INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

UNIDAD DE APRENDIZAJE

SISTEMAS DISTRIBUIDOS

PRACTICA No. 5

OBJETOS DISTRIBUIDOS

ALUMNO

GÓMEZ GALVAN DIEGO YAEL

GRUPO

7CM1

PROFESOR

CARRETO ARELLANO CHADWICK

FECHA DE ENTREGA

26 DE MARZO DE 2025

INDICE

ANTESCEDENTES

La evolución de la tecnología informática ha impulsado el paso de sistemas centralizados hacia sistemas distribuidos cada vez más complejos. En las primeras décadas de la computación, predominaban los modelos centralizados (por ejemplo, grandes mainframes) donde todo el procesamiento residía en una sola máquina. Con la popularización de las redes de computadoras, surgió el paradigma cliente-servidor, permitiendo que múltiples equipos (clientes) interactuaran con servidores remotos para compartir datos y funcionalidades. Este cambio marcó el inicio de los sistemas distribuidos modernos, donde diversas aplicaciones colaboran a través de una infraestructura de red.

Paralelamente, la programación orientada a objetos se consolidó como una metodología principal en el desarrollo de software debido a su capacidad para modularizar la lógica en entidades (objetos) que encapsulan estado y comportamiento. La combinación de estos dos enfoques, distribución en red y orientación a objetos, dio lugar a nuevos modelos de diseño de software en los que los objetos pueden comunicarse más allá de los límites de un solo proceso o máquina. Así nacen los modelos de objetos distribuidos, que extienden el concepto tradicional de objeto a entornos donde múltiples procesos interactúan mediante llamadas remotas.

En este contexto, diferentes tecnologías fueron desarrolladas para soportar objetos distribuidos. A nivel multi-plataforma, emergieron estándares como CORBA (Common Object Request Broker Architecture) y protocolos basados en componentes (COM/DCOM en entornos Windows). En el ecosistema Java, la solución nativa fue Java RMI (Remote Method Invocation), introducida a mediados de los años noventa. Java RMI proporcionó una manera sencilla y tipada de invocar métodos de objetos ubicados en distintas máquinas virtuales Java (JVM) a través de la red, manteniendo una sintaxis cercana a la de las invocaciones locales pero con el soporte del lenguaje para manejar la distribución y posibles fallos en la comunicación.

Objetos Distribuidos y Java RMI

Un objeto distribuido es una entidad de software cuyos métodos pueden ser invocados desde otros procesos o máquinas a través de una red, manteniendo la apariencia de una llamada local. Esto extiende el modelo orientado a objetos tradicional al contexto distribuido: los objetos no están confinados a una sola máquina, sino que pueden residir en diferentes nodos de un sistema. La idea central es que un programa cliente pueda interactuar con un objeto remoto como si fuera parte de su propio espacio de direcciones, solicitando operaciones y obteniendo respuestas mediante invocaciones de métodos.

Entre las ventajas de utilizar objetos distribuidos destaca la posibilidad de reutilizar e integrar componentes de software de manera modular a escala de red. Un objeto distribuido encapsula su estado interno y expone operaciones que pueden ser utilizadas por clientes remotos, preservando los principios de encapsulación y abstracción incluso a través de los límites de la red. Esto

facilita la construcción de sistemas complejos dividiéndolos en servicios u objetos especializados que interactúan remotamente, en lugar de concentrar toda la lógica en una única aplicación monolítica difícil de escalar o mantener.

Además, el modelo de objetos distribuidos proporciona un tipado fuerte y coherente en las interacciones remotas. Tanto el cliente como el servidor acuerdan los tipos de datos y la interfaz del objeto remoto, lo que reduce errores de interpretación que podrían ocurrir al enviar mensajes sin formato en sistemas basados únicamente en sockets. Al emplear invocaciones de métodos en vez de protocolos de comunicación ad-hoc, el desarrollador trabaja con un nivel de abstracción más alto y cede al middleware la tarea de enviar datos por la red, esperar respuestas y gestionar posibles fallos. En suma, este enfoque mejora la transparencia en la invocación (hacer una llamada remota se parece en su forma a una local) y aumenta la productividad en el desarrollo de aplicaciones distribuidas.

Java RMI (Remote Method Invocation) es la tecnología que proporciona la plataforma Java para implementar este modelo de objetos distribuidos. En Java RMI, un objeto remoto se define mediante una *interfaz remota* que extiende java.rmi.Remote. Esta interfaz declara los métodos que podrán llamarse desde otro proceso, especificando que dichos métodos pueden lanzar java.rmi.RemoteException (lo cual obliga a manejar en el cliente posibles errores de red). En el lado del servidor, se desarrolla una clase que implementa esta interfaz remota; comúnmente la clase extiende de java.rmi.server.UnicastRemoteObject, lo cual facilita su exportación como objeto remoto único accesible vía RMI. Para que los clientes remotos puedan encontrar este objeto, el servidor lo registra en un servicio de directorio conocido como RMI Registry (mediante la herramienta rmiregistry o creando un registro embebido en la aplicación). El *rmiregistry* es un proceso que típicamente se ejecuta en la misma máquina que el servidor y mantiene un mapa de nombres a referencias de objetos remotos activos. El servidor, al iniciar, crea una instancia del objeto remoto e invoca un método de registro (por ejemplo Naming.rebind("NombreServicio", referenciaObjeto)) o utiliza la API de java.rmi.registry.Registry para asociar un nombre lógico con la referencia de su objeto remoto. De esta manera, el objeto queda publicado en el registro y disponible para ser localizado por los clientes.

En el lado cliente, el proceso es complementario: el cliente necesita conocer la dirección (host y puerto) donde se ejecuta el RMI Registry del servidor, así como el nombre lógico bajo el cual se registró el objeto remoto deseado. Empleando la clase java.rmi.Naming (por ejemplo, Naming.lookup("rmi://servidor:1099/NombreServicio")) o mediante una referencia obtenida a la interfaz Registry, el cliente realiza una búsqueda (*lookup*) que le devuelve una referencia remota al objeto distribuido. En la práctica, lo que el cliente obtiene es un stub, que actúa como representante local (proxy) del objeto remoto. El stub es una clase, generalmente generada de forma automática por RMI, que implementa la misma interfaz remota que el objeto real, de modo que cuando el cliente invoca un método, en realidad está llamando a un método del stub.

