Egibide

Unidad Didáctica 3:

PROGRAMACIÓN DE COMUNICACIONES EN RED

1 INTRODUCCIÓN

La programación en red permite que dos programas se comuniquen entre sí a través de una red. Para ello usamos el modelo **Cliente-Servidor** y los **Sockets**.

2 CONCEPTOS BÁSICOS

2.1 Modelo Cliente-Servidor

Cliente: Programa que inicia la comunicación y solicita un servicio (ejemplo: navegador web).El cliente es el proceso que permite al usuario formular los requerimientos y pasarlos al servidor, se le conoce con el término front-end.

Servidor: Programa que espera peticiones y responde con información (ejemplo: servidor web).espera peticiones y responde. Es conocido como back-end. El servidor normalmente maneja todas las funciones relacionadas con la mayoría de las reglas del negocio y los recursos de datos.

2.2 Dirección IP

Identifica de forma única un dispositivo dentro de una red.

Ejemplos: 192.168.1.10, 172.16.0.2

2.3 Puerto

Número que identifica una aplicación o servicio dentro de un ordenador.

Es un número entre **0 y 65535**.

Ejemplos importantes:

80 → HTTP (web)

443 → HTTPS (web segura)

25 → SMTP (correo)

Grupo de puertos	Rango de puertos	Descripción
Puertos bien conocidos o puertos del sistema	0 - 1023	Los usan los protocolos estándar y los servicios del SO
Puertos registrados	1024- 49151	Reservados por empresas y organizaciones para sus propios servicios
Puertos efímeros	49152 - 65535	De libre disposición y uso para aplicaciones cliente y servidor

2.4 Socket

Un socket es un proceso o hilo existente en la máquina cliente y en la máquina servidor, que sirve en última instancia para que el programa servidor y el cliente lean y escriban la información. Esta información será la transmitida por las diferentes capas de red. Los procesos envían/reciben mensajes a/desde sus sockets.

Es la combinación de IP+puerto. Permite que dos programas se conecten y envíen datos entre sí.

Ejemplo de socket: 192.168.1.5:5000

3 TCP vs UDP

Son los dos protocolos principales de la capa de transporte en Internet. Permiten que los datos viajen de un dispositivo a otro.

3.1 TCP

- Orientado a la conexión. Fiable.
- Garantiza entrega y orden. No pierde datos.
- Es algo mas lento.
- Se suso principalmente en web, email, chats, etc.
- Tiene control de errores.
- Ejemplo: WhatsApp.

3.2 UDP

- No es orientado a la conexión. No fiable.
- No garantiza ni entrega ni orden. Se pueden perder datos.
- Es mas rápido.
- Se usa principalmente en Juegos online, streaming, etc..
- No controla errores.
- Ejemplo; Videollamadas, Twitch(plataforma online de juegos).

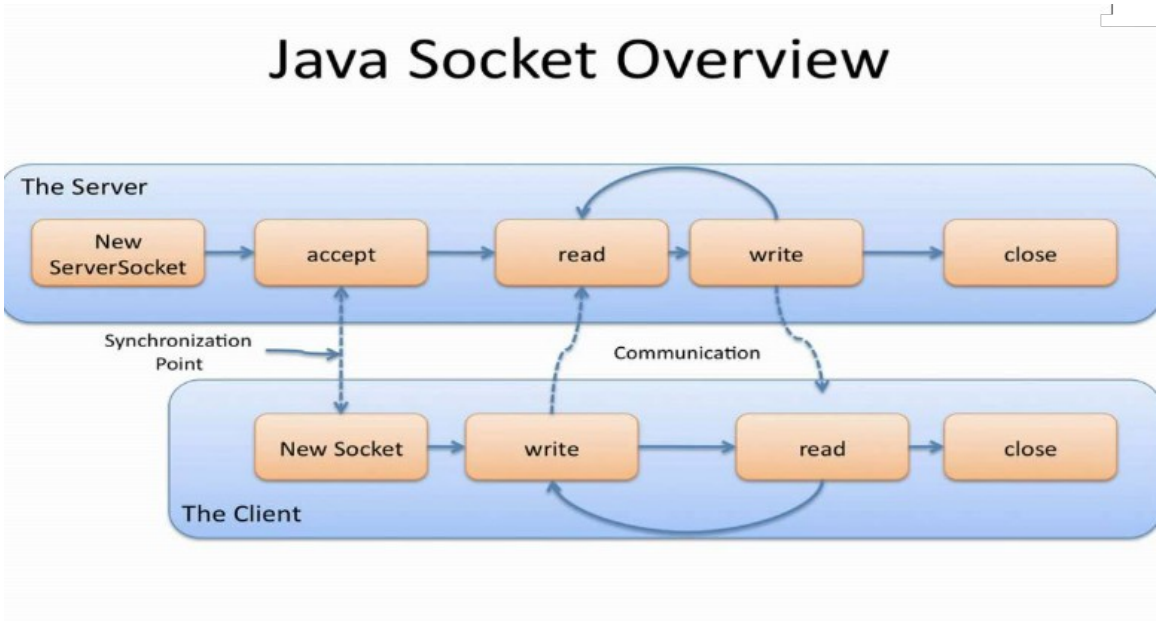
3.3 ¿Cuando usar cada uno?

