INSTITUTO POLITÉCTICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

Carrera:

Ingeniería en Sistemas Computacionales

Unidad de aprendizaje:

Sistemas Distribuidos

Actividad:

Practica 5 – Objetos Distribuidos

Integrante:

Hernández Vázquez Jorge Daniel

Profesor:

Chadwick Carreto Arellano

Fecha de entrega:

26/03/2025

Índice de contenido

Antecedente 1
Características de los Objetos Distribuidos 1
Modelos de Comunicación en Objetos Distribuidos 2
Evolución de los Modelos de Comunicación en Sistemas Distribuidos 6
Comparación entre Modelos de Objetos Distribuidos 7
Planteamiento del problema 7
Propuesta de solución 8
Materiales y métodos empleados 9
Desarrollo de la solución 10
Instrucciones para ejecutar el proyecto 19
Resultados 21
Conclusión 24
Bibliografía 26
Anexos 27
Código PanaderiaRMI.java 27
Código PanaderiaServidorRMI.java 27
Código ServidorRMI.java 29
Código ClienteRMI.java 30

Índice de Figuras

Figura 1 3
Figura 2 4
Figura 3 5
Figura 4 6
Figura 5 11
Figura 6 11
Figura 7 12
Figura 8 13
Figura 9 13
Figura 10. 14
Figura 11. 14
Figura 12. 15
Figura 13. 15

Figura 14. 16
Figura 15. 16
Figura 16. 17
Figura 17. 17
Figura 18. 18
Figura 19 19
Figura 20 21
Figura 21 21
Figura 22 21
Figura 23 22
Figura 24 22
Figura 25 22
Figura 26 23
Figura 27 23
Figura 28 23
Figura 29 23
Figura 30 24

Antecedente

En la programación orientada a objetos (POO), un objeto es una entidad que encapsula datos y comportamientos, definidos a través de atributos y métodos. Cuando estos objetos deben interactuar en un entorno distribuido, surge el concepto de objetos distribuidos, los cuales pueden ser invocados y manipulados de manera remota desde diferentes máquinas dentro de una red.

Los objetos distribuidos son una evolución de la arquitectura cliente-servidor y buscan superar sus limitaciones, permitiendo una distribución más eficiente de la carga de trabajo y promoviendo el desacoplamiento entre componentes. En este contexto, se utilizan tecnologías como Remote Method Invocation (RMI), CORBA, DCOM y otros modelos de middleware que facilitan la comunicación entre objetos distribuidos.

Características de los Objetos Distribuidos

Los objetos distribuidos presentan características clave que los diferencian de los objetos tradicionales en un entorno local:

- **Transparencia de ubicación:** Los usuarios y desarrolladores pueden invocar métodos sin preocuparse por la ubicación física del objeto.
- **Interoperabilidad:** Los objetos pueden comunicarse a través de diferentes plataformas y sistemas operativos.
- **Encapsulamiento y reutilización:** Al igual que en la POO convencional, los objetos distribuidos encapsulan datos y comportamientos, lo que favorece la modularidad y reutilización del código.
- **Manejo de concurrencia y estado:** Pueden gestionar múltiples peticiones simultáneamente y conservar su estado entre invocaciones.
- **Seguridad y control de acceso:** En entornos distribuidos, la autenticación y autorización juegan un papel fundamental para evitar accesos no autorizados.

Y a diferencia de la Arquitectura Cliente-Servidor, sabemos claramente que la arquitectura cliente-servidor ha sido ampliamente utilizada para desarrollar aplicaciones distribuidas, presenta limitaciones como el acoplamiento entre clientes y servidores, dificultades en la escalabilidad y el riesgo de que el servidor se convierta en un cuello de botella. Los objetos distribuidos ofrecen mejoras en estos aspectos:

Característica	Cliente-Servidor	Objetos Distribuidos
Distribución de carga	Centralizada en el servidor	Distribuida entre múltiples nodos
Escalabilidad	Limitada por la capacidad del servidor	Mejora con la replicación de objetos
Reutilización de componentes	Baja, código separado en capas fijas	Alta, objetos autónomos y modulares
Manejo de fallos	Dependencia en el servidor central	Replicación y tolerancia a fallos

Históricamente, esta arquitectura surgió con el desarrollo de las redes de computadoras y se consolidó con la popularización de Internet. En sus inicios, la computación se basaba en modelos centralizados con mainframes, donde múltiples terminales dependían de un único sistema. Sin embargo, con la aparición de computadoras personales y redes más avanzadas, se hizo viable distribuir las tareas entre

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:

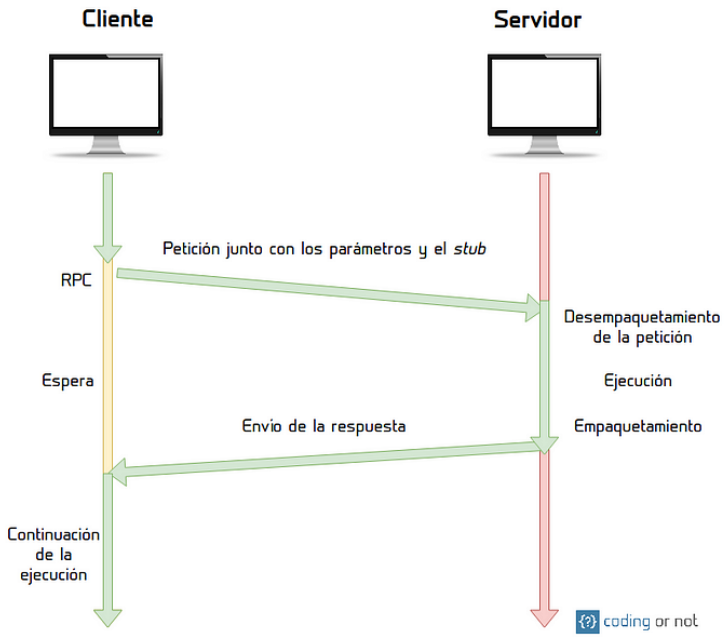


Figura 1. Diagrama del funcionamiento básico de RPC.

Invocación de Métodos Remotos (RMI)

Para abordar las limitaciones de RPC en la programación orientada a objetos, se desarrolló la tecnología Java RMI (Remote Method Invocation). A diferencia de RPC, que trabaja con funciones independientes, RMI permite invocar métodos sobre objetos remotos, manteniendo la semántica de la orientación a objetos en sistemas distribuidos.

Entre sus características más relevantes, RMI permite el paso de objetos completos entre cliente y servidor. Esto significa que no solo se pueden enviar datos primitivos, sino que es posible transferir estructuras de datos complejas con su comportamiento asociado. Además, RMI oculta los detalles de

la comunicación en red, permitiendo que los métodos remotos sean invocados de la misma forma que los métodos locales.

Otra característica clave de RMI es el Registro RMI, un servicio que permite a los clientes localizar objetos remotos en la red. Este registro facilita la interacción entre componentes distribuidos sin necesidad de conocer direcciones IP o puertos específicos.

A pesar de sus ventajas, RMI tiene una limitación importante: su dependencia de la Máquina Virtual de Java (JVM). Esto restringe su interoperabilidad con otros lenguajes de programación, lo que lo hace menos viable en entornos heterogéneos. En muchos casos, CORBA se presenta como una alternativa más flexible en términos de interoperabilidad entre plataformas.

Y podemos notar que los roles de cliente y servidor aplican sólo a un llamado. Un objeto servidor luego puede ser también cliente al hacer otro llamado remoto.

- **Marshalling**: es el proceso de codificación de los parámetros.
- **Stub**: es un objeto que encapsula el método que deseamos invocar remotamente. Así el llamado remoto es semejante a un llamado local. Éste prepara información con la identificación el objeto remoto a invocar, el método a invocar y codificación de los parámetros (Marshalling).
- **Skeleton**: es el objeto en el lado servidor que decodifica los parámetros, ubica el objeto llamado, llama el método deseado, codifica el valor retornado, y envía la información de regreso al stub.