El stub se encarga entonces de serializar (convertir a un formato transmisible) los parámetros de la invocación, establecer la comunicación con el servidor a través de la red y enviar la petición al objeto remoto verdadero.

Históricamente, la comunicación RMI utilizó un esquema de stub/skeleton explícito. El stub, en el cliente, empacaba la llamada y la enviaba; el skeleton (esqueleto), en el servidor, recibía la petición, la desempaquetaba y llamaba al método correspondiente en el objeto real, enviando de vuelta el resultado. En las primeras versiones de Java, el desarrollador tenía que generar las clases stub (y skeleton) usando la herramienta rmic a partir de la implementación del objeto remoto. Las versiones modernas de Java eliminan la necesidad de generar un skeleton manualmente (el framework RMI lo maneja internamente) y pueden generar el stub dinámicamente en tiempo de ejecución, siempre que la interfaz remota esté disponible. No obstante, el concepto fundamental no cambia: sigue existiendo un objeto proxy en el cliente y un mecanismo en el servidor que hace de intermediario, asegurando que la invocación remota llegue al objeto destino correcto.

Una vez establecido lo anterior, la comunicación entre las JVM cliente y servidor durante una llamada RMI ocurre de forma transparente. Cuando el cliente invoca un método remoto mediante el stub, la infraestructura RMI en la JVM cliente abre (o reutiliza) una conexión de socket hacia la JVM del servidor (por defecto en el puerto 1099 u otro configurado para RMI) usando un protocolo especializado (JRMP en la implementación estándar de RMI). El stub envía entonces la información de la invocación: identificador del objeto remoto solicitado, nombre del método y parámetros serializados. En el lado servidor, el sistema RMI recibe esa solicitud entrante, identifica qué objeto remoto y método se desean ejecutar, y delega la ejecución a un hilo de servicio. Ese hilo invocará el método real sobre la instancia del objeto remoto en el servidor, tal como si fuese una llamada local dentro de esa JVM. Cuando el método termina (ya sea retornando un valor o lanzando una excepción), el resultado o la excepción se encapsula y se envía de regreso al cliente a través de la misma conexión. El stub en el cliente recibe la respuesta, la deserializa reconvirtiéndola al objeto Java apropiado (o excepción), y finalmente la entrega al código llamador como retorno del método invocado. Para el desarrollador de la aplicación, este flujo completo se manifiesta simplemente como una llamada a un método que puede devolver el resultado esperado o lanzar una RemoteException en caso de error, igual que cualquier método podría lanzar una excepción.

Gracias a este mecanismo, Java RMI logra que los desarrolladores construyan aplicaciones distribuidas complejas sin tener que lidiar directamente con los detalles de la comunicación de bajo nivel. La plataforma se encarga de proveer la transparencia de invocación remota: el énfasis del programador puede permanecer en la lógica de negocio del objeto (qué hace cada método) en lugar de como enviar y recibir datos por la red. Cabe destacar que RMI también permite pasar objetos complejos como parámetros o resultados en las invocaciones (mediante serialización automática), e incluso transferir referencias a objetos remotos como parte de una llamada, de

modo que el receptor obtenga un stub y pueda a su vez invocar métodos remotos en ese objeto. Esta flexibilidad habilita patrones avanzados de interacción, como invocaciones encadenadas entre varios servidores u objetos callback donde el servidor invoca métodos de un objeto remoto provisto por el cliente. En definitiva, el modelo de objetos distribuidos soportado por Java RMI proporciona una base sólida y conveniente para desarrollar sistemas distribuidos manteniendo los principios de la orientación a objetos, a la vez que aborda las complejidades inherentes de la comunicación en red.

Modelo de Múltiples Clientes y Múltiples Servidores

En un sistema distribuido realista no suele haber un único cliente consumiendo un único servicio, sino múltiples clientes que acceden simultáneamente a múltiples servidores o servicios distribuidos. Este modelo de "múltiples clientes/múltiples servidores" se refiere a una arquitectura en la que varios procesos servidor ofrecen diferentes recursos o funcionalidades, y un conjunto potencialmente grande de clientes los utiliza de forma concurrente a través de la red. En el contexto de objetos distribuidos con RMI, esto significa que puede haber diversos objetos remotos (posiblemente en distintos servidores físicos o en distintos procesos) disponibles, y numerosos clientes realizando invocaciones remotas sobre ellos al mismo tiempo. Cada cliente puede conectarse a uno o varios servidores según las necesidades de la aplicación, y cada servidor está preparado para atender concurrentemente las solicitudes provenientes de diferentes clientes.

La arquitectura lógica de un sistema con múltiples servidores bajo el enfoque de objetos distribuidos suele organizar los componentes por roles o por servicios. Por ejemplo, en una plataforma bancaria distribuida podríamos tener separado el servicio de gestión de cuentas, el servicio de procesamiento de pagos y el servicio de préstamos, cada uno implementado como uno o varios objetos remotos residiendo quizás en diferentes servidores. Alternativamente, podríamos replicar el mismo servicio en varias instancias de servidor para atender una gran cantidad de clientes (balanceo de carga). En cualquier caso, los clientes necesitan un mecanismo para conectarse con el servidor adecuado: gracias a RMI, esto se logra a través del registro de objetos remotos. Puede haber un registro RMI en cada servidor (accesible en un puerto específico de ese host) donde los clientes buscan el objeto deseado según el servicio que requieran, o un registro centralizado que actúe como punto único de acceso para listar todos los servicios disponibles en la plataforma (por ejemplo, un servidor dedicado que conoce las referencias de objetos en diversos nodos). Los clientes, una vez que obtienen la referencia (stub) de un objeto remoto de determinado servidor, establecen la comunicación directamente con ese servidor para invocar los métodos. De manera similar, es factible que los propios servidores actúen como clientes de otros servidores en este modelo: un objeto remoto puede invocar métodos de otro objeto remoto en otra JVM si la lógica del negocio lo requiere, permitiendo encadenar servicios (por ejemplo, el servidor de pagos podría invocar al servidor de cuentas para actualizar un saldo durante una transferencia).