Usa **TCP** cuando: los datos no pueden perderse (mensajes, archivos, login...).

Usa **UDP** cuando: importa más la velocidad que la fiabilidad (streaming, sensores, videojuegos).

4 SOCKETS TCP

TCP funciona como una llamada telefónica: primero se conecta, luego se habla, y finalmente se cuelga.



4.1 Flujo básico

Servidor:

1. Abre un puerto y queda esperando.
2. Cuando un cliente se conecta, lo atiende.
3. Envía/recibe datos.
4. Cierra la conexión.

Cliente:

1. Intenta conectarse al servidor (IP + puerto).
2. Intercambia datos.
3. Cierra la conexión.

4.2 TCP en Java

La interfaz Java que da soporte a sockets TCP está constituida por las clases ServerSocket y Socket.

ServerSocket: es utilizada por un servidor para crear un socket en el puerto en el que escucha las peticiones de conexión de los clientes. Su método accept toma una petición de conexión de la cola, o si la cola está vacía, se bloquea hasta que llega una petición. El resultado de ejecutar accept es una instancia de Socket, a través del cual el servidor tiene acceso a los datos enviados por el cliente.

Socket: es utilizada tanto por el cliente como por el servidor. El cliente crea un socket especificando el nombre DNS del host y el puerto del servidor, así se crea el socket local y además se conecta con el servicio.

Esta clase proporciona los métodos getInputStream y getOutputStream para acceder a los dos streams asociados a un socket (recordemos que son bidireccionales), y devuelve tipos de datos InputStream y OutputStream, respectivamente, a partir de los cuales podemos construir BufferedReader y PrintWriter , respectivamente, para poder procesar los datos de forma más sencilla.

Os dejo el enlace a la documentación de Java:

Socket (Java Platform SE 8)

ServerSocket (Java Platform SE 8)

4.3 Detalles de Sockets TCP

- Creación y conexión del socket (java.net.Socket).
  - Al llamar al constructor se crea el socket y si se indica la dirección y puerto del servidor ya se conecta con la máquina y puerto indicados.
  - Constructores:
    - Socket()
    - Socket(InetAddress dir, int puerto)
    - Socket(String nombre, int puerto)
  - Métodos
    - public void close() throws IOException Se encarga de cerrar el socket.
    - public InetAddress getInetAddress () Retorna la dirección IP remota a la que se conecta el socket.
    - public InputStream getInputStream () throws IOException Retorna un input stream para la lectura de bytes desde el socket.
    - public int getLocalPort() Retorna el puerto local al que está conectado el socket.
    - public OutputStream getOutputStream () throws IOException Retorna un output stream para la escritura de bytes hacia el socket.
    - public int getPort () Retorna el puerto remoto al que está conectado el socket.
- Creación del socket servidor (java.net.ServerSocket)
  - Al llamar al constructor se crea el socket servidor y si se indica el puerto ya se asocia a ese puerto.

- Constructores:
  - ServerSocket()
  - ServerSocket(int puerto)

Hay que recordar, que es fundamental que el puerto escogido sea conocido por el cliente, en caso contrario, no se podría establecer la conexión.

