INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO

Práctica 3

Sockets TCP con hilos de ejecución Java

Desarrollo de sistemas distribuidos

Sánchez Torres Kevin Alexander

4CM12

Tabla de contenido

Introducción

Los sockets con hilos de ejecución (threads) son una técnica común para manejar múltiples conexiones o tareas en redes de manera concurrente. En este contexto, un socket permite la comunicación entre dos nodos (por ejemplo, entre un cliente y un servidor) en una red, mientras que los hilos de ejecución permiten que un programa ejecute múltiples tareas simultáneamente.

Funcionamiento de los sockets con hilos:

1. Creación del socket: El servidor crea un socket que estará "escuchando" conexiones entrantes de los clientes. El cliente también crea su propio socket para conectarse al servidor.

2. Hilo principal del servidor: El servidor, después de abrir su socket, espera solicitudes de conexión. Una vez que llega una solicitud de un cliente, se puede crear un hilo para manejar esa conexión específica, de manera que el servidor pueda seguir aceptando nuevas solicitudes sin bloquearse.

3. Hilos para cada conexión: Por cada cliente que se conecte al servidor, se crea un hilo independiente. Cada hilo manejará las comunicaciones (enviar y recibir datos) entre el servidor y ese cliente en particular. Esto permite que múltiples clientes se conecten y se comuniquen con el servidor simultáneamente, sin interferir unos con otros.

4. Comunicación concurrente: Los hilos permiten que el servidor maneje múltiples conexiones a la vez, lo que es ideal para

aplicaciones donde varias solicitudes pueden llegar al mismo tiempo (por ejemplo, en un chat o un servidor web).

Ejemplo

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

public class ServidorMultihilo {
    public static void main(String[] args) throws IOException {
        ServerSocket servidorSocket = new ServerSocket(1234);
        System.out.println("Servidor escuchando en el puerto 1234");

        while (true) {
            Socket clienteSocket = servidorSocket.accept();
            System.out.println("Cliente conectado");

            // Crear un hilo para manejar la comunicación con este cliente
            HiloCliente cliente = new HiloCliente(clienteSocket);
            cliente.start();
        }
    }
}

class HiloCliente extends Thread {
    private Socket clienteSocket;
    private PrintWriter out;
    private BufferedReader in;
```

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

    public void run() {
        try {
            out = new PrintWriter(clienteSocket.getOutputStream(), true);
            in = new BufferedReader(new
InputStreamReader(clienteSocket.getInputStream()));

            String inputLine;
            while ((inputLine = in.readLine()) != null) {
                System.out.println("Mensaje recibido: " + inputLine);
                out.println("Respuesta del servidor: " + inputLine);
            }

            in.close();
            out.close();
            clienteSocket.close();

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Explicación del ejemplo:

- **ServerSocket:** El servidor escucha en el puerto 1234.
- **accept():** Espera una conexión de un cliente. Cuando un cliente se conecta, se crea un nuevo Socket.
- **Hilos de cliente:** Cada vez que un cliente se conecta, se crea un nuevo hilo (HiloCliente) para manejar la comunicación con ese cliente.
- **run():** Cada hilo ejecuta su código de manera independiente, permitiendo que múltiples clientes puedan ser atendidos simultáneamente.

Ventajas de usar hilos con sockets:

- **Eficiencia:** El servidor no se bloquea mientras atiende a un cliente. Otros clientes pueden ser atendidos simultáneamente.
- **Escalabilidad:** Se pueden manejar muchas conexiones concurrentes.

Consideraciones:

- **Sincronización:** Al usar hilos, es necesario tener cuidado con las condiciones de carrera (race conditions) si los hilos comparten recursos.
- **Sobrecarga de hilos:** Crear un hilo para cada conexión puede sobrecargar el sistema si el número de clientes es muy alto. Para evitar esto, se pueden usar pools de hilos (thread pools) que limitan la cantidad de hilos simultáneos.

Objetivo

El objetivo de este proyecto es implementar un sistema cliente-servidor que permita realizar operaciones matemáticas básicas (suma, resta, multiplicación y división) de manera concurrente utilizando sockets y hilos de ejecución en Java.

Problemática

En muchos sistemas cliente-servidor, la concurrencia es un reto importante, especialmente cuando múltiples clientes intentan conectarse simultáneamente. Sin el uso de hilos, un servidor puede bloquearse

mientras atiende a un cliente, impidiendo que otros puedan conectarse y ser atendidos al mismo tiempo. Esta falta de concurrencia reduce la eficiencia y escalabilidad del sistema. El reto de este proyecto es implementar un sistema que pueda gestionar múltiples conexiones de manera simultánea sin sacrificar rendimiento o precisión en los cálculos.

Solución

Para implementar hilos de ejecución se crea un objeto implementando la palabra clave “Thread”, así, cada vez que un usuario se conecte al servidor, se creara un hilo.

Servidor

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

public class Servidor {
    public static void main(String[] args) {
        try (ServerSocket serverSocket = new ServerSocket(12345)) {
            System.out.println("Servidor iniciado...");
            while (true) {
                Socket socket = serverSocket.accept();
                new Thread(new ManejadorCliente(socket)).start();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

class ManejadorCliente implements Runnable {
    private Socket socket;

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

    @Override
    public void run() {
        System.out.println("Nuevo cliente conectado: " + socket.getInetAddress());
        try (BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true)) {
            String inputLine;
            while ((inputLine = in.readLine()) != null) {
                if (inputLine.equalsIgnoreCase("salir")) {
                    break;
                }
                String[] parts = inputLine.split(" ");
                double num1 = Double.parseDouble(parts[0]);
                double num2 = Double.parseDouble(parts[1]);
                String operador = parts[2];
                double resultado = 0;

                switch (operador) {
                    case "+":
                        resultado = num1 + num2;
                        break;
                    case "-":
                        resultado = num1 - num2;
                        break;
```