En la siguiente Figura podemos observar un diagrama detallado del funcionamiento de RMI:

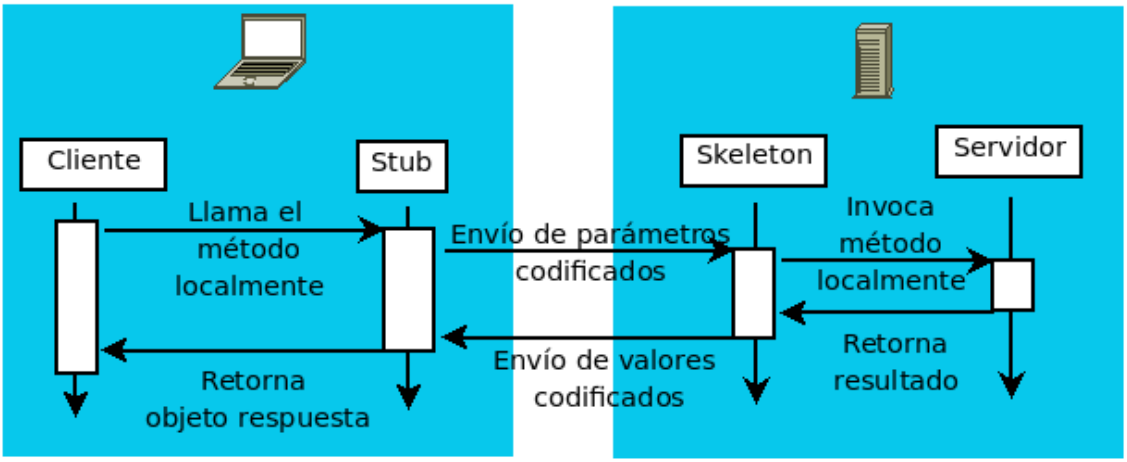


Figura 2. Diagrama del funcionamiento básico de RMI.

Aún cuando el proceso de la figura anterior es complejo, RMI lo hace en gran medida automático y en gran media transparente para el programador.

CORBA (Common Object Request Broker Architecture)

El Common Object Request Broker Architecture (CORBA) fue desarrollado por el Object Management Group (OMG) con el propósito de permitir la comunicación entre objetos distribuidos sin importar su lenguaje de implementación o plataforma subyacente. CORBA resuelve problemas de interoperabilidad al proporcionar una arquitectura estandarizada que permite que aplicaciones escritas en diferentes lenguajes, como Java, C++ o Python, puedan comunicarse sin necesidad de realizar conversiones manuales.

El núcleo de CORBA es el Object Request Broker (ORB), un intermediario que maneja las solicitudes de invocación de métodos remotos entre clientes y servidores. Cuando un cliente desea invocar un método en un objeto remoto, el ORB se encarga de localizar el objeto, realizar la llamada y devolver el resultado al cliente.

Entre sus ventajas, CORBA ofrece interoperabilidad, transparencia de localización y soporte para múltiples protocolos de transporte, como TCP/IP y HTTP. Sin embargo, una de sus principales desventajas es su complejidad. La configuración de CORBA puede ser difícil de manejar y la sobrecarga de comunicación puede hacer que sea menos eficiente en comparación con tecnologías más modernas.

Con la llegada de enfoques más livianos como REST y gRPC, CORBA ha perdido popularidad en muchos entornos de desarrollo. No obstante, sigue siendo una opción viable en sistemas donde la interoperabilidad entre distintos lenguajes es un requisito crítico. En la siguiente Figura podemos observar el esquema de la arquitectura CORBA.

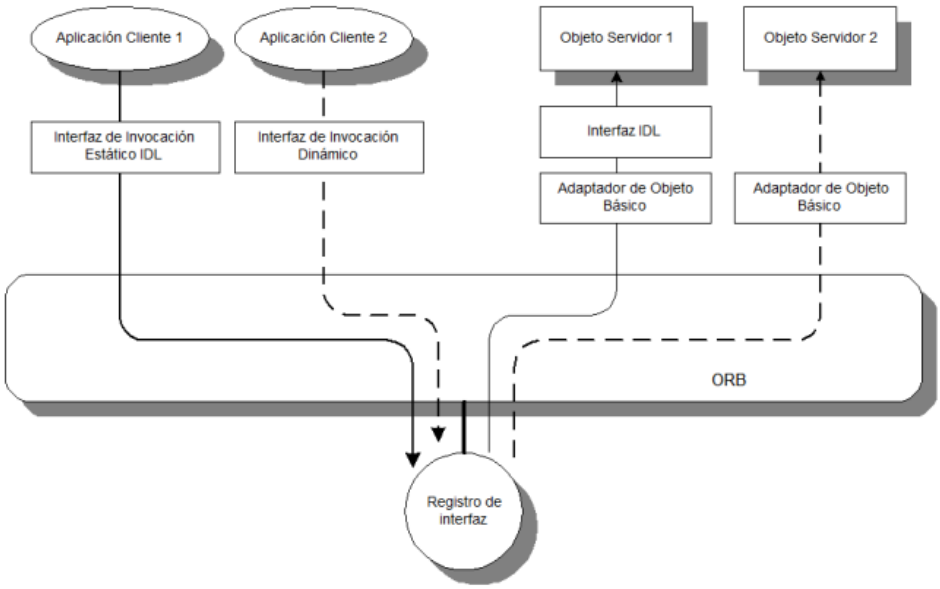


Figura 3. Esquema de la arquitectura CORBA.

DCOM (Distributed Component Object Model)

El Distributed Component Object Model (DCOM) es una tecnología de Microsoft que extiende el modelo COM (Component Object Model) para permitir la comunicación entre componentes distribuidos en una red. A diferencia de RPC y CORBA, que están diseñados para invocar funciones

o métodos en sistemas heterogéneos, DCOM está optimizado para entornos Windows, lo que facilita su integración en aplicaciones empresariales que dependen del ecosistema de Microsoft.

Una de las principales características de DCOM es su modelo basado en interfaces. Los objetos distribuidos en DCOM interactúan a través de interfaces bien definidas, lo que permite la reutilización de componentes y la evolución de aplicaciones sin afectar la compatibilidad con versiones anteriores. Además, DCOM incorpora mecanismos de seguridad integrados, como autenticación y control de acceso a través de Active Directory.

A pesar de estas ventajas, DCOM presenta varias limitaciones. Su fuerte dependencia de Windows lo hace poco adecuado para entornos heterogéneos, y su configuración puede ser compleja, especialmente en redes con restricciones de seguridad. Además, la comunicación en DCOM tiende a ser más pesada en comparación con soluciones modernas basadas en estándares abiertos.

Debido a estas limitaciones, DCOM ha sido reemplazado en gran medida por tecnologías más modernas como .NET Remoting y los servicios web basados en SOAP y REST. Sin embargo, sigue siendo utilizado en entornos empresariales donde la compatibilidad con sistemas heredados de Microsoft es una prioridad. En la siguiente Figura podemos observar el diagrama de funcionamiento para DCOM.

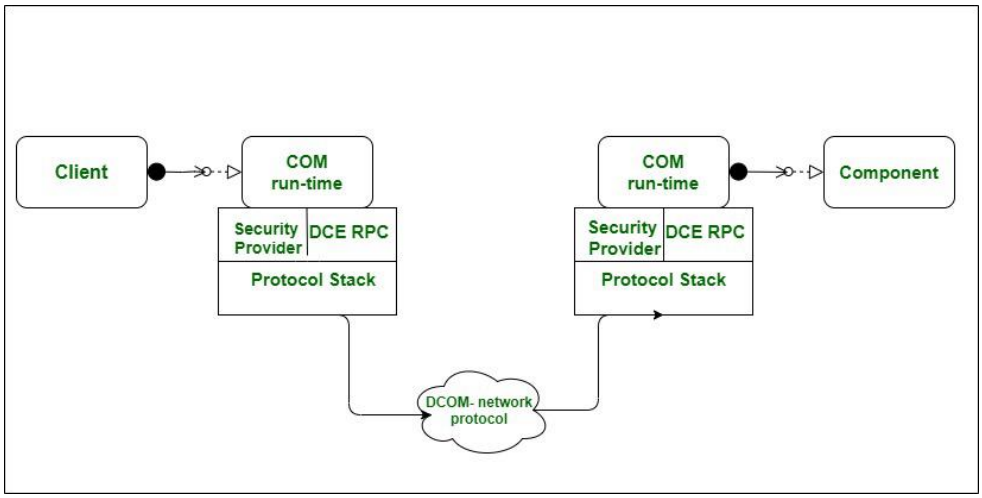


Figura 4. Diagrama básico del funcionamiento de DCOM.

Evolución de los Modelos de Comunicación en Sistemas Distribuidos

La evolución de la comunicación en sistemas distribuidos ha estado impulsada por la necesidad de mejorar la interoperabilidad, la escalabilidad y la facilidad de desarrollo.

Si bien RMI, CORBA y DCOM fueron fundamentales en la construcción de aplicaciones distribuidas en las décadas de 1990 y 2000, la aparición de **arquitecturas basadas en microservicios** y **servicios web** ha llevado a la adopción de nuevos enfoques, como:

- **REST (Representational State Transfer):** Utiliza HTTP como protocolo de comunicación y se ha convertido en el estándar de facto para la integración de sistemas distribuidos en la web.
- **gRPC:** Desarrollado por Google, permite la comunicación eficiente entre servicios utilizando **Protocol Buffers** en lugar de JSON o XML, lo que reduce el consumo de ancho de banda.

- **Arquitecturas basadas en eventos:** Tecnologías como **Kafka** y **RabbitMQ** permiten la comunicación asíncrona entre sistemas distribuidos, mejorando la escalabilidad y tolerancia a fallos.