- Métodos:
  - public Socket accept () throws IOException Sobre un ServerSocket se puede realizar una espera de conexión por parte del cliente mediante el método accept(). Hay que decir, que este método es de bloqueo, el proceso espera a que se realice una conexión por parte del cliente para seguir su ejecución. Una vez que se ha establecido una conexión por el cliente, este método devolverá un objeto tipo Socket, a través del cual se establecerá la comunicación con el cliente.
  - public void close () throws IOException Se encarga de cerrar el socket.
  - public InetAddress getInetAddress () Retorna la dirección IP remota a la cual está conectado el socket. Si no lo está retornará null .
  - public int getLocalPort () Retorna el puerto en el que está escuchando el socket.

4.4 Envío y Recepción de Datos mediante Sockets

Streams en los Sockets

Para comunicarnos a través de un socket se utilizan dos flujos (streams):

Tipo de Stream	Método para obtenerlo
Entrada (leer del socket)	InputStream getInputStream()
Salida (escribir al socket)	OutputStream getOutputStream()

Estos streams trabajan a nivel de byte, por lo que es común envolverlos en clases más cómodas para manejar texto o datos primitivos.

- Lectura de datos (Entrada)

  Opciones disponibles:

  a) InputStream + BufferedReader (para texto)

  Permite leer líneas de texto fácilmente:

```
BufferedReader inSocket = new BufferedReader(
    new InputStreamReader(miSocket.getInputStream()) );
String linea = inSocket.readLine();
```

  Usa readLine() para leer texto línea por línea.
  Ideal para protocolos basados en texto.

  b) DataInputStream (entrada de datos primitivos o texto UTF)

```
DataInputStream inSocket;
try {
    inSocket = new DataInputStream(miSocket.getInputStream());
} catch (IOException e) {
    System.out.println(e);
}
```

Métodos útiles:

Método	Descripción
readInt()	Lee un entero
readDouble()	Lee un número decimal
readChar()	Lee un carácter
readUTF()	Lee texto en formato UTF
read()	Lee un byte

Ideal cuando se reciben datos binarios o tipos primitivos de Java.

- Envío de datos (Salida)

  Opciones disponibles

  a) PrintWriter o PrintStream (para texto)

```
PrintStream outSocket;
try {
    outSocket = new PrintStream(miSocket.getOutputStream());
} catch (IOException e) {
    System.out.println(e);
}
```

  Métodos destacados:

  - println() → envía una línea de texto.
  - write() → escribe bytes.

  Especial para enviar texto fácilmente.
  Se usa tanto en cliente como en servidor.

  b) DataOutputStream (para datos primitivos o texto UTF)

```
DataOutputStream outSocket;
try {
    outSocket = new DataOutputStream(miSocket.getOutputStream());
} catch (IOException e) {
    System.out.println(e);
}
```

  Métodos útiles:

Método	Envia...
writeInt(int)	Entero
writeDouble(double)	Decimal
writeChar(char)	Carácter
writeUTF(String)	Texto en UTF
writeBytes(String)	Cadena como bytes puros

Ideal para enviar datos binarios o tipos primitivos de Java.

Resumen visual

Capa	Para leer	Para escribir	Ideal para...
Texto	BufferedReader + InputStreamReader	PrintWriter / PrintStream	Comunicación tipo chat, comandos, texto
Datos primitivos	DataInputStream	DataOutputStream	Envío de números, binarios, UTF

Cliente vs Servidor

- Cliente:
  - Lee con DataInputStream o BufferedReader
  - Escribe con PrintStream o DataOutputStream
- Servidor:
  - Usa las mismas clases según el tipo de dato a enviar o recibir.

4.5 Cliente TCP (pasos)

1. Crear socket indicando IP y puerto.
2. Obtener flujos de entrada y salida.
3. Enviar y recibir datos.
4. Cerrar conexión.

