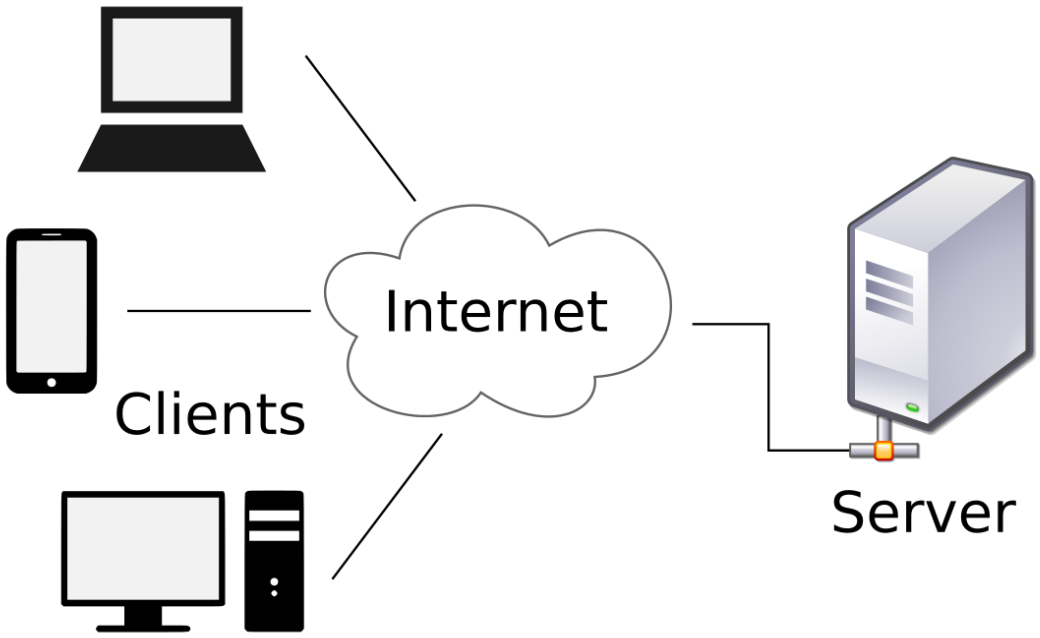
INSTITUTO POLITÉCNICO NACIONAL

Escuela Superior de Cómputo

Sistemas Distribuidos

Profesor: Chadwick Carreto Arrellano



7CM1

Práctica 3 Múltiples C un Servidor

Juan Fernando León Medellín

ANTECEDENTE

1. Introducción

En el desarrollo de aplicaciones de redes, es fundamental comprender la arquitectura cliente-servidor y cómo manejar múltiples conexiones simultáneas. En esta práctica, se implementa un servidor multicliente en **Java** que gestiona solicitudes de retiro de dinero de una cuenta compartida.

El enfoque utilizado es **programación concurrente** con **hilos (Threads)** para permitir que varios clientes interactúen con el servidor sin bloquearse entre sí. Además, se emplea sincronización para evitar problemas de concurrencia en el acceso al saldo de la cuenta bancaria.

2. Comunicación Cliente-Servidor

2.1 Modelo Cliente-Servidor

El modelo **cliente-servidor** es un paradigma de comunicación en el que una entidad (cliente) solicita servicios a otra entidad (servidor), que los proporciona. Es ampliamente utilizado en aplicaciones de redes, desde servidores web hasta sistemas bancarios.

En este caso, el servidor es responsable de recibir solicitudes de retiro de los clientes, verificar la disponibilidad de fondos y responder con el resultado de la operación.

2.2 Sockets en Java

Los **sockets** son puntos finales de comunicación entre dos programas que se ejecutan en una red. En Java, se implementan mediante las clases:

- ServerSocket → Permite que un servidor escuche conexiones entrantes.
- Socket → Representa la conexión entre un cliente y el servidor.
- BufferedReader y PrintWriter → Facilitan la lectura y escritura de datos en la conexión.

El servidor acepta conexiones en un puerto específico, y los clientes se conectan a ese puerto para enviar y recibir datos.

3. Concurrencia y Sincronización

3.1 Programación Concurrente con Hilos (Threads)

Para manejar múltiples clientes simultáneamente sin bloquear el servidor, se utiliza **programación concurrente** mediante **hilos (Threads)**. Cada cliente es manejado en un hilo independiente, lo que permite que el servidor atienda varias solicitudes al mismo tiempo.

```
new Thread(new ManejadorCliente(cliente)).start();
```

Cada vez que un cliente se conecta, el servidor crea un **nuevo hilo** para procesar su solicitud.

3.2 Sincronización con synchronized

Cuando varios clientes intentan retirar dinero de la misma cuenta, puede haber problemas de **condiciones de carrera** (race conditions), donde múltiples hilos modifican el saldo simultáneamente y causan inconsistencias. Para evitar esto, se usa **synchronization** en Java:

```
synchronized (cuenta) {
  if (cuenta.retirar(cantidad)) {
    // Operación exitosa
  } else {
    // Saldo insuficiente
  }
}
```

Esto garantiza que solo **un cliente a la vez** pueda acceder y modificar el saldo.

4. Límite de Clientes y Manejo de Recursos

Para evitar que el servidor se sobrecargue, se establece un límite de **tres clientes simultáneos**. Cuando el límite se alcanza, nuevas conexiones son rechazadas hasta que un cliente finalice su sesión.