A pesar del desplazamiento de tecnologías como RMI, CORBA y DCOM en favor de soluciones más modernas, estas siguen siendo relevantes en sistemas heredados y en entornos donde se requiere una fuerte integración con plataformas específicas. La elección del modelo de comunicación adecuado dependerá de los requisitos del sistema, incluyendo el rendimiento, la interoperabilidad y la facilidad de implementación.

Comparación entre Modelos de Objetos Distribuidos

Existen múltiples tecnologías que han sido desarrolladas para facilitar la comunicación entre objetos distribuidos. Algunas de las más relevantes como vimos son:

Característica	RMI	CORBA	DCOM
Lenguaje	Exclusivo de Java	Multilenguaje	Principalmente Windows
Facilidad de uso	Alta (nativo de Java)	Complejo	Complejo
Interoperabilidad	Limitada a Java	Amplia	Limitada a Windows
Rendimiento	Bueno para entornos Java	Alta latencia en algunas implementaciones	Razonable en entornos Windows
Estado actual	Sigue en uso en aplicaciones Java	En desuso	Reemplazado por .NET

Planteamiento del problema

El objetivo de esta práctica es simular un sistema distribuido para la gestión de una panadería, donde se contará con un inventario de productos (pan) que podrá ser consultado y actualizado a través de una red. El sistema permitirá a múltiples clientes conectarse de forma remota para realizar operaciones como verificar el stock disponible, comprar pan y recibir actualizaciones en tiempo real sobre el inventario.

Para asegurar la correcta operación del sistema, se deben implementar mecanismos que garanticen la sincronización del inventario entre los diferentes clientes. Esto evitará inconsistencias y asegurará que las transacciones se realicen sobre información actualizada. Asimismo, el sistema deberá ser capaz de gestionar conexiones y desconexiones de los clientes, registrando estas actividades para mantener un control adecuado del acceso.

Además, se requiere un sistema de producción automática de pan, simulado mediante un proceso interno que incrementará periódicamente el stock disponible. Esta funcionalidad permitirá que el inventario se actualice sin intervención manual, asegurando la continuidad del servicio incluso en momentos de alta demanda.

El desafío principal radica en garantizar la consistencia de los datos cuando múltiples clientes intenten comprar pan simultáneamente. Para ello, será necesario implementar mecanismos de bloqueo y transacciones en la base de datos que aseguren que no ocurran condiciones de carrera o inconsistencias en el stock.

Por último, el sistema debe ser fácil de utilizar para los clientes, proporcionando una interfaz sencilla a través de la cual puedan realizar sus compras y consultar la disponibilidad de pan en tiempo real. Esta práctica me permitirá comprender los principios fundamentales de los sistemas distribuidos, incluyendo la comunicación remota, la gestión de concurrencia y la sincronización de datos mediante la tecnología Java RMI, además de los objetos distribuidos.

Propuesta de solución

Para abordar el problema planteado, se propone el desarrollo de un sistema distribuido utilizando **Java RMI (Remote Method Invocation)**, que permitirá la comunicación entre un servidor central y múltiples clientes.

La solución consistirá en los siguientes componentes principales:

1. **Servidor RMI:**
    - Gestionará el inventario de la panadería mediante una conexión a una base de datos MySQL.
    - Permitirá a los clientes consultar el stock disponible y realizar compras de manera concurrente.
    - Implementará mecanismos de sincronización y control de concurrencia utilizando transacciones SQL y bloqueos para evitar inconsistencias.
    - Simulará la producción continua de pan a través de un proceso interno (horno) que añadirá stock en intervalos regulares.
2. **Clientes RMI:**
    - Podrán conectarse al servidor de forma remota para consultar el stock, realizar compras y recibir actualizaciones inmediatas del inventario.
    - Tendrán una interfaz de línea de comandos para interactuar con el sistema de manera sencilla y eficiente.
    - Registrarán su conexión y desconexión, permitiendo al servidor llevar un control de los clientes activos.
3. **Base de Datos MySQL:**
    - Almacenará y gestionará la información del inventario.
    - Permitirá la ejecución de consultas y actualizaciones de forma segura mediante transacciones y bloqueos exclusivos.
4. **Mecanismos de Concurrencia:**
    - Se implementarán transacciones SQL con la instrucción FOR UPDATE para evitar condiciones de carrera al consultar y actualizar el inventario.
    - El uso de métodos sincronizados en Java garantizará la consistencia de los datos en operaciones críticas.
5. **Registro de Actividades:**

- El servidor registrará las conexiones y desconexiones de los clientes, además de las compras realizadas, permitiendo un monitoreo constante del sistema.

Con esta propuesta, se espera desarrollar un sistema distribuido funcional, eficiente y robusto que simule con éxito las operaciones de una panadería en línea, resolviendo los desafíos de consistencia de datos y sincronización en un entorno con múltiples clientes.

Materiales y métodos empleados

Para llevar a cabo la práctica y desarrollar la simulación del sistema distribuido basado en RMI con objetos remotos, se utilizaron las siguientes herramientas y metodologías:

Materiales (Herramientas utilizadas):

1. **Lenguaje de programación: Java**
   - Se utilizó Java debido a su robustez en aplicaciones distribuidas y su soporte para RMI (Remote Method Invocation), facilitando la comunicación entre el servidor y los clientes de manera remota.
2. **Entorno de desarrollo: Visual Studio Code**
   - Se empleó Visual Studio Code como editor de código debido a su integración con Java y su eficiencia en el desarrollo de aplicaciones distribuidas.
3. **JDK (Java Development Kit):**
   - Se utilizó JDK 17 para compilar y ejecutar el código, asegurando compatibilidad con las últimas versiones de Java.
4. **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.
5. **Biblioteca adicional: 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 Empleados

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:

1. **Arquitectura Cliente-Servidor Distribuida:**
   - Se implementó un sistema cliente-servidor distribuido mediante RMI, donde el servidor gestiona el inventario y los clientes interactúan de forma remota.
2. **Concurrencia en Base de Datos:**

- Se utilizó JDBC con transacciones SQL para manejar compras simultáneas y evitar inconsistencias en el inventario.

3. **Sincronización de Datos:**
    - A través de RMI, los clientes consultan y actualizan el inventario en tiempo real, asegurando que todos trabajen con datos actualizados.
4. **Manejo de Conexiones Concurrentes:**
    - El servidor maneja múltiples clientes de manera concurrente mediante hilos (threads), optimizando el procesamiento de solicitudes.

Con estas herramientas y métodos, se desarrolló un sistema eficiente para la simulación de una panadería distribuida con objetos distribuidos y RMI.

Desarrollo de la solución

La solución implementada para la gestión distribuida del inventario en la panadería se basa en un modelo Cliente/Servidor con múltiples servidores sincronizados a través de una base de datos MySQL. Se ha diseñado un sistema que permite a varios clientes conectarse simultáneamente a distintos servidores, realizar compras y actualizar el inventario sin inconsistencias.

A continuación, se detallan los componentes principales de la solución y su implementación:

1. PanaderiaRMI.java (Interfaz Remota)

Este archivo define la interfaz PanaderiaRMI, que contiene los métodos remotos que los clientes pueden invocar. Todos estos métodos deben ser declarados con la excepción RemoteException para manejar posibles errores en la comunicación remota.

- **Objetivo:** Esta interfaz permite que los clientes interactúen con el servidor panadería de manera remota.
- **Métodos Definidos:**
    - obtenerStock(): Obtiene el stock de pan disponible.
    - comprarPan(): Permite a los clientes comprar pan, disminuyendo el stock.
    - añadirStock(): Aumenta el stock con la cantidad de pan horneado.
    - registrarConexion(): Registra la conexión de un cliente, indicando su nombre y dirección IP.
    - registrarDesconexion(): Registra cuando un cliente se desconecta.

En la siguiente figura podemos observar la definición de la interfaz remota que define cómo interactuarán los clientes con el servidor.

```java
import java.rmi.Remote;
import java.rmi.RemoteException;

// Interfaz remota para la panadería
// Esta interfaz define los métodos que pueden ser invocados remotamente
public interface PanaderiaRMI extends Remote {
    int obtenerStock() throws RemoteException; // Método para obtener el stock de panes
    boolean comprarPan(String nombreCliente, int cantidad) throws RemoteException; // Método para comprar panes
    void añadirStock(int cantidad) throws RemoteException; //método para añadir stock de panes
    void registrarConexion(String nombreCliente, String ipCliente) throws RemoteException; // Método para registrar la conexión del cliente
    void registrarDesconexion(String nombreCliente) throws RemoteException; // Método para registrar la desconexión del cliente
}
```

Figura 5. Definición de la interfaz remota/métodos remotos.

2. PanaderiaServidorRMI.java (Implementación del Servidor)

Este archivo implementa la interfaz PanaderiaRMI y contiene la lógica del servidor, incluida la gestión del inventario (almacenado en MySQL), la compra de pan, la adición de stock, y la gestión de conexiones de clientes.