Este esquema de múltiples servidores distribuidos ofrece importantes ventajas frente a una arquitectura monolítica tradicional. En un diseño monolítico, toda la funcionalidad del sistema (por ejemplo, todas las operaciones bancarias) se ejecutaría en un solo proceso o máquina. Eso puede simplificar ciertos aspectos iniciales, pero rápidamente presenta limitaciones en entornos de gran escala: un monolito es difícil de escalar (sólo se puede escalar duplicando la aplicación entera en otro servidor, lo que no separa carga por funcionalidad), es un único punto de fallo (si ese servidor cae, todo el servicio bancario queda inaccesible) y las actualizaciones o mantenimiento afectan a todo el sistema a la vez. En cambio, una arquitectura distribuida con múltiples servidores permite escalar de forma más granular: se pueden añadir más servidores para un servicio particular que esté recibiendo alta demanda (por ejemplo, más instancias del servidor de pagos si ese módulo es muy utilizado) sin tener que duplicar necesariamente otros componentes menos utilizados. También mejora la tolerancia a fallos: si un módulo o servidor específico falla, los demás pueden continuar operando; incluso se pueden implementar mecanismos de redundancia donde otro servidor asuma el rol del que falló. Igualmente, el desarrollo y mantenimiento se benefician de la separación de responsabilidades: cada servicio (objeto distribuido o conjunto de objetos) puede evolucionar o actualizarse de forma relativamente independiente, siempre y cuando respete las interfaces públicas acordadas, lo que es más complejo en un monolito fuertemente acoplado.

En comparación con un esquema basado únicamente en sockets y protocolos manuales, el enfoque de objetos distribuidos con RMI simplifica grandemente las conexiones entre múltiples clientes y servidores. Si se usaran sockets crudos, cada servidor debería manejar conexiones de muchos clientes concurrentemente, lo que implica crear y gestionar hilos para cada conexión, definir un protocolo de mensajes claro para intercambiar datos, y parsear esas comunicaciones en ambos extremos. Además, habría que codificar la lógica para que los clientes sepan a qué puerto y formato comunicarse con cada tipo de servicio, y ocuparse de detalles como la sincronización de accesos concurrentes al recurso cuando múltiples peticiones llegan a la vez por el socket. Con RMI, gran parte de ese trabajo se automatiza: cada servidor expone métodos a través de una interfaz bien definida y registra sus objetos; los clientes localizan el objeto y luego invocan métodos sin preocuparse por cómo se empaquetan o envían los datos. La concurrencia es manejada en el servidor por el propio sistema RMI que, como se mencionó, lanza un hilo por cada invocación entrante. La comunicación está tipada y es más segura, en el sentido de que si un cliente espera un método transferirFondos(double monto) de un objeto remoto, no existe ambigüedad en la interacción: se llama a ese método con un double y el servidor sabe exactamente cómo procesarlo, comparado con interpretar un mensaje textual donde podría haber errores de formato. En resumen, usar RMI en un entorno de múltiples clientes/servidores reduce la complejidad del código de comunicación, evita tener que reinventar mecanismos de red y permite centrarse en la lógica distributiva.

Un ejemplo aplicado al entorno bancario ayuda a ilustrar estos beneficios. Imaginemos una red bancaria donde existen numerosos cajeros automáticos, aplicaciones de banca en línea y

terminales internas en sucursales (todos actuando como clientes) que necesitan acceder a las operaciones bancarias centrales. En lugar de conectar todos estos clientes a un solo servidor central monolítico, la entidad puede desplegar varios servidores distribuidos por función o ubicación: por ejemplo, un servidor principal que maneja cuentas y balances generales, servidores regionales que procesan las transacciones de cajeros en cada zona geográfica, y quizás servidores especializados en operaciones de gran volumen como compensaciones interbancarias. Con RMI, cada uno de estos servidores podría ofrecer objetos remotos para las operaciones bajo su cargo (el servidor de cuentas podría ofrecer un objeto remoto GestorDeCuenta, los servidores regionales un objeto ServicioATM, etc.). Los cajeros automáticos se conectarían al servicio de su región para las operaciones cotidianas, pero si se requiere una transacción que involucra datos globales, el servicio regional podría a su vez consultar al servidor principal mediante otra invocación remota. Todo este entramado funcionaría de forma transparente: cada cliente o servidor simplemente invoca métodos en los objetos remotos adecuados. La plataforma Java RMI maneja las conexiones y la traducción de datos. El resultado es un sistema bancario mucho más escalable y flexible que puede atender a miles de usuarios simultáneamente, adaptarse al crecimiento (añadiendo más servidores o instancias cuando la carga aumenta) y mantener la consistencia y disponibilidad de las operaciones incluso si alguna de las máquinas presenta problemas. De esta manera, el enfoque de múltiples clientes y servidores distribuidos proporciona la robustez necesaria en un escenario crítico como el bancario, superando las limitaciones de un modelo centralizado o de soluciones de red de bajo nivel.

Sincronización entre Servidores en Entornos Distribuidos

Un aspecto crítico en sistemas distribuidos es la correcta sincronización de las operaciones concurrentes para evitar inconsistencias. La concurrencia ocurre cuando dos o más hilos de ejecución (threads) o procesos intentan acceder o modificar al mismo tiempo un recurso compartido. Si no se controla, esto puede derivar en condiciones de carrera (race conditions), en las cuales el resultado final depende del intercalado no determinista de operaciones concurrentes. Un ejemplo típico en un sistema bancario sería el de dos transacciones intentando actualizar el saldo de la misma cuenta bancaria simultáneamente desde distintos puntos de la red (por ejemplo, un pago en línea y un retiro en cajero sobre la misma cuenta a la vez). Sin algún mecanismo de coordinación, es posible que ambas operaciones lean un saldo inicial común y luego cada una sobrescriba el valor, provocando una pérdida de una de las actualizaciones (un monto "desaparece" porque la última escritura ganó la carrera). Este tipo de error puede comprometer gravemente la integridad de los datos financieros, de ahí la importancia de diseñar el sistema considerando la sincronización desde el principio.

En entornos de un solo proceso, los lenguajes de programación proporcionan herramientas para manejar estas situaciones, protegiendo las secciones críticas mediante exclusión mutua. En Java, el concepto de monitor está integrado a cada objeto: la palabra clave synchronized permite que un método o bloque de código sea accedido por un solo hilo a la vez, garantizando que ningún

otro hilo ejecute esa sección crítica simultáneamente. Adicionalmente, Java ofrece las primitivas wait() y notify()/notifyAll() que, usadas dentro de un bloque sincronizado, permiten que un hilo libere el candado (lock) del objeto monitor y espere a que alguna condición cambie, mientras otro hilo puede notificarle cuando dicha condición se haya cumplido, coordinando así la ejecución de hilos de manera ordenada. Estos mecanismos funcionan muy bien en programas monolíticos con múltiples hilos en la misma JVM. Sin embargo, ¿cómo se aplican en un contexto distribuido donde los hilos están en procesos separados potencialmente en máquinas distintas?

