INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

UNIDAD DE APRENDIZAJE

SISTEMAS DISTRIBUIDOS

PRACTICA No. 3

MODELO MULTICLIENTE - SERVIDOR

ALUMNO

GÓMEZ GALVAN DIEGO YAEL

GRUPO

7CM1

PROFESOR

CARRETO ARELLANO CHADWICK

FECHA DE ENTREGA

10 DE MARZO DE 2025

INDICE

ANTESCEDENTES

El modelo Cliente – Servidor ha sido una arquitectura fundamental en el desarrollo de sistemas informáticos y redes de comunicación desde los años 80, con la expansión de las redes de computadoras y el crecimiento del internet. Su principal propósito es permitir la comunicación entre múltiples clientes y un servidor centralizado, donde el servidor gestiona los recursos y responde a las solicitudes de los clientes. Este modelo se popularizo con la aparición de las primeras aplicaciones de bases de datos centralizadas y sistemas de información distribuidos, evolucionando posteriormente con la llegada de la web y los servicios en la nube.

Inicialmente, la arquitectura Cliente – Servidor estaba diseñada para manejar una única conexión a la vez, procesando una solicitud a la vez antes de aceptar otra conexión. Sin embargo, con el crecimiento del acceso a internet y la demanda de sistemas capaces de atender múltiples usuarios simultáneamente, surgió la necesidad de extender este modelo a un esquema multicliente, donde un solo servidor puede gestionar múltiples conexiones concurrentes. Para lograr esto, se han desarrollado diversas estrategias como el uso de hilos, donde cada cliente es manejado de manera independiente sin afectar la experiencia de los demás usuarios,

La implementación de servidores multicliente es esencial en sistemas bancarios, comercio electrónico, servidores web y plataformas de comunicación en tiempo real, ya que permite a los usuarios acceder simultáneamente a los servicios sin interrupciones ni tiempos de espera prolongados. Para garantizar la correcta opción en un entorno multicliente, es crucial implementar mecanismos de sincronización y manejo eficiente de recursos, evitando problemas como condiciones de carrera, accesos concurrentes no controlados y sobrecarga del servidor.

Uno de los métodos más utilizados en la programación de servidores multiclientes es el uso de sockets y programación multihilo, permitiendo que cada cliente tenga una conexión independiente con el servidor. En este contexto, cada cliente que se conecta es atendido por un hilo independiente, asegurando que múltiples usuarios puedan interactuar simultáneamente con el servidor sin afectar su rendimiento global.

El estudio de este modelo es clave en el desarrollo de aplicaciones distribuidas y redes de computadoras, introduciendo conceptos como comunicación a través de sockets, concurrencia, sincronización de datos y escalabilidad en sistemas distribuidos. Estos principios son ampliamente aplicados en servidores modernos como aplicaciones bancarias, videojuegos en línea, servidores web, plataformas de mensajería instantánea.

PLANTEAMIENTO DEL PROBLEMA

En la actualidad, los sistemas informáticos deben gestionar múltiples usuarios simultáneamente, asegurando que cada cliente puede acceder a los servicios sin interferencias ni tiempos de espera excesivos. Este problema se vuelve específicamente relevantes en sistemas bancarios, donde los usuarios requieren realizar operaciones como consulta de saldo, depósitos, retiros y transferencia en tiempo real

Un servidor que maneja solo una conexión a la vez no es eficiente en un entorno donde múltiples usuarios necesitan acceso simultaneo. Si el servidor atiende secuencialmente cada solicitud, se generarán largos tiempo de espera y una mala experiencia de usuario. Esto podría resultar en demoras en las transacciones, afectando la confiabilidad del sistema y la satisfacción de los clientes.

Bajo este contexto, surge la necesidad de desarrollar un sistema que permita la conexión de múltiples clientes simultáneamente, asegurando que cada usuario pueda realizar operaciones bancarias de manera fluida y sin interrupciones. Sin embargo, la implementación de un servidor multicliente con concurrencia introduce nuevos desafíos, como la sincronización de datos compartidos, el manejo eficiente de múltiples conexiones y la seguridad de las transacciones.