- **Objetivo:** Este archivo implementa la lógica del servidor de panadería que se encarga de gestionar el inventario y realizar operaciones como la compra de pan y la adición de stock.
- **Métodos Implementados:**
    - obtenerStock(): Recupera el stock disponible de pan en la base de datos.
    - comprarPan(): Permite a un cliente comprar una cantidad específica de pan, decrementando el stock de manera segura utilizando transacciones y bloqueos.
    - añadirStock(): Aumenta el stock de pan en la base de datos, simula el proceso de producción.
    - HornoPanadero: Un hilo que simula la producción de pan cada cierto intervalo.

Constructor PanaderiaServidorRMI(): Este constructor inicializa el servidor RMI y arranca un hilo que simula el proceso de producción de pan (horneado) en intervalos regulares.

- **super()**: Llama al constructor de UnicastRemoteObject, que es necesario para habilitar la funcionalidad RMI en esta clase.
- **Horno Panadero**: El hilo hornoPanadero se inicia para simular la producción continua de pan cada ciertos segundos.

En la siguiente figura podemos observar el proceso de inicialización del servidor y el hilo para la generación de panes:

```java
protected PanaderiaServidorRMI() throws RemoteException {
    super();

    // Iniciar el hilo que generará panes en intervalos
    Thread hornoPanadero = new Thread(new HornoPanadero(this));
    hornoPanadero.start();
}
```

Figura 6. Inicialización del servidor y arranque del hilo.

Método obtenerStock(): Este método obtiene la cantidad de pan disponible en la base de datos.

- Se conecta a la base de datos usando JDBC, realizando una consulta SELECT para obtener el stock de pan disponible.
- El stock es leído de la tabla inventario, específicamente para el producto Pan.

En la siguiente figura podemos observar cómo se realiza la consulta a la base de datos para obtener el stock disponible:

```java
@Override
public int obtenerStock() throws RemoteException {
    int stock = 0;
    try {
        Class.forName(className:"com.mysql.cj.jdbc.Driver");

        try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);
             Statement stmt = conn.createStatement();
             ResultSet rs = stmt.executeQuery(sql:"SELECT stock FROM inventario WHERE producto='Pan'")) {
            if (rs.next()) {
                stock = rs.getInt(columnLabel:"stock");
            }
        }
    } catch (ClassNotFoundException | SQLException e) {
        e.printStackTrace();
    }
    return stock;
}
```

Figura 7. Método para obtener el stock, mostrando la consulta SQL y la extracción del stock.

Método comprarPan(): Este método permite que un cliente compre una cantidad específica de pan, actualizando el stock en la base de datos.

- Sincronización: Utiliza synchronized para asegurar que las compras se realicen de manera segura, evitando problemas de concurrencia.
- Se utiliza una transacción para asegurar que el stock no se modifique de manera incorrecta.
- Si el cliente tiene suficiente stock, se realiza la actualización del inventario, y se confirma la transacción.

En la siguiente figura podemos observar el proceso de compra de pan, que incluye la verificación del stock, la actualización de la base de datos y la transacción:

```java
@Override
public synchronized boolean comprarPan(String nombreCliente, int cantidad) throws RemoteException {
    try {
        Class.forName(className:"com.mysql.cj.jdbc.Driver");

        try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD)) {
            conn.setAutoCommit(autoCommit:false);

            try (Statement stmt = conn.createStatement();
                 ResultSet rs = stmt.executeQuery(sql:"SELECT stock FROM inventario WHERE producto='Pan' FOR UPDATE")) {
                if (rs.next()) {
                    int stockActual = rs.getInt(columnLabel:"stock");
                    if (stockActual >= cantidad) {
                        try (PreparedStatement pstmt = conn.prepareStatement(
                                sql:"UPDATE inventario SET stock = stock - ? WHERE producto='Pan'")) {
                            pstmt.setInt(parameterIndex:1, cantidad);
                            pstmt.executeUpdate();
                        }
                        conn.commit();
                        int nuevoStock = stockActual - cantidad;
                        System.out.println("[" + LocalDateTime.now() + "] Cliente '" + nombreCliente + "' compró " + cantidad + " panes. Stock restante: " + nuevoStock);
                        return true;
                    }
                }
            }
            conn.rollback();
        }
    } catch (ClassNotFoundException | SQLException e) {
        e.printStackTrace();
    }
    return false;
}
```

Figura 8. Bloque de código del Flujo de transacción para la compra de pan.

Método añadirStock(): Este método aumenta el stock de pan en la base de datos (simulando la producción de pan).

- Este método incrementa el stock de pan en la base de datos.
- Tras añadir el stock, se realiza una consulta adicional para obtener el stock actualizado y mostrarlo.

En la siguiente figura podemos observar cómo se realiza la actualización del stock de pan después de la producción:

```java
@Override
public void añadirStock(int cantidad) {
    try {
        Class.forName(className:"com.mysql.cj.jdbc.Driver");

        try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);
             PreparedStatement pstmt = conn.prepareStatement(sql:"UPDATE inventario SET stock = stock + ? WHERE producto='Pan'")) {
            pstmt.setInt(parameterIndex:1, cantidad);
            pstmt.executeUpdate();

            // Capturar RemoteException aquí
            try {
                int nuevoStock = obtenerStock(); // Obtener el stock actualizado después de la operación
                System.out.println("Se hornearon " + cantidad + " panes. Nuevo stock: " + nuevoStock);
            } catch (RemoteException e) {
                System.out.println("Error al obtener el stock: " + e.getMessage());
            }
        }
    } catch (ClassNotFoundException | SQLException e) {
        e.printStackTrace();
    }
}
```

Figura 9. Bloque de código mostrando el incremento del stock y la actualización en la base de datos.

Método registrarConexion() y registrarDesconexion(): Estos métodos registran la conexión y desconexión de los clientes.

- **registrarConexion()**: Imprime un mensaje con la información del cliente (nombre e IP) cuando se conecta.
- **registrarDesconexion()**: Imprime un mensaje cuando un cliente se desconecta.

En la siguiente figura podemos observar cómo se registran las conexiones y desconexiones de los clientes:

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

// Implementación para registrar la desconexión del cliente
@Override
public void registrarDesconexion(String nombreCliente) throws RemoteException {
    System.out.println("Cliente desconectado: " + nombreCliente);
}
```

Figura 10. Métodos para registrar la conexión y desconexión de los clientes.

Clase Interna HornoPanadero (Simula la Producción de Pan): Esta clase es un hilo que simula la producción de pan cada 10 segundos.

- El hilo HornoPanadero simula la producción continua de pan cada 10 segundos.
- Llama al método añadirStock(5) para incrementar el stock en la base de datos.

En la siguiente figura podemos observar cómo el hilo HornoPanadero simula la producción de pan cada 10 segundos:

```
// HornoPanadero que generará panes cada cierto tiempo
private static class HornoPanadero implements Runnable {
    private final PanaderiaServidorRMI servidor;

    public HornoPanadero(PanaderiaServidorRMI servidor) {
        this.servidor = servidor;
    }

    @Override
    public void run() {
        try {
            while (true) {
                Thread.sleep(millis:10000); // Espera 10 segundos
                servidor.añadirStock(cantidad:5);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Figura 11. Clase interna para simular la producción de pan.

Declaración de la Clase y Configuración de la Base de Datos:

- PanaderiaServidorRMI: Esta clase implementa la interfaz PanaderiaRMI, lo que le permite exponer métodos para ser invocados remotamente a través de RMI.
- UnicastRemoteObject: Al extender UnicastRemoteObject, la clase puede actuar como un objeto remoto accesible por los clientes.
- URL: Contiene la dirección de conexión a la base de datos MySQL. En este caso, la base de datos se encuentra en localhost, con el nombre panaderia.
- USER y PASSWORD: Son las credenciales para acceder a MySQL. Se utiliza "root" tanto para el usuario como para la contraseña, que es común en entornos de desarrollo.
- useSSL=false y serverTimezone=UTC: Estas opciones son necesarias para evitar advertencias y asegurar la correcta interpretación de la zona horaria.

En la siguiente figura podemos observar cómo se establece la configuración de conexión a la base de datos MySQL:

```
public class PanaderiaServidorRMI extends UnicastRemoteObject implements PanaderiaRMI {
    private static final String URL = "jdbc:mysql://localhost:3306/panaderia?useSSL=false&serverTimezone=UTC";
    private static final String USER = "root";
    private static final String PASSWORD = "root";
```

Figura 12. Establecimiento de la conexión a la base de datos.

3. ServidorRMI (Servidor de objetos remotos RMI): Se desarrolló una clase que implementa la interfaz remota y gestiona la lógica de negocio, proporcionando servicios accesibles a los clientes a través de la red. El servidor RMI es responsable de escuchar las solicitudes de los clientes, ejecutar los métodos remotos definidos en la interfaz y devolver los resultados correspondientes.

