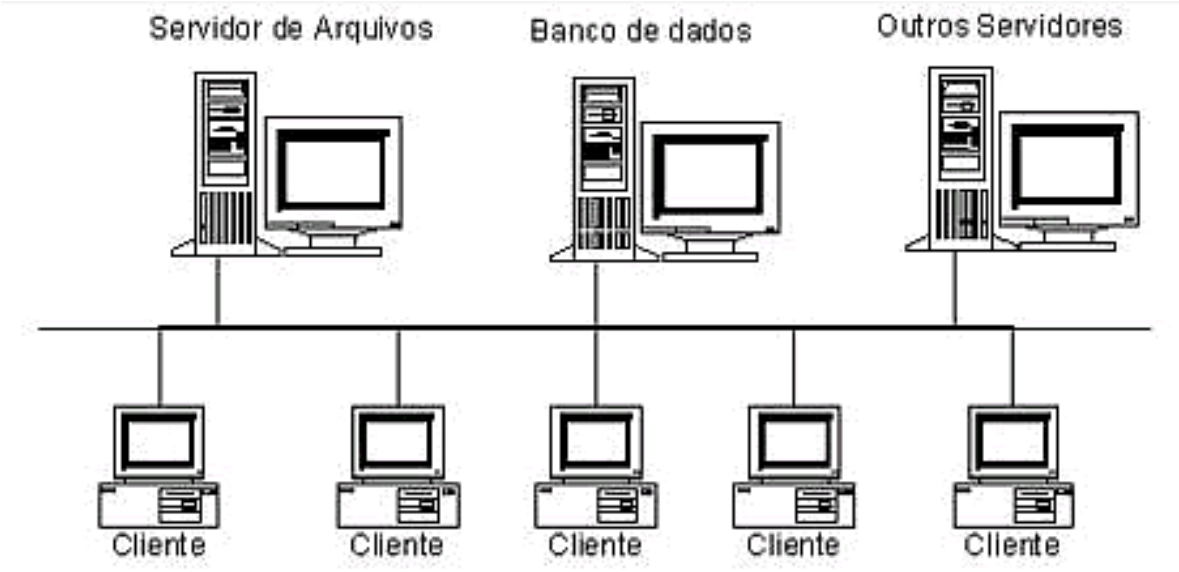
INSTITUTO POLITÉCNICO NACIONAL

Escuela Superior de Cómputo

Sistemas Distribuidos

Profesor: Chadwick Carreto Arrellano



7CM1

Práctica 4 “Múltiples Clientes, múltiples servidores”

Juan Fernando León Medellín

ANTECEDENTE

En los sistemas **distribuidos**, el uso de **sockets** para la implementación de sistemas cliente-servidor es una práctica comúnmente utilizada. A lo largo de los años, la implementación de **múltiples servidores** y **múltiples clientes** ha sido una solución eficiente para manejar un gran volumen de conexiones simultáneas y distribuir la carga de trabajo de manera equilibrada entre varios nodos.

1. Modelos Cliente-Servidor

El **modelo cliente-servidor** es un diseño arquitectónico ampliamente utilizado en la informática para distribuir tareas y servicios entre los proveedores de recursos o servicios (servidores) y los consumidores de esos recursos (clientes). Este modelo es fundamental en aplicaciones de redes, como **sitios web**, **sistemas de mensajería instantánea**, **juegos multijugador en línea**, **sistemas bancarios**, y muchas otras aplicaciones modernas que requieren interacción en tiempo real.

En un sistema cliente-servidor tradicional, un solo servidor está encargado de manejar todas las solicitudes de los clientes. Sin embargo, este modelo puede ser ineficiente si el número de clientes es alto, ya que la carga en el servidor aumenta de forma significativa.

2. Múltiples Servidores y Balanceo de Carga

La necesidad de **escalar** un sistema cliente-servidor para manejar un mayor número de usuarios llevó a la adopción de la arquitectura **multi-servidor**. En este tipo de arquitectura, el servidor principal o **dispatcher** recibe las solicitudes de los clientes y las distribuye entre varios servidores secundarios, de acuerdo con un algoritmo de **balanceo de carga**. Esta técnica permite distribuir las solicitudes de manera equilibrada y evita que un solo servidor se sobrecargue, mejorando el rendimiento general del sistema.

El **balanceo de carga** puede realizarse de distintas formas, entre ellas:

- **Round-robin**: Donde las solicitudes se distribuyen de manera secuencial entre los servidores disponibles.
- **Least Connections**: En el que la solicitud se dirige al servidor con la menor cantidad de conexiones activas.
- **IP Hash**: Donde la solicitud se asigna a un servidor en función de una clave hash derivada de la dirección IP del cliente.