Por lo tanto, es necesario explorar una solución que permita gestionar múltiples clientes de manera concurrente, sin afectar la integridad de los datos y garantizando un acceso eficiente a los recursos del sistema

PROPUESTA DE SOLUCIÓN

Para abordar el problema identificado, se implementará un sistema Cliente – Servidor multicliente utilizando Java y sockets. La solución consistirá en lo siguiente:

- ✓ Un servidor que manejara múltiples clientes mediante hilos independientes, asegurando que cada usuario pueda conectarse sin afectar a los demás
- ✓ Clientes concurrentes que podrán autenticarse con un numero de cuenta y NIP, permitiéndoles realizar operaciones como consulta de saldo, depósitos, retiros y transferencias
- ✓ Sincronización de datos compartidos para evitar problemas de concurrencia, asegurando que cada transacción se realice correctamente sin afectar los datos de otros clientes

El sistema utilizara sockets TCP para la comunicación entre clientes y servidor. Cada cliente tendrá una conexión establecida con el servidor y este, a su vez, gestionará cada solicitud a través de hilos individuales. De esta manera, se garantizará que múltiples usuarios puedan interactuar con el servidor sin generar bloqueos ni tiempos de espera innecesarios.

La solución permita explorar conceptos claves como comunicación en red con sockets, concurrencia con hilos, sincronización de datos y administración eficiente de múltiples conexiones en un entorno distribuido

MATERIALES Y METODOS

Para la realización de modelo, requieran los siguientes recursos:

- ✓ Lenguaje de programación: Java
- ✓ Entorno de desarrollo: NetBeans IDE 21
- ✓ Librerías utilizadas:
  - Java.net para el manejo de sockets TCP
  - Java.io para la lectura y escritura de datos entre el cliente y servidor
  - Java.util.concurrent para la gestión de hilos y concurrencia
- ✓ Sistema operativo: Windows 11
- ✓ Protocolo de comunicación: TCP (Transmission Control Protocol), que garantiza una conexión estable entre el cliente y servidor

DESARROLLO DE SOLUCIÓN

Para implementar el modelo Cliente – Servidor con múltiples clientes se desarrolló un sistema que simula el funcionamiento de un cajero automático (ATM), donde múltiples clientes pueden conectarse a un servidor bancario, ingresar sus credenciales y realiza operaciones bancarias como consulta de saldo, envió de depósitos, retiro de dinero y envío de transferencias.

La implementación del sistema se implementó con tres módulos principales: ServidorMulticliente, Cliente, Cuenta Bancaria, donde cada una de las clases tienen funciones específicas dentro del modelo Cliente – Servidor, permitiendo la interacción de múltiples usuarios y el servidor.

Clase Servidor Multicliente

El servidor se encarga de recibir las conexiones de múltiples clientes y gestionarlas de manera concurrente. Para lograr esto, se implementó un ServerSocket que escucha las conexiones entrantes en el puerto 3000.

Cuando un cliente se conecta, el servidor crea un hilo independiente para manejar su sesión, permitiendo que múltiples clientes puedan realizar transacciones simultáneamente sin que uno afecte el rendimiento del otro, tal como se muestra en el código siguiente.

```
public static void main(String[] args) {
    cuentas.put("1234567890", new CuentaBancaria("1234567890", "Juan Perez", 5000.0, 1234));
    cuentas.put("2468024680", new CuentaBancaria("2468024680", "Jesus Gonzalez", 10000.0, 2468));
    cuentas.put("3692581470", new CuentaBancaria("3692581470", "Santiago Muñoz", 6348.3, 3692));
    cuentas.put("4826048260", new CuentaBancaria("4826048260", "Rodrigo Peralta", 1.80, 4826));
    cuentas.put("6284062840", new CuentaBancaria("6284062840", "Ernesto Caballero", 1000000.1, 6284));

    try(ServerSocket ss = new ServerSocket(3000)) {
        System.out.println("Servidor bancario iniciado en el puerto 3000...");
        while (true) {
            Socket cl = ss.accept();
            System.out.println("Cliente conectado desde puerto "+cl.getPort());
            //Creando hilo que atiende a cliente
            new Thread(new MClientes(cl)).start();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Aquí cada cliente recibe un hilo dedicado, lo que permite manejar múltiples clientes de forma concurrente.

Cuando un cliente se conecta, este envía su número de cuenta y NIP para autenticarse. Aquí el servidor valida esa información que se encuentra en el HashMap de cuentas bancarias que funge como base de datos