Java RMI, al permitir invocaciones remotas, posibilita también extender el concepto de monitor a través de la red. Una forma de lograr la sincronización en un entorno distribuido es mediante el uso de un servidor de sincronización, es decir, un objeto remoto especial que actúa como árbitro o guardián de ciertos recursos compartidos. Este objeto remoto puede implementar métodos sincronizados (synchronized) de tal manera que imponga exclusión mutua entre todas las invocaciones que reciba. Por ejemplo, supongamos que se diseña un objeto remoto MonitorTransacciones con un método procesarTransaccion(...) marcado como synchronized. Si múltiples clientes o incluso múltiples servidores invocan remotamente procesarTransaccion al mismo tiempo (quizá para procesar transacciones sobre una misma cuenta global), la primera invocación que llegue obtendrá el bloqueo del objeto MonitorTransacciones en el servidor de sincronización, y las demás quedarán bloqueadas esperando. Aunque cada llamada proviene de una JVM distinta, en el lado del servidor todas las peticiones concurrentes son gestionadas por hilos que intentan entrar en el mismo método sincronizado del mismo objeto; Java asegura que sólo un hilo a la vez ejecuta el código dentro del método. Las invocaciones adicionales permanecerán encoladas automáticamente hasta que la invocación en curso termine y libere el monitor, momento en el cual la siguiente puede entrar. Así, el objeto remoto compartido está funcionando como un monitor distribuido: múltiples procesos remotos logran una exclusión mutua al acceder a un recurso crítico a través de un único punto de control.

El uso de wait() y notify() en este contexto distribuido también es viable y útil. Continuando con el objeto MonitorTransacciones, podríamos imaginar que implementa una lógica en la cual si una condición necesaria no se cumple para procesar la transacción en ese momento (por ejemplo, si el monto excede cierto límite y se requiere una autorización, o si un recurso aún no está disponible), el método remoto invoca wait() sobre sí mismo. Esto haría que el hilo que estaba atendiendo esa llamada remota entre en espera liberando el bloqueo del objeto remoto monitor, permitiendo que otras operaciones puedan ejecutarse mientras tanto. Cuando la condición cambie (por ejemplo, otro método remoto u otro hilo interno del servidor invoca notify() o notifyAll en ese mismo objeto remoto tras completar cierta acción pendiente), el hilo en espera se despertará y reintentará completar su operación. Desde la perspectiva del cliente que invocó originalmente, la llamada remota permanecerá bloqueada (sin retornar) hasta que el servidor haya podido terminar completamente la operación tras el notify(). En efecto, el cliente experimenta un comportamiento similar a una espera sincronizada tradicional, con la diferencia

de que toda la gestión de la espera ocurrió en el lado servidor utilizando las primitivas de monitor habituales.

Cabe resaltar que, aunque este enfoque de monitores distribuidos mediante RMI puede resolver muchos problemas de concurrencia, también se deben tener en cuenta consideraciones adicionales. La latencia de la red implica que mantener un candado distribuido durante mucho tiempo puede retrasar a otros procesos que esperan, por lo que las secciones críticas remotas deben mantenerse lo más breves y eficientes posible. Asimismo, si un cliente que poseía el bloqueo se quedara congelado o desconectado sin liberar la sección crítica, podría ser necesario implementar mecanismos de tiempo de espera (*timeout*) o estrategias de recuperación para evitar un interbloqueo permanente en el sistema. En sistemas bancarios reales, con grandes volúmenes de operaciones, a menudo se complementa este nivel de sincronización con soluciones de más alto nivel, como gestores de transacciones distribuidas o bases de datos que manejan el bloqueo de registros, las cuales garantizan atomicidad y aislamiento en las operaciones. No obstante, el principio subyacente es equivalente: asegurar que ciertas operaciones críticas no sean interferidas por otras al mismo tiempo. Java RMI brinda la flexibilidad para implementar estos esquemas a nivel de aplicación cuando se requiera, permitiendo construir un servidor de sincronización o coordinar directamente la exclusión mutua en los métodos remotos de los objetos compartidos. De este modo, se logra preservar la consistencia y la integridad de los datos en un sistema distribuido, incluso bajo altas cargas de concurrencia.

PLANTEAMIENTO DEL PROBLEMA

En la actualidad, los sistemas distribuidos son una parte fundamental del desarrollo de software moderno, ya que permiten distribuir procesos, recursos y servicios a través de múltiples nodos interconectados por una red. Este tipo de arquitectura ofrece ventajas significativas en términos de escalabilidad, disponibilidad, rendimiento y modularidad. Sin embargo, también plantea desafíos técnicos importantes, especialmente en lo que respecta a la comunicación entre componentes, el control de concurrencia, la integridad de los datos compartidos y la sincronización entre procesos remotos.

Tradicionalmente, muchos sistemas distribuidos se han desarrollado utilizando esquemas de comunicación directa a través de sockets. Si bien este enfoque puede ser eficaz en entornos simples o controlados, su uso implica una serie de limitaciones. En primer lugar, obliga al programador a encargarse del manejo detallado de las conexiones, de la serialización y deserialización de datos, y de la implementación manual de protocolos de comunicación entre procesos. Esto no solo incrementa la complejidad del código, sino que también introduce mayores probabilidades de error, dificulta el mantenimiento del sistema y reduce su capacidad de adaptación a cambios o crecimiento.

Además, cuando estos sistemas distribuidos involucran múltiples clientes y múltiples servidores —como es común en plataformas que brindan diferentes servicios simultáneamente— la falta de un modelo de comunicación unificado y tipado dificulta la coordinación efectiva entre componentes. A esto se suma la problemática del acceso concurrente a recursos compartidos, lo cual puede derivar en condiciones de carrera, pérdidas de datos o comportamientos indeterminados si no se implementan mecanismos de sincronización adecuados. Por tanto, resulta necesario buscar enfoques que permitan encapsular la comunicación remota de forma más segura, organizada y mantenible, al tiempo que se garanticen mecanismos sólidos para la gestión de concurrencia.