3. Concurrencia y Uso de Hilos

Uno de los mayores desafíos al manejar múltiples clientes en un sistema distribuido es la **concurrencia**. Cada cliente debe ser atendido de manera simultánea sin interferir con otros. Para lograr esto, se utilizan **hilos** (threads) en los servidores, permitiendo que cada cliente se maneje en un hilo independiente. Esta técnica es crucial para la **escalabilidad** y **desempeño** de los sistemas multi-cliente, ya que permite que múltiples solicitudes sean procesadas al mismo tiempo.

4. Sockets en Java

Java es un lenguaje de programación ampliamente utilizado en el desarrollo de aplicaciones de red debido a su robustez, facilidad para manejar conexiones de red y soporte de hilos. El uso de **sockets** en Java permite que los programadores creen aplicaciones que pueden comunicarse entre sí a través de una red, sin importar el sistema operativo que estén utilizando.

- El **Socket** en Java proporciona una interfaz de bajo nivel para la comunicación entre máquinas a través de redes, permitiendo la transmisión de datos en ambas direcciones (cliente-servidor).
- **ServerSocket**, por su parte, se utiliza en el servidor para escuchar y aceptar conexiones entrantes de los clientes.

5. Aplicaciones Prácticas y Casos de Uso

Las prácticas de implementación de sistemas cliente-servidor multi-cliente y multi-servidor se utilizan en una variedad de aplicaciones modernas:

- **Sistemas de mensajería instantánea** (por ejemplo, WhatsApp, Telegram) en donde múltiples usuarios se comunican con el servidor central.
- **Aplicaciones de comercio electrónico** (como Amazon), donde los usuarios envían solicitudes de compra, y los servidores gestionan los pedidos y las transacciones.
- **Juegos en línea multijugador**, donde miles de jugadores interactúan con el servidor para obtener información en tiempo real sobre su entorno.
- **Sistemas bancarios** y **sistemas de reservas** donde múltiples clientes realizan transacciones concurrentemente.

PLANTEAMIENTO DEL PROBLEMA

En la práctica a realizar, se busca implementar un sistema de **comunicación cliente-servidor** utilizando **sockets en Java** con el enfoque de **múltiples servidores y múltiples clientes**. Este tipo de arquitectura es común en sistemas distribuidos donde múltiples clientes necesitan interactuar con varios servidores para cumplir con tareas específicas de procesamiento de datos. La práctica tiene como objetivo comprender cómo manejar de manera eficiente múltiples conexiones simultáneas y balancear las cargas entre servidores de forma dinámica.

Los clientes, en este caso, se conectarán a un servidor principal que se encarga de **redirigir las solicitudes a diferentes servidores secundarios**. Estos servidores secundarios serán responsables de procesar los mensajes enviados por los clientes y devolver una respuesta.

El problema específico a resolver es el siguiente:

1. **Conexión de múltiples clientes**: Se busca que varios clientes puedan conectarse al mismo tiempo y realizar una solicitud al servidor principal sin afectar el rendimiento del sistema.
2. **Redirección de clientes a servidores secundarios**: El servidor principal debe ser capaz de redirigir las solicitudes de los clientes a diferentes servidores secundarios, distribuyendo la carga de manera equilibrada. Esto asegura que no haya un solo servidor sobrecargado y permite una mejor utilización de los recursos disponibles.
3. **Comunicación eficiente entre cliente y servidor**: Se debe garantizar que los mensajes entre los clientes y los servidores sean enviados y recibidos correctamente, y que la respuesta generada por el servidor secundario sea adecuada para el cliente.
4. **Escalabilidad del sistema**: El sistema debe ser escalable, permitiendo agregar más servidores secundarios según sea necesario, para manejar una mayor cantidad de clientes de manera eficiente.
5. **Sincronización y manejo de hilos**: Cada cliente debe ser atendido de manera concurrente sin interferir con otros clientes. Para lograr esto, se utilizarán **hilos** en el servidor principal y en los servidores secundarios, asegurando que múltiples clientes sean atendidos simultáneamente sin problemas de sincronización.

PROPUESTA DE SOLUCIÓN

La propuesta de solución para la práctica de **comunicación cliente-servidor con múltiples clientes y múltiples servidores** se basa en la implementación de una arquitectura distribuida utilizando **sockets** en Java. El objetivo principal es distribuir eficientemente las solicitudes de los clientes a varios servidores, asegurando una **alta disponibilidad**, **escalabilidad** y **balanceo de carga**. Esta arquitectura permitirá que múltiples clientes puedan interactuar con el sistema de manera simultánea sin afectar el rendimiento global del servidor.