Creación del Servidor RMI:

- Aquí se crea una instancia de la clase PanaderiaServidorRMI, que implementa los métodos definidos en la interfaz PanaderiaRMI.
- Este objeto actuará como el servidor que gestionará las solicitudes de los clientes.

En la siguiente figura podemos observar la creación del objeto servidor:

```
// Crear el servidor RMI
PanaderiaServidorRMI servidor = new PanaderiaServidorRMI();
```

Figura 13. Creación del servidor RMI.

Creación del Registro RMI:

- **LocateRegistry.createRegistry(puerto):** Crea un registro RMI en el puerto especificado (por defecto yo coloque, el 1099).
- **Excepción y Manejo de Error:** Si el puerto 1099 está ocupado, el sistema captura la excepción y crea el registro en el puerto alternativo 1098.
- **Mensajes de Consola:** Informan al usuario si el registro fue exitoso o si se utilizó un puerto diferente.

En la siguiente figura se muestra la creación del registro RMI:

```java
// Crear el registro RMI en el puerto
int puerto = 1099;
try {
    LocateRegistry.createRegistry(puerto);
    System.out.println("Registro RMI creado en el puerto " + puerto);
} catch (Exception e) {
    System.out.println("El puerto " + puerto + " ya está en uso, usando otro puerto.");
    LocateRegistry.createRegistry(port:1098); // Cambiar al puerto 1098 si el 1099 está ocupado
}
```

Figura 14. Creación del registro RMI.

Registro del Servidor con el Nombre "Panadería":

- **Naming.rebind():** Vincula el objeto remoto (servidor) a un nombre lógico ("Panaderia") en el registro RMI. Esto permite que los clientes lo localicen y accedan a sus métodos remotos.
- **rmi://localhost/Panaderia:** Especifica la URL para acceder al servidor desde la máquina local.
- **Mensaje de Consola:** Confirma que el servidor RMI de panadería está en funcionamiento.

En la siguiente figura se observa el registro del servidor con el nombre "Panadería":

```java
// Registrar el servidor con el nombre "Panaderia"
Naming.rebind(name:"rmi://localhost/Panaderia", servidor);
System.out.println(x:"Servidor RMI de panadería iniciado...");
```

Figura 15. Registro del servidor con el nombre "Panaderia".

4. ClienteRMI (Cliente de objetos remotos RMI): El ClienteRMI.java es una clase que se conecta al servidor RMI para invocar los métodos remotos definidos en la interfaz. En el cliente, se utiliza el registro RMI para buscar el objeto remoto en el servidor, estableciendo una conexión entre ambos. Una vez establecida la conexión, el cliente puede invocar los métodos del servidor como si estuvieran siendo ejecutados localmente. El cliente también es responsable de gestionar la interacción con el usuario y de manejar las respuestas del servidor, mostrando la información solicitada o realizando acciones adicionales según el comportamiento definido por la simulación de la panadería.

Conexión con el servidor RMI y solicitud del Nombre del Cliente y Obtención de la IP:

- Naming.lookup(): Permite al cliente buscar el objeto remoto registrado con el nombre "Panaderia" en el servidor RMI.
- Conversión a Interfaz RMI: El objeto obtenido es convertido a la interfaz PanaderiaRMI, lo que permite invocar sus métodos remotos.
- Mensaje de Confirmación: Si la conexión es exitosa, se muestra un mensaje al cliente.
- Ingreso del Nombre: El cliente ingresa su nombre mediante Scanner.
- Obtención de la IP: Se usa InetAddress.getLocalHost().getHostAddress() para obtener la dirección IP del cliente.
- Esta información será enviada al servidor para registrar la conexión.

En la siguiente figura se ilustra la conexión del cliente al servidor RMI y la obtención del nombre y la IP del cliente:

```java
public static void main(String[] args) {
    try {
        // Conexión con el servidor RMI
        PanaderiaRMI panaderia = (PanaderiaRMI) Naming.lookup(name:"rmi://localhost/Panaderia");
        System.out.println(x:"Conexión exitosa con el servidor RMI");

        Scanner scanner = new Scanner(System.in);

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

Figura 16. Conexión con el servidor RMI y obtención del nombre e IP.

Registro de la Conexión en el Servidor y Menú Principal del Cliente:

- El cliente llama al método remoto registrarConexion() para notificar al servidor su conexión.
- Mensaje de Bienvenida: El cliente recibe una confirmación de conexión exitosa.
- Menú Interactivo: Presenta tres opciones principales al usuario.
- Scanner: Captura la opción seleccionada.
- Este bucle continuará ejecutándose hasta que el usuario decida salir.

En la siguiente figura podemos observar el registro de la conexión en el servidor y menú principal del cliente:

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

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

    int opcion = scanner.nextInt();
```

Figura 17. Registro de la conexión en el servidor y menú principal del cliente.

Visualización del stock, compra de pan y desconexión del servidor:

- **Método Remoto:** Se llama al método obtenerStock() en el servidor para conocer la cantidad de panes disponibles.
- **Visualización del Stock:** El cliente recibe y muestra el stock actual.
- **Ingreso de Cantidad:** El cliente ingresa cuántos panes desea comprar.Método Remoto: Se llama a comprarPan() en el servidor, quien valida si hay suficiente stock.
- **Respuesta:**
    - Si la compra fue exitosa, se muestra la cantidad restante de panes.
    - Si no hay suficiente stock, se informa al cliente.
- **Salida del Sistema:** El cliente decide salir seleccionando la opción 3.

- **Notificación de Desconexión:** Llama al método remoto registrarDesconexion() para notificar al servidor.
- **Mensaje de Despedida:** El cliente recibe un mensaje de despedida.

En la siguiente figura se muestra la consulta de stock, la interacción del cliente al realizar una compra y la desconexión del cliente:

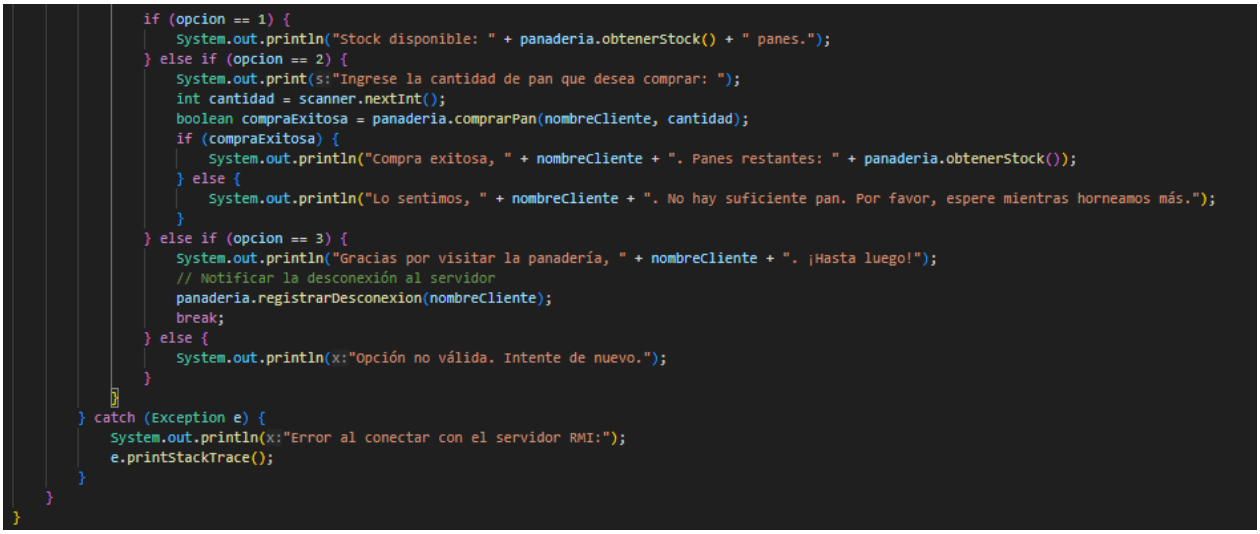


Figura 18. Operaciones que puede realizar el cliente desde el menú.

Con estos cuatro códigos (PanaderiaRMI, PanaderiaServidorRMI, ServidorRMI y ClienteRMI), se implementa correctamente la propuesta de solución del sistema distribuido para la panadería utilizando RMI.

Esta implementación permite:

- La **gestión del inventario** en tiempo real a través del servidor, asegurando la consistencia de los datos.
- La **sincronización** de las operaciones de compra, utilizando mecanismos de concurrencia y bloqueos en la base de datos.
- La **comunicación eficiente** entre clientes y servidores mediante objetos distribuidos.
- El **registro y monitoreo** de las conexiones de los clientes, proporcionando mayor control sobre las interacciones del sistema.
- La **generación automática de panes** con el horno panadero, asegurando un reabastecimiento constante del inventario.

