



INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

Carrera:

Ingeniería en Sistemas Computacionales

Unidad de aprendizaje:

Sistemas Distribuidos

Actividad:

Practica 4 – MultiCliente/MultiServidor

Integrante:

Hernández Vázquez Jorge Daniel

Profesor:

Chadwick Carreto Arellano

Fecha de entrega:

17/03/2025

INSTITUTO POLITÉCNICO NACIONAL



Índice de contenido

Antecedente.....	1
Modelo Cliente/Servidor	1
Modelo multicliente-multiservidor	2
Métodos de sincronización en Sistemas Multicliente/Multiservidor	3
Algoritmo de consenso.....	4
Sistemas centralizados.....	5
Sistemas distribuidos.....	5
Sistemas basados en token	6
Planteamiento del problema	7
Propuesta de solución.....	7
Materiales y métodos empleados	8
Desarrollo de la solución.....	10
Resultados	22
Conclusión.....	28
Bibliografía	29
Anexos.....	30
Código PanaderiaCliente.java	30
Código PanaderiaServidor.java.....	34
Código ServidorDescubrimiento.java	39

Índice de Figuras

Figura 1	1
Figura 2	10
Figura 3	11
Figura 4	11
Figura 5	13
Figura 5	14
Figura 6	14
Figura 7	15
Figura 8	15
Figura 9	16
Figura 10	16
Figura 11	18

Figura 12	19
Figura 13	20
Figura 14	21
Figura 15	21
Figura 16	22
Figura 16	23
Figura 17	23
Figura 18	23
Figura 19	24
Figura 20	24
Figura 21	24
Figura 22	25
Figura 23	25
Figura 24	25
Figura 25	25
Figura 25	26
Figura 26	26
Figura 27	27
Figura 28	27
Figura 29	27
Figura 30	28

Antecedente

En los sistemas distribuidos, la arquitectura multiservidor es una solución clave para mejorar la escalabilidad y la disponibilidad de servicios al permitir que múltiples servidores trabajen de manera conjunta para atender las solicitudes de los clientes. Este enfoque es esencial en aplicaciones que requieren alta disponibilidad y rendimiento, como plataformas de comercio electrónico, servicios en la nube y sistemas de grandes bases de datos. En un sistema multiservidor, cada servidor puede manejar un conjunto específico de clientes, distribuyendo la carga de trabajo y evitando que un solo servidor se convierta en un cuello de botella.

Uno de los principales desafíos en una arquitectura multiservidor es la gestión eficiente de la sincronización entre los servidores. Los recursos compartidos, como bases de datos o inventarios, deben estar actualizados en tiempo real para evitar inconsistencias, especialmente cuando varios servidores están manejando la misma información. Esto requiere la implementación de mecanismos de sincronización robustos y eficientes.

En este contexto, el uso de un servidor de descubrimiento juega un papel fundamental al facilitar la asignación dinámica de clientes a los servidores disponibles. Este servidor actúa como un intermediario que dirige las solicitudes de los clientes a los servidores correspondientes, optimizando la carga y mejorando la eficiencia general del sistema.

El enfoque multiservidor también implica el uso de tecnologías como sockets y hilos (threads) para gestionar las conexiones y comunicaciones entre los servidores y los clientes, asegurando una interacción fluida y sin interrupciones.

Modelo Cliente/Servidor

El modelo Cliente/Servidor es una de las arquitecturas más utilizadas en la informática moderna, ya que permite la comunicación eficiente entre dispositivos en una red. Su concepto principal radica en la diferenciación entre dos tipos de procesos: los servidores, encargados de ofrecer y gestionar recursos o servicios, y los clientes, que consumen dichos servicios mediante solicitudes específicas. Esta separación facilita la centralización de la información y la delegación de responsabilidades, lo que mejora la administración de los sistemas distribuidos y optimiza el acceso a los datos.

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

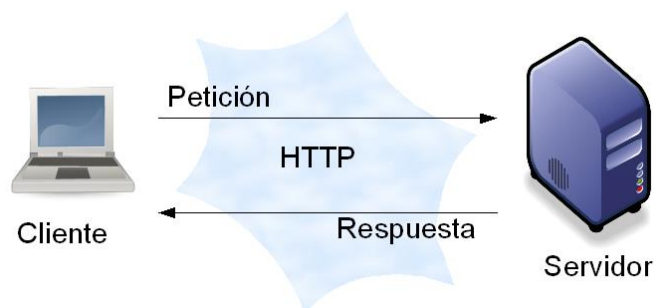


Figura 1. Diagrama del funcionamiento Modelo Cliente/Servidor.

clientes y servidores, permitiendo mayor flexibilidad y eficiencia. En la actualidad, esta arquitectura es la base de numerosos servicios, desde aplicaciones web hasta sistemas en la nube y videojuegos en línea. La comunicación entre clientes y servidores se basa en el intercambio de mensajes a través de protocolos de red. Entre los más utilizados se encuentran TCP/IP, que proporciona una transmisión

confiable, HTTP/HTTPS, empleado en la web, FTP, para transferencia de archivos, y SSH, que permite conexiones seguras. Estos protocolos garantizan la correcta transmisión de datos y establecen reglas claras para la comunicación entre dispositivos. En la siguiente Figura podemos observar un diagrama de cómo funciona este tipo de modelo.

De igual manera, uno de los elementos clave en la implementación de sistemas Cliente/Servidor son los sockets, que actúan como puntos de conexión entre procesos en una red. En términos simples, un socket es una interfaz de comunicación que permite enviar y recibir datos, ya sea en una misma máquina o en dispositivos remotos. Existen dos tipos principales: los sockets de flujo, que utilizan TCP y garantizan una comunicación confiable y ordenada, y los sockets de datagrama, basados en UDP, que son más rápidos, pero no garantizan la entrega de datos. Para esta práctica, se emplearán sockets de flujo debido a su fiabilidad y control en la transmisión de información.

A pesar de sus ventajas, el modelo Cliente/Servidor enfrenta desafíos importantes. Uno de ellos es la gestión de concurrencia, ya que un servidor puede recibir múltiples solicitudes simultáneamente. Para manejar esta carga, se emplean mecanismos como hilos, procesos concurrentes o balanceo de carga. Otro problema es la dependencia del servidor, pues si este falla sin un sistema de respaldo, los clientes quedan inoperantes. Para evitarlo, se implementan estrategias como clustering y réplicas de servidores, e incluso la caída del servidor, ya que debido a la disposición centralizada y a la dependencia en un modelo cliente-servidor, la caída del servidor conlleva la caída de todo el sistema. Si el servidor se cae, los clientes dejan de funcionar porque no pueden recibir las respuestas necesarias del servidor.

Con el avance de la tecnología, esta arquitectura ha evolucionado hacia modelos más dinámicos, como los microservicios, que dividen las aplicaciones en componentes independientes, y la computación en la nube, que permite escalabilidad global y alta disponibilidad. Estas innovaciones han optimizado el rendimiento y la confiabilidad de los sistemas basados en Cliente/Servidor.

En general, esta arquitectura sigue siendo un pilar fundamental en el desarrollo de sistemas informáticos. Su flexibilidad y eficiencia la han convertido en la base de múltiples aplicaciones y servicios.

Modelo multicliente-multiservidor

El modelo multicliente/multiservidor es una extensión del tradicional modelo cliente-servidor, en el cual múltiples clientes pueden interactuar simultáneamente con múltiples servidores. Este modelo se ha vuelto fundamental en el diseño de sistemas distribuidos, ya que permite mejorar la escalabilidad, disponibilidad y eficiencia en el manejo de solicitudes. Su uso es común en aplicaciones web, bases de datos distribuidas, computación en la nube y sistemas empresariales que requieren alta concurrencia.

A diferencia del modelo cliente-servidor tradicional, donde un único servidor centraliza la gestión de datos y servicios, el modelo multicliente/multiservidor distribuye la carga de trabajo en varios servidores. Esto permite que los clientes puedan conectarse a diferentes servidores según la disponibilidad, optimizando los tiempos de respuesta y evitando la sobrecarga en un solo punto.

Además, la redundancia de servidores incrementa la tolerancia a fallos, ya que si un servidor deja de funcionar, los clientes pueden ser redirigidos a otro servidor sin que el servicio se vea interrumpido.

Uno de los principales retos de este modelo es la sincronización de datos entre los servidores. Si varios clientes realizan modificaciones simultáneamente en diferentes servidores, es crucial garantizar que los cambios sean consistentes en todo el sistema. Para abordar este problema, se emplean diversas estrategias de sincronización, como replicación de datos, control de concurrencia y algoritmos de consenso, los cuales garantizan la coherencia del sistema distribuido.

La implementación de este modelo puede realizarse mediante diferentes enfoques. Un método común es el balanceo de carga, donde un componente intermedio, como un **balanceador de carga**, distribuye las solicitudes de los clientes entre los servidores de manera equitativa o según su disponibilidad. Otra estrategia es la segmentación de datos, en la cual cada servidor es responsable de un subconjunto específico de información, reduciendo la redundancia y optimizando el acceso. En aplicaciones que requieren alta disponibilidad, se puede emplear la replicación, donde los servidores mantienen copias actualizadas de los datos para garantizar que la información esté disponible en todo momento.

Si bien este modelo ofrece muchas ventajas en términos de escalabilidad y tolerancia a fallos, su implementación conlleva desafíos adicionales en términos de sincronización y coordinación entre los servidores. La correcta elección de los métodos de sincronización y control de concurrencia es clave para evitar problemas como la inconsistencia de datos, la sobrecarga de comunicación y los conflictos de acceso.

Métodos de sincronización en Sistemas Multicliente/Multiservidor

La sincronización en sistemas multicliente/multiservidor es un aspecto crítico para garantizar la coherencia y disponibilidad de los datos en entornos donde múltiples clientes pueden acceder y modificar información simultáneamente. Existen diversos enfoques para lograr esta sincronización, los cuales pueden clasificarse en métodos basados en bloqueos, replicación de datos y estrategias optimistas de concurrencia.

Uno de los métodos más tradicionales es el **bloqueo pesimista**, en el cual un recurso queda bloqueado cuando un cliente lo está modificando, impidiendo que otros accedan a él hasta que la operación haya finalizado. Este método es efectivo para evitar conflictos, pero puede generar cuellos de botella, ya que otros clientes deben esperar a que se libere el recurso antes de continuar. Por otro lado, el **bloqueo optimista** permite que múltiples clientes accedan simultáneamente a los datos, pero antes de realizar cambios se verifica si ha ocurrido una modificación concurrente. Si se detecta un conflicto, el sistema puede optar por descartar la transacción o solicitar una resolución manual.