Además, se maneja adecuadamente el cierre de conexiones y la liberación de recursos para evitar fugas de memoria y errores en la comunicación.

PLAMTEAMIENTO DEL PROBLEMA

En los sistemas bancarios en línea, es fundamental permitir que múltiples clientes realicen transacciones simultáneamente sin que ello afecte la integridad del saldo de las cuentas. Un problema común en estos sistemas es la concurrencia, donde varios usuarios intentan acceder y modificar el mismo recurso (saldo bancario) al mismo tiempo, lo que puede generar inconsistencias.

Actualmente, en una implementación básica de un servidor bancario, si varios clientes intentan retirar dinero de una cuenta compartida al mismo tiempo, pueden ocurrir errores como:

- Condiciones de carrera (race conditions), donde dos clientes pueden retirar dinero al mismo tiempo sin que el sistema actualice correctamente el saldo.
- Inconsistencia en los datos, cuando múltiples retiros superan el saldo disponible debido a la falta de sincronización en el acceso al recurso compartido.
- Bloqueo del servidor, si no se maneja adecuadamente la concurrencia y el servidor no puede atender múltiples clientes a la vez.

Para abordar este problema, se requiere el desarrollo de un servidor multicliente en Java que permita a hasta tres clientes conectarse simultáneamente y realizar retiros de una cuenta bancaria compartida, garantizando la consistencia del saldo y evitando accesos simultáneos descontrolados.

PROPUESTA DE SOLUCIÓN

Para solucionar el problema de concurrencia en el retiro de dinero, se desarrollará un **servidor multicliente en Java** que permita la conexión de hasta **tres clientes simultáneos**. Se utilizarán **sockets** para la comunicación entre el servidor y los clientes, y **hilos (Threads)** para manejar múltiples conexiones de manera concurrente.

El acceso al saldo de la cuenta será controlado mediante **sincronización (synchronized)**, evitando condiciones de carrera y asegurando que las transacciones sean procesadas correctamente.

Componentes de la Solución

- Se implementará un **servidor Java** que escuche en un puerto específico (por ejemplo, 1234).
- Cuando un cliente se conecte, el servidor creará un **hilo (Thread)** para manejar su solicitud de manera independiente, permitiendo que otros clientes puedan conectarse simultáneamente.
- Se establecerá un límite de **tres clientes conectados al mismo tiempo**.

Gestión del Saldo Compartido

- Se creará una clase **CuentaBancaria** que almacene el saldo y ofrezca un método retirar(int cantidad), el cual estará sincronizado para evitar accesos simultáneos conflictivos.
- Cuando un cliente solicite un retiro, el servidor verificará si hay suficiente saldo antes de autorizar la transacción.

```
public class CuentaBancaria {
  private int saldo;
  public CuentaBancaria(int saldoInicial) {
    this.saldo = saldoInicial;
  }
  public synchronized boolean retirar(int cantidad) {
    if (cantidad > saldo) {
      return false; // Saldo insuficiente
    } else {
      saldo -= cantidad;
      return true; // Retiro exitoso
    }
  }
  public synchronized int getSaldo() {
    return saldo;
  }
}
```

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_PRACTICA3.git

RESULTADOS

Cliente 1:

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program F
Conexión establecida con el servidor...
Escribe tu nombre y presiona la tecla <enter>
Fernando
Coloca la cantidad de dinero a retirar
200
Recibiendo mensaje desde el servidor:
Hola Fernando, retiro exitoso de 200 pesos. Saldo restante: 800

Process finished with exit code 0
```

Cliente 2:

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Progra
Conexión establecida con el servidor...
Escribe tu nombre y presiona la tecla <enter>
Andrea
Coloca la cantidad de dinero a retirar
150
Recibiendo mensaje desde el servidor:
Hola Andrea, retiro exitoso de 150 pesos. Saldo restante: 650

Process finished with exit code 0
```

Cliente 3:

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program
Conexión establecida con el servidor...
Escribe tu nombre y presiona la tecla <enter>
Uriel
Coloca la cantidad de dinero a retirar
500
Recibiendo mensaje desde el servidor:
Hola Uriel, retiro exitoso de 500 pesos. Saldo restante: 150

Process finished with exit code 0
```

Servidor

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\
Servidor iniciado en el puerto 1234
Cliente conectado desde /127.0.0.1:61293
Cliente conectado desde /127.0.0.1:61320
Cliente [Fernando] retiró: 200 pesos. Saldo restante: 800
Cliente [Andrea] retiró: 150 pesos. Saldo restante: 650
Cliente conectado desde /127.0.0.1:61710
Cliente [Uriel] retiró: 500 pesos. Saldo restante: 150
```

CONCLUSIÓN

En esta práctica, se ha desarrollado un servidor multicliente en Java que permite a tres clientes simultáneamente realizar retiros de una cuenta bancaria compartida, asegurando la correcta actualización del saldo mediante el uso de sincronización.

Se logró implementar un modelo cliente-servidor con concurrencia, en el que cada cliente es manejado en un hilo independiente, permitiendo que el servidor pueda atender varias solicitudes al mismo tiempo sin bloquearse. Además, se aplicaron mecanismos de control de acceso al recurso compartido (el saldo de la cuenta), evitando condiciones de carrera y asegurando que las transacciones sean seguras y consistentes.