Figura 1. Socket servidor 1.

```
                    case "*":
                        resultado = num1 * num2;
                        break;
                    case "/":
                        if (num2 != 0) {
                            resultado = num1 / num2;
                        } else {
                            out.println("Error: Division por cero");
                            continue;
                        }
                        break;
                    default:
                        out.println("Operador no válido");
                        continue;
                }
                out.println("Resultado: " + resultado);
                System.out.println("Operación realizada: " + num1 + " " + operador + " " + num2 + " = " + resultado);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            System.out.println("Cliente desconectado: " + socket.getInetAddress());
        }
    }
}
```

Figura 2. Socket servidor 2.

Cliente

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

public class Cliente {
    public static void main(String[] args) {
        try (Socket socket = new Socket("localhost", 12345);
            BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
            BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in))) {
            String userInput;
            while (true) {
                System.out.print("Ingrese dos números y un operador (+, -, *, /) separados por espacio (o 'salir' para terminar): ");
                userInput = stdIn.readLine();
                out.println(userInput);
                if (userInput.equalsIgnoreCase("salir")) {
                    break;
                }
                System.out.println("Respuesta del servidor: " + in.readLine());
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Figura 3. Socket cliente.

Descripción de la solución

El código va realizando las siguientes tareas.

```java
public class Servidor {
    public static void main(String[] args) {
        try (ServerSocket serverSocket = new ServerSocket(12345)) {
            System.out.println("Servidor iniciado...");
            while (true) {
                Socket socket = serverSocket.accept();
                new Thread(new ManejadorCliente(socket)).start();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Figura 4. Creación de socket e hilo.

- **ServerSocket serverSocket = new ServerSocket(12345):** Se crea un servidor que escucha en el puerto 12345. El ServerSocket permite aceptar conexiones entrantes.
- **while (true):** El servidor entra en un bucle infinito, esperando conexiones de clientes.
- **serverSocket.accept():** Este método bloquea la ejecución hasta que un cliente se conecta. Cuando eso ocurre, se acepta la conexión y se crea un nuevo Socket para interactuar con el cliente.
- **new Thread(new ManejadorCliente(socket)).start():** Cada vez que un cliente se conecta, se crea un nuevo hilo para manejar esa conexión de manera concurrente. La clase ManejadorCliente (ver más abajo) implementa la lógica de interacción con el cliente.