Ejemplo Cliente TCP:

```
Socket socket = new Socket("localhost", 5000);
 PrintStream salida = new PrintStream(socket.getOutputStream());
salida.println("Hola servidor");
socket.close();
```

4.6 Servidor TCP (pasos)

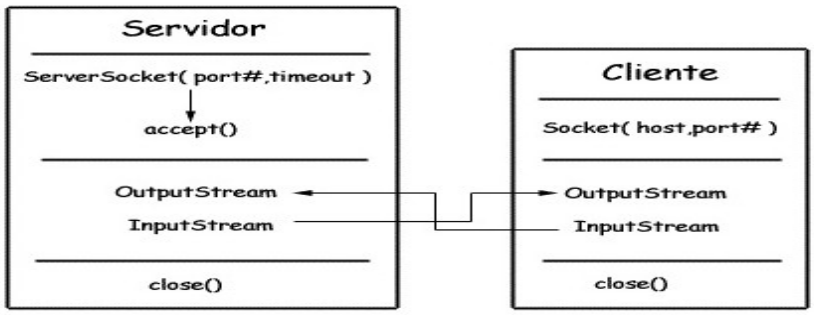
1. Crear ServerSocket con puerto.
2. Esperar cliente (accept).

3. Comunicar mediante flujos.
4. Cerrar conexión.

Ejemplo Servidor TCP:

```
ServerSocket servidor = new ServerSocket(5000);
Socket cliente = servidor.accept();
PrintStream ps = new PrintStream(cliente.getOutputStream());
ps.println("Conectado al servidor");
cliente.close();
servidor.close();
```

4.7 Código Cliente -Servidor TCP sencillo



Código Servidor TCP sencillo en Java

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

public class ServidorTCP {
      public static void main(String[] args) throws IOException {
      ServerSocket servidor = new ServerSocket(5000);
      System.out.println("Servidor esperando conexión...");

      Socket cliente = servidor.accept(); // Espera conexión
      System.out.println("Cliente conectado");
```

```
      BufferedReader entrada = new BufferedReader(new
            InputStreamReader(cliente.getInputStream()));
      PrintWriter salida = new
PrintWriter(cliente.getOutputStream(), true);

        salida.println("Hola, soy el servidor");
        System.out.println("Mensaje del cliente: " +
entrada.readLine());

        cliente.close();
      servidor.close();
      }
}
```

Código Cliente TCP sencillo en Java

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

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

      BufferedReader entrada = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
        PrintWriter salida = new
PrintWriter(socket.getOutputStream(), true);

        System.out.println("Servidor dice: " +
entrada.readLine());
        salida.println("Hola servidor, soy el cliente");

      socket.close();
        }
}
```

5 Servidor Multihilo TCP

El servidor TCP básico solo atiende a un cliente cada vez.
Con hilos (threads), puede atender varios clientes simultáneamente.

5.1 Estructura básica

```
while (true) {
    Socket cliente = servidor.accept();  // Llega un cliente
    new HiloCliente(cliente).start();    // Se atiende en un hilo
separado
}
```

5.2 Código completo de servidor multihilo TCP

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

public class ServidorMultihilo {
    public static void main(String[] args) throws Exception {
        ServerSocket servidor = new ServerSocket(5000);
        System.out.println("Servidor activo...");

        while (true) {
            Socket cliente = servidor.accept();
            System.out.println("Cliente conectado");
            new HiloCliente(cliente).start();
        }
    }
}

class HiloCliente extends Thread {
    private Socket socket;

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

    public void run() {
        try {
            BufferedReader entrada = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            PrintWriter salida = new
PrintWriter(socket.getOutputStream(), true);

            salida.println("Conexión establecida con el servidor
multihilo");
            System.out.println("Cliente dice: " + entrada.readLine());

            socket.close();
        } catch (Exception e) {
            System.out.println("Error en hilo: " + e.getMessage());
        }
    }
}
```

6 SOCKETS UDP

UDP funciona como enviar una carta sin confirmar si llega. No hay conexión previa.

Funcionamiento básico:

Servidor UDP:

1. Abre un puerto
2. Espera datagramas
3. Procesa y responde
4. No hay conexión real.

Cliente UDP:

1. No necesita puerto fijo
2. Envia datagramas
3. Opcional: recibe respuesta
4. No hay confirmación

6.1 Sockets UDP en Java

La interfaz Java que da soporte a sockets UDP está constituida por las clases DatagramPacket y DatagramSocket.

DatagramSocket: Representa el socket UDP, tanto para envío como para recepción.

DatagramPacket:Representa el datagrama que se envía o recibe.

6.1.1 DatagramSocket

DatagramSocket representa un socket UDP.
Sirve tanto para enviar como recibir datagramas.

A diferencia de los sockets TCP (Socket y ServerSocket), aquí no se establece una conexión previa entre cliente y servidor.
Cada datagrama viaja de manera independiente.

Métodos principales

- DatagramSocket():Crea un socket UDP sin asociarlo a ningún puerto específico. El sistema elige uno libre.
- DatagramSocket(int port):Crea un socket y lo asocia a un puerto (habitual en servidores).
- void send(DatagramPacket p):Envía un datagrama.
- void receive(DatagramPacket p):Espera (bloquea) hasta recibir un datagrama.
- void close(): Cierra el socket.
- int getLocalPort():Devuelve el puerto local del socket.
- InetAddress getLocalAddress():Devuelve la IP local.

Enlace a la documentación de Java: DatagramSocket (Java Platform SE 8)

Ejemplo básico:

```
DatagramSocket socket = new DatagramSocket(5000); // Escucha en puerto 5000

byte[] buffer = new byte[1024];

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

socket.receive(paquete); // Espera a recibir un mensaje
```

6.1.1 DatagramPacket

Un DatagramPacket es el contenedor del mensaje UDP.
Puede representar tanto un datagrama de envío como de recepción.

Contiene:

- Los datos (en forma de array de bytes).
- La dirección IP de destino/origen.
- El puerto de destino/origen.

- La longitud de los datos.

Métodos básicos

- DatagramPacket(byte[] buf, int length):Se usa para recibir datos (el socket rellenará el buffer).
- DatagramPacket(byte[] buf, int length, InetAddress address, int port):Se usa para enviar datos a una dirección y puerto concretos
- getData():Devuelve el buffer de bytes recibido.
- getLength():Devuelve la cantidad real de bytes recibidos.
- getAddress():Devuelve la dirección IP del emisor.
- getPort():Devuelve el puerto del emisor.
- setData(byte[] buf): Cambia el buffer de datos.
- setLength(int length):Ajusta la longitud del mensaje.

Enlace a la documentación de Java: DatagramPacket (Java Platform SE 8)

Ejemplo para enviar:

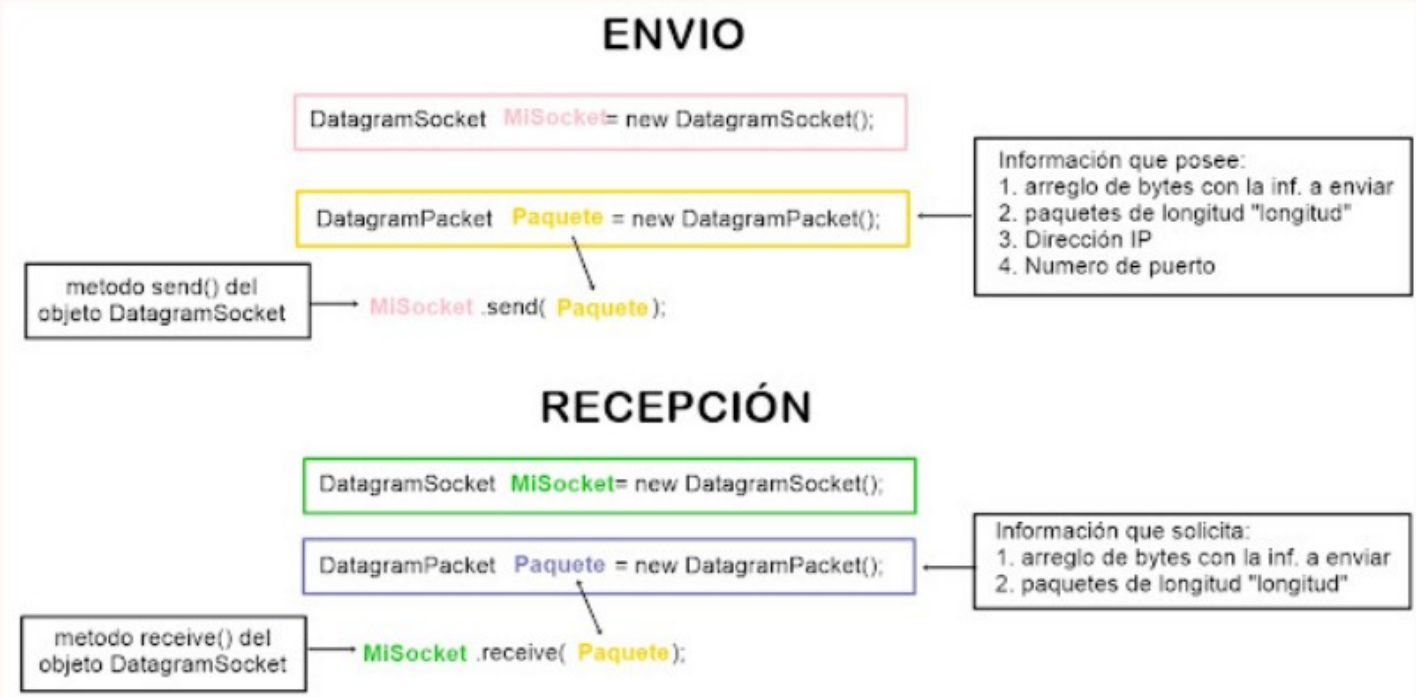
```
byte[] datos = "Hola UDP".getBytes();