Otro enfoque ampliamente utilizado es la **replicación de datos**, donde la información se mantiene en múltiples servidores para mejorar la disponibilidad y tolerancia a fallos. Dependiendo del método de replicación, la sincronización puede ser síncrona o asíncrona. En la **replicación síncrona**, cada modificación realizada en un servidor se propaga inmediatamente a los demás, asegurando consistencia en tiempo real, pero con el costo de mayor latencia. En la **replicación asíncrona**, los cambios se propagan con cierto retraso, lo que mejora el rendimiento, pero puede ocasionar inconsistencias temporales en los datos.

Para mitigar estos problemas, algunos sistemas implementan **algoritmos de control de concurrencia distribuida**, como el **relogeo de registros (Write-Ahead Logging, WAL)**, en el que todas las operaciones se registran en un log antes de aplicarse a la base de datos. Esto permite recuperar el estado anterior en caso de fallos y facilita la sincronización de datos en entornos distribuidos.

Un método alternativo es el uso de **tokens de acceso**, en el cual un cliente solo puede modificar datos si posee un token especial. Este mecanismo evita conflictos de escritura simultánea sin necesidad de bloqueos complejos. Sin embargo, la gestión de tokens puede volverse costosa en sistemas altamente dinámicos.

La elección del método de sincronización depende del tipo de aplicación y los requerimientos de coherencia de datos. En sistemas financieros o de bases de datos críticas, se prefieren enfoques estrictos como la replicación síncrona o los bloqueos pesimistas. En cambio, en aplicaciones como redes sociales o juegos en línea, donde la disponibilidad y velocidad son más importantes que la consistencia absoluta, se suelen emplear métodos optimistas o replicación asíncrona.

Algoritmo de consenso

Los algoritmos de consenso son fundamentales en sistemas distribuidos, ya que permiten que múltiples nodos acuerden un estado único y coherente de los datos sin la necesidad de un servidor centralizado. Su importancia radica en la necesidad de garantizar consistencia en entornos donde múltiples servidores pueden recibir actualizaciones simultáneamente y deben asegurarse de que todos los nodos reflejen la misma información.

Uno de los algoritmos más utilizados es **Paxos**, desarrollado por Leslie Lamport, el cual garantiza consenso en sistemas distribuidos asumiendo que una mayoría de los nodos está operativa. Paxos funciona mediante un proceso de proposición y aceptación en el que los nodos acuerdan un valor sin depender de una única entidad centralizada. Su principal ventaja es su tolerancia a fallos, ya que sigue funcionando incluso si algunos nodos fallan o se desconectan temporalmente.

Otra alternativa popular es **Raft**, diseñado para ser más comprensible y fácil de implementar que Paxos. Raft divide el proceso de consenso en tres fases: **elección de líder**, **replicación de log** y **compromiso de entradas**. En este modelo, un nodo actúa como líder y coordina la replicación de datos entre los demás nodos seguidores. Raft se utiliza en sistemas como bases de datos distribuidas y almacenamiento en la nube debido a su simplicidad y eficiencia en la toma de decisiones.

Además de estos algoritmos, existen enfoques como **Two-Phase Commit (2PC)** y **Three-Phase Commit (3PC)**, diseñados para garantizar la consistencia en transacciones distribuidas. En **2PC**, una transacción se divide en dos fases: preparación y confirmación. Durante la fase de preparación, todos los nodos deben aceptar la transacción antes de que pueda ser comprometida. Sin embargo, 2PC tiene la desventaja de que, si un nodo falla en el momento equivocado, el sistema puede quedar bloqueado. **3PC** introduce una fase adicional para evitar bloqueos, permitiendo que los nodos recuperen el estado anterior en caso de fallos intermedios.

Otro método es el **gossip protocol**, el cual distribuye información entre nodos de manera similar a la propagación de rumores. Cada nodo comparte su estado con un subconjunto aleatorio de otros nodos, asegurando eventualmente que toda la red alcance un consenso. Este protocolo es útil en redes descentralizadas y sistemas P2P donde la latencia y la tolerancia a fallos son más importantes que la consistencia inmediata.

La elección del algoritmo de consenso adecuado depende del tipo de sistema distribuido. Paxos y Raft son ideales para bases de datos y almacenamiento distribuido, mientras que los protocolos gossip y commit son útiles en redes descentralizadas y aplicaciones con alta tolerancia a fallos. Estos algoritmos forman la base de muchos sistemas modernos y garantizan la estabilidad y confiabilidad de la infraestructura digital.

Sistemas centralizados

Los sistemas centralizados se caracterizan por la presencia de un único servidor o entidad central que gestiona todos los recursos, la comunicación y la toma de decisiones dentro del sistema. En este modelo, todos los clientes dependen de un servidor principal para acceder a datos, ejecutar procesos o coordinar transacciones. Es una arquitectura ampliamente utilizada en aplicaciones tradicionales, como bases de datos monolíticas, servidores de autenticación y redes empresariales cerradas.

Una de las principales ventajas de los sistemas centralizados es su simplicidad en términos de administración y control. Como existe un único punto de gestión, las actualizaciones, configuraciones y políticas de seguridad pueden implementarse de manera uniforme. Además, la coherencia de los datos es más fácil de garantizar, ya que no se requiere replicación ni sincronización entre múltiples nodos.

Sin embargo, esta arquitectura presenta importantes limitaciones. La más evidente es el **punto único de falla**: si el servidor central experimenta problemas técnicos o se sobrecarga debido a un alto número de solicitudes, todo el sistema puede colapsar. Además, la escalabilidad es limitada, ya que a medida que aumenta el número de clientes, el servidor puede volverse un cuello de botella, afectando el rendimiento y los tiempos de respuesta.

Para mitigar estos problemas, algunas soluciones centralizadas incorporan técnicas como la replicación de servidores o la segmentación de datos. Sin embargo, estas soluciones pueden volverse costosas y difíciles de administrar. Como resultado, muchos sistemas han migrado a arquitecturas más flexibles, como los sistemas distribuidos, para mejorar la escalabilidad y la tolerancia a fallos.

Sistemas distribuidos

Los sistemas distribuidos son aquellos en los que múltiples nodos (servidores o computadoras) trabajan conjuntamente para proporcionar un servicio o resolver un problema de manera coordinada. A diferencia de los sistemas centralizados, en los sistemas distribuidos no existe una única entidad de control, sino que las tareas y los datos se distribuyen entre varios nodos que colaboran para garantizar la disponibilidad y el rendimiento del sistema.

Este tipo de sistemas es ampliamente utilizado en aplicaciones modernas, como servicios en la nube, bases de datos distribuidas, redes peer-to-peer (P2P) y blockchain. Su principal ventaja radica en la **tolerancia a fallos**, ya que, al distribuir la carga de trabajo entre múltiples nodos, el fallo de uno de ellos no compromete el funcionamiento del sistema completo. Además, los sistemas distribuidos ofrecen **alta escalabilidad**, permitiendo agregar nuevos nodos sin afectar significativamente el rendimiento general.

Sin embargo, la implementación de un sistema distribuido presenta múltiples desafíos. Uno de los más importantes es la **coherencia de datos**, ya que, al existir múltiples copias de la información en diferentes nodos, es necesario asegurarse de que todas las versiones sean consistentes. Para ello, se utilizan algoritmos de sincronización y consenso, como **Paxos**, **Raft** o **Two-Phase Commit (2PC)**, que garantizan que todas las actualizaciones sean aplicadas de manera uniforme en toda la red.

Otro desafío es la **latencia en la comunicación**. En sistemas centralizados, la comunicación es directa entre el cliente y el servidor, mientras que, en un sistema distribuido, las solicitudes pueden atravesar múltiples nodos antes de completarse. Para mitigar este problema, se emplean estrategias como **almacenamiento en caché**, **replicación de datos** y **balanceo de carga**, que optimizan la distribución de las solicitudes y reducen la latencia.

Los sistemas distribuidos pueden clasificarse en diferentes arquitecturas, como:

- **Sistemas distribuidos homogéneos:** todos los nodos tienen la misma función y capacidades, lo que facilita la administración del sistema.
- **Sistemas distribuidos heterogéneos:** los nodos pueden tener diferentes funciones y capacidades, lo que permite una mayor flexibilidad, pero también requiere mecanismos avanzados de coordinación.
- **Sistemas descentralizados:** no existe una jerarquía clara entre los nodos, como ocurre en redes peer-to-peer o blockchain.

Gracias a su flexibilidad y robustez, los sistemas distribuidos han reemplazado en muchos casos a los sistemas centralizados, especialmente en aplicaciones que requieren alta disponibilidad, grandes volúmenes de datos y procesamiento paralelo.

Sistemas basados en token

Los sistemas basados en token son una categoría especial de sistemas distribuidos que utilizan **tokens de control** como mecanismo para coordinar el acceso a recursos compartidos. En estos sistemas, un token es un objeto digital que otorga permisos a un nodo o usuario para realizar ciertas acciones, como acceder a datos, modificar registros o ejecutar procesos críticos.

Uno de los usos más comunes de los sistemas basados en token es el **control de concurrencia en entornos distribuidos**. En este contexto, el token actúa como un permiso exclusivo que debe poseer un nodo antes de poder realizar una operación en un recurso compartido. Esto permite evitar conflictos de acceso simultáneo sin necesidad de emplear bloqueos pesados o coordinaciones complejas entre nodos.

Existen varias estrategias para la gestión de tokens en sistemas distribuidos:

- **Token único (Token Ring):** Se utiliza un solo token que circula entre los nodos siguiendo un orden predefinido. Solo el nodo que posee el token puede ejecutar ciertas operaciones. Este método garantiza exclusión mutua y evita bloqueos, pero puede generar tiempos de espera prolongados si el número de nodos es alto.
- **Múltiples tokens (Distributed Token-Based Systems):** Se distribuyen varios tokens entre los nodos para permitir concurrencia en operaciones distintas. Este enfoque mejora el rendimiento, pero requiere mecanismos adicionales para prevenir la duplicación o pérdida de tokens.
- **Tokens en redes descentralizadas (Blockchain y Criptografía):** En sistemas como blockchain, los tokens representan activos digitales y se utilizan para validar transacciones y garantizar la seguridad de la red sin necesidad de una autoridad centralizada.

Un ejemplo práctico de sistemas basados en token es el **algoritmo de exclusión mutua de Ricart y Agrawala**, que utiliza mensajes de solicitud y respuesta para determinar cuál nodo tiene prioridad en el acceso a un recurso. Otro enfoque es el **algoritmo de Suzuki-Kasami**, donde un nodo debe solicitar un token especial antes de acceder a una sección crítica, garantizando que solo un nodo a la vez pueda modificar el recurso compartido.

El uso de tokens también se extiende a áreas como la seguridad informática, donde se emplean tokens de autenticación para verificar la identidad de los usuarios en sistemas distribuidos. Estos tokens

pueden ser estáticos (como claves API) o dinámicos (como los tokens de acceso en OAuth), proporcionando un nivel adicional de seguridad y control en la comunicación entre clientes y servidores.