A continuación, se detalla la solución propuesta para los distintos aspectos del problema planteado:

---

1. Implementación de un Servidor Principal (Dispatcher)

El servidor principal, también conocido como **dispatcher**, se encargará de gestionar las conexiones de los clientes y de **redirigir** estas solicitudes hacia varios servidores secundarios. Este servidor actuará como un **punto de entrada único** para los clientes, gestionando las conexiones entrantes y distribuyéndolas a los servidores disponibles según un algoritmo de balanceo de carga.

- **Escucha de conexiones**: El servidor principal escuchará las solicitudes de los clientes en un puerto determinado.
- **Distribución de clientes**: Una vez que un cliente se conecta, el servidor principal redirige la solicitud a uno de los servidores secundarios disponibles. Este proceso se puede realizar utilizando un algoritmo de balanceo de carga (como Round-Robin).

Ventajas:

- Asegura que el servidor principal no se sobrecargue.
- Permite que múltiples servidores manejen la carga de trabajo de manera equitativa.
- Facilita la implementación de nuevos servidores según la demanda.

2. Implementación de Múltiples Servidores Secundarios

Los **servidores secundarios** serán responsables de procesar las solicitudes enviadas por el servidor principal. Cada servidor secundario manejará las solicitudes de un cliente específico y devolverá la respuesta correspondiente.

- **Concurrencia de solicitudes**: Cada servidor secundario debe ser capaz de manejar múltiples conexiones de clientes de manera concurrente. Esto se logra mediante el uso de **hilos** (threads) que permiten que cada cliente sea atendido en su propio hilo sin interferir con otros clientes.

Ventajas:

- Mejora el rendimiento al procesar solicitudes de manera simultánea.
- Optimiza el uso de recursos del servidor.
- La infraestructura de múltiples servidores proporciona redundancia y alta disponibilidad.

---

3. Uso de Hilos para Manejo Concurrente

Para permitir que el servidor principal y los servidores secundarios puedan manejar múltiples conexiones simultáneas, se utilizarán **hilos**. Cada solicitud de un cliente se procesará en su propio hilo, lo que permite que el servidor continúe atendiendo otras solicitudes sin bloqueos.

- **Servidor principal**: El servidor principal puede manejar múltiples clientes a través de **hilos de aceptación de conexiones**, donde cada cliente es atendido de manera concurrente en un hilo independiente.
- **Servidores secundarios**: Al recibir la solicitud de un cliente, los servidores secundarios también asignarán un hilo para manejar la comunicación con ese cliente.

Ventajas:

- El uso de hilos mejora la eficiencia y permite manejar grandes volúmenes de solicitudes.
- La aplicación se vuelve más escalable, ya que cada nuevo cliente no afecta el rendimiento global del servidor.

4. Balanceo de Carga

El servidor principal implementará un algoritmo de **balanceo de carga** que distribuye las solicitudes de manera equitativa entre los servidores secundarios. Existen diferentes métodos para balancear la carga, y la elección del algoritmo dependerá de los requerimientos de rendimiento y disponibilidad:

- **Round-Robin**: Este algoritmo distribuye las conexiones de manera equitativa entre los servidores disponibles en un orden secuencial.
- **Least Connections**: Este algoritmo distribuye las solicitudes al servidor con el menor número de conexiones activas, lo que ayuda a equilibrar la carga de manera más dinámica.
- **Random**: Asigna aleatoriamente las solicitudes a los servidores disponibles.

Ventajas:

- El balanceo de carga evita que un solo servidor se sobrecargue y asegura que todos los servidores disponibles reciban solicitudes de manera equitativa.

---

5. Comunicación Cliente-Servidor

La comunicación entre los clientes y los servidores se realiza mediante **sockets**, donde los clientes envían solicitudes al servidor y esperan una respuesta. El proceso es el siguiente:

1. **Cliente**: Inicia una conexión con el servidor principal, enviando su solicitud.
2. **Servidor principal**: Recibe la solicitud, redirige la solicitud a un servidor secundario y espera la respuesta.
3. **Servidor secundario**: Procesa la solicitud, genera la respuesta y la envía de vuelta al servidor principal.
4. **Cliente**: Recibe la respuesta del servidor y termina la comunicación.

Ventajas:

- La comunicación mediante sockets permite interacciones en tiempo real entre los clientes y los servidores.
- Es una solución sencilla y flexible para implementar en redes locales o distribuidas.

Cliente Bancario

- Cada cliente podrá conectarse al servidor e ingresar su nombre y la cantidad que desea retirar.
- Si el retiro es exitoso, el servidor enviará un mensaje de confirmación con el nuevo saldo disponible.
- Si el saldo es insuficiente, el cliente recibirá un mensaje indicando que no puede realizar el retiro.