InetAddress destino = InetAddress.getByName("localhost");

DatagramPacket paquete = new DatagramPacket(datos, datos.length, destino, 5000);

socket.send(paquete);
```

Ejemplo para recibir:

byte[] buffer = new byte[1024];//se inicializa a 1024 porque es un tamaño suficiente para mensajes “estándar”. Se puede inicializar a 2048 o mas si se esperan mensajes mas grandes.

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

socket.receive(recibido);



6.2 Cliente UDP

```
DatagramSocket socket = new DatagramSocket();
String mensaje = "Hola";
byte[] buffer = mensaje.getBytes();
InetAddress destino = InetAddress.getLocalHost();
DatagramPacket paquete = new DatagramPacket(buffer, buffer.length,
destino, 5000);
socket.send(paquete);
socket.close();
```

6.3 Servidor UDP

```
DatagramSocket socket = new DatagramSocket(5000);
byte[] buffer = new byte[1024];
DatagramPacket recibido = new DatagramPacket(buffer, buffer.length);
socket.receive(recibido);
System.out.println(new String(recibido.getData()).trim());
socket.close();
```

6.4 Ejemplo Cliente-Servidor UDP sencillo

Código Servidor UDP

```
import java.net.*;

public class ServidorUDP {
    public static void main(String[] args) throws Exception {
        DatagramSocket socket = new DatagramSocket(12345);
        byte[] buffer = new byte[1024];

        System.out.println("Servidor UDP abierto...");

        DatagramPacket paquete = new DatagramPacket(buffer,
            buffer.length);
        socket.receive(paquete); // Espera mensaje

        String mensaje = new String(paquete.getData()).trim();
        System.out.println("Mensaje recibido: " + mensaje);

        socket.close();
          }
}
```

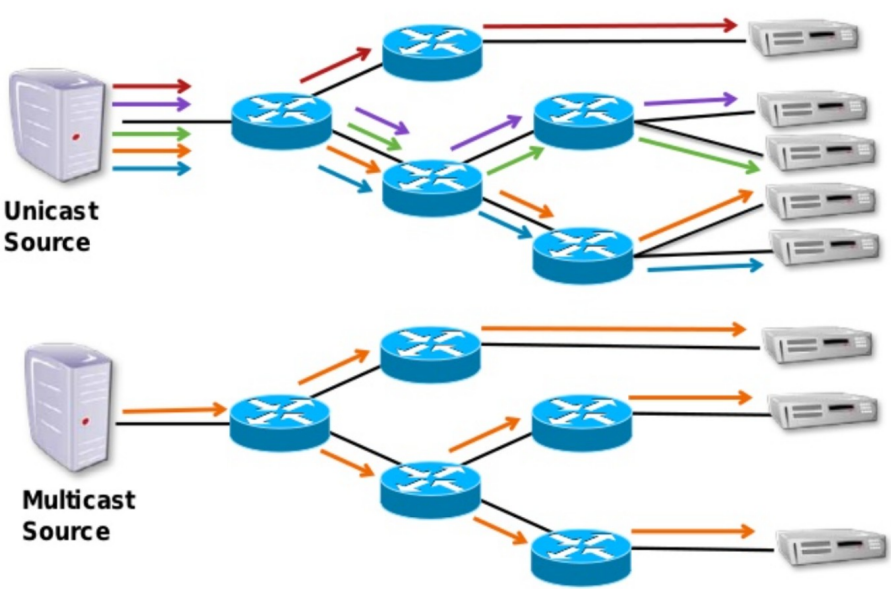
Código Cliente UDP