Esta situación se vuelve particularmente crítica en el contexto de los sistemas bancarios, donde la seguridad, la integridad de la información y la consistencia de las operaciones son aspectos esenciales. En aplicaciones bancarias distribuidas —como las que permiten a los usuarios realizar consultas de saldo, depósitos, retiros y transferencias desde múltiples dispositivos— es común implementar un modelo de múltiples clientes conectándose a múltiples servidores especializados en diferentes funciones. Cuando este modelo se basa exclusivamente en sockets, la falta de abstracción complica el desarrollo, la depuración y la extensión del sistema. Además, no contar con una estrategia robusta de sincronización puede generar errores graves, como actualizaciones inconsistentes del saldo de una cuenta si dos operaciones se ejecutan simultáneamente sin control.

A partir de este escenario, se identifica como problema principal la limitada escalabilidad, mantenibilidad y seguridad que presenta el modelo tradicional de múltiples clientes y múltiples

servidores basado en sockets en un sistema bancario distribuido. La necesidad de manejar de forma manual la comunicación y la concurrencia, sumada al riesgo de interferencias entre operaciones críticas, representa una barrera significativa para construir aplicaciones confiables, seguras y sostenibles en este dominio.

Por tanto, se plantea como solución la transición hacia un modelo basado en objetos distribuidos utilizando Java RMI, el cual permite una mejor modularización del sistema, una comunicación más limpia entre componentes a través de interfaces remotas, y una mayor facilidad para escalar o extender la aplicación. Además, se propone el desarrollo e integración de un servidor de sincronización como objeto remoto, que actúe como monitor distribuido y regule el acceso a las secciones críticas, garantizando así la integridad de las operaciones bancarias. Esta reingeniería del sistema busca no solo superar las limitaciones del modelo anterior, sino también proporcionar una arquitectura distribuida moderna, segura, extensible y alineada con los principios de la ingeniería de software actual.

PROPUESTA DE SOLUCIÓN

Con el objetivo de superar las limitaciones identificadas en el modelo tradicional de múltiples clientes y múltiples servidores basado en sockets, se propone la reingeniería del sistema bancario hacia una arquitectura distribuida fundamentada en el uso de **objetos distribuidos mediante Java RMI**. Esta transición permitirá encapsular los diferentes servicios del sistema como objetos remotos, mejorando la modularidad, escalabilidad y mantenibilidad del sistema.

La solución consiste en diseñar e implementar un conjunto de servidores especializados, cada uno expuesto como un objeto remoto accesible mediante interfaces RMI. Cada servidor atenderá una funcionalidad específica del sistema bancario:

- ✓ **Servidor de autenticación**, encargado de validar las credenciales de los usuarios.
- ✓ **Servidor de saldo**, responsable de gestionar las consultas y actualizaciones del saldo de las cuentas.
- ✓ **Servidor de transacciones**, encargado de procesar operaciones bancarias como depósitos, retiros y transferencias.
- ✓ **Servidor de sincronización**, que actuará como un monitor distribuido para regular el acceso a las secciones críticas, evitando condiciones de carrera y garantizando la integridad de las operaciones.

Los clientes del sistema se conectarán a estos objetos distribuidos a través del registro RMI, localizando cada servicio por su nombre lógico. Una vez obtenida la referencia al objeto remoto, podrán invocar métodos directamente sobre el servicio, de forma transparente, como si se tratara de una llamada local. Esto elimina la necesidad de gestionar sockets y protocolos de bajo nivel, y permite centralizar la lógica de negocio en componentes especializados.

La incorporación de un servidor de sincronización resulta clave para resolver el problema de concurrencia. Este componente controlará el acceso exclusivo a operaciones críticas, utilizando sincronización en métodos remotos (synchronized) junto con las primitivas wait() y notifyAll() de Java. De esta forma, se garantiza que las operaciones bancarias que afectan el estado del sistema se realicen sin interferencias, manteniendo la coherencia de los datos.

La solución propuesta no solo mejora la estructura interna del sistema, sino que también habilita su evolución futura hacia entornos más complejos. Por ejemplo, podría adaptarse fácilmente a arquitecturas de microservicios, desplegarse en múltiples nodos o incluso integrarse con bases de datos distribuidas. Todo esto sin modificar sustancialmente la lógica de negocio, gracias a la separación de responsabilidades y al uso de interfaces bien definidas.

Esta propuesta responde a la necesidad de construir sistemas distribuidos más seguros, eficientes y mantenibles, particularmente en contextos donde la concurrencia y la integridad de los datos son factores críticos, como ocurre en las plataformas bancarias.

MATERIALES Y METODOS

Para la implementación de la solución propuesta se utilizaron las siguientes herramientas y tecnologías:

- ✓ Lenguaje de programación: Java SE, versión 22.
- ✓ Modelo de objetos distribuidos: Java RMI, utilizado para implementar la comunicación remota entre objetos en diferentes JVM.
- ✓ IDE de desarrollo: NetBeans 21, utilizado para el desarrollo, compilación y ejecución de los módulos del sistema.
- ✓ Consola de comandos: Utilizada para iniciar el registro RMI y verificar rutas de ejecución.
- ✓ Sistema operativo: Windows 11

DESARROLLO DE SOLUCIÓN

La solución propuesta fue implementada a través de una arquitectura modular compuesta por múltiples objetos distribuidos, utilizando Java RMI como tecnología principal para la comunicación remota entre servidores y clientes. El sistema fue diseñado para simular un entorno bancario en el que diversos usuarios pueden autenticarse, consultar saldos, realizar depósitos, retiros o transferencias, todo ello mediante una estructura orientada a servicios distribuidos, con sincronización concurrente entre procesos.

Estructura general del sistema

El sistema está dividido en cuatro módulos funcionales principales, cada uno representado por una interfaz remota y su correspondiente implementación:

- ✓ Autenticación: define y expone un método remoto para validar el número de cuenta y el NIP del usuario.
- ✓ Saldo: permite consultar y actualizar el saldo de una cuenta bancaria.
- ✓ Transacciones: contiene la lógica para ejecutar depósitos, retiros y transferencias.
- ✓ Sincronización: actúa como monitor remoto para controlar el acceso a las operaciones críticas del sistema.

Cada uno de estos módulos fue registrado en el *rmiregistry*, utilizando nombres lógicos únicos para que pudieran ser localizados fácilmente por los clientes. Las interfaces remotas definen la funcionalidad visible del servicio, y las clases que las implementan encapsulan la lógica de negocio asociada.