En comparación con otros métodos de sincronización, los sistemas basados en token ofrecen varias ventajas, como la reducción de la latencia en la coordinación y la eliminación de posibles bloqueos por espera mutua. No obstante, presentan desafíos como la posible pérdida del token, lo que puede requerir mecanismos adicionales para su regeneración o redistribución en caso de fallos.

Los sistemas centralizados, distribuidos y basados en token representan diferentes enfoques para el diseño y gestión de arquitecturas de software y redes. Mientras que los sistemas centralizados destacan por su simplicidad y control, los sistemas distribuidos ofrecen escalabilidad y tolerancia a fallos. Por otro lado, los sistemas basados en token proporcionan soluciones eficientes para la sincronización y control de concurrencia en entornos distribuidos. La elección entre estas arquitecturas depende de los requerimientos específicos de cada aplicación, como la necesidad de disponibilidad, rendimiento y seguridad en la gestión de datos y procesos.

Planteamiento del problema

El objetivo de esta práctica es desarrollar un sistema distribuido para la gestión de una panadería, permitiendo que múltiples clientes interactúen con diferentes servidores de manera simultánea sin generar inconsistencias en el inventario. Para ello, se implementará un modelo de arquitectura multicliente-multiservidor en el que cada servidor maneja operaciones de compra y actualización del stock, sincronizándose con una base de datos compartida para mantener la coherencia de la información.

- El sistema cuenta con múltiples servidores de panadería, cada uno ejecutándose en un puerto diferente y gestionando conexiones de clientes.
- Los clientes pueden conectarse a cualquier servidor disponible a través de un servidor de descubrimiento, el cual registra y asigna dinámicamente las conexiones.
- El inventario es administrado de manera centralizada mediante una base de datos compartida, lo que permite a los servidores reflejar en tiempo real cualquier cambio realizado en las existencias de los productos.
- Se deben implementar mecanismos de control de concurrencia para evitar problemas de sobreventa y garantizar la integridad de los datos.

Dado que el sistema opera en un entorno distribuido, es fundamental gestionar correctamente la sincronización del inventario y la concurrencia en las transacciones. Para ello, cada servidor deberá asegurar que los datos reflejados en la base de datos sean consistentes, evitando escenarios en los que dos clientes realicen compras simultáneamente sobre un mismo producto sin que se actualice correctamente el stock. Además, es necesario garantizar la tolerancia a fallos, de modo que si un servidor deja de estar disponible, los clientes puedan redirigir sus solicitudes a otro servidor activo sin interrumpir su operación.

Propuesta de solución

Para abordar el problema de la gestión distribuida del inventario en la panadería, se propone el desarrollo de un sistema multicliente-multiservidor basado en una arquitectura distribuida con

sincronización de datos a través de una base de datos compartida. Este sistema garantizará la consistencia de la información en tiempo real, permitiendo que múltiples clientes realicen compras sin generar conflictos en el stock.

La solución se estructura en los siguientes componentes clave:

1. **Servidor de descubrimiento:** Será el punto central encargado de registrar y gestionar las conexiones de los servidores de la panadería. Este componente permitirá a los clientes obtener la dirección de un servidor activo para conectarse y realizar sus operaciones.
2. **Servidores de panadería:** Cada servidor funcionará como una instancia independiente que atenderá múltiples clientes de forma concurrente. Estos servidores estarán sincronizados con la base de datos compartida para reflejar los cambios en el inventario de manera consistente y evitar discrepancias entre diferentes instancias.
3. **Base de datos compartida:** Todos los servidores accederán a una misma base de datos MySQL, donde se almacenará el inventario de productos. Se implementarán mecanismos de control de concurrencia para evitar problemas de acceso simultáneo que puedan generar errores en la actualización de las existencias.
4. **Manejo de sesiones persistentes:** Para evitar la pérdida del historial de compras de los clientes, se implementará un sistema de sesiones que les permitirá mantener su estado incluso si cambian de servidor. Esto se logrará asignando un identificador único a cada cliente y almacenando su información en la base de datos.
5. **Control de concurrencia y sincronización:** Dado que múltiples clientes pueden realizar compras al mismo tiempo, se aplicarán técnicas como el bloqueo de registros en la base de datos o el uso de transacciones ACID para asegurar que las operaciones se realicen de forma segura y sin inconsistencias.

Con esta solución, se busca desarrollar un sistema escalable y confiable que permita a los clientes interactuar de manera fluida con la panadería, manteniendo la coherencia de los datos en todo momento. La implementación de una arquitectura distribuida con sincronización en la base de datos garantizará que el inventario se mantenga actualizado, incluso cuando varias transacciones se realicen simultáneamente desde distintos servidores.

Materiales y métodos empleados

Para llevar a cabo la práctica y desarrollar la simulación de la panadería con el modelo Multicliente/multiservidor, se utilizaron las siguientes herramientas y metodologías:

Materiales (Herramientas utilizadas)

1. **Lenguaje de programación: Java**
 - Se eligió Java debido a su robusto manejo de hilos y sus herramientas integradas para la concurrencia, como la palabra clave `synchronized`.
2. **Entorno de desarrollo: Visual Studio Code (VS Code)**
3. **JDK (Java Development Kit):**
 - Se usó el JDK para compilar y ejecutar el programa.

4. Sistema Operativo: Windows

- La práctica se desarrolló y ejecutó en un sistema operativo Windows.

Métodos Empleados

Para la implementación del sistema multicliente-multiservidor con sincronización de inventario, se emplearon diversos métodos y técnicas que garantizaron la correcta comunicación, concurrencia y persistencia de datos en el entorno distribuido. A continuación, se describen los principales métodos utilizados:

1. **Arquitectura MultiCliente-MultiServidor Distribuida:** Se diseñó una arquitectura en la que múltiples servidores pueden atender a varios clientes simultáneamente. Los clientes se conectan a un servidor disponible a través del servidor de descubrimiento, permitiendo una distribución eficiente de la carga y evitando puntos únicos de fallo.
2. **Gestión de Concurrencia en Base de Datos:** Para garantizar la integridad del inventario, se implementaron mecanismos de concurrencia en MySQL, utilizando transacciones y bloqueo de registros. Esto evita inconsistencias en los datos cuando múltiples clientes realizan operaciones sobre el mismo producto de manera simultánea.
3. **Persistencia de Datos:** Se utilizó MySQL como sistema de gestión de bases de datos (DBMS) para almacenar la información del inventario y las sesiones de los clientes. Se aplicaron técnicas de normalización para optimizar el almacenamiento y evitar redundancia de datos.
4. **Sincronización de Datos entre Servidores:** Como cada servidor puede recibir y procesar solicitudes de manera independiente, la sincronización con la base de datos se realizó mediante consultas atómicas y confirmaciones de transacciones (commit) para garantizar que todos los servidores trabajen con información actualizada en tiempo real.
5. **Uso de Hilos para el Manejo de Clientes Concurrentes:** Cada servidor maneja múltiples clientes simultáneamente mediante la implementación de hilos (Threads). Cada solicitud de un cliente es atendida en un hilo separado, lo que permite la ejecución concurrente de múltiples operaciones sin afectar el rendimiento del sistema.
6. **Servidor de Descubrimiento para Balanceo de Carga:** Para distribuir equitativamente las conexiones de los clientes entre los servidores de panadería, se utilizó un servidor de descubrimiento que asigna dinámicamente un servidor disponible. Esto mejora la escalabilidad del sistema y reduce la sobrecarga en un solo nodo.
7. **Modelo de Comunicación Basado en Sockets:** La comunicación entre clientes y servidores se estableció mediante sockets en Java, permitiendo la transmisión eficiente de datos en una red. Se definieron protocolos específicos para el intercambio de información, asegurando la correcta interpretación de las solicitudes y respuestas.

Estos métodos garantizaron el correcto funcionamiento del sistema distribuido, asegurando la coherencia del inventario, la concurrencia de múltiples clientes y la persistencia de los datos en la base de datos.

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. Servidor de Descubrimiento (Registro de Servidores Activos)

El sistema cuenta con un **servidor de descubrimiento**, el cual tiene la función de mantener un registro de los servidores disponibles y redirigir a los clientes hacia ellos. Al iniciarse, cada servidor de panadería se registra en el servidor de descubrimiento, indicando que está listo para recibir conexiones.

Cuando un cliente desea conectarse, primero consulta al servidor de descubrimiento, el cual le asigna un servidor activo disponible, garantizando así un balanceo de carga y evitando sobrecargas en un solo servidor.

Entonces, el Servidor de Descubrimiento es un componente fundamental del sistema distribuido, ya que mantiene un registro actualizado de los servidores de panadería disponibles. Su función principal es gestionar las solicitudes de registro y consulta de servidores, permitiendo a los clientes conectarse a una instancia de servidor activa.

Para su implementación, se utiliza un **ServerSocket** que escucha en un puerto específico (5050) y gestiona las conexiones tanto de servidores de panadería como de clientes. En la siguiente figura, se muestra el bloque de código donde se inicia el servidor y se aceptan conexiones entrantes.

```
import java.io.*;
import java.net.*;
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;

public class ServidorDescubrimiento {
    private static final int PUERTO_DESCUBRIMIENTO = 5050;
    private static final Map<String, Integer> servidoresPanaderia = new ConcurrentHashMap<>(); // Mapa concurrente para manejo de múltiples hilos

    Run|Debug
    public static void main(String[] args) {
        // Iniciar el servidor de descubrimiento
        try (ServerSocket serverSocket = new ServerSocket(PUERTO_DESCUBRIMIENTO)) {
            System.out.println("Servidor de descubrimiento iniciado en el puerto " + PUERTO_DESCUBRIMIENTO);

            while (true) {
                // Aceptar las conexiones de los servidores de panadería
                Socket socketCliente = serverSocket.accept();
                new Thread(() -> manejarCliente(socketCliente)).start(); // Manejo concurrente de clientes
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Figura 2. Código de inicialización del Servidor de Descubrimiento.

Cada vez que un servidor de panadería se conecta, se registra enviando su número de puerto al servidor de descubrimiento. Este dato es almacenado en una estructura de tipo **ConcurrentHashMap**, lo que permite un acceso seguro en entornos multihilo.

Y cuando un cliente desea conectarse a un servidor de panadería, envía una solicitud con el mensaje "SOLICITUD_PUERTO". El servidor de descubrimiento responde con la lista de puertos de los servidores registrados, permitiendo que el cliente seleccione una opción. El código que implementa esta funcionalidad se muestra en la siguiente figura.

