerto = 1099; // Puerto por defecto

            // Crear un registro RMI en el puerto especificado
            try {
                LocateRegistry.createRegistry(puerto); // Crear un
registro RMI en el puerto especificado
                System.out.println("Registro RMI creado en el
puerto " + puerto); // Mensaje de éxito
            } catch (Exception e) {
                System.out.println("El puerto " + puerto + " ya
está en uso, intentando con el puerto 1098.");
                LocateRegistry.createRegistry(1098); // Crear un
registro RMI en el puerto 1098
                puerto = 1098; // Actualizar el puerto
            }

            // Registrar el servidor con el nombre "Cine"
            Naming.rebind("rmi://localhost:" + puerto + "/Cine",
servidor); // Registrar el servidor con el nombre "Cine"
            System.out.println("Servidor RMI de cine iniciado en
el puerto " + puerto);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Código Fuente CineServidorRMI.java

```
/*
 * Nombre: Cardoso Osorio Atl Yosafat
 * Grupo: 7CM1
 * Materia: Sistemas Distribuidos
 * Practica 5: Objetos Distribuidos
 * Fecha: 26 de Marzo del 2025
 */
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.sql.*;
import java.time.LocalDateTime;

public class CineServidorRMI extends UnicastRemoteObject
implements CineRMI {// Clase que implementa la interfaz remota
    private static final String URL =
"jdbc:mysql://localhost:3306/cine?useSSL=false&serverTimezone=UTC"
; // URL de la base de datos
    private static final String USER = "root"; // Usuario de la
base de datos
    private static final String PASSWORD = "Atl1God$"; //
Contraseña de la base de datos

    protected CineServidorRMI() throws RemoteException { //
Constructor
        super(); // Llama al constructor de la clase padre
        // Iniciar el hilo que generará boletos en intervalos
        Thread repositorBoletos = new Thread(new
RepositorBoletos(this));
        repositorBoletos.start(); // Inicia el hilo
    }

    @Override // Implementación de los métodos remotos
    public int obtenerBoletosDisponibles() throws RemoteException
{
        int stock = 0; // Inicializar el stock
        try (Connection conn = DriverManager.getConnection(URL,
USER, PASSWORD);
```

```
            Statement stmt = conn.createStatement(); // Crear una
sentencia
            ResultSet rs = stmt.executeQuery("SELECT stock FROM
boletos WHERE producto='Boleto'")
        ) { // Ejecutar una consulta
            if (rs.next()) { /// Si hay un resultado
                stock = rs.getInt("stock");
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return stock;
    }

    @Override // Implementación de los métodos remotos
    public synchronized boolean comprarBoleto(String
nombreCliente, int cantidad) throws RemoteException {
        try (Connection conn = DriverManager.getConnection(URL,
USER, PASSWORD)) { // Crear una conexión
            conn.setAutoCommit(false); // Deshabilitar el modo de
autocommit

            try (Statement stmt = conn.createStatement(); // Crear
una sentencia
                 ResultSet rs = stmt.executeQuery("SELECT stock
FROM boletos WHERE producto='Boleto' FOR UPDATE")) {
                if (rs.next()) { // Si hay un resultado
                    int stockActual = rs.getInt("stock"); //
Obtener el stock actual
                    if (stockActual >= cantidad) { // Si hay
suficientes boletos
                        try (PreparedStatement pstmt =
conn.prepareStatement( // Crear una sentencia preparada
                                "UPDATE boletos SET stock = stock
- ? WHERE producto='Boleto'")) { // Actualizar el stock
                            pstmt.setInt(1, cantidad); //
Establecer el valor del parámetro
                            pstmt.executeUpdate(); // Ejecutar la
sentencia
```

```
                        }
                        conn.commit(); // Confirmar la transacción
                        System.out.println("[" +
LocalDateTime.now() + "] Cliente '" + nombreCliente + "' compró "
+ cantidad + " boletos. Stock restante: " + (stockActual -
cantidad));
                        return true; // Compra exitosa
                    }
                }
            }
            conn.rollback();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return false;
    }

    public void anadirStock(int cantidad) throws RemoteException {
// Método para añadir boletos al stock
        try (Connection conn = DriverManager.getConnection(URL,
USER, PASSWORD);
             PreparedStatement pstmt =
conn.prepareStatement("UPDATE boletos SET stock = stock + ? WHERE
producto='Boleto'")) {
            pstmt.setInt(1, cantidad); // Establecer el valor del
parámetro
            pstmt.executeUpdate();
            int nuevoStock = obtenerBoletosDisponibles(); //
Obtener el stock actualizado
            System.out.println("Se añadieron " + cantidad + "
boletos. Nuevo stock: " + nuevoStock);
        } catch (SQLException e) { // Capturar excepciones
            e.printStackTrace(); // Imprimir la traza de la pila
        }
    }

    @Override
```

```
    public void registrarConexion(String nombreCliente, String
ipCliente) throws RemoteException { // Método para registrar la
conexión de un cliente
        System.out.println("[" + LocalDateTime.now() + "] Cliente
conectado: " + nombreCliente + " desde la IP: " + ipCliente); //
Imprimir mensaje
    }

    @Override // Implementación de los métodos remotos
    public void registrarDesconexion(String nombreCliente) throws
RemoteException {
        System.out.println("Cliente desconectado: " +
nombreCliente);
    }

    // Hilo que repone boletos automáticamente
    private static class RepositorBoletos implements Runnable {
        private final CineServidorRMI servidor;

        public RepositorBoletos(CineServidorRMI servidor) {
            this.servidor = servidor; // Inicializar el servidor
        }

        @Override
        public void run() { // Método run
            try { // Capturar excepciones
                while (true) { // Bucle infinito
                    Thread.sleep(10000); // Espera 10 segundos
                    servidor.anadirStock(10); // Añadir 10 boletos
al stock
                }
            } catch (InterruptedException | RemoteException e) {
                e.printStackTrace();
            }
        }
    }
}
```