Manejo de Concurrencia y Sincronización

- Se utilizarán **hilos (Threads)** para manejar múltiples clientes de manera concurrente sin bloquear el servidor.
- Se aplicará **sincronización (synchronized)** en el acceso al saldo de la cuenta para evitar inconsistencias en los retiros simultáneos.
- Se implementará un **contador de clientes conectados** para evitar que más de tres clientes accedan simultáneamente.

Flujo de Funcionamiento

1. El servidor inicia y espera conexiones en el puerto 1234.
2. Un cliente se conecta e ingresa su nombre y la cantidad a retirar.

   El servidor valida la solicitud:

   - Si hay saldo suficiente, se realiza el retiro y se envía un mensaje de confirmación.
   - Si el saldo es insuficiente, se notifica al cliente.
3. El servidor permite que hasta **tres clientes** realicen retiros al mismo tiempo.
4. Una vez que un cliente finaliza su operación, el servidor queda disponible para nuevas conexiones.

DESARROLLO DE LA SOLUCIÓN

En el siguiente enlace se puede encontrar el código de la práctica:

https://github.com/JFernandoLe/SD_PRACTICA4.git

RESULTADOS

Cliente 1. El desea realizar una transferencia bancaria y vemos como el servidor con puerto 6000 realizó esa transacción.

```
PS C:\Users\leonm\OneDrive\Escritorio\documentos\6 semestre\Sistemas Distribuidos\Practica4\src> java Cliente
Conectado al Servidor Principal.
Introduce un mensaje: Hola, necesito una tranferencia
Respuesta del servidor: Servidor 6000 recibió: Hola, necesito una tranferencia
PS C:\Users\leonm\OneDrive\Escritorio\documentos\6 semestre\Sistemas Distribuidos\Practica4\src>
```

Cliente 2. Él desea realizar un depósito bancario y vemos como el servidor con puerto 6001 realizó esa operación.

```
PS C:\Users\leonm\OneDrive\Escritorio\documentos\6 semestre\Sistemas Distribuidos\Practica4\src> java Cliente
Conectado al Servidor Principal.
Introduce un mensaje: Hola, necesito realizar un deposito bancario
Respuesta del servidor: Servidor 6001 recibió: Hola, necesito realizar un deposito bancario
PS C:\Users\leonm\OneDrive\Escritorio\documentos\6 semestre\Sistemas Distribuidos\Practica4\src>
```

Servidor principal. Este servidor se encarga de balancear la carga de los clientes entre los servidores.

```
PS C:\Users\leonm\OneDrive\Escritorio\documentos\6 semestre\Sistemas Distribuidos\Practica4\src> java ServidorPrincipal
Servidor Principal escuchando en el puerto 5000
Cliente conectado desde: /127.0.0.1
Cliente conectado desde: /127.0.0.1
```

Servidor secundario 1. Este servidor procesa la operación del cliente 1.

```
PS C:\Users\leonm\OneDrive\Escritorio\documentos\6 semestre\Sistemas Distribuidos\Practica4\src> java ServidorSecundario 6000
Servidor Secundario escuchando en el puerto 6000
Procesando mensaje: Hola, necesito una tranferencia
```

Servidor secundario 2. Este servidor se encarga de la operación del cliente 2.

```
PS C:\Users\leonm\OneDrive\Escritorio\documentos\6 semestre\Sistemas Distribuidos\Practica4\src> java ServidorSecundario 6001
Servidor Secundario escuchando en el puerto 6001
Procesando mensaje: Hola, necesito realizar un deposito bancario
```

CONCLUSIÓN

La implementación de un sistema cliente-servidor con múltiples clientes y múltiples servidores es una solución eficiente para manejar aplicaciones distribuidas que requieren escalar y gestionar múltiples solicitudes de manera simultánea. A través de la utilización de **sockets** en Java y **hilos** para la concurrencia, se logra una comunicación eficiente y el procesamiento de varias solicitudes sin afectar el rendimiento global del sistema.

El uso de un **servidor principal (dispatcher)** para gestionar las conexiones entrantes y distribuirlas entre los **servidores secundarios** es esencial para garantizar un **balanceo de carga** adecuado. Este enfoque optimiza el uso de recursos y asegura que la infraestructura del sistema pueda manejar un alto volumen de conexiones de manera equilibrada, evitando la sobrecarga de un solo servidor.

Además, la flexibilidad y escalabilidad del sistema permiten agregar servidores adicionales según sea necesario, adaptándose a la demanda y asegurando que el rendimiento no se vea comprometido con el aumento de la cantidad de usuarios.