```
// Maneja la conexión de un servidor de panadería
private static void manejarCliente(Socket socket) {
    try (BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        PrintWriter out = new PrintWriter(socket.getOutputStream(), autoFlush:true)) {

        // Recibir el mensaje del cliente
        String mensaje = in.readLine();

        // Dentro de manejarCliente(), en el caso de solicitud de puerto:
        if ("SOLICITUD_PUERTO".equals(mensaje)) {
            if (!servidoresPanaderia.isEmpty()) {
                // Obtener todos los puertos registrados como una lista separada por comas
                String puertos = String.join(delimiter:", ", servidoresPanaderia.values().stream().map(String::valueOf).toArray(String[]::new));
                out.println(puertos); // Enviar la lista de puertos al cliente
                System.out.println("Se ha enviado la lista de puertos de panadería: " + puertos);
            } else {
                out.println(x:"No hay servidores de panadería disponibles.");
            }
        } else {
            // Si el mensaje no es una solicitud de puerto, manejarlo como un registro de servidor
            int puerto = Integer.parseInt(mensaje); // Convertir el mensaje a puerto
            System.out.println("Servidor de panadería registrado en el puerto: " + puerto);

            // Registrar el servidor de panadería en el mapa (servidores disponibles)
            servidoresPanaderia.put("panaderia" + puerto, puerto);

            // Responder al servidor de panadería que se registró correctamente
            out.println("Servidor registrado correctamente en el puerto: " + puerto);

            // (Opcional) Imprimir todos los servidores registrados para verificación
            System.out.println("Servidores de panadería registrados: " + servidoresPanaderia);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Figura 3. Código de manejo de solicitudes de clientes en el Servidor

Para mantener la integridad del sistema, cuando un servidor de panadería se desconecta, se debe eliminar de la lista de servidores activos. En la siguiente figura, se muestra el código que permite realizar esta acción.

```
// Método para registrar un servidor (llamado desde el servidor de panadería)
public static void registrarServidor(int puerto) {
    try (Socket socket = new Socket(host:"192.168.100.12", PUERTO_DESCUBRIMIENTO);
        PrintWriter out = new PrintWriter(socket.getOutputStream(), autoFlush:true)) {
        // Enviar el puerto del servidor de panadería al servidor de descubrimiento
        out.println(puerto); // Enviar el puerto como mensaje
        System.out.println("Servidor de panadería con puerto " + puerto + " se ha registrado.");
    } catch (IOException e) {
        e.printStackTrace();
    }
}

// Método para desregistrar un servidor (llamado desde el servidor de panadería)
public static void desregistrarServidor(int puerto) {
    try (Socket socket = new Socket(host:"192.168.100.12", PUERTO_DESCUBRIMIENTO);
        PrintWriter out = new PrintWriter(socket.getOutputStream(), autoFlush:true)) {
        // Desregistrar el servidor de panadería
        servidoresPanaderia.remove("panaderia" + puerto);
        System.out.println("Servidor de panadería desregistrado en el puerto: " + puerto);
        out.println("Desregistro exitoso para el servidor en el puerto: " + puerto);
    } catch (IOException e) {
        e.printStackTrace();
    }
}

// Método para obtener los servidores registrados (opcional)
public static Map<String, Integer> obtenerServidoresRegistrados() {
    return new HashMap<>(servidoresPanaderia); // Retorna una copia del mapa para evitar cambios directos
}
```

Figura 4. Código de registro y eliminación de servidores desconectados de Descubrimiento.

Gracias a este mecanismo, los clientes pueden distribuirse entre los servidores de panadería disponibles, asegurando una correcta sincronización del inventario y evitando que intenten conectarse a servidores inactivos.

2. Servidor de Panadería (Atención a Clientes y Gestión del Inventario)

Cada servidor de panadería opera como un nodo independiente dentro del sistema distribuido. Su función es recibir y procesar solicitudes de los clientes, permitiéndoles consultar el stock de productos, realizar compras y mantener la sesión activa sin perder información.

El servidor de panadería se conecta directamente a la base de datos MySQL para leer y actualizar la información del inventario. Además, maneja múltiples clientes simultáneamente utilizando hilos (Threads), lo que permite que varias personas realicen transacciones sin bloqueos innecesarios.

Cuando un cliente realiza una compra, el servidor valida la disponibilidad del producto y, si la compra es exitosa, actualiza la base de datos para reflejar el nuevo estado del inventario.

En general, el Servidor de Panadería es el componente encargado de atender las solicitudes de los clientes y gestionar el inventario de productos, en este caso, el pan. Su funcionamiento incluye la gestión de conexiones con los clientes, la visualización del stock disponible, y la compra de productos.

El servidor escucha en un puerto específico, maneja múltiples clientes de manera concurrente utilizando un **ThreadPoolExecutor** y realiza operaciones sobre la base de datos para consultar y modificar el stock. Además, existe un mecanismo que simula el proceso de horneado de pan, aumentando el stock de manera periódica. En la siguiente figura, se muestra el bloque de código que inicializa el servidor y gestiona la atención a los clientes, además se muestra cómo el servidor se registra y desregistra en el servidor de descubrimiento al iniciar y detenerse, respectivamente. Este

proceso asegura que el servidor de panadería esté disponible para los clientes durante su funcionamiento.

```
public class PanaderiaServidor {
    public static void iniciarServidor(int puerto) {
        ThreadPoolExecutor pool = (ThreadPoolExecutor) Executors.newFixedThreadPool(nThreads:5);

        // Registrar el servidor en el servidor de descubrimiento antes de iniciar el servicio
        ServidorDescubrimiento.registrarServidor(puerto);

        new HornoPanadero().start(); // Hilo para hornear pan

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

            while (true) {
                Socket socketCliente = serverSocket.accept();
                pool.execute(new HiloCliente(socketCliente));
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            // Desregistrar el servidor en el servidor de descubrimiento cuando se detiene
            ServidorDescubrimiento.desregistrarServidor(puerto);
        }
    }
}

Run | Debug
public static void main(String[] args) {
    // Verificar que se pasen puertos como argumentos
    if (args.length == 0) {
        System.err.println(x:"Debe proporcionar al menos un puerto.");
        return;
    }

    // Iterar sobre los puertos pasados y crear un hilo para cada uno
    for (String puertoStr : args) {
        try {
            int puerto = Integer.parseInt(puertoStr); // Convertir cada argumento a entero
            // Iniciar un hilo para cada puerto
            new Thread(() -> iniciarServidor(puerto)).start();
        } catch (NumberFormatException e) {
            System.err.println("Error: El puerto '" + puertoStr + "' no es válido.");
        }
    }
}
```

Figura 5. Código de inicialización del Servidor de Panadería y atención a clientes.

Cuando un cliente se conecta, el servidor lo identifica y le presenta un menú con varias opciones. Entre ellas, el cliente puede consultar el stock disponible, realizar compras de pan o salir de la sesión. El siguiente bloque de código muestra cómo se gestiona la comunicación con los clientes y la interacción con la base de datos para verificar el stock y procesar las compras.


```

class HiloCliente extends Thread {
    private final Socket socket;

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

    @Override
    public void run() {
        try {
            BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            PrintWriter out = new PrintWriter(socket.getOutputStream(), autoFlush:true)
        } {
            // Leer el nombre del cliente y dar la bienvenida
            String nombreCliente = in.readLine();
            String ipCliente = socket.getInetAddress().getHostAddress();
            System.out.println(nombreCliente + " se ha conectado desde la IP: " + ipCliente + " en el puerto: " + socket.getPort());
            out.println("Bienvenido a la Panadería, " + nombreCliente + "!");

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

                String opcionStr = in.readLine();
                if (opcionStr == null) break;

                int opcion;
                try {
                    opcion = Integer.parseInt(opcionStr);
                } catch (NumberFormatException e) {
                    out.println(x:"Opción no válida. Intente de nuevo.");
                    continue;
                }

                if (opcion == 1) {
                    out.println("Stock disponible: " + InventarioDB.obtenerStock() + " panes.");
                } else if (opcion == 2) {
                    out.println(x:"Ingrese la cantidad de pan que desea comprar:");
                    String cantidadStr = in.readLine();
                    int cantidad;
                    try {
                        cantidad = Integer.parseInt(cantidadStr);
                    } catch (NumberFormatException e) {
                        out.println(x:"Cantidad no válida. Intente de nuevo.");
                        continue;
                    }
                }
            }
        }
    }
}

```

Figura 5. Código de interacción con los clientes y gestión de stock en el Servidor de Panadería.

```

        boolean compraExitosa = InventarioDB.comprarPan(cantidad);
        if (compraExitosa) {
            out.println("Compra exitosa. Panes restantes: " + InventarioDB.obtenerStock());
            System.out.println("Cliente " + nombreCliente + " compró " + cantidad + " panes. Stock restante: " + InventarioDB.obtenerStock());
        } else {
            out.println(x:"No hay suficiente pan. Por favor, espere mientras hornecemos más.");
        }

        } else if (opcion == 3) {
            out.println(x:"Gracias por visitar la panadería. ¡Hasta luego!");
            System.out.println(nombreCliente + " se ha desconectado.");
            break;
        } else {
            out.println(x:"Opción no válida. Intente de nuevo.");
        }
    }

    catch (IOException e) {
        System.err.println("Error en la conexión con el cliente: " + e.getMessage());
    } finally {
        try {
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

Figura 6. Código de interacción con los clientes y gestión de stock en el Servidor de Panadería.

Para realizar una compra, el servidor consulta el stock disponible y, si hay suficiente cantidad de pan, actualiza el inventario en la base de datos. Si no hay suficiente stock, se notifica al cliente y se le

informa que debe esperar a que se horneen más panes. Este mecanismo está implementado en el siguiente bloque de código.

```
public static synchronized boolean comprarPan(int cantidad) {
    try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD)) {
        conn.setAutoCommit(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();
                    return true;
                }
            }
        }
        conn.rollback();
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return false;
}
```

Figura 7. Código de compra de pan y actualización de inventario.

El proceso de horneado de pan está implementado en un hilo independiente que se ejecuta continuamente, añadiendo más panes al inventario cada cierto tiempo. A continuación, se presenta el código que simula el horneado de pan.

```
public static void añadirStock(int cantidad) {
    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();
        System.out.println("Se hornearon " + cantidad + " panes. Nuevo stock: " + obtenerStock());
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

class HornoPanadero extends Thread {
    @Override
    public void run() {
        while (true) {
            try {
                Thread.sleep(millis:10000); // Cada 10 segundos se hornean más panes
                InventarioDB.añadirStock(cantidad:5);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Figura 8. Código de simulación de horneado de pan.

Por último, en la siguiente figura, se muestra como se realiza la conexión con la base de datos y se obtiene el stock actual de panes en la panadería.

```

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

class InventarioDB {
    static {
        try {
            Class.forName(className:"com.mysql.cj.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

    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";

    public static int obtenerStock() {
        int stock = 0;
        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 (SQLException e) {
            e.printStackTrace();
        }
        return stock;
    }
}

```

Figura 9. Conexión a la base de datos y obtener stock de pan.

Gracias a este enfoque, el servidor de panadería puede manejar múltiples clientes de manera eficiente, gestionando el inventario y asegurando que los clientes puedan realizar compras de manera fluida.

