Implementación de Comunicaciones Simultáneas

Ahora nos centraremos en **cómo gestionar múltiples conexiones simultáneas en un servidor**, asegurando que varios clientes puedan interactuar con él al mismo tiempo. Para ello, desarrollaremos distintas estrategias, desde las más simples hasta las más avanzadas.

Estructura del Tema

1. **Servidor TCP Multicliente con Hilos** *(Lo más básico, cada cliente tiene su propio hilo).*
2. **Optimización con `ThreadPoolExecutor`** *(Manejo eficiente de múltiples clientes sin sobrecargar el servidor).*
3. **Servidor No Bloqueante con `NIO` (Non-blocking I/O)** *(Manejo de cientos de conexiones simultáneamente con pocos recursos).*
4. **Servidor UDP Multicliente** *(Comunicaciones sin conexión persistente, ideal para juegos o streaming).*
5. **Chat en Tiempo Real para múltiples clientes** *(Ejemplo completo para poner en práctica la comunicación simultánea).*

1.- Servidor TCP Multicliente con Hilos

Este método usa **un hilo por cliente**, permitiendo que cada cliente tenga su propia comunicación con el servidor.

Servidor TCP Multicliente

Cada vez que un cliente se conecta, se lanza un nuevo hilo.

```
package com.miapp.servidor;

import java.io.*;

import java.net.ServerSocket;

import java.net.Socket;
```

```
public class ServidorTCP {

    public static void main(String[] args) {

        int puerto = 5000;


        try (ServerSocket serverSocket = new ServerSocket(puerto)) {

            System.out.println("Servidor TCP en espera de
conexiones...");


            while (true) {

                Socket cliente = serverSocket.accept();

                System.out.println("Cliente conectado: " +
cliente.getInetAddress());


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

            }

        } catch (IOException e) {

            e.printStackTrace();

        }

    }

}


class ManejadorCliente implements Runnable {

    private Socket cliente;
```

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

    @Override
    public void run() {
        try (BufferedReader reader = new BufferedReader(new
InputStreamReader(cliente.getInputStream()));
             BufferedWriter writer = new BufferedWriter(new
OutputStreamWriter(cliente.getOutputStream()))) {

            String mensaje;
            while ((mensaje = reader.readLine()) != null) {
                System.out.println("Cliente dice: " + mensaje);
                writer.write("Servidor recibió: " + mensaje + "\n");
                writer.flush();
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                cliente.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
```

```
        }
    }
}
```

Cliente TCP

```
package com.miapp.cliente;

import java.io.*;

import java.net.Socket;


public class ClienteTCP {

    public static void main(String[] args) {

        try (Socket socket = new Socket("localhost", 5000);

             BufferedReader reader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));

             BufferedWriter writer = new BufferedWriter(new
OutputStreamWriter(socket.getOutputStream()));

             BufferedReader teclado = new BufferedReader(new
InputStreamReader(System.in))) {


            System.out.println("Conectado al servidor. Escribe
mensajes:");


            String mensaje;

            while ((mensaje = teclado.readLine()) != null) {

                writer.write(mensaje + "\n");
```

```
                writer.flush();


                String respuesta = reader.readLine();

                System.out.println("Servidor: " + respuesta);

            }

        } catch (IOException e) {

            e.printStackTrace();

        }

    }

}
```

Cada cliente tendrá su propio hilo, pero si hay demasiados clientes, el servidor puede colapsar.

2.- Optimización con `ThreadPoolExecutor`

En vez de crear un hilo por cliente, usamos un pool de hilos reutilizables.

```
package com.miapp.servidor;

import java.io.*;

import java.net.ServerSocket;

import java.net.Socket;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;
```

```
public class ServidorOptimizado {

    private static final int PUERTO = 5000;

    private static final int MAX_CLIENTES = 10;


    public static void main(String[] args) {

        ExecutorService pool =
Executors.newFixedThreadPool(MAX_CLIENTES);


        try (ServerSocket serverSocket = new ServerSocket(PUERTO)) {

            System.out.println("Servidor optimizado en espera de
conexiones...");


            while (true) {

                Socket cliente = serverSocket.accept();

                System.out.println("Cliente conectado: " +
cliente.getInetAddress());

                pool.execute(new ManejadorCliente(cliente));

            }

        } catch (IOException e) {

            e.printStackTrace();

        }

    }

}
```

Esto mejora el rendimiento, pero aún usa hilos bloqueantes.

3.- Servidor No Bloqueante con NIO

Usa `Selector` y `Channel` para manejar cientos de conexiones con pocos recursos.

```
package com.miapp.servidor;


import java.io.IOException;

import java.net.InetSocketAddress;

import java.nio.ByteBuffer;

import java.nio.channels.*;

import java.util.Iterator;


public class ServidorNIO {

    public static void main(String[] args) throws IOException {

        Selector selector = Selector.open();

        ServerSocketChannel serverChannel =
ServerSocketChannel.open();

        serverChannel.bind(new InetSocketAddress(5000));

        serverChannel.configureBlocking(false);

        serverChannel.register(selector, SelectionKey.OP_ACCEPT);


        System.out.println("Servidor NIO esperando conexiones...");
```

```
        while (true) {
            selector.select();
            Iterator<SelectionKey> keys =
selector.selectedKeys().iterator();

            while (keys.hasNext()) {
                SelectionKey key = keys.next();
                keys.remove();

                if (key.isAcceptable()) {
                    SocketChannel cliente = serverChannel.accept();
                    cliente.configureBlocking(false);
                    cliente.register(selector,
SelectionKey.OP_READ);
                    System.out.println("Cliente conectado.");
                } else if (key.isReadable()) {
                    SocketChannel cliente = (SocketChannel)
key.channel();
                    ByteBuffer buffer = ByteBuffer.allocate(256);
                    cliente.read(buffer);
                    buffer.flip();
                    cliente.write(buffer);
                }
            }
        }
```

```
    }
}
```

Mucho más eficiente para manejar múltiples conexiones simultáneas sin bloquear el servidor.

4.- Servidor UDP Multicliente

Múltiples clientes pueden enviar mensajes sin necesidad de una conexión persistente.

```
package com.miapp.servidor;

import java.net.DatagramPacket;

import java.net.DatagramSocket;


public class ServidorUDP {

    public static void main(String[] args) throws Exception {

        DatagramSocket socket = new DatagramSocket(5000);

        byte[] buffer = new byte[1024];


        System.out.println("Servidor UDP esperando mensajes...");


        while (true) {

            DatagramPacket paquete = new DatagramPacket(buffer,
buffer.length);

            socket.receive(paquete);
```

```
            String mensaje = new String(paquete.getData(), 0,
paquete.getLength());

            System.out.println("Mensaje recibido: " + mensaje);

        }

    }

}
```

UDP es útil cuando la velocidad es más importante que la confiabilidad.

5.- Chat en Tiempo Real

Varios clientes pueden enviar y recibir mensajes simultáneamente en un chat. 🔹 *Aquí usaríamos WebSockets o NIO para un sistema interactivo.*