Implementación técnica

El primer paso fue la implementación del servidor de autenticación, cuya clase ImpAutenticacion almacena un conjunto predefinido de cuentas con sus respectivos NIPs:

```java
public class ImpAutenticacion extends UnicastRemoteObject implements IAutenticacion {

    private Map<String, Integer> cuentas;

    public ImpAutenticacion() throws RemoteException {
        super();
        cuentas = new HashMap<>();
        cuentas.put("1234567890", 1234);
        cuentas.put("2468024680", 2468);
        cuentas.put("3692581470", 3692);
        cuentas.put("4826048260", 4826);
        cuentas.put("6284062840", 6284);
        System.out.println("[" + LocalTime.now() + "] Servidor de Autenticación inicializado con cuentas de prueba.");
    }

    @Override
    public String autenticar(String cuenta, int nip) throws RemoteException {
        System.out.println("[" + LocalTime.now() + "] Solicitud de autenticación para cuenta: " + cuenta);
        if (cuentas.containsKey(cuenta) && cuentas.get(cuenta).equals(nip)) {
            System.out.println("[" + LocalTime.now() + "] Autenticación exitosa para cuenta: " + cuenta);
            return "Autenticacion Exitosa";
        } else {
            System.out.println("[" + LocalTime.now() + "] Fallo en autenticación para cuenta: " + cuenta);
            return "Error en autenticación";
        }
    }
}
```

Este objeto remoto fue registrado bajo el nombre "AutenticacionService", y es consultado por el cliente al iniciar sesión.

En segundo lugar, se desarrolló el servidor de saldo (ImpSaldo), encargado de gestionar un mapa de cuentas y sus saldos:

```java
public class ImpSaldo extends UnicastRemoteObject implements ISaldo {
    private ConcurrentHashMap<String, Double> cuentas;
    public ImpSaldo() throws RemoteException {
        super();
        cuentas = new ConcurrentHashMap<>();
        cuentas.put("1234567890", 9500.0);
        cuentas.put("2468024680", 10000.0);
        cuentas.put("3692581470", 6348.3);
        cuentas.put("4826048260", 100.80);
        cuentas.put("6284062840", 1150000.0);
        System.out.println("[" + LocalTime.now() + "] Servidor de Saldo inicializado con cuentas de prueba.");
    }
    @Override
    public double consultarSaldo(String cuenta) throws RemoteException {
        System.out.println("[" + LocalTime.now() + "] Consulta de saldo solicitada para cuenta: " + cuenta);
        if (cuentas.containsKey(cuenta)) {
            System.out.println("[" + LocalTime.now() + "] Saldo actual: $" + cuentas.get(cuenta));
            return cuentas.get(cuenta);
        } else {
            System.out.println("[" + LocalTime.now() + "] Cuenta no encontrada: " + cuenta);
            return -1; // Cuenta no encontrada
        }
    }
    @Override
    public String actualizarSaldo(String cuenta, double nuevoSaldo) throws RemoteException {
        System.out.println("[" + LocalTime.now() + "] Solicitud de actualización de saldo para cuenta: " + cuenta + " -> Nuevo saldo: $" + nuevoSaldo);
        if (cuentas.containsKey(cuenta)) {
            cuentas.put(cuenta, nuevoSaldo);
            System.out.println("[" + LocalTime.now() + "] Saldo actualizado correctamente para cuenta: " + cuenta);
            return "Nuevo saldo $" + consultarSaldo(cuenta);
        } else {
            System.out.println("[" + LocalTime.now() + "] Error: Cuenta no encontrada para actualizar saldo.");
            return "Cuenta no encontrada, no se puede actualizar saldo.";
        }
    }
}
```

Esta clase remota expone métodos para consultar y actualizar el saldo de una cuenta bancaria. Para fines de prueba, se inicializó con cuentas y saldos variados. Su registro en el RMI se hizo bajo el nombre "SaldoService".

Posteriormente, se diseñó el servidor de transacciones (ImpTransacciones), el cual realiza invocaciones remotas a los servicios de saldo y sincronización para llevar a cabo sus operaciones:

```
public ImpTransacciones() throws RemoteException {
    super();
    try {
        Registry registry = LocateRegistry.getRegistry("localhost");
        saldo = (ISaldo) registry.lookup("SaldoService");
        sync = (ISincronizacion) registry.lookup("SyncService");
        System.out.println("[" + LocalTime.now() + "] Servidor de Transacciones conectado correctamente a los servicios remotos.");
    } catch (Exception e) {
        System.err.println("[" + LocalTime.now() + "] Error al conectar con servicios de saldo o sincronización: " + e.getMessage());
        throw new RemoteException("Servicios no disponibles.");
    }
}
```

```
@Override
public String depositar(String cuenta, double monto) throws RemoteException {
    System.out.println("[" + LocalTime.now() + "] Cliente solicitó depósito en cuenta: " + cuenta + " por $" + monto);
    if (!sync.solicitarAcceso()) {
        return "Error: No se pudo obtener acceso a la transacción.";
    }

    try {
        double saldoActual = saldo.consultarSaldo(cuenta);
        if (saldoActual == -1) return "Cuenta no encontrada.";
        saldoActual += monto;
        String resultado = saldo.actualizarSaldo(cuenta, saldoActual);
        System.out.println("[" + LocalTime.now() + "] Depósito completado para cuenta " + cuenta + ". Nuevo saldo: $" + saldoActual);
        return resultado;
    } finally {
        sync.liberarAcceso();
    }
}
```

```
@Override
public String retirar(String cuenta, double monto) throws RemoteException {
    System.out.println("[" + LocalTime.now() + "] Cliente solicitó retiro en cuenta: " + cuenta + " por $" + monto);
    if (!sync.solicitarAcceso()) {
        return "Error: No se pudo obtener acceso a la transacción.";
    }

    try {
        double saldoActual = saldo.consultarSaldo(cuenta);
        if (saldoActual == -1) return "Cuenta no encontrada.";
        if (monto <= 0 || monto > saldoActual) return "Error: Monto inválido o saldo insuficiente.";
        saldoActual -= monto;
        String resultado = saldo.actualizarSaldo(cuenta, saldoActual);
        System.out.println("[" + LocalTime.now() + "] Retiro realizado en cuenta " + cuenta + ". Nuevo saldo: $" + saldoActual);
        return resultado;
    } finally {
        sync.liberarAcceso();
    }
}
```

Esta clase implementa los métodos depositar, retirar y transferir, y para cada operación primero consulta el saldo actual, valida los parámetros y actualiza el saldo si la operación es válida. Su nombre en el registro es "TransaccionesService".