3. Base de Datos Centralizada (Persistencia y Sincronización de Inventario)

Para evitar inconsistencias en el inventario, todos los servidores de panadería están conectados a una base de datos MySQL centralizada. Esta base de datos actúa como el punto de sincronización entre los servidores, garantizando que cada transacción se registre y refleje en todos los nodos del sistema.

La base de datos centralizada es un componente esencial del sistema de panadería, ya que permite la persistencia del inventario de productos (en este caso, el pan) y la sincronización de la información entre los diferentes servidores. A continuación, se describe el diseño de la base de datos, las consultas SQL utilizadas, y cómo se establece la conexión utilizando el conector JDBC.

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 10. 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.

Consultas SQL

Las consultas SQL son fundamentales para obtener y actualizar el inventario. A continuación, se detallan las consultas principales utilizadas para consultar el stock y realizar compras de pan.

Consulta para obtener el stock disponible:

```
ResultSet rs = stmt.executeQuery(sql:"SELECT stock FROM inventario WHERE producto='Pan'");
```

Esta consulta se ejecuta cada vez que un cliente solicita ver el stock disponible. El valor devuelto es el número de panes disponibles en el inventario.

Consulta para comprar pan:

```
sql:"UPDATE inventario SET stock = stock - ? WHERE producto='Pan'");
```

Esta consulta se utiliza cuando un cliente decide comprar pan. La cantidad solicitada se resta del stock disponible, pero solo si hay suficiente cantidad en el inventario. La transacción es manejada de manera **transaccional** para asegurar que el stock no sea modificado simultáneamente por diferentes clientes.

Consulta para añadir stock (simulando el proceso de horneado de pan):

```
PreparedStatement pstmt = conn.prepareStatement(sql:"UPDATE inventario SET stock = stock + ? WHERE producto='Pan'");
```

Esta consulta se ejecuta periódicamente en un hilo que simula el horneado de pan, agregando más unidades al inventario.

Conector JDBC

El conector **JDBC** (Java Database Connectivity) es utilizado para establecer la conexión entre la aplicación y la base de datos MySQL. Para interactuar con la base de datos, se utiliza el siguiente código en Java:

```

class InventarioDB {
    static {
        try {
            Class.forName(className:"com.mysql.cj.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

    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";

    public static int obtenerStock() {
        int stock = 0;
        try {
            Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);
            Statement stat = conn.createStatement();
            ResultSet rs = stat.executeQuery(sql:"SELECT stock FROM Inventario WHERE producto='Pan'"); {
                if (rs.next()) {
                    stock = rs.getInt(columnLabel:"stock");
                }
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return stock;
    }

    public static synchronized boolean comprarPan(int cantidad) {
        try {
            Connection conn = DriverManager.getConnection(URL, USER, PASSWORD); {
                conn.setAutoCommit(autoCommit:false);

                try {
                    Statement stat = conn.createStatement();
                    ResultSet rs = stat.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();
                                    return true;
                                }
                            }
                        }
                    }
                } catch (SQLException e) {
                    e.printStackTrace();
                }
                conn.rollback();
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return false;
    }

    public static void añadirStock(int cantidad) {
        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();
                System.out.println("Se hornearon " + cantidad + " panes. Nuevo stock: " + obtenerStock());
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

```

Figura 11. Código utilizado para la conexión con la BD con JDBC.

Para ello y poder realizar la conexión hay que instalar el conector desde la pagina oficial de MySQL.

4. Cliente de Panadería (Interacción con los Usuarios)

El cliente es una aplicación que permite a los usuarios conectarse al sistema distribuido y realizar compras en la panadería. Al iniciar, el cliente se comunica con el servidor de descubrimiento para obtener un servidor disponible y establecer una conexión.

El cliente presenta un menú con opciones para:

1. Consultar el stock de productos.
2. Realizar una compra.
3. Cerrar sesión y desconectarse.

Cada interacción con el cliente se envía al servidor asignado, el cual procesa la solicitud y devuelve la respuesta correspondiente.

En la siguiente figura, se muestra el código del cliente de panadería, el cual establece la conexión con los servidores y maneja la interacción con el usuario para realizar compras en línea. Este código es crucial para que el cliente pueda interactuar con el sistema de panadería de manera eficiente. En el inicio del código, definimos la dirección y puerto del servidor de descubrimiento. Este servidor es responsable de asignar al cliente un servidor de panadería disponible.

Posteriormente, creamos objetos para manejar la comunicación entre el cliente y el servidor, tales como el socket, el flujo de entrada (BufferedReader) y el flujo de salida (PrintWriter). También inicializamos un objeto Scanner para leer la entrada del usuario.

```
import java.io.*;
import java.net.*;
import java.util.*;

public class PanaderiaCliente {
    public static void main(String[] args) {
        String servidorDescubrimiento = "192.168.100.12"; // Dirección del servidor de descubrimiento
        int puertoDescubrimiento = 5050; // Puerto del servidor de descubrimiento

        final Socket[] socket = new Socket[1]; // Usamos un array para poder modificar su valor dentro del hook
        final BufferedReader[] in = new BufferedReader[1]; // Se declara como final
        final PrintWriter[] out = new PrintWriter[1]; // Se declara como final
        Scanner scanner = new Scanner(System.in);

        // Registrar un shutdown hook para manejar la interrupción del programa
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            try {
                System.out.println(x:"Cerrando recursos correctamente...");
                if (socket[0] != null && !socket[0].isClosed()) {
                    socket[0].close();
                }
                if (in[0] != null) {
                    in[0].close();
                }
                if (out[0] != null) {
                    out[0].close();
                }
            } catch (IOException e) {
                System.err.println("Error cerrando la conexión: " + e.getMessage());
            }
        }));
    }
}
```

Figura 12. Código de inicialización de la PanaderiaCliente.

Después, el cliente se conecta al servidor de descubrimiento a través de un socket. Una vez conectados, se inicializan los flujos de entrada y salida para la comunicación, se envía una solicitud al servidor de descubrimiento para obtener un puerto de uno de los servidores de panadería disponibles, donde el cliente recibe la lista de puertos disponibles del servidor de descubrimiento y se procesan los puertos disponibles, separándolos por comas y convirtiéndolos a una lista de enteros y de manera aleatoria el cliente selecciona un puerto aleatorio de la lista de puertos disponibles para conectarse al servidor de panadería.

```

// Conectar al servidor de descubrimiento
try {
    socket[0] = new Socket(servidorDescubrimiento, puertoDescubrimiento); // Asignamos el socket en el array
    in[0] = new BufferedReader(new InputStreamReader(socket[0].getInputStream()));
    out[0] = new PrintWriter(socket[0].getOutputStream(), autoFlush:true);

    // Solicitar un puerto de servidor de panadería
    System.out.println(x:"Solicitando un puerto de panadería...");
    out[0].println(x:"SOLICITUD_PUERTO"); // Enviar solicitud para obtener un puerto de servidor

    // Leer la lista de puertos disponibles
    String respuesta = in[0].readLine();
    if (respuesta != null && !respuesta.equals(anObject:"No hay servidores de panadería disponibles.")) {
        // Extraer los puertos de servidores disponibles (puedes modificar esto según el formato de la respuesta)
        List<Integer> puertosDisponibles = new ArrayList<>();
        String[] puertos = respuesta.split(regex:",");
        for (String puertoStr : puertos) {
            puertosDisponibles.add(Integer.parseInt(puertoStr.trim()));
        }

        // Elegir un puerto aleatoriamente
        Random random = new Random();
        int puertoPanaderia = puertosDisponibles.get(random.nextInt(puertosDisponibles.size()));
        System.out.println("Conectando al servidor de panadería en el puerto " + puertoPanaderia);
    }
}

```

Figura 13. Conexión al servidor de descubrimiento y a PanaderiaSevidor.

En la siguiente Figura, podemos ver el bloque de código donde el cliente se conecta al servidor de panadería usando el puerto asignado, y se crean los flujos de entrada y salida correspondientes. El cliente solicita el nombre del usuario y verifica que no esté vacío antes de continuar, y una vez obtenido el nombre el cliente lo envía al servidor de panadería y espera un mensaje de bienvenida, además de mostrar las opciones disponibles y se le solicita al cliente que seleccione alguna opción. y luego la envía al servidor para procesar la solicitud.

Finalmente, se manejan las operaciones de compra y se cierra la conexión correctamente. Este código es un ejemplo claro de cómo el cliente interactúa con el servidor de panadería para realizar compras y cómo se maneja la conexión y desconexión de los recursos.

```

// Ahora conectarse al servidor de panadería en el puerto asignado
try (Socket panaderiaSocket = new Socket(servidorDescubrimiento, puertoPanaderia);
    BufferedReader panaderiaIn = new BufferedReader(new InputStreamReader(panaderiaSocket.getInputStream()));
    PrintWriter panaderiaOut = new PrintWriter(panaderiaSocket.getOutputStream(), autoFlush:true)) {

    // Solicitar y validar el nombre del cliente
    String nombre = null;
    boolean nombreValido = false;

    while (!nombreValido) {
        System.out.print("Ingrese su nombre: ");
        if (scanner.hasNextLine()) {
            nombre = scanner.nextLine();
        } else {
            break; // Salir si no se puede leer del scanner
        }
        if (nombre != null && !nombre.trim().isEmpty()) {
            nombreValido = true; // El nombre es válido
        } else {
            System.out.println("El nombre no puede estar vacío.");
        }
    }

    // Si el nombre no es válido (null o vacío), no se envía nada
    if (nombre != null && !nombre.trim().isEmpty()) {
        panaderiaOut.println(nombre); // Enviar nombre al servidor
    }

    // Mensaje de bienvenida del servidor
    String mensajeBienvenida = panaderiaIn.readLine();
    if (mensajeBienvenida != null) {
        System.out.println(mensajeBienvenida);
    }

    while (true) {
        String linea;
        while (!(!linea = panaderiaIn.readLine()).contains(":Seleccione una opción")) {
            System.out.println(linea);
        }
        System.out.println(linea); // Imprime "Seleccione una opción:"

        System.out.print("Opción: ");
        String opcionStr = scanner.nextLine();
        panaderiaOut.println(opcionStr); // Enviar opción al servidor

        if (opcionStr.equals(anObject:"1") || opcionStr.equals(anObject:"3")) {
            System.out.println("Servidor: " + panaderiaIn.readLine());
            if (opcionStr.equals(anObject:"3")) break; // Salir si elige salir
        } else if (opcionStr.equals(anObject:"2")) {
            System.out.println(panaderiaIn.readLine());
            String cantidad = scanner.nextLine();
            panaderiaOut.println(cantidad); // Enviar cantidad al servidor
            String respuestaCompra = panaderiaIn.readLine();
            System.out.println("Servidor: " + respuestaCompra);
        } else {
            System.out.println("Opción no válida.");
        }
    }
}

```

Figura 14. Asignación del puerto y operaciones disponibles desde el cliente.

```

        System.out.println("Saliendo de la panadería... ¡Hasta luego!");
    } catch (IOException e) {
        System.err.println("Error al conectarse al servidor de panadería: " + e.getMessage());
    }
} else {
    System.err.println("Error: " + respuesta);
}
} catch (IOException e) {
    System.err.println("Error de conexión al servidor de descubrimiento: " + e.getMessage());
} finally {
    try {
        // Asegurarse de cerrar los recursos correctamente
        if (socket[0] != null && !socket[0].isClosed()) {
            socket[0].close();
        }
        if (in[0] != null) {
            in[0].close();
        }
        if (out[0] != null) {
            out[0].close();
        }
    } catch (IOException e) {
        System.err.println("Error cerrando la conexión: " + e.getMessage());
    }
}
}

```

Figura 15. Asignación del puerto y operaciones disponibles desde el cliente.