```java
public void run(){
    try {
        BufferedReader ent = new BufferedReader(new InputStreamReader(cl.getInputStream()));
        PrintWriter sal = new PrintWriter(cl.getOutputStream(), true);
        // Recibiendo número de cuenta
        String numCuenta = ent.readLine();
        // Recibiendo NIP
        int nip = Integer.parseInt(ent.readLine());
        // Validando número de cuenta y NIP
        if (cuentas.containsKey(numCuenta)) {
            CuentaBancaria cuenta = cuentas.get(numCuenta);
            if (cuenta.autenticar(nip)) {
                System.out.println("Cliente "+cuenta.getNumeroCuenta()+" autenticado correctamente");
                sal.println("Bienvenido " + cuenta.getTitular());
                MOperaciones(ent,sal,cuenta);
            } else {
                sal.println("NIP incorrecto.");
            }
        } else {
            sal.println("Número de cuenta no encontrado.");
        }
    } catch (IOException e) {
        System.out.println("Cliente desconectado inesperadamente desde puerto "+cl.getPort());
    } finally {
        try {
            cl.close();
            System.out.println("Cliente desconectado desde puerto "+cl.getPort());
        } catch (IOException e) {
            System.out.println("Error al cerrar la conexión con el cliente.");
        }
    }
}
```

Si la autenticación es exitosa, el servidor permite al cliente continuar con sus operaciones bancarias. En caso de que no, este notifica al cliente el error en la autenticación.

Entre las diversas operaciones que puede realizar el cliente, se encuentra la consulta de saldo en la cuenta, que es seleccionada en el servidor con el envío del comando “C”, con “D” podemos seleccionar la opción de depositar dinero, donde no solicitara el monto a depositar, con “R” podemos seleccionar la opción de retirar dinero, donde nos solicitara el monto a retirar y se validara que la cuenta tenga los fondos suficientes, con “T” podemos seleccionar la opción de transferir dinero, donde nos solicitara el número de cuenta destino y el monto a transferir, validando el saldo permita dicho movimiento y en caso de no querer realizar alguna operación de las antes mencionadas, se encuentra la opción salir con el comando “S”, el cual envía un mensaje de confirmación al cliente y cierra la conexión con él, tal como se muestra a continuación.

```
switch (operacion) {
    case "C":   sal.println("Saldo actual: $" + cuenta.consultarSaldo());
                break;
    case "D":   double montoD = Double.parseDouble(ent.readLine());
                cuenta.depositar(montoD);
                sal.println("Deposito exitoso. Nuevo saldo: $" + cuenta.consultarSaldo());
                break;
    case "R":   sal.println(cuenta.consultarSaldo());
                double montoR = Double.parseDouble(ent.readLine());
                if (montoR > cuenta.consultarSaldo()) {
                    sal.println("Fondos insuficientes. Saldo actual: $" + cuenta.consultarSaldo());
                } else {
                    cuenta.retirar(montoR);
                    sal.println("Retiro exitoso. Nuevo saldo: $" + cuenta.consultarSaldo());
                }
                break;
    case "T":   sal.println(cuenta.consultarSaldo());
                String cuentaDestino = ent.readLine();
                double montoT = Double.parseDouble(ent.readLine());
                if (cuentas.containsKey(cuentaDestino)) {
                    CuentaBancaria destino = cuentas.get(cuentaDestino);
                    if (cuenta.transferir(destino, montoT)) {
                        sal.println("Transferencia exitosa. Nuevo saldo: $" + cuenta.consultarSaldo());
                    } else {
                        sal.println("Saldo insuficiente");
                    }
                } else
                    sal.println("Cuenta de destino no encontrada.");
                break;
    case "S":   sal.println("Sesion finalizada.");
                break;
}
```

Clase Cliente

El cliente es la entidad dentro del modelo Cliente – Servidor encargada de establecer la comunicación con el servidor y permitir la interacción del usuario con el sistema. Su función principal es enviar solicitudes al servidor y recibir las respuestas correspondientes para que el usuario pueda realizar diversas operaciones bancarias-

El cliente inicia estableciendo una conexión con el servidor a través de un socket TCP en el puerto 3000. Para ello, se establece un socket junto con flujos de entrada y salida para el intercambio de información

```
Scanner scan = new Scanner(System.in)){
Socket cl = new Socket("localhost",3000);
BufferedReader ent = new BufferedReader(new InputStreamReader(cl.getInputStream()));
PrintWriter sal = new PrintWriter(cl.getOutputStream(), true);
```

Antes de realizar cualquier operación, el cliente debe autenticarse ingresando su numero de cuenta y NIP, datos que serán enviados a servidor para su validación.