```
class ManejadorCliente implements Runnable {
    private Socket socket;

    public ManejadorCliente(Socket socket) {
        this.socket = socket;
    }
```

Figura 5. Clase ManejadorCliente.

Esta clase maneja la comunicación entre el servidor y cada cliente de forma concurrente. Implementa la interfaz Runnable, lo que le permite ser ejecutada en un hilo.

- ManejadorCliente(Socket socket): Constructor que recibe el Socket que representa la conexión con un cliente.

```java
@Override
public void run() {
    System.out.println("Nuevo cliente conectado: " + socket.getInetAddress());
    try (BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream(
        PrintWriter out = new PrintWriter(socket.getOutputStream(), true)) {
        String inputLine;
        while ((inputLine = in.readLine()) != null) {
            if (inputLine.equalsIgnoreCase("salir")) {
                break;
            }
            String[] parts = inputLine.split(" ");
            double num1 = Double.parseDouble(parts[0]);
            double num2 = Double.parseDouble(parts[1]);
            String operador = parts[2];
            double resultado = 0;

            switch (operador) {
                case "+":
                    resultado = num1 + num2;
                    break;
                case "-":
                    resultado = num1 - num2;
                    break;
                case "*":
                    resultado = num1 * num2;
                    break;
                case "/":
                    if (num2 != 0) {
                        resultado = num1 / num2;
                    } else {
                        out.println("Error: División por cero");
```

Figura 6. Operaciones.

- **run():** Método que se ejecuta en un hilo separado cuando se inicia el cliente. Se encarga de manejar la comunicación con el cliente.
- **BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream())):** Se crea un BufferedReader para leer los datos que el cliente envía al servidor.

- **PrintWriter out = new PrintWriter(socket.getOutputStream(), true):** Se crea un PrintWriter para enviar datos del servidor al cliente.
- **while ((inputLine = in.readLine()) != null):** Bucle que escucha y procesa las operaciones matemáticas que envía el cliente.
- inputLine.equalsIgnoreCase("salir"): Si el cliente envía la palabra "salir", se rompe el bucle y se cierra la conexión.
- **inputLine.split(" "):** Divide la entrada del cliente en tres partes: los dos números y el operador.
- **switch (operador):** Realiza la operación matemática solicitada (suma, resta, multiplicación, división).

```
public class Cliente {
    public static void main(String[] args) {
        try (Socket socket = new Socket("localhost", 12345);
            BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputSt
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
            BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in))) {
            String userInput;
            while (true) {
                System.out.print("Ingrese dos números y un operador (+, -, *, /) separados
                userInput = stdIn.readLine();
                out.println(userInput);
                if (userInput.equalsIgnoreCase("salir")) {
                    break;
                }
                System.out.println("Respuesta del servidor: " + in.readLine());
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Figura 7. Cliente.

- **Socket socket = new Socket("localhost", 12345)**: El cliente se conecta al servidor en el puerto 12345 de la máquina local (localhost).

- BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream())): Se utiliza para recibir datos del servidor.
- PrintWriter out = new PrintWriter(socket.getOutputStream(), true): Se utiliza para enviar datos al servidor.
- BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in)): Permite leer datos ingresados por el usuario en la consola.
- while (true): Bucle que permite al usuario introducir operaciones repetidamente hasta que escriba "salir".

- stdIn.readLine(): Lee la entrada del usuario (dos números y un operador).
- out.println(userInput): Envía la entrada al servidor.
- if (userInput.equalsIgnoreCase("salir")): Si el usuario escribe "salir", se cierra la conexión.
- System.out.println("Respuesta del servidor: " + in.readLine()): Recibe y muestra el resultado de la operación que envía el servidor.