5. Gestión de Concurrency y Sincronización

Dado que múltiples servidores pueden procesar compras simultáneamente, se implementaron mecanismos de control de concurrencia, como:

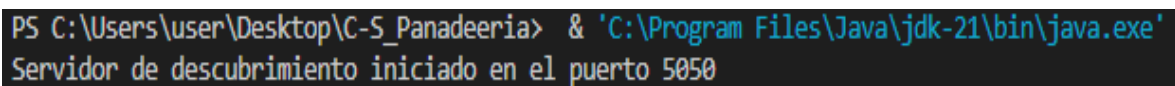
- **Bloqueo de registros en la base de datos**, evitando que dos transacciones modifiquen el mismo producto al mismo tiempo.
- **Confirmaciones de transacción (commit)** para asegurar que los cambios en el inventario sean reflejados de manera consistente en la base de datos.
- **Manejo de hilos en los servidores**, permitiendo que múltiples clientes sean atendidos sin afectar el rendimiento del sistema.

Con la ayuda de estos tres códigos: `PanaderiaCliente`, `PanaderiaServidor` y `ServidorDescubrimiento`, podemos realizar la práctica de manera eficiente y funcional. Estos códigos permiten establecer la comunicación entre los cliente y los servidores de panadería, facilitando la gestión del inventario y la interacción con los usuarios. Gracias a su estructura modular, se logra una correcta distribución de tareas entre los diferentes componentes del sistema, asegurando una experiencia fluida y sincronizada.

Resultados

Al ejecutar la solución propuesta, el sistema demuestra su capacidad para gestionar múltiples clientes de manera simultánea, garantizando una interacción eficiente y sin conflictos con el servidor de la panadería. A continuación, se detallan los resultados obtenidos al ejecutar el código tanto en el servidor como en los clientes.

En primer lugar, al iniciar el servidor de descubrimiento en el puerto configurado, el servidor permanece a la espera de conexiones entrantes, y se pone a disposición de los clientes mediante la dirección IP de la red local, lo que permite que distintos equipos dentro de la misma red local puedan conectarse sin inconvenientes. Cuando un cliente se conecta, el servidor de panadería acepta la conexión y crea un hilo independiente para manejar la interacción con dicho cliente. Esto permite que múltiples clientes se conecten y gestionen sus transacciones de manera concurrente, sin afectar el desempeño del sistema.



```
PS C:\Users\user\Desktop\C-S_Panadeeria> & 'C:\Program Files\Java\jdk-21\bin\java.exe'
Servidor de descubrimiento iniciado en el puerto 5050
```

Figura 16. Ejecución del Servidor de descubrimiento en el puerto 5050.

El servidor de panadería también es capaz de manejar el stock de pan, asegurando que las interacciones con los clientes se realicen de forma sincronizada. Si varios clientes intentan comprar pan al mismo tiempo, el sistema es capaz de gestionar el acceso a los recursos compartidos (como el stock de pan), evitando condiciones de carrera y actualizando correctamente la cantidad disponible de pan.

Al ejecutar el código cliente, el sistema solicita al usuario que ingrese su nombre y le ofrece un menú con varias opciones de interacción:

1. **Ver el stock de pan:** Al seleccionar esta opción, el servidor envía la cantidad actual de panes disponibles. El cliente puede verificar en todo momento la cantidad disponible de pan en el inventario.
2. **Comprar pan:** Si el usuario selecciona esta opción, se solicita la cantidad de pan que desea comprar. Si el stock es suficiente, la compra se procesa correctamente, y el servidor actualiza la cantidad de pan restante. Si el stock es insuficiente, el cliente recibe un mensaje que le indica que debe esperar a que el horno produzca más panes.
3. **Salir:** Esta opción permite que el cliente se desconecte del servidor, cerrando la conexión de manera ordenada.

Los mensajes del servidor muestran cómo se gestionan las solicitudes de los clientes, garantizando una experiencia de usuario fluida y sin errores. Además, en el servidor se lleva un registro de todas las interacciones, lo que facilita la observación del comportamiento del sistema durante las pruebas de concurrencia.

Para comprobar la capacidad concurrente del sistema, se realizaron pruebas en las que se conectaron varios clientes simultáneamente. Como se puede observar en las siguientes imágenes, se establecieron varias conexiones, y ambos clientes pudieron consultar el stock y realizar compras de manera independiente, sin interferencias entre ellos.

Además, se realizaron conexiones desde dos equipos distintos dentro de la misma red local. Al ejecutar el código en diferentes dispositivos, se pudo verificar que el cliente se conecta correctamente a los servidores de panadería. Esto se demuestra mediante la obtención de la dirección IP del cliente, que es capturada por el servidor al momento de aceptar la conexión. En el código del servidor, el objeto Socket se utiliza para obtener la dirección IP del cliente mediante el método `getInetAddress()` y luego se usa `getHostAddress()` para obtener la dirección IP en formato de cadena, como "192.168.1.5".

Entonces, una vez ejecutado el Servidor de Descubrimiento, podemos empezar a abrir distintos servidores para la Panadería en puertos diferentes.

```
C:\Users\user\Desktop\C-S_Panadeeria>java PanaderiaServidor 10002
Servidor de panadería con puerto 10002 se ha registrado.
Servidor de panadería iniciado en el puerto 10002
```

Figura 16. Ejecución del Servidor Panaderia en el puerto 10002.

```
PS C:\Users\user\Desktop\C-S_Panadeeria> java PanaderiaServidor 5002
Servidor de panadería con puerto 5002 se ha registrado.
Servidor de panadería iniciado en el puerto 5002
```

Figura 17. Ejecución del Servidor Panaderia en el puerto 5002.

```
PS C:\Users\user\Desktop\C-S_Panadeeria> java PanaderiaServidor 8723
Servidor de panadería con puerto 8723 se ha registrado.
Servidor de panadería iniciado en el puerto 8723
```

Figura 18. Ejecución del Servidor Panaderia en el puerto 8723.

Una vez ejecutados cada PanaderiaServidor, podemos abrir tantos servidores como deseemos o queramos utilizar, en mi caso solo abri 3 servidores en distintos puertos, y en el Servidor Descubrimiento, nos indica que servidores están registrados o disponibles para los clientes.

```
Servidor de descubrimiento iniciado en el puerto 5050
Servidor de panadería registrado en el puerto: 10002
Servidores de panadería registrados: {panaderia10002=10002}
Servidor de panadería registrado en el puerto: 5002
Servidores de panadería registrados: {panaderia5002=5002, panaderia10002=10002}
Servidor de panadería registrado en el puerto: 8723
Servidores de panadería registrados: {panaderia5002=5002, panaderia10002=10002, panaderia8723=8723}
```

Figura 19. Mensajes de registro en el ServidorDescubrimiento.

Ahora, teniendo ejecutados cada servidor, podemos ejecutar el cliente en distintos dispositivos dentro de la red local y como vemos a continuación, veremos las opciones disponibles para cada cliente, y como se conectan distintos clientes en cada servidor.

```
Solicitando un puerto de panadería...
Conectando al servidor de panadería en el puerto 8723
Ingrese su nombre: Paco
Bienvenido a la Panadería, Paco!

--- Menú Panadería ---
1. Ver stock
2. Comprar pan
3. Salir
Seleccione una opción:
Opción: █
```

Figura 20. Conexión de un cliente en el servidor de panaderia en el puerto 8723.

```
PS C:\Users\user\Desktop\C-S_Panadeeria> java .\PanaderiaCliente.java
Solicitando un puerto de panadería...
Conectando al servidor de panadería en el puerto 10002
Ingrese su nombre: Luis
Bienvenido a la Panadería, Luis!

--- Menú Panadería ---
1. Ver stock
2. Comprar pan
3. Salir
Seleccione una opción:
Opción: █
```

Figura 21. Conexión de un cliente en el servidor de panaderia en el puerto 10002.

```

PS C:\Users\user\Desktop\C-S_Panadeeria> java .\PanaderiaCliente.java
Solicitando un puerto de panadería...
Conectando al servidor de panadería en el puerto 5002
Ingrese su nombre: Elena
Bienvenido a la Panadería, Elena!

--- Menú Panadería ---
1. Ver stock
2. Comprar pan
3. Salir
Seleccione una opción:
Opción:

```

Figura 22. Conexión de un cliente en el servidor de panaderia en el puerto 5002.

Y en cada Servidor de Panaderia, nos indica que clientes se han conectado, desde que IP y que puerto.

```

Luis se ha conectado desde la IP: 192.168.100.12 en el puerto: 4367
Paco se ha conectado desde la IP: 192.168.100.12 en el puerto: 4215
Elena se ha conectado desde la IP: 192.168.100.12 en el puerto: 4539

```

Figura 23. Mensajes de conexión de los clientes en cada servidor de panaderia.

De igual forma en cada servidor de panadería, como vimos anteriormente se hornean panes cada cierto tiempo como vemos a continuación.

```

Servidor de panadería con puerto 10002 se ha registrado.
Servidor de panadería iniciado en el puerto 10002
Se hornearon 5 panes. Nuevo stock: 3825
Se hornearon 5 panes. Nuevo stock: 3830
Se hornearon 5 panes. Nuevo stock: 3835
Se hornearon 5 panes. Nuevo stock: 3840
Se hornearon 5 panes. Nuevo stock: 3845

```

Figura 24. Mensajes de horneado de panes nuevos en los servidores.

Respecto al cliente, este puede realizar distintas operaciones, entre ellas tenemos ver el stock, realizar compras y salir de la panadería (Finalizar el programa).

```

--- Menú Panadería ---
1. Ver stock
2. Comprar pan
3. Salir
Seleccione una opción:
Opción: 1
Servidor: Stock disponible: 5280 panes.

```

Figura 25. Ver el stock desde el cliente.

```
--- Menú Panadería ---
1. Ver stock
2. Comprar pan
3. Salir
Seleccione una opción:
Opción: 1
Servidor: Stock disponible: 5595 panes.

--- Menú Panadería ---
1. Ver stock
2. Comprar pan
3. Salir
Seleccione una opción:
Opción:

--- Menú Panadería ---
1. Ver stock
2. Comprar pan
3. Salir
Seleccione una opción:
Opción: 1
Servidor: Stock disponible: 5595 panes.
```

Figura 25. Ver el stock desde múltiples clientes conectados a distintos servidores.