```
//Solicitando numero de cuenta
System.out.print("Ingrese su numero de cuenta: ");
String numCuenta = scan.nextLine();
//Solicitando NIP
System.out.print("Ingrese su NIP: ");
nip = scan.nextInt();
//Enviar info al servidor
sal.println(numCuenta);
sal.println(nip);
sal.flush();
//Recibiendo respueta de servidor
String autenticacion = ent.readLine();
System.out.println("\n"+autenticacion);
//
if(!autenticacion.startsWith("Bienvenido")){
    cl.close();
    return;
}
```

Si la autenticación es correcta, se muestra un menú interactivo donde el cliente puede seleccionar distintas operaciones e interactuar con su cuenta bancaria

```
switch(opc){
    case 1: sal.println("C");
            sal.flush();
            System.out.print("\n");
            break;
    case 2: System.out.print("\nIngrese el monto a depositar: $");
            montoD = scan.nextInt();
            sal.println("D");
            sal.println(montoD);
            sal.flush();
            break;
    case 3: sal.println("R");
            System.out.print("\nSaldo disponible: $"+ent.readLine());
            System.out.print("\nIngrese el monto a retirar: $");
            montoR = scan.nextInt();
            sal.println(montoR);
            sal.flush();
            break;
    case 4: sal.println("T");
            System.out.print("\nSaldo disponible: $"+ent.readLine());
            System.out.print("\nIngrese el el numero de cuenta destino: ");
            scan.nextLine();
            cuentaD = scan.nextLine();
            sal.println(cuentaD);
            sal.flush();
            System.out.print("Ingresa el monto a depositar: $");
            montoD = scan.nextInt();
            sal.println(montoD);
            sal.flush();
            break;
    case 5: sal.println("S");
            sal.flush();
            break;
}
```

Cuando el usuario selecciona la opción de salir, el cliente envía el comando correspondiente y espera su confirmación antes de cerrar la conexión

Clase CuentaBancaria

Esta clase permite gestionar las cuentas bancarias, a través del almacenamiento de los datos de cada usuario y el uso de diversos métodos para realizar las operaciones financieras de manera segura.

Cada cuenta bancaria tiene los siguientes atributos:

- ✓ Número de cuenta
- ✓ Nombre del titular
- ✓ Saldo disponible
- ✓ NIP de acceso

Los métodos implementados en dicha clase son los siguientes:

```
public CuentaBancaria(String numeroCuenta, String titular, double saldo, int nip){
    this.numeroCuenta = numeroCuenta;
    this.titular = titular;
    this.saldo = saldo;
    this.nip = nip;
}
```

Este método es el constructor que se encarga de inicializar los datos de la cuenta

```
public String getTitular(){
    return titular;
}
```

Este método se encarga de devolver el nombre del titular de la cuenta

```
public synchronized String getNumeroCuenta(){
    return numeroCuenta;
}
```

Este método se encarga de devolver el numero de cuenta del usuario

```
public double consultarSaldo(){
    return saldo;
}
```

Este método retorna el saldo actual de la cuenta

```java
public void depositar(double monto){
    if(monto > 0){
        saldo += monto;
        System.out.println("Deposito exitoso. Nuevo saldo $"+saldo);
    }
}
```

Este método aumenta el saldo de la cuenta si el monto ingresado es valido

```java
public void retirar(double monto){
    if(monto < saldo){
        saldo -= monto;
        System.out.println("Retiro exitoso. Nuevo saldo $"+saldo);
    }else{
        System.out.println("La cantidad debe ser menor al saldo total de la cuenta.");
    }
}
```

Este método disminuye el saldo de la cuenta si hay suficientes fondos disponibles

```java
public boolean autenticar(int claveIngresada) {
    return nip==claveIngresada;
}
```

Este método valida que el NIP ingresado sea el correcto para la cuenta

```java
public synchronized boolean transferir(CuentaBancaria cuentaD, double monto) {
    if(monto > 0 && monto <= saldo){
        this.saldo -= monto;
        cuentaD.depositar(monto);
        return true;
    }
    return false;
}
```

Este método permite el envío de dinero entre cuentas. Si el saldo de la cuenta origen es mayor o igual al monto a transferir, se ejecuta la transferencia

RESULTADOS

Al correr el servidor, este se inicializa y se coloca en espera de conexiones de clientes al puerto 3000. Al recibir un cliente, lo muestra en pantalla junto con el puerto por donde se encuentra conectado, después de que este se autentique, el servidor imprime su número de cuenta que es su identificador único, para que después de realizar las operaciones que requiera y concluya el servicio, este muestre en pantalla el cliente que se desconecto