En conjunto, estos componentes garantizan el correcto funcionamiento del sistema distribuido, cumpliendo con los objetivos planteados en la propuesta de solución.

Diseño de la Base de Datos

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

```
CREATE TABLE inventario (
    id INT PRIMARY KEY AUTO_INCREMENT,
    producto VARCHAR(50) NOT NULL,
    stock INT NOT NULL
);
```

Figura 19. Estructura de la tabla principal de la Base de Datos.

Esta tabla tiene tres columnas:

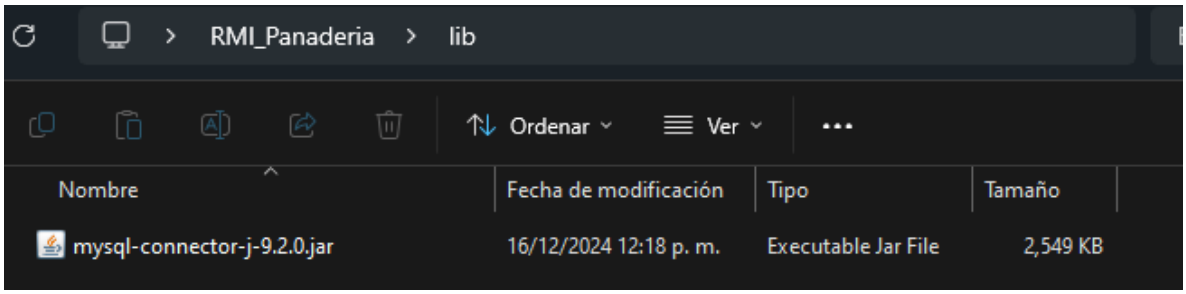
- id: Un identificador único para cada producto (es un campo autoincremental).
- producto: El nombre del producto (en este caso, "Pan").
- stock: La cantidad disponible del producto.

Instrucciones para ejecutar el proyecto

Para poder ejecutar este proyecto, es necesario estar dentro de la carpeta donde se encuentran todos los archivos. En mi caso, la ruta del proyecto es: C:\Users\user\Desktop\RMI_Panaderia.

1. Conector JDBC

Primero, debemos asegurarnos de tener el conector JDBC necesario para la conexión con MySQL. En este caso, el archivo mysql-connector-j-9.2.0.jar se encuentra en la subcarpeta **lib** dentro de la carpeta del proyecto.



2. Compilación de los códigos

Abrimos una terminal y navegamos a la carpeta del proyecto. Para compilar todos los códigos fuente, ejecutamos el siguiente comando:

```
PS C:\Users\user\Desktop\RMI_Panaderia> javac -d bin ClienteRMI.java PanaderiaRMI.java ServidorRMI.java PanaderiaServ
idorRMI.java
```

Este comando compilará los archivos .java y almacenará las clases resultantes en la carpeta **bin**, que se creará automáticamente si no existe.

3. Compilación de nuevo (opcional)

Después de compilar, ejecutamos nuevamente el siguiente comando para asegurarnos de que todos los archivos .java se compilen correctamente:

```
PS C:\Users\user\Desktop\RMI_Panaderia> javac *.java
```

Esto garantizará que todas las clases estén correctamente compiladas.

4. Ejecución del servidor RMI

Una vez compilados sin errores, es momento de ejecutar el servidor RMI. Para ello, debesmos ejecutar el siguiente comando desde la terminal, asegurándote de incluir el conector JDBC en el classpath:

```
PS C:\Users\user\Desktop\RMI_Panaderia> java -cp ";lib/mysql-connector-j-9.2.0.jar" ServidorRMI
Registro RMI creado en el puerto 1099
Servidor RMI de panadería iniciado...
```

Al ejecutar este comando, el servidor RMI se iniciará y se mostrará un mensaje indicando que el servidor RMI ha sido creado y está en ejecución.

5. Ejecución del cliente RMI

Con el servidor RMI en ejecución, ahora podemos iniciar el cliente. Para ello, desde la misma terminal y estando dentro de la carpeta del proyecto, ejecutamos el siguiente comando:

```
PS C:\Users\user\Desktop\RMI_Panaderia> java ClienteRMI
```

El cliente se conectará al servidor RMI y podremos interactuar con la panadería de forma distribuida.

Entonces, el motivo por el cual solo se ejecutan los archivos ServidorRMI y ClienteRMI es que los otros dos archivos, PanaderiaRMI y PanaderiaServidorRMI, contienen las definiciones de las interfaces y la lógica del servidor, pero no son aplicaciones ejecutables por sí solas. El archivo PanaderiaRMI es una interfaz remota que define los métodos que el servidor ofrece a los clientes, y PanaderiaServidorRMI implementa esta interfaz para proporcionar la funcionalidad real, como la gestión del inventario y la compra de productos.

El ServidorRMI es el encargado de crear el registro RMI y poner a disposición los servicios del servidor para los clientes, por lo que es el único archivo que necesita ejecutarse para poner en marcha el sistema. El ClienteRMI se conecta al servidor y permite a los usuarios interactuar con él. Ambos archivos son independientes y se ejecutan de manera separada para establecer la comunicación entre el servidor y los clientes.

No es necesario ejecutar un comando adicional para abrir el puerto RMI, ya que el propio servidor ServidorRMI se encarga de crear el registro RMI y abrir el puerto en el que escucha. Por lo tanto, al ejecutar únicamente ServidorRMI y ClienteRMI, se garantiza que el sistema funcione correctamente sin necesidad de ejecutar manualmente otros componentes.

Resultados

El sistema de panadería distribuido basado en RMI funciona correctamente, permitiendo que varios clientes interactúen con el servidor de manera simultánea. Los clientes pueden consultar el inventario, realizar compras y visualizar el stock actualizado en tiempo real. Cada operación es gestionada eficientemente por el servidor, asegurando la consistencia de los datos en todo momento.

Durante las pruebas, se verificó que el sistema maneja adecuadamente múltiples clientes conectados al mismo tiempo. Las compras realizadas por distintos usuarios son procesadas de manera sincronizada, evitando problemas de concurrencia o inconsistencias en el inventario. Asimismo, el mecanismo de reabastecimiento automático con el horno panadero opera de forma continua, incrementando el stock en intervalos de tiempo definidos.

El servidor mantiene un registro de las conexiones y desconexiones de los clientes, mostrando información en tiempo real sobre las interacciones. Además, cualquier intento de compra sin stock suficiente es gestionado apropiadamente, notificando al cliente sobre la falta de panes y evitando transacciones inválidas.

A continuación, se presentan capturas de las pruebas realizadas para evidenciar el correcto funcionamiento del sistema:

```
PS C:\Users\user\Desktop\RMI_Panaderia> java -cp ";lib/mysql-connector-j-9.2.0.jar" ServidorRMI
Registro RMI creado en el puerto 1099
Servidor RMI de panadería iniciado...
```

Figura 20. Ejecución del ServidorRMI.

```
PS C:\Users\user\Desktop\RMI_Panaderia> java ClienteRMI
Conexión exitosa con el servidor RMI
Por favor, ingrese su nombre: Elena
¡Bienvenido, Elena!

--- Menú Panadería ---
1. Ver stock
2. Comprar pan
3. Salir
Seleccione una opción: 1
Stock disponible: 100 panes.
```

Figura 21. Conexión exitosa de un cliente y consulta del inventario.

```
Registro RMI creado en el puerto 1099
Servidor RMI de panadería iniciado...
Se hornearon 5 panes. Nuevo stock: 15
Se hornearon 5 panes. Nuevo stock: 20
Se hornearon 5 panes. Nuevo stock: 25
Se hornearon 5 panes. Nuevo stock: 30
Se hornearon 5 panes. Nuevo stock: 35
Se hornearon 5 panes. Nuevo stock: 40
Se hornearon 5 panes. Nuevo stock: 45
[2025-03-26T02:12:03.614414400] Cliente conectado: Elena desde la IP: 192.168.56.1
Se hornearon 5 panes. Nuevo stock: 50
Se hornearon 5 panes. Nuevo stock: 55
Se hornearon 5 panes. Nuevo stock: 60
Se hornearon 5 panes. Nuevo stock: 65
Se hornearon 5 panes. Nuevo stock: 70
```

Figura 22. Mensajes obtenidos desde el servidor.

```
--- Menú Panadería ---
1. Ver stock
2. Comprar pan
3. Salir
Seleccione una opción: 1
Stock disponible: 90 panes.

--- Menú Panadería ---
1. Ver stock
2. Comprar pan
3. Salir
Seleccione una opción: 2
Ingrese la cantidad de pan que desea comprar: 50
Compra exitosa, Elena. Panes restantes: 40
```

Figura 23. Consulta de stock y compra de pan con un cliente.