Cuando varios clientes realizan compras de pan de manera simultánea, el sistema demuestra su capacidad para manejar múltiples transacciones sin generar inconsistencias

```
--- Menú Panadería ---
1. Ver stock
2. Comprar pan
3. Salir
Seleccione una opción:
Opción: 2
Ingrese la cantidad de pan que desea comprar:
500
Servidor: Compra exitosa. Panes restantes: 3940

--- Menú Panadería ---
1. Ver stock
2. Comprar pan
3. Salir
Seleccione una opción:
Opción:

--- Menú Panadería ---
1. Ver stock
2. Comprar pan
3. Salir
Seleccione una opción:
Opción: 2
Ingrese la cantidad de pan que desea comprar:
500
Servidor: Compra exitosa. Panes restantes: 3440

--- Menú Panadería ---
1. Ver stock
2. Comprar pan
3. Salir
Seleccione una opción:
Opción: 2
Ingrese la cantidad de pan que desea comprar:
500
Servidor: Compra exitosa. Panes restantes: 2945
```

Figura 26. Realizar compra de pan desde múltiples clientes conectados a distintos servidores.

En general, la solución implementada permite que varios clientes compren pan simultáneamente, gestionando el stock de manera eficiente y evitando inconsistencias. Gracias a la sincronización del servidor y las verificaciones de stock, las transacciones se procesan de forma correcta, incluso cuando múltiples clientes intentan comprar al mismo tiempo.

Y en el servidor, se muestran los siguientes mensajes de compra por cada cliente.

```
Cliente Luis compró 500 panes. Stock restante: 3940
Se hornearon 5 panes. Nuevo stock: 3445
```

```
Cliente Elena compró 500 panes. Stock restante: 3440
Se hornearon 5 panes. Nuevo stock: 2955
```

```
Cliente Paco compró 500 panes. Stock restante: 2945
Se hornearon 5 panes. Nuevo stock: 2950
```

Figura 27. Mensajes hacia el servidor de las compras de cada cliente.

Los mensajes llegan de manera única a cada servidor, y como podemos ver las compras se realizaron de manera concurrente, casi al mismo tiempo, es por ello por ejemplo que de la compra de Luis a la compra de Elena el nuevo stock es diferente, y de igual forma con la compra de Paco.

También, algo por mencionar es que en el ServidorDescubrimiento se indica que puertos son mandados a cada cliente.

```
Se ha enviado la lista de puertos de panadería: 5002, 10002, 8723
Se ha enviado la lista de puertos de panadería: 5002, 10002, 8723
Se ha enviado la lista de puertos de panadería: 5002, 10002, 8723
```

Figura 28. Mensaje de los puertos enviados a cada cliente en el Servidor de Descubrimiento.

Finalmente, la operación de salir, en este caso como se vio en el código se cierran correctamente los recursos.

```
Saliendo de la panadería... ¡Hasta luego!
Cerrando recursos correctamente...
PS C:\Users\user\Desktop\C-S_Panadeeria> |
Opcion: 2
Ingrese la cantidad de pan que desea comprar:
500
Servidor: Compra exitosa. Panes restantes: 2945

--- Menú Panadería ---
1. Ver stock
2. Comprar pan
3. Salir
Seleccione una opción:
Opción: 3
Servidor: Gracias por visitar la panadería. ¡Hasta luego!
Saliendo de la panadería... ¡Hasta luego!
Cerrando recursos correctamente...
PS C:\Users\user\Desktop\C-S_Panadeeria> |
```

Figura 29. Opción de salir y cierre de recursos correctamente en cada cliente.

En general, el sistema demuestra una alta capacidad para manejar múltiples clientes de manera simultánea, gestionar la sincronización del stock de pan, y proporcionar una experiencia de usuario satisfactoria. Además, las pruebas de concurrencia y conectividad entre distintos dispositivos dentro de la misma red local validan el funcionamiento adecuado del sistema.

Cabe decir que, para poder observar estos resultados, como se menciona es necesario tener instalado y descargado el conector JDBC de MySQL, y para poder ejecutarlo desde terminal es necesario añadir el archivo .jar del conector descargado a las variables de entorno del sistema, específicamente al CLASSPATH, donde debemos colocar la ruta del archivo .jar.

Variables del sistema	
Variable	Valor
CLASSPATH	.;C:\Users\user\Desktop\C-S_Panadeeria\lib\mysql-connector-j-9.2....

Figura 30. Archivo .jar añadido a la variable CLASSPATH en variables de entorno.

Conclusión

En esta práctica, pude profundizar en el concepto de sistemas multiservidor, donde la principal dificultad radicó en la gestión eficiente de múltiples servidores para atender las solicitudes de los clientes de manera concurrente. Aunque el diseño básico del servidor y los clientes se mantiene similar a prácticas anteriores, se introdujeron mejoras importantes como la implementación de un servidor de descubrimiento que facilita la distribución de conexiones entre varios servidores de panadería.

Al permitir que los clientes se conecten a diferentes servidores dentro de un mismo entorno, logré comprender mejor los desafíos de la sincronización entre estos servidores y cómo asegurar que las bases de datos compartidas mantengan la coherencia del inventario, sin que un servidor afecte negativamente al resto. La utilización de múltiples servidores permitió mejorar la escalabilidad y la eficiencia al distribuir las cargas de trabajo y reducir el riesgo de sobrecargar un único servidor.

El uso de un servidor de descubrimiento también fue esencial para dirigir las solicitudes de los clientes a los servidores disponibles de manera dinámica, lo cual resalta la importancia de la gestión de recursos en un sistema distribuido. Esta técnica refleja cómo los sistemas pueden escalar horizontalmente, es decir, añadiendo más servidores para mejorar el rendimiento y la capacidad de respuesta, un concepto clave en aplicaciones de alto tráfico.

A lo largo de la práctica, pude observar cómo la coordinación entre los distintos servidores permite gestionar eficientemente el inventario y atender múltiples clientes simultáneamente. Esto me permitió entender mejor los mecanismos internos de un sistema multiservidor y los retos asociados, como la sincronización de datos y la gestión de conexiones concurrentes.

En general, esta práctica me brindó una comprensión más profunda de cómo funcionan los sistemas distribuidos en un entorno multiservidor y cómo se pueden implementar soluciones para garantizar una distribución equitativa de recursos, minimizar tiempos de espera y mantener la integridad de la información a través de múltiples servidores. Esto me ha motivado a seguir explorando y mejorando la eficiencia de los sistemas distribuidos, especialmente en entornos de mayor escala.

Bibliografía

Daemon4, "Arquitectura Cliente-Servidor," 2024. [En línea]. Available: <https://www.daemon4.com/empresa/noticias/arquitectura-cliente-servidor/>. [Último acceso: 16 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].

J. Geeks, "Introducción a la programación con sockets en Java," Geeks for Geeks, 2023. [En línea]. Available: <https://www.geeksforgeeks.org/socket-programming-in-java>. [Último acceso: 16 Marzo 2025].

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

J. Geeks, "Programación de servidores multiclientes en Java," Geeks for Geeks, 2023. [En línea]. Available: <https://www.geeksforgeeks.org/multithreaded-server-in-java>. [Último acceso: 16 Marzo 2025].

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

Stack Overflow, "How to create a multithreaded server in Java," Stack Overflow, 2024. [En línea]. Available: <https://stackoverflow.com/questions/123456/how-to-create-a-multithreaded-server-in-java>. [Último acceso: 16 Marzo 2025].

J. Geeks, "Introducción a la programación multicliente en Java," Geeks for Geeks, 2023. [En línea]. Available: <https://www.geeksforgeeks.org/multithreaded-client-server-architecture-in-java/>. [Último acceso: 16 Marzo 2025].

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

Anexos

Código PanaderiaCliente.java

```
import java.io.*;
import java.net.*;
import java.util.*;

public class PanaderiaCliente {
    public static void main(String[] args) {
        String servidorDescubrimiento = "192.168.100.12"; // Dirección del
servidor de descubrimiento
        int puertoDescubrimiento = 5050; // Puerto del servidor de
descubrimiento

        final Socket[] socket = new Socket[1]; // Usamos un array para
poder modificar su valor dentro del hook
        final BufferedReader[] in = new BufferedReader[1]; // Se declara
como final
        final PrintWriter[] out = new PrintWriter[1]; // Se declara como
final
        Scanner scanner = new Scanner(System.in);

        // Registrar un shutdown hook para manejar la interrupción del
programa
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            try {
                System.out.println("Cerrando recursos correctamente...");
                if (socket[0] != null && !socket[0].isClosed()) {
                    socket[0].close();
                }
                if (in[0] != null) {
                    in[0].close();
                }
                if (out[0] != null) {
                    out[0].close();
                }
            } catch (IOException e) {
                System.err.println("Error cerrando la conexión: " +
e.getMessage());
            }
        }));

        // Conectar al servidor de descubrimiento
        try {
```

```

        socket[0] = new Socket(servidorDescubrimiento,
puertoDescubrimiento); // Asignamos el socket en el array
        in[0] = new BufferedReader(new
InputStreamReader(socket[0].getInputStream()));
        out[0] = new PrintWriter(socket[0].getOutputStream(), true);

        // Solicitar un puerto de servidor de panadería
        System.out.println("Solicitando un puerto de panadería...");
        out[0].println("SOLICITUD_PUERTO"); // Enviar solicitud para
obtener un puerto de servidor

        // Leer la lista de puertos disponibles
        String respuesta = in[0].readLine();
        if (respuesta != null && !respuesta.equals("No hay servidores de
panadería disponibles.)) {
            // Extraer los puertos de servidores disponibles (puedes
modificar esto según el formato de la respuesta)
            List<Integer> puertosDisponibles = new ArrayList<>();
            String[] puertos = respuesta.split(",");
            for (String puertoStr : puertos) {
                puertosDisponibles.add(Integer.parseInt(puertoStr.trim()
));
            }

            // Elegir un puerto aleatoriamente
            Random random = new Random();
            int puertoPanaderia =
puertosDisponibles.get(random.nextInt(puertosDisponibles.size()));
            System.out.println("Conectando al servidor de panadería en
el puerto " + puertoPanaderia);

            // Ahora conectarse al servidor de panadería en el puerto
asignado
            try (Socket panaderiaSocket = new
Socket(servidorDescubrimiento, puertoPanaderia);
                BufferedReader panaderiaIn = new BufferedReader(new
InputStreamReader(panaderiaSocket.getInputStream()));
                PrintWriter panaderiaOut = new
PrintWriter(panaderiaSocket.getOutputStream(), true)) {

                // Solicitar y validar el nombre del cliente
                String nombre = null;
                boolean nombreValido = false;

                while (!nombreValido) {