Resultado

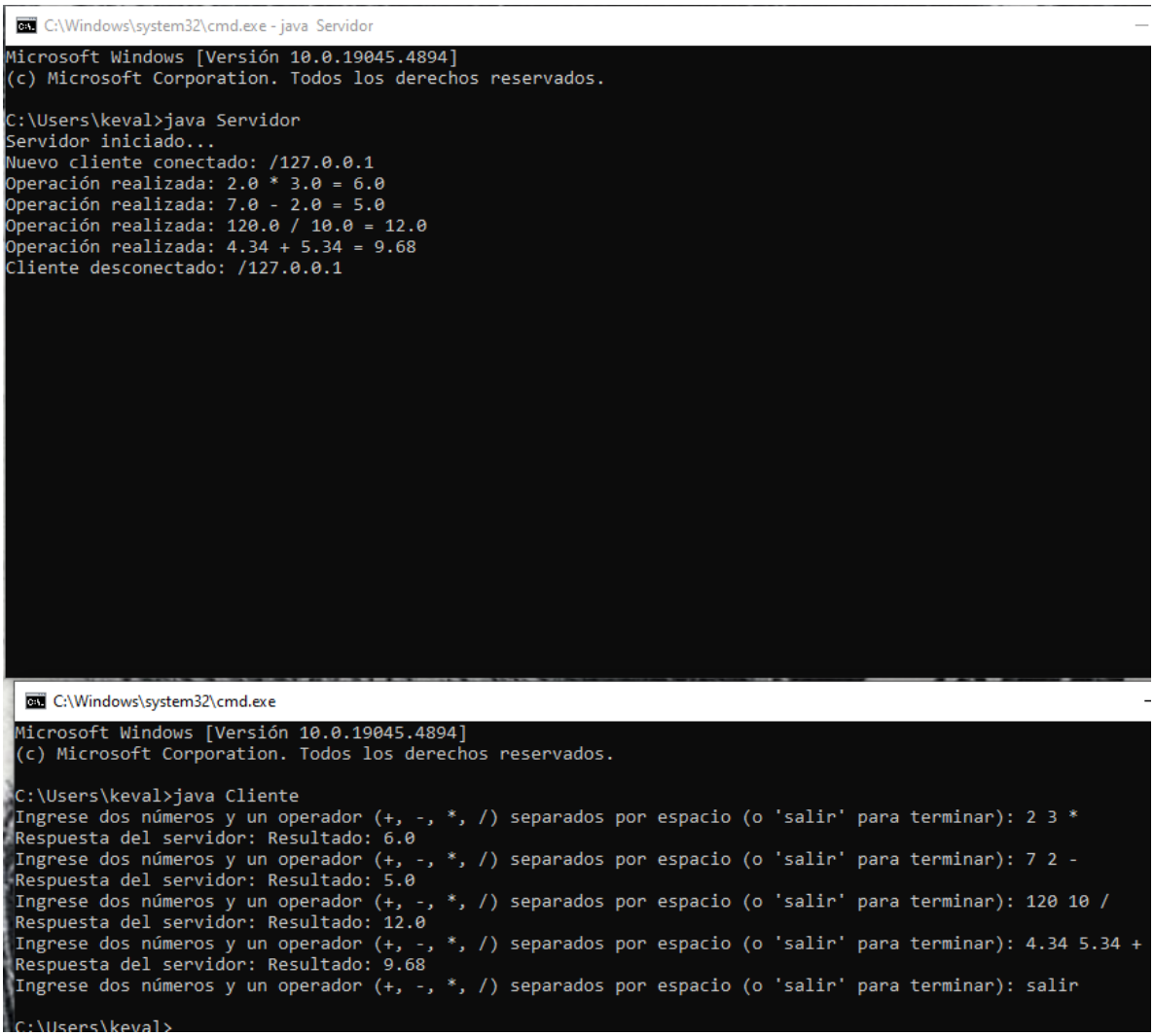

Figura 8. Ejecución.

Conclusión

La implementación de este sistema cliente-servidor concurrente en Java demuestra la eficacia del uso de hilos para manejar múltiples conexiones simultáneamente sin bloquear el servidor. La solución es escalable y eficiente, permitiendo que varios clientes interactúen con el servidor de manera fluida. Además, este proyecto proporciona una base sólida para futuras mejoras o extensiones, como la inclusión de más operaciones matemáticas o el manejo de errores más avanzados. Este enfoque puede aplicarse a sistemas que requieren procesamiento concurrente de

múltiples clientes en tiempo real, como en aplicaciones de chat o sistemas distribuidos.