UD4 - 04. Establecimiento y finalización de conexiones.

Introducción

El proceso de comunicación en red implica dos fases fundamentales:

1. **Establecimiento de la conexión**: Se inicia la comunicación entre un cliente y un servidor.
2. **Finalización de la conexión**: Se cierra la comunicación una vez que se ha completado el intercambio de datos.

El control correcto de estas etapas es esencial para garantizar que los recursos del sistema (como sockets y flujos de datos) se liberen adecuadamente y evitar errores, bloqueos o consumo excesivo de recursos.

---

1. Establecimiento de Conexiones

Pasos para Establecer una Conexión TCP

1. **Cliente**
   - Crea un `Socket` indicando la dirección IP y el puerto del servidor.
   - Abre los flujos de entrada y salida para enviar y recibir datos.
2. **Servidor**
   - Crea un `ServerSocket` y lo asocia a un puerto específico.
   - Queda a la espera de peticiones entrantes utilizando el método `accept()`, que devuelve un `Socket` conectado al cliente.

Ejemplo en Java

Cliente:

```
import java.io.OutputStream;
import java.io.PrintWriter;
import java.net.Socket;

public class ClienteTCP {
    public static void main(String[] args) {
        try (Socket socket = new Socket("localhost", 5000);
```

```
             OutputStream output = socket.getOutputStream();
             PrintWriter writer = new PrintWriter(output, true)) {

            writer.println("Hola, servidor");
            System.out.println("Mensaje enviado al servidor.");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Servidor:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.ServerSocket;
import java.net.Socket;

public class ServidorTCP {
    public static void main(String[] args) {
        try (ServerSocket serverSocket = new ServerSocket(5000)) {
            System.out.println("Servidor esperando conexiones...");
            Socket cliente = serverSocket.accept();
            System.out.println("Cliente conectado.");

            BufferedReader reader = new BufferedReader(new
InputStreamReader(cliente.getInputStream()));
            String mensaje = reader.readLine();
            System.out.println("Mensaje recibido: " + mensaje);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

2. Establecimiento de Conexiones UDP

En el protocolo UDP no se establece una conexión persistente, sino que se envían y reciben paquetes de forma independiente.

Ejemplo en Java

Cliente UDP:

```
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class ClienteUDP {
    public static void main(String[] args) {
        try (DatagramSocket socket = new DatagramSocket()) {
            String mensaje = "Hola, servidor UDP";
            byte[] buffer = mensaje.getBytes();
            InetAddress direccion =
InetAddress.getByName("localhost");
            DatagramPacket paquete = new DatagramPacket(buffer,
buffer.length, direccion, 5001);
            socket.send(paquete);
            System.out.println("Paquete UDP enviado.");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Servidor UDP:

```
import java.net.DatagramPacket;
import java.net.DatagramSocket;

public class ServidorUDP {
    public static void main(String[] args) {
        try (DatagramSocket socket = new DatagramSocket(5001)) {
            byte[] buffer = new byte[1024];
            DatagramPacket paquete = new DatagramPacket(buffer,
buffer.length);
```

```
            System.out.println("Servidor UDP esperando
paquetes...");
            socket.receive(paquete);

            String mensaje = new String(paquete.getData(), 0,
paquete.getLength());
            System.out.println("Mensaje recibido: " + mensaje);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

3. Finalización de Conexiones

Proceso de Finalización en TCP

Para cerrar una conexión TCP correctamente:

1. El cliente o el servidor llama al método `close()` del `Socket`.
2. Esto envía un mensaje `FIN` al otro extremo de la conexión.
3. La conexión se cierra completamente cuando ambos extremos han enviado y recibido el mensaje de cierre.

Ejemplo de Cierre de Conexión:

```
import java.io.PrintWriter;
import java.net.Socket;

public class ClienteCierre {
    public static void main(String[] args) {
        try (Socket socket = new Socket("localhost", 5000);
             PrintWriter writer = new
PrintWriter(socket.getOutputStream(), true)) {

            writer.println("Mensaje final");
            System.out.println("Cerrando conexión...");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

En este ejemplo, el método `close()` se ejecuta automáticamente al salir del bloque `try-with-resources`.

Finalización en UDP

En UDP no es necesario cerrar explícitamente la conexión, ya que no hay un canal persistente. Sin embargo, es recomendable liberar los recursos cerrando el `DatagramSocket`:

```
socket.close();
```

Errores Comunes en el Establecimiento y Cierre de Conexiones

1. **SocketException: Address already in use**: Se produce si el puerto ya está siendo usado.
2. **ConnectException: Connection refused**: Ocurre cuando el servidor no está disponible o no escucha en el puerto indicado.
3. **EOFException**: Error de fin de archivo al intentar leer desde un flujo cerrado.

Recomendaciones para el Manejo de Conexiones

- **Uso de `try-with-resources`**: Cerrar automáticamente los recursos para evitar fugas.
- **Establecer un tiempo de espera (`setSoTimeout`)**: Evitar bloqueos si el otro extremo no responde.
- **Manejo de Excepciones**: Capturar y registrar los errores para evitar caídas inesperadas de la aplicación.

Conclusión

El establecimiento y cierre de conexiones son aspectos críticos en la programación de servicios en red. En TCP, es importante gestionar correctamente el ciclo de vida del `Socket`, mientras que en UDP el manejo es más sencillo, al no haber una conexión persistente. Utilizar buenas prácticas, como el cierre automático de recursos y la gestión adecuada de errores, garantiza la estabilidad y eficiencia de las aplicaciones de red.