```
Servidor bancario iniciado en el puerto 3000...
Cliente conectado desde puerto 52664
Cliente conectado desde puerto 52666
Cliente 1234567890 autenticado correctamente
Cliente desconectado desde puerto 52664
```

El servidor permite conectar múltiples clientes, como se muestra a continuación donde se conectaron dos clientes y se autenticaron correctamente.

```
Servidor bancario iniciado en el puerto 3000...
Cliente conectado desde puerto 52697
Cliente conectado desde puerto 52698
Cliente 1234567890 autenticado correctamente
Cliente 2468024680 autenticado correctamente
```

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

Bienvenido Juan Perez

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

```
Ingrese su numero de cuenta: 2468024680
Ingrese su NIP: 2468

Bienvenido Jesus Gonzalez

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

A través del menú interactivo, cada cliente puede realizar operaciones sobre su cuenta bancaria

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

Saldo actual: $5000.0
```

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

Ingrese el monto a depositar: $160
Deposito exitoso. Nuevo saldo: $5160.0
```

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

Saldo disponible: $5160.0
Ingrese el monto a retirar: $80
Retiro exitoso. Nuevo saldo: $5080.0
```

Para la opción de transferencia, permite enviar dinero a otras cuentas, donde se actualiza en tiempo real los saldos de la cuenta, como se muestra a continuación:

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

Saldo disponible: $5080.0
Ingrese el el numero de cuenta destino: 2468024680
Ingresa el monto a depositar: $2500
Transferencia exitosa. Nuevo saldo: $2580.0
```

En la imagen anterior, el cliente Juan Pérez realiza una transferencia a la cuenta de Jesús González de $2,500 pesos

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

Saldo actual: $10120.0
```

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

Saldo actual: $12620.0
```

En la imagen anterior se muestra el saldo de la cuenta antes y después de recibir la transferencia realizada. Se observa que la información se actualiza correctamente de manera instantánea, asegurando la sincronización adecuada y evitando inconsistencias en los datos.

Cuando los clientes terminan sus operaciones bancarias y concluyen su sesión, se les envía un mensaje de cierre y se notifica en el servidor su desconexión.

```
Cliente desconectado desde puerto 52717
Cliente desconectado desde puerto 52715

        MENU
1. Consultar saldo
2. Depositar dinero
3. Retirar dinero
4. Tranferencia
5. Salir
Seleccione la opcion deseada: 5
Sesion finalizada.
```

CONCLUSIONES

La implementación del modelo Cliente – Servidor multicliente en esta practica me ha permitido comprender en profundidad el funcionamiento de los sistemas distribuidos y la concurrencia en aplicaciones de red. A diferencia del modelo inicial de un solo cliente y servidor, donde únicamente se atendía una conexión a la vez, en esta versión se logró que múltiples clientes pudieran interactuar simultáneamente con el servidor, permitiendo consultas de saldo, depósitos, retiros y transferencias sin interrupciones.

Uno de los principales aprendizajes que me deja esta práctica fue la implementación de hilos en el servidor para manejar múltiples conexiones en paralelo, asegurando que cada cliente tuviera su propio canal de comunicación sin afectar la experiencia de otros usuarios. Esto mejoro significativamente la escalabilidad y eficiencia del sistema, evitando que un cliente tuviera que esperar a que otro finalizara su operación

Otra mejora clave fue la sincronización de los datos compartidos, utilizando synchronized. Esto garantizo que múltiples clientes pudieran realizar operaciones simultáneamente sin generar inconsistencias en los saldos de las cuentas. Además, se implemento el manejo seguro de conexiones, asegurando que cada cliente cerrara su sesión correctamente al finalizar sus operaciones, evitando problemas como conexiones no liberadas o bloqueos en el sistema

Finalmente, con esta práctica pude tener una compresión mas clara del uso de sockets en Java, el manejo de la concurrencia con hilos y la importancia de la sincronización en aplicaciones distribuidas. Además, se demostró que el modelo Cliente – Servidor multicliente es mucho mas eficiente y realista para cualquier sistema que deba gestionar múltiples usuarios simultáneamente.