```

```

        System.out.print("Ingrese su nombre: ");
        if (scanner.hasNextLine()) {
            nombre = scanner.nextLine();
        } else {
            break; // Salir si no se puede leer del scanner
        }
        if (nombre != null && !nombre.trim().isEmpty()) {
            nombreValido = true; // El nombre es válido
        } else {
            System.out.println("El nombre no puede estar
vacío.");
        }
    }

    // Si el nombre no es válido (null o vacío), no se envía
nada

    if (nombre != null && !nombre.trim().isEmpty()) {
        panaderiaOut.println(nombre); // Enviar nombre al
servidor
    }

    // Mensaje de bienvenida del servidor
    String mensajeBienvenida = panaderiaIn.readLine();
    if (mensajeBienvenida != null) {
        System.out.println(mensajeBienvenida);
    }

    while (true) {
        String linea;
        while (!(linea =
panaderiaIn.readLine()).contains("Seleccione una opción")) {
            System.out.println(linea);
        }
        System.out.println(linea); // Imprime "Seleccione
una opción:"

        System.out.print("Opción: ");
        String opcionStr = scanner.nextLine();
        panaderiaOut.println(opcionStr); // Enviar opción al
servidor

        if (opcionStr.equals("1") || opcionStr.equals("3"))
{
            System.out.println("Servidor: " +
panaderiaIn.readLine());

```

```

        if (opcionStr.equals("3")) break; // Salir si
elige salir
    } else if (opcionStr.equals("2")) {
        System.out.println(panaderiaIn.readLine());
        String cantidad = scanner.nextLine();
        panaderiaOut.println(cantidad); // Enviar
cantidad al servidor
        String respuestaCompra = panaderiaIn.readLine();
        System.out.println("Servidor: " +
respuestaCompra);
    } else {
        System.out.println("Opción no válida.");
    }
}

    System.out.println("Saliendo de la panadería... ¡Hasta
luego!");
} catch (IOException e) {
    System.err.println("Error al conectarse al servidor de
panadería: " + e.getMessage());
}
} else {
    System.err.println("Error: " + respuesta);
}
} catch (IOException e) {
    System.err.println("Error de conexión al servidor de
descubrimiento: " + e.getMessage());
} finally {
    try {
        // Asegurarse de cerrar los recursos correctamente
        if (socket[0] != null && !socket[0].isClosed()) {
            socket[0].close();
        }
        if (in[0] != null) {
            in[0].close();
        }
        if (out[0] != null) {
            out[0].close();
        }
    } catch (IOException e) {
        System.err.println("Error cerrando la conexión: " +
e.getMessage());
    }
}
}
}

```

```
}
```

Código PanaderiaServidor.java

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

class InventarioDB {
    static {
        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

    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";

    public static int obtenerStock() {
        int stock = 0;
        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 (SQLException e) {
            e.printStackTrace();
        }
        return stock;
    }

    public static synchronized boolean comprarPan(int cantidad) {
        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();
                    return true;
                }
            }
            conn.rollback();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return false;
    }

    public static void añadirStock(int cantidad) {
        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();
            System.out.println("Se hornearon " + cantidad + " panes. Nuevo
stock: " + obtenerStock());
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

class HornoPanadero extends Thread {
    @Override
    public void run() {
        while (true) {

```

```

        try {
            Thread.sleep(10000); // Cada 10 segundos se hornean más
panes
            InventarioDB.añadirStock(5);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

class HiloCliente extends Thread {
    private final Socket socket;

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

    @Override
    public void run() {
        try (
            BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            PrintWriter out = new PrintWriter(socket.getOutputStream(),
true)
        ) {
            // Leer el nombre del cliente y dar la bienvenida
            String nombreCliente = in.readLine();
            String ipCliente = socket.getInetAddress().getHostAddress();
            System.out.println(nombreCliente + " se ha conectado desde la
IP: " + ipCliente + " en el puerto: " + socket.getPort());
            out.println("Bienvenido a la Panadería, " + nombreCliente +
"!");

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

                String opcionStr = in.readLine();
                if (opcionStr == null) break;
            }
        }
    }
}

```

```

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

        if (opcion == 1) {
            out.println("Stock disponible: " +
InventarioDB.obtenerStock() + " panes.");
        } else if (opcion == 2) {
            out.println("Ingrese la cantidad de pan que desea
comprar:");

            String cantidadStr = in.readLine();
            int cantidad;
            try {
                cantidad = Integer.parseInt(cantidadStr);
            } catch (NumberFormatException e) {
                out.println("Cantidad no válida. Intente de
nuevo.");
                continue;
            }
            boolean compraExitosa =
InventarioDB.comprarPan(cantidad);
            if (compraExitosa) {
                out.println("Compra exitosa. Panes restantes: " +
InventarioDB.obtenerStock());
                System.out.println("Cliente " + nombreCliente + "
compró " + cantidad + " panes. Stock restante: " +
InventarioDB.obtenerStock());
            } else {
                out.println("No hay suficiente pan. Por favor,
espere mientras horneamos más.");
            }
        } else if (opcion == 3) {
            out.println("Gracias por visitar la panadería. ¡Hasta
luego!");
            System.out.println(nombreCliente + " se ha
desconectado.");
            break;
        } else {
            out.println("Opción no válida. Intente de nuevo.");
        }
    }
}

```



```

    } catch (IOException e) {
        System.err.println("Error en la conexión con el cliente: " +
e.getMessage());
    } finally {
        try {
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
}

public class PanaderiaServidor {
    public static void iniciarServidor(int puerto) {
        ThreadPoolExecutor pool = (ThreadPoolExecutor)
Executors.newFixedThreadPool(5);

        // Registrar el servidor en el servidor de descubrimiento antes de
iniciar el servicio
        ServidorDescubrimiento.registrarServidor(puerto);

        new HornoPanadero().start(); // Hilo para hornear pan

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

            while (true) {
                Socket socketCliente = serverSocket.accept();
                pool.execute(new HiloCliente(socketCliente));
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            // Desregistrar el servidor en el servidor de descubrimiento
cuando se detiene
            ServidorDescubrimiento.desregistrarServidor(puerto);
        }
    }

    public static void main(String[] args) {
        // Verificar que se pasen puertos como argumentos
        if (args.length == 0) {
            System.err.println("Debe proporcionar al menos un puerto.");
        }
    }
}

```

```

        return;
    }

    // Iterar sobre los puertos pasados y crear un hilo para cada uno
    for (String puertoStr : args) {
        try {
            int puerto = Integer.parseInt(puertoStr); // Convertir cada
argumento a entero
            // Iniciar un hilo para cada puerto
            new Thread(() -> iniciarServidor(puerto)).start();
        } catch (NumberFormatException e) {
            System.err.println("Error: El puerto '" + puertoStr + "' no
es válido.");
        }
    }
}
}
}

```

Código ServidorDescubrimiento.java

```

import java.io.*;
import java.net.*;
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;

public class ServidorDescubrimiento {
    private static final int PUERTO_DESCUBRIMIENTO = 5050;
    private static final Map<String, Integer> servidoresPanaderia = new
ConcurrentHashMap<>(); // Mapa concurrente para manejo de múltiples hilos

    public static void main(String[] args) {
        // Iniciar el servidor de descubrimiento
        try (ServerSocket serverSocket = new
ServerSocket(PUERTO_DESCUBRIMIENTO)) {
            System.out.println("Servidor de descubrimiento iniciado en el
puerto " + PUERTO_DESCUBRIMIENTO);

            while (true) {
                // Aceptar las conexiones de los servidores de panadería
                Socket socketCliente = serverSocket.accept();
                new Thread(() -> manejarCliente(socketCliente)).start(); //
Manejo concurrente de clientes
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

```

    }

    // Maneja la conexión de un servidor de panadería
    private static void manejarCliente(Socket socket) {
        try (BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            PrintWriter out = new PrintWriter(socket.getOutputStream(),
true)) {

            // Recibir el mensaje del cliente
            String mensaje = in.readLine();

            // Dentro de manejarCliente(), en el caso de solicitud de
puerto:
            if ("SOLICITUD_PUERTO".equals(mensaje)) {
                if (!servidoresPanaderia.isEmpty()) {
                    // Obtener todos los puertos registrados como una lista
separada por comas
                    String puertos = String.join(", ",
servidoresPanaderia.values().stream().map(String::valueOf).toArray(String[]:
:new));
                    out.println(puertos); // Enviar la lista de
puertos al cliente
                    System.out.println("Se ha enviado la lista de
puertos de panadería: " + puertos);
                } else {
                    out.println("No hay servidores de panadería
disponibles.");
                }
            } else {
                // Si el mensaje no es una solicitud de puerto, manejarlo
como un registro de servidor
                int puerto = Integer.parseInt(mensaje); // Convertir el
mensaje a puerto
                System.out.println("Servidor de panadería registrado en el
puerto: " + puerto);

                // Registrar el servidor de panadería en el mapa (servidores
disponibles)
                servidoresPanaderia.put("panaderia" + puerto, puerto);

                // Responder al servidor de panadería que se registró
correctamente
                out.println("Servidor registrado correctamente en el puerto:
" + puerto);
            }
        }
    }
}

```

```

        // (Opcional) Imprimir todos los servidores registrados para
verificación
        System.out.println("Servidores de panadería registrados: " +
servidoresPanaderia);
    }
} catch (IOException e) {
    e.printStackTrace();
}
}

// Método para registrar un servidor (llamado desde el servidor de
panadería)
public static void registrarServidor(int puerto) {
    try (Socket socket = new Socket("192.168.100.12",
PUERTO_DESCUBRIMIENTO);
        PrintWriter out = new PrintWriter(socket.getOutputStream(),
true)) {
        // Enviar el puerto del servidor de panadería al servidor de
descubrimiento
        out.println(puerto); // Enviar el puerto como mensaje
        System.out.println("Servidor de panadería con puerto " + puerto
+ " se ha registrado.");
    } catch (IOException e) {
        e.printStackTrace();
    }
}

// Método para desregistrar un servidor (llamado desde el servidor de
panadería)
public static void desregistrarServidor(int puerto) {
    try (Socket socket = new Socket("192.168.100.12",
PUERTO_DESCUBRIMIENTO);
        PrintWriter out = new PrintWriter(socket.getOutputStream(),
true)) {
        // Desregistrar el servidor de panadería
        servidoresPanaderia.remove("panaderia" + puerto);
        System.out.println("Servidor de panadería desregistrado en el
puerto: " + puerto);
        out.println("Desregistro exitoso para el servidor en el puerto:
" + puerto);
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

```
}

// Método para obtener los servidores registrados (opcional)
public static Map<String, Integer> obtenerServidoresRegistrados() {
    return new HashMap<>(servidoresPanaderia); // Retorna una copia del
mapa para evitar cambios directos
}
}
```