Para resolver los problemas de concurrencia, se integró un servidor de sincronización (ImpSincronizacion), cuyo objeto remoto implementa métodos sincronizados para garantizar que solo un cliente pueda ejecutar una operación crítica a la vez:

```
public class ImpSincronizacion extends UnicastRemoteObject implements ISincronizacion {
    private boolean enUso = false;
    public ImpSincronizacion() throws RemoteException {
        super();
    }
    @Override
    public synchronized boolean solicitarAcceso() throws RemoteException {
        System.out.println("[" + LocalTime.now() + "] Cliente remoto solicitó acceso a la sección crítica.");

        try {
            while (enUso) {
                System.out.println("[" + LocalTime.now() + "] Sección crítica ocupada. Cliente en espera...");
                wait();
            }

            enUso = true;
            System.out.println("[" + LocalTime.now() + "] Acceso concedido. Cliente ingresó a la sección crítica.");
            return true;
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            throw new RemoteException("Error al solicitar acceso (interrumpido)", e);
        }
    }
    @Override
    public synchronized void liberarAcceso() throws RemoteException {
        System.out.println("[" + LocalTime.now() + "] Cliente liberó la sección crítica. Acceso disponible.");
        enUso = false;
        notifyAll();
    }
}
```

Esto simula el comportamiento de un monitor distribuido, esencial en sistemas que manejan información financiera.

Finalmente, se implementó el cliente, una aplicación de consola que permite al usuario autenticarse e interactuar con los servicios remotos:

```java
public class Cliente {
    public static void main(String[] args) throws RemoteException, NotBoundException {
        int nip, opc = 0, montoR;
        double montoD;
        String cuentaD;

        Registry registry = LocateRegistry.getRegistry("localhost");
        IAutenticacion auth = (IAutenticacion) registry.lookup("AutenticacionService");
        ISaldo saldoStub = (ISaldo) registry.lookup("SaldoService");
        ITransacciones transStub = (ITransacciones) registry.lookup("TransaccionesService");

        Scanner scan = new Scanner(System.in);

        // Solicitar datos de autenticación
        System.out.print("Ingrese su numero de cuenta: ");
        String numCuenta = scan.nextLine();

        System.out.print("Ingrese su NIP: ");
        nip = scan.nextInt();

        String respuesta = auth.autenticar(numCuenta, nip);
        System.out.println(respuesta);

        if (!respuesta.startsWith("Autenticacion Exitosa")) {
            return;
        }
```

```
while (true) {
    System.out.println("\nSeleccione la opcion deseada:");
    System.out.println("1. Consultar saldo");
    System.out.println("2. Depositar dinero");
    System.out.println("3. Retirar dinero");
    System.out.println("4. Transferencia");
    System.out.println("5. Salir");

    opc = scan.nextInt();
    scan.nextLine(); // limpiar buffer

    switch (opc) {
        case 1: // Consultar saldo
            double saldo = saldoStub.consultarSaldo(numCuenta);
            if (saldo == -1) {
                System.out.println("\nCuenta no encontrada.");
            } else {
                System.out.println("\nSaldo Actual: $" + saldo);
            }
            break;

        case 2: // Depositar
            System.out.println("Ingrese el monto a depositar: $");
            montoD = scan.nextDouble();
            scan.nextLine(); // limpiar buffer
            respuesta = transStub.depositar(numCuenta, montoD);
            System.out.println(respuesta);
            break;

        case 3: // Retirar
            System.out.println("Ingrese el monto a retirar: $");
            montoR = scan.nextInt();
            scan.nextLine(); // limpiar buffer
            respuesta = transStub.retirar(numCuenta, montoR);
            System.out.println(respuesta);
            break;
```

```
case 4: // Transferencia
    double saldoDisponible = saldoStub.consultarSaldo(numCuenta);
    System.out.println("\nSaldo disponible: $" + saldoDisponible);

    System.out.print("Ingrese el numero de cuenta destino: ");
    cuentaD = scan.nextLine();

    System.out.print("Ingrese el monto a transferir: $");
    montoD = scan.nextDouble();
    scan.nextLine(); // limpiar buffer

    respuesta = transStub.transferir(numCuenta, cuentaD, montoD);
    System.out.println("\n"+respuesta);
    break;

case 5:
    System.out.println("Sesion finalizada.");
    return;

default:
    System.out.println("Opción no válida.");
    break;
```

Flujo de ejecución del sistema

1. El cliente se conecta al registro RMI en el servidor y realiza una búsqueda del objeto remoto de autenticación.
2. Tras introducir el número de cuenta y el NIP, el cliente invoca el método remoto autenticar().
3. Si la autenticación es exitosa, el cliente accede al menú principal de operaciones bancarias.
4. Al seleccionar una operación, el cliente realiza una búsqueda del objeto remoto correspondiente (TransaccionesService o SaldoService).
5. Antes de ejecutar una operación crítica, el servicio de transacciones solicita acceso exclusivo al servidor de sincronización.
6. Una vez concedido el acceso, se ejecuta la lógica, se actualiza el saldo y se libera el recurso.
7. El resultado se imprime al cliente, repitiendo el proceso hasta que decida salir.

Validación de funcionamiento

Durante la etapa de pruebas, se verificó que las operaciones se ejecutaran correctamente en escenarios con múltiples clientes simultáneos. Se comprobó que los accesos al saldo se

sincronizan adecuadamente, evitando inconsistencias incluso en transferencias cruzadas entre cuentas. Además, se realizaron pruebas de conexión, registro de servicios y manejo de excepciones remotas para garantizar la robustez del sistema.

RESULTADOS

Una vez completada la implementación de los distintos módulos del sistema bancario distribuido con Java RMI, se realizaron diversas pruebas funcionales y de concurrencia para validar el correcto funcionamiento del sistema. A continuación, se presentan los principales resultados obtenidos durante la ejecución del proyecto.