Y en el servidor se nos muestra lo siguiente:

```
Se hornearon 5 panes. Nuevo stock: 90
[2025-03-26T02:15:08.504803600] Cliente 'Elena' compró 50 panes. Stock restante: 40
Se hornearon 5 panes. Nuevo stock: 45
```

Figura 24. Mensaje desde el servidor de compra y actualización de stock.

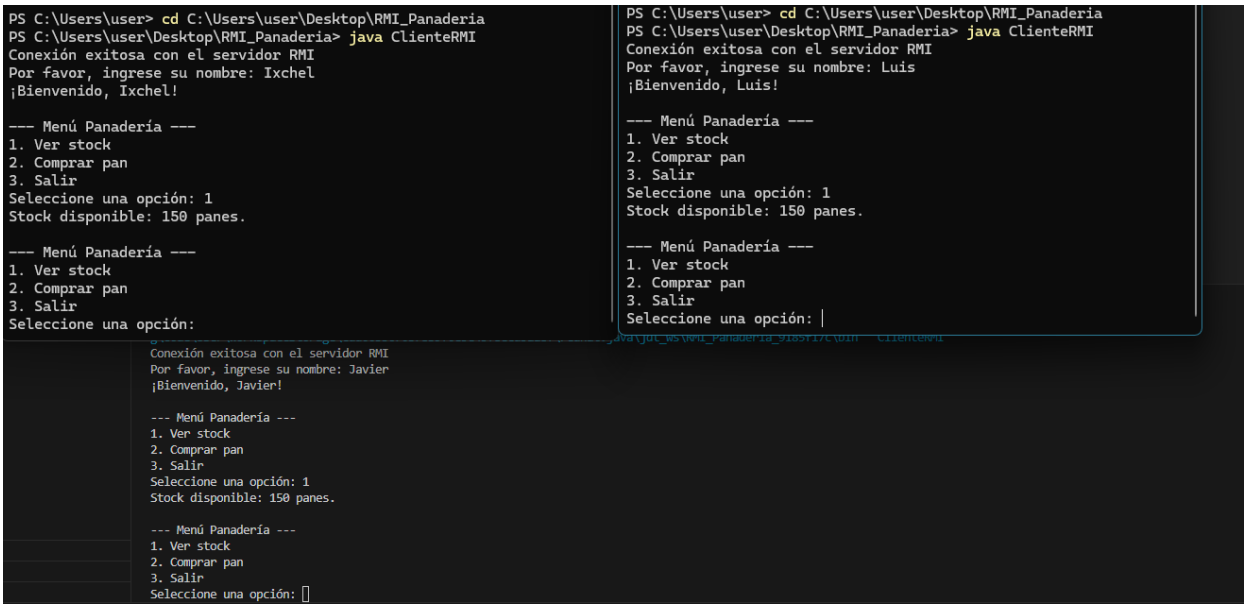


Figura 25. Conexión con varios clientes y consulta de stock desde cada uno.

```
[2025-03-26T02:17:35.617072100] Cliente conectado: Luis desde la IP:
192.168.56.1
Se hornearon 5 panes. Nuevo stock: 120
[2025-03-26T02:17:51.100486300] Cliente conectado: Javier desde la IP
: 192.168.56.1
Se hornearon 5 panes. Nuevo stock: 125
Se hornearon 5 panes. Nuevo stock: 130
Se hornearon 5 panes. Nuevo stock: 135
[2025-03-26T02:18:22.484933300] Cliente conectado: Ixchel desde la IP
: 192.168.56.1
```

Figura 26. Mensajes obtenidos desde el servidor.

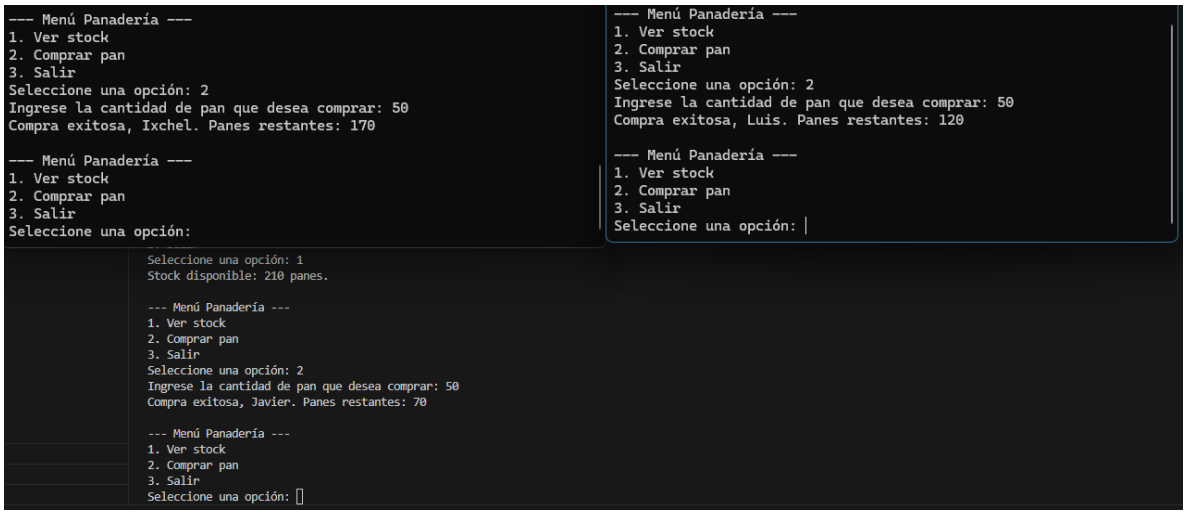


Figura 27. Prueba con varios clientes al intentar comprar pan simultáneamente.

```
[2025-03-26T02:21:08.183277800] Cliente 'Ixchel' compró 50 panes. Stock restante: 170
[2025-03-26T02:21:11.347365] Cliente 'Luis' compró 50 panes. Stock restante: 120
[2025-03-26T02:21:13.771766800] Cliente 'Javier' compró 50 panes. Stock restante: 70
Se hornearon 5 panes. Nuevo stock: 75
```

Figura 28. Mensajes del servidor de cada compra de los clientes.

```
--- Menú Panadería ---
1. Ver stock
2. Comprar pan
3. Salir
Seleccione una opción: 2
Ingrese la cantidad de pan que desea comprar: 250
Lo sentimos, Elena. No hay suficiente pan. Por favor, espere mientras horneamos más.
```

Figura 29. Mensaje al intentar comprar más pan del stock disponible.

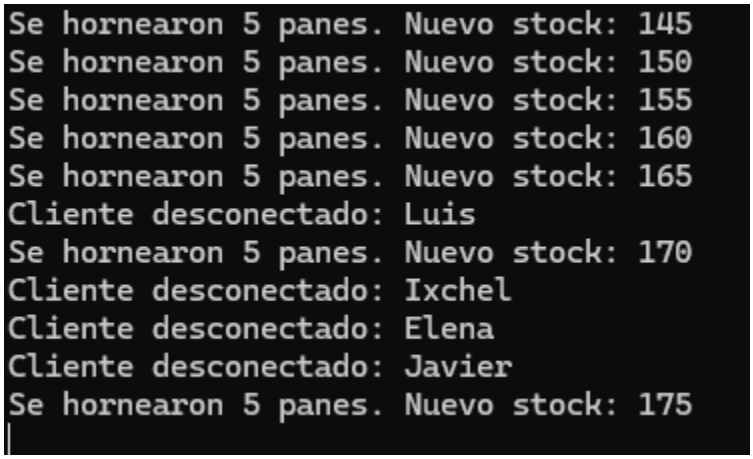


Figura 30. Mensajes del servidor de la desconexión de los clientes y funcionamiento del hornoPanadero, añadiendo stock periódicamente.

Estos resultados demuestran la robustez y confiabilidad del sistema distribuido implementado, cumpliendo con los objetivos planteados en la propuesta de solución.

Conclusión

Esta práctica me permitió adentrarme por primera vez en el mundo de los objetos distribuidos y el uso de RMI (Remote Method Invocation) para la comunicación entre aplicaciones en un entorno distribuido. Antes de realizar esta práctica, únicamente conocía lo visto durante las clases, sin embargo, el realizarlo e implementarlo en mi ejemplo, fue una experiencia enriquecedora el hecho de comprender más a fondo cómo funciona la interacción entre servidores y clientes a través de la invocación remota de métodos. Implementar un sistema de panadería distribuido me ayudó a visualizar los desafíos y soluciones asociados con la sincronización de datos, la gestión de concurrencia y la comunicación eficiente entre componentes distribuidos.

A lo largo del desarrollo, adquirí un mejor entendimiento sobre cómo RMI facilita la construcción de aplicaciones distribuidas mediante la creación de interfaces y la transferencia de objetos de manera transparente. Además, pude experimentar la importancia de manejar excepciones y asegurar la consistencia de los datos, especialmente al trabajar con bases de datos en tiempo real. La implementación de métodos remotos para la compra y actualización de inventarios, junto con la gestión de conexiones de clientes, reforzó mis conocimientos sobre programación concurrente y sincronización de recursos.