```
import java.net.*;

public class ClienteUDP {
       public static void main(String[] args) throws Exception {
      DatagramSocket socket = new DatagramSocket();
        InetAddress destino = InetAddress.getByName("localhost");

        String mensaje = "Hola servidor UDP";
        byte[] datos = mensaje.getBytes();

        DatagramPacket paquete = new DatagramPacket(datos,
            datos.length,destino, 12345);
        socket.send(paquete); // Envía mensaje

        socket.close();
      }
}
```

7 MULTICAST (UDP A MÚLTIPLES DESTINATARIOS)

¿Qué es Multicast?

Es una forma de enviar un mensaje a varios ordenadores a la vez.
Se usa para streaming de vídeo, juegos online, difusión de eventos en LAN.
Usa direcciones IP especiales del rango: 224.0.0.0 a 239.255.255.255
Un equipo envía el mensaje → todos los que estén suscritos al "grupo" lo reciben.



7.1 Esquema general para un servidor multicast

Se crea el socket multicast. No hace falta especificar puerto

MuslticastSocket ms = new MulticastSocket();

Se define el pueto multicast: *int Puerto = 12345;*

Se crea el grupo multicast

InetAddress grupo = InetAddress.getByName("225.0.0.1");

Se crea el datagrama

DatagramPacket paquete = new DatagramPacket(msg.getBytes(), msg.length(), grupo, Puerto);

Se envía el paquete al grupo: *ms.send(paquete);*

Se cierra el socket: *ms.close();*

7.2 Esquema general para un cliente multicast

Se crea un socket multicast en el puerto establecido

MulticastSocket ms = new MulticastSocket(12345);

Se configura la IP del grupo al que nos conectaremos

InetAddress grupo = InetAddress.getByName("225.0.0.1");

Se une al grupo: *ms.joinGroup(grupo);*

Recibe el paquete del servidor multicast

byte[] buf = new byte[1000];

DatagramPacket recibido = new DatagramPacket(buf, buf.length);

ms.receive(recibido);

Salimos del grupo multicast: *ms.leaveGroup(grupo);*

Se cierra el socket: *ms.close();*

7.3 Código Servidor Multicast (envía mensaje)

```
import java.net.*;

public class ServidorMulticast {
    public static void main(String[] args) throws Exception {
        MulticastSocket socket = new MulticastSocket();
        InetAddress grupo = InetAddress.getByName("230.0.0.1"); //
                  Dirección multicast

        String mensaje = "Hola a todos los del grupo multicast";
        byte[] datos = mensaje.getBytes();

        DatagramPacket paquete = new DatagramPacket(datos, datos.length,
            grupo, 55557);
        socket.send(paquete);

        System.out.println("Mensaje multicast enviado");
        socket.close();
    }
}
```

Código Cliente Multicast (escucha mensaje)

```
import java.net.*;

public class ClienteMulticast {
 public static void main(String[] args) throws Exception {
     MulticastSocket socket = new MulticastSocket(55557);
     InetAddress grupo = InetAddress.getByName("230.0.0.1");

     socket.joinGroup(grupo); // Unirse al grupo

     byte[] buffer = new byte[1024];
     DatagramPacket paquete = new DatagramPacket(buffer,
   buffer.length);

     System.out.println("Esperando mensaje multicast...");
     socket.receive(paquete);
```

```
        String recibido = new String(paquete.getData()).trim();
        System.out.println("Mensaje recibido: " + recibido);

        socket.leaveGroup(grupo);
        socket.close();
    }
}
```

8 Envío de Objetos con TCP

Además de enviar texto o números, en Java es posible enviar objetos completos entre cliente y servidor usando sockets TCP.
Para hacerlo, se usa serialización: convertir un objeto a bytes, enviarlo y reconstruirlo.

8.1 ¿Qué se necesita?

- La clase del objeto debe implementar la interfaz Serializable.
- Se utilizan las clases ObjectOutputStream y ObjectInputStream.

Ejemplo de clase a enviar