1. Inicio del sistema distribuido

Cada uno de los servidores fue ejecutado por separado desde NetBeans, registrando su objeto remoto en el *rmiregistry*. La correcta ejecución se verificó al observar en consola mensajes de inicialización

```
[23:00:32.286289700] Servidor de Saldo inicializado con cuentas de prueba
Servidor de Saldo RMI listo.

[23:00:41.566445100] Servidor de Transacciones conectado correctamente a los servicios remotos.
Servidor de Transacciones RMI listo.

Servidor de Sincronizacion RMI listo.

[23:00:34.939693700] Servidor de Autenticacion inicializado con cuentas de prueba.
Servidor de Autenticacion RMI listo.
```

2. Autenticación de usuarios

Al ejecutar el cliente, se solicita al usuario ingresar su número de cuenta y NIP. Se probó con datos correctos e incorrectos para comprobar la validación.

- Con datos válidos: se muestra “Autenticación Exitosa”.
- Con datos inválidos: se devuelve “Error en autenticación”.

```
Ingrese su numero de cuenta: 1234567890
Ingrese su NIP: 1234
Autenticacion Exitosa


Ingrese su numero de cuenta: 2468024680
Ingrese su NIP: 2469
Error en autenticaci□n
```

3. Consulta de saldo

Una vez autenticado, el usuario selecciona la opción para consultar saldo. El cliente invoca el método remoto consultarSaldo() y muestra en pantalla el monto actual asociado a la cuenta.

```
Seleccione la opcion deseada:
1. Consultar saldo
2. Depositar dinero
3. Retirar dinero
4. Transferencia
5. Salir
1

Saldo Actual: $9500.0
```

4. Depósitos y retiros

Para probar las operaciones críticas, se ejecutaron depósitos y retiros desde diferentes clientes de forma simultánea. Se validó que:

- El sistema bloquea el recurso mientras se procesa una operación.
- El servidor de sincronización actúa correctamente.
- Se actualiza el saldo tras cada operación.

Durante estas pruebas, se lanzaron múltiples clientes ejecutando depósitos y transferencias al mismo tiempo. El sistema respondió de forma ordenada y sin inconsistencias de saldo, confirmando la efectividad del servidor de sincronización implementado con wait() y notifyAll().

```
Seleccione la opcion deseada:
1. Consultar saldo
2. Depositar dinero
3. Retirar dinero
4. Transferencia
5. Salir
4

Saldo disponible: $9500.0
Ingrese el numero de cuenta destino: 3692581470
Ingrese el monto a transferir: $9000

Transferencia exitosa. Nuevo saldo en cuenta: $500.0
```

Cliente 1. Realizando trasferencia a cliente 2

```
Seleccione la opcion deseada:
1. Consultar saldo
2. Depositar dinero
3. Retirar dinero
4. Transferencia
5. Salir
3

Ingrese el monto a retirar: $6000
Nuevo saldo $9348.3
```

Cliente 2. Realizando retiro de $6000 teniendo saldo de $6348.3 y recibiendo deposito de $9000 de cliente 2

```
Servidor de Sincronizacion RMI listo.
[23:09:16.042134800] Cliente remoto solicit acceso a la secci n cr tica.
[23:09:16.049137300] Acceso concedido. Cliente ingres  a la secci n cr tica.
[23:09:16.097922400] Cliente liber  la secci n cr tica. Acceso disponible.
[23:09:16.274187300] Cliente remoto solicit acceso a la secci n cr tica.
[23:09:16.274187300] Acceso concedido. Cliente ingres  a la secci n cr tica.
[23:09:16.282190500] Cliente liber  la secci n cr tica. Acceso disponible.
```

Servidor de sincronización mostrando mensajes de solicitud y liberación de acceso

5. Transferencias

En la operación de transferencia, el cliente origen y el destino son distintos. Se verificó que el sistema:

- Consulta el saldo actual de ambas cuentas.
- Valida el monto disponible.
- Actualiza ambos saldos correctamente.
- Solo permite una transferencia activa a la vez gracias al monitor remoto.

```
Seleccione la opcion deseada:
1. Consultar saldo
2. Depositar dinero
3. Retirar dinero
4. Transferencia
5. Salir
4

Saldo disponible: $9500.0
Ingrese el numero de cuenta destino: 3692581470
Ingrese el monto a transferir: $9000

Transferencia exitosa. Nuevo saldo en cuenta: $500.0
```

CONCLUSIONES

Durante el desarrollo de esta práctica pude comprender de forma práctica cómo funciona el modelo de objetos distribuidos, así como su aplicación real para la construcción de sistemas distribuidos robustos y organizados. La implementación de un sistema bancario distribuido me permitió no solo reforzar conceptos clave como el modularidad, la comunicación remota y el diseño orientado a servicios, sino también enfrentar y resolver retos propios del trabajo con múltiples procesos que interactúan entre sí.

Una de las cosas que más me agradaron fue la posibilidad de encapsular la lógica de cada módulo del sistema dentro de objetos remotos accesibles mediante interfaces. Esto facilitó la separación de responsabilidades y la reutilización del código, a diferencia del modelo anterior basado en sockets, donde toda la lógica estaba más acoplada y resultaba más propensa a errores de conexión o mal manejo de datos.

Sin embargo, también enfrenté varias dificultades técnicas. Al principio, tuve problemas con el registro de los servicios en el rmiregistry, especialmente con la configuración de rutas, el reconocimiento de clases remotas y el control de puertos ocupados. Fue necesario comprender mejor cómo funciona el proceso de registro y cómo asegurarse de que todas las clases necesarias estén accesibles desde las diferentes JVM. Además, al ser una arquitectura distribuida, el orden de arranque de los servidores y la comunicación entre ellos requería atención detallada para evitar errores de referencia o RemoteException.

Otra problemática importante fue la gestión de concurrencia. Al tratarse de operaciones bancarias, cualquier error en la sincronización podía comprometer la integridad del sistema. Por eso, implementar un servidor de sincronización como objeto remoto me ayudó a consolidar el uso de wait() y notifyAll() en entornos distribuidos, y a entender la importancia de garantizar la exclusión mutua en operaciones críticas como retiros o transferencias.

En general, esta práctica me permitió ver las ventajas reales del uso de RMI, como el encapsulamiento, la limpieza del código y la capacidad de escalar el sistema de manera ordenada. Aprendí que, si bien los objetos distribuidos facilitan mucho la lógica de negocio y la comunicación entre servicios, su correcta implementación requiere planificación, comprensión del modelo cliente-servidor y manejo cuidadoso de errores y concurrencia.