Si bien RMI fue una herramienta útil para comprender los conceptos fundamentales de los objetos distribuidos, también pude reflexionar sobre sus limitaciones, en comparación con tecnologías más modernas, como lo vimos en clase. Actualmente, soluciones como los microservicios, servicios web REST o gRPC, y las plataformas en la nube ofrecen mayor flexibilidad, escalabilidad y facilidad de mantenimiento para sistemas distribuidos complejos. Estas alternativas permiten una mayor interoperabilidad entre diferentes lenguajes de programación y facilitan la implementación de arquitecturas más robustas y resilientes, aunque eso no quita que estos métodos se sigan utilizando por algunas empresas.

En general, esta práctica fue un excelente punto de partida para entender los principios fundamentales de los objetos distribuidos y la comunicación remota. Aunque existen tecnologías más avanzadas, la

experiencia adquirida con RMI me proporciono una base sólida para explorar otras soluciones y aplicar estos conocimientos en futuros proyectos. Además, me permitió valorar la evolución de los sistemas distribuidos y reconocer la importancia de elegir la tecnología adecuada según las necesidades de un proyecto, me pareció muy interesante esta practica y mas sencilla, ya que incluso las líneas de código disminuyeron a comparación de las practicas anteriores.

Bibliografía

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].

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: 26 Marzo 2025].

A. 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: 26 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: 26 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: 26 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: 26 Marzo 2025].

Anexos

Código PanaderiaRMI.java

```
import java.rmi.Remote;
import java.rmi.RemoteException;

// Interfaz remota para la panadería
// Esta interfaz define los métodos que pueden ser invocados remotamente
public interface PanaderiaRMI extends Remote {
    int obtenerStock() throws RemoteException; // Método para obtener el stock de panes
    boolean comprarPan(String nombreCliente, int cantidad) throws RemoteException; // Método
para comprar panes
    void añadirStock(int cantidad) throws RemoteException; //método para añadir stock de panes
    void registrarConexion(String nombreCliente, String ipCliente) throws RemoteException; //
Método para registrar la conexión del cliente
    void registrarDesconexion(String nombreCliente) throws RemoteException; // Método para
registrar la desconexión del cliente
}
```

Código PanaderiaServidorRMI.java

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

public class PanaderiaServidorRMI extends UnicastRemoteObject implements PanaderiaRMI {
    private static final String URL =
"jdbc:mysql://localhost:3306/panaderia?useSSL=false&serverTimezone=UTC";
    private static final String USER = "root";
    private static final String PASSWORD = "root";


    protected PanaderiaServidorRMI() throws RemoteException {
        super();

        // Iniciar el hilo que generará panes en intervalos
        Thread hornoPanadero = new Thread(new HornoPanadero(this));
        hornoPanadero.start();
    }

    @Override
    public int obtenerStock() throws RemoteException {
        int stock = 0;
        try {
            Class.forName("com.mysql.cj.jdbc.Driver");

            try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);
                 Statement stmt = conn.createStatement();
                 ResultSet rs = stmt.executeQuery("SELECT stock FROM inventario WHERE
producto='Pan'")) {
                if (rs.next()) {
                    stock = rs.getInt("stock");
                }
            }
        } catch (ClassNotFoundException | SQLException e) {
```

```
            e.printStackTrace();
        }
        return stock;
    }

    @Override
    public synchronized boolean comprarPan(String nombreCliente, int cantidad) throws
RemoteException {
        try {
            Class.forName("com.mysql.cj.jdbc.Driver");

            try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD)) {
                conn.setAutoCommit(false);

                try (Statement stmt = conn.createStatement();
                     ResultSet rs = stmt.executeQuery("SELECT stock FROM inventario WHERE
producto='Pan' FOR UPDATE")) {
                    if (rs.next()) {
                        int stockActual = rs.getInt("stock");
                        if (stockActual >= cantidad) {
                            try (PreparedStatement pstmt = conn.prepareStatement(
                                    "UPDATE inventario SET stock = stock - ? WHERE
producto='Pan'")) {
                                pstmt.setInt(1, cantidad);
                                pstmt.executeUpdate();
                            }
                            conn.commit();
                            int nuevoStock = stockActual - cantidad;
                            System.out.println("[" + LocalDateTime.now() + "] Cliente '" +
nombreCliente + "' compró " + cantidad + " panes. Stock restante: " + nuevoStock);
                            return true;
                        }
                    }
                }
                conn.rollback();
            }
        } catch (ClassNotFoundException | SQLException e) {
            e.printStackTrace();
        }
        return false;
    }

    @Override
    public void añadirStock(int cantidad) {
        try {
            Class.forName("com.mysql.cj.jdbc.Driver");

            try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);
                 PreparedStatement pstmt = conn.prepareStatement("UPDATE inventario SET stock
= stock + ? WHERE producto='Pan'")) {
                pstmt.setInt(1, cantidad);
                pstmt.executeUpdate();

                // Capturar RemoteException aquí
                try {
                    int nuevoStock = obtenerStock(); // Obtener el stock actualizado después
de la operación
```

```
                    System.out.println("Se hornearon " + cantidad + " panes. Nuevo stock: " +
nuevoStock);
                } catch (RemoteException e) {
                    System.out.println("Error al obtener el stock: " + e.getMessage());
                }
            }
        } catch (ClassNotFoundException | SQLException e) {
            e.printStackTrace();
        }
    }

    // Implementación para registrar la conexión del cliente
    @Override
    public void registrarConexion(String nombreCliente, String ipCliente) throws
RemoteException {
        System.out.println("[" + LocalDateTime.now() + "] Cliente conectado: " + nombreCliente
+ " desde la IP: " + ipCliente);
    }

    // Implementación para registrar la desconexión del cliente
    @Override
    public void registrarDesconexion(String nombreCliente) throws RemoteException {
        System.out.println("Cliente desconectado: " + nombreCliente);
    }

    // HornoPanadero que generará panes cada cierto tiempo
    private static class HornoPanadero implements Runnable {
        private final PanaderiaServidorRMI servidor;

        public HornoPanadero(PanaderiaServidorRMI servidor) {
            this.servidor = servidor;
        }

        @Override
        public void run() {
            try {
                while (true) {
                    Thread.sleep(10000); // Espera 10 segundos
                    servidor.añadirStock(5);
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Código ServidorRMI.java

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

public class ServidorRMI {
    public static void main(String[] args) {
        try {
            // Crear el servidor RMI
            PanaderiaServidorRMI servidor = new PanaderiaServidorRMI();

            // Crear el registro RMI en el puerto
```

```
            int puerto = 1099;
            try {
                LocateRegistry.createRegistry(puerto);
                System.out.println("Registro RMI creado en el puerto " + puerto);
            } catch (Exception e) {
                System.out.println("El puerto " + puerto + " ya está en uso, usando otro
puerto.");
                LocateRegistry.createRegistry(1098); // Cambiar al puerto 1098 si el 1099 está
ocupado
            }

            // Registrar el servidor con el nombre "Panaderia"
            Naming.rebind("rmi://localhost/Panaderia", servidor);
            System.out.println("Servidor RMI de panadería iniciado...");

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Código ClienteRMI.java

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

public class ClienteRMI {
    public static void main(String[] args) {
        try {
            // Conexión con el servidor RMI
            PanaderiaRMI panaderia = (PanaderiaRMI)
Naming.lookup("rmi://localhost/Panaderia");
            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
            panaderia.registrarConexion(nombreCliente, ipCliente);
            System.out.println("¡Bienvenido, " + nombreCliente + "!");

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

                int opcion = scanner.nextInt();

                if (opcion == 1) {
```

```
                    System.out.println("Stock disponible: " + panaderia.obtenerStock() + "
panes.");
                } else if (opcion == 2) {
                    System.out.print("Ingrese la cantidad de pan que desea comprar: ");
                    int cantidad = scanner.nextInt();
                    boolean compraExitosa = panaderia.comprarPan(nombreCliente, cantidad);
                    if (compraExitosa) {
                        System.out.println("Compra exitosa, " + nombreCliente + ". Panes
restantes: " + panaderia.obtenerStock());
                    } else {
                        System.out.println("Lo sentimos, " + nombreCliente + ". No hay
suficiente pan. Por favor, espere mientras horneamos más.");
                    }
                } else if (opcion == 3) {
                    System.out.println("Gracias por visitar la panadería, " + nombreCliente +
". ¡Hasta luego!");
                    // Notificar la desconexión al servidor
                    panaderia.registrarDesconexion(nombreCliente);
                    break;
                } else {
                    System.out.println("Opción no válida. Intente de nuevo.");
                }
            }
        } catch (Exception e) {
            System.out.println("Error al conectar con el servidor RMI:");
            e.printStackTrace();
        }
    }
}
```