```
import java.io.Serializable;

public class Persona implements Serializable {
    private String nombre;
    private int edad;

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
}
```

Servidor: recibe objetos

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

public class ServidorObjeto {
    public static void main(String[] args) throws Exception {
        ServerSocket servidor = new ServerSocket(6000);
        System.out.println("Esperando objeto...");
```

```
        Socket socket = servidor.accept();
        ObjectInputStream inObjeto = new
ObjectInputStream(socket.getInputStream());

        Persona p = (Persona) inObjeto.readObject(); // Se recibe el
objeto
        System.out.println("Recibido: " + p);

        socket.close();
        servidor.close();
    }
}
```

Cliente: envía objetos

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

public class ClienteObjeto {
    public static void main(String[] args) throws Exception {
        Socket socket = new Socket("localhost", 6000);

        Persona persona = new Persona("Ana", 25);

        ObjectOutputStream outObjeto = new
ObjectOutputStream(socket.getOutputStream());
        outObjeto.writeObject(persona); // Envía el objeto

        socket.close();
    }
}
```

8.2 Puntos a tener en cuenta

Si la clase del objeto no es Serializable, dará error.

Siempre se crea primero el ObjectOutputStream, después el ObjectInputStream, para evitar bloqueos.

Solo funciona en TCP, porque UDP no garantiza recibir todos los bytes en orden.

9 Envío de Objetos con Sockets UDP

UDP no permite enviar objetos directamente porque no tiene conexión ni streams de objetos como TCP.
Por eso, se debe convertir el objeto a bytes (serializarlo manualmente) antes de enviarlo.

9.1 ¿Como se hace?

- Se usa ByteArrayOutputStream + ObjectOutputStream → para transformar el objeto a un array de bytes.
- Luego se envía con un DatagramPacket (UDP).
- Para recibirlo, hacemos el proceso inverso con ByteArrayInputStream + ObjectInputStream.

9.2 Ejemplo completo – Enviar objeto por UDP

Objeto a enviar (Serializable)

```
import java.io.Serializable;

	public class Persona implements Serializable {
	    public String nombre;
	    public int edad;

	    public Persona(String nombre, int edad) {
	        this.nombre = nombre;
	        this.edad = edad;
	    }
	}
```

Cliente UDP – Envía objeto

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

	public class ClienteUDPObjeto {
	    public static void main(String[] args) throws Exception {
	        DatagramSocket socket = new DatagramSocket();
	        InetAddress destino = InetAddress.getByName("localhost");

	        // Crear objeto
	        Persona persona = new Persona("Luis", 30);

	        // Convertir objeto a bytes
	        ByteArrayOutputStream baos = new ByteArrayOutputStream();
	        ObjectOutputStream oos = new ObjectOutputStream(baos);
	        oos.writeObject(persona);
	        oos.flush();
	        byte[] datos = baos.toByteArray();

	        // Enviar por UDP
	        DatagramPacket paquete = new DatagramPacket(datos, datos.length,
	destino, 6000);
	        socket.send(paquete);
```

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

Servidor UDP – Recibe objeto

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

    public class ServidorUDPObjeto {
        public static void main(String[] args) throws Exception {
            DatagramSocket socket = new DatagramSocket(6000);
            byte[] buffer = new byte[1024];

            DatagramPacket paquete = new DatagramPacket(buffer,
    buffer.length);
            System.out.println("Esperando objeto...");
            socket.receive(paquete);

            // Convertir bytes a objeto
            ByteArrayInputStream bais = new
    ByteArrayInputStream(paquete.getData());
            ObjectInputStream ois = new ObjectInputStream(bais);
            Persona personaRecibida = (Persona) ois.readObject();
            System.out.println("Objeto recibido: " + personaRecibida.nombre
    + ", edad " + personaRecibida.edad);

            socket.close();
        }
    }
```

9.3 Limitaciones importantes de UDP para objetos

UDP solo permite paquetes de hasta 64KB.

El objeto puede no llegar o llegar incompleto si pierde algún fragmento.

Si se envían varios objetos pueden llegar desordenados.