
UT2 Programación de comunicaciones en red

Módulo - Programación de Servicios y Procesos
Ciclo - Desarrollo de Aplicaciones Multiplataforma
IES María Ana Sanz

Resumen de contenidos

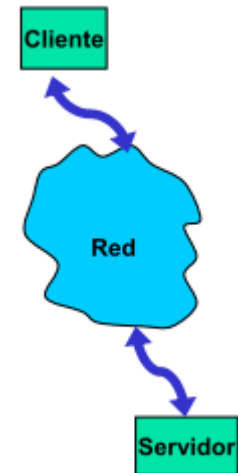
- Aplicaciones cliente/servidor
- Conceptos básicos de la pila de protocolos TCP/IP
- Protocolos de transporte TCP y UDP
- ¿Qué es un puerto?
- Java y la programación en red
 - el paquete *java.net*
- Programación de sockets en Java
 - Socket TCP (*socket stream*)
 - Socket UDP (*socket datagram*)
- Servidores concurrentes

Programación en red

- **Programación en red**
 - escritura de programas que se comunican unos con otros a través de una red (local o Internet)
- En general, las aplicaciones que tienen componentes ejecutándose en diferentes máquinas se denominan **aplicaciones distribuidas**
 - usualmente consisten en relaciones **cliente/servidor**
- La programación en red está altamente integrada en Java
 - Java proporciona toda una API de clases para facilitar la comunicación de los programas a través de la red
 - **java.net**

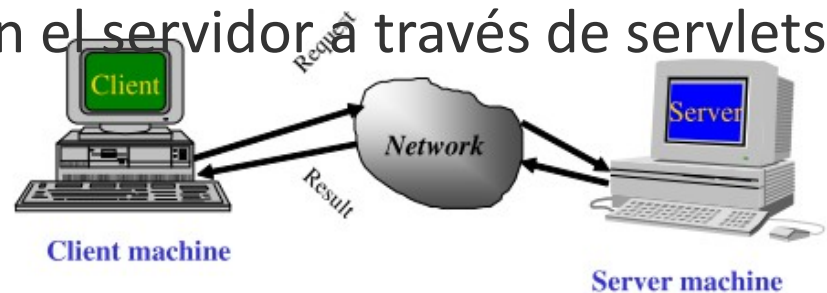
Modelo Cliente / Servidor

- Una aplicación que sigue este modelo consta de dos partes o extremos que se ejecutan en distintas máquinas
 - o en la misma - *localhost*
- **Servidor**
 - proporciona un “servicio” a varios clientes, responde a sus peticiones, las espera
- **Cliente**
 - solicita un servicio, realiza peticiones, las inicia
- Una implementación común de este modelo de petición/respuesta se da entre los servidores web y los navegadores

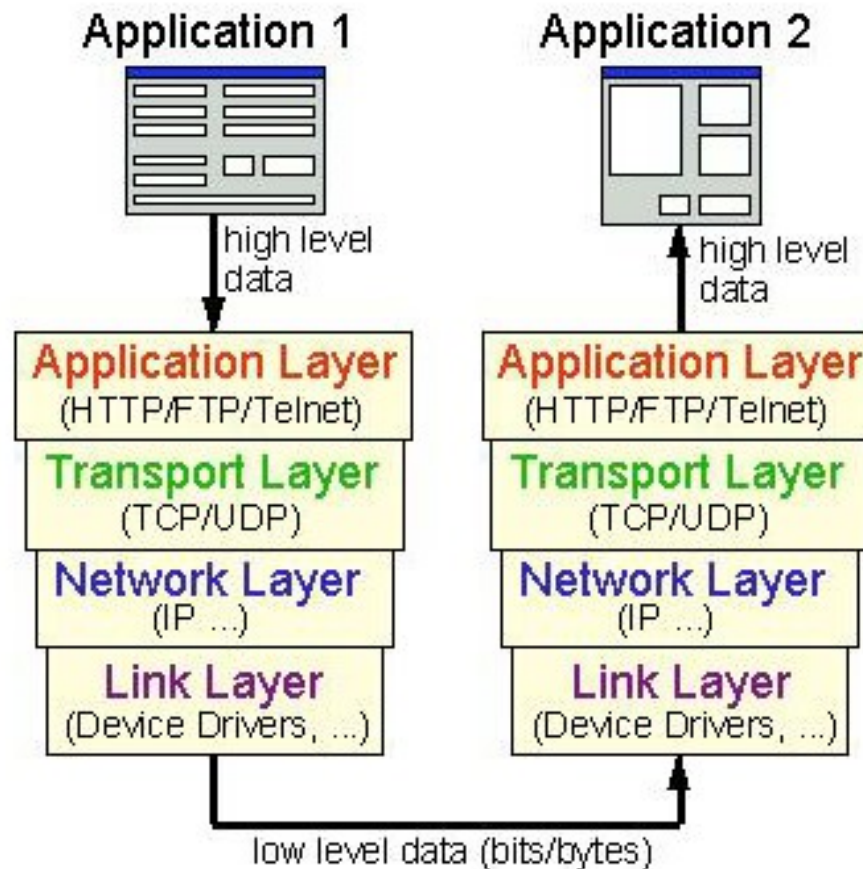


Modelo Cliente / Servidor

- Ambas partes dialogan a través de un **protocolo de comunicación**
 - conjunto de reglas establecidas para el intercambio de información
- Diferentes estrategias de comunicación entre aplicaciones en red que permite la tecnología Java
 - comunicación de aplicaciones a través de sockets
 - comunicación de aplicaciones a través de RMI
 - clientes Internet se comunican con el servidor a través de servlets
 - aplicaciones web Java
 - web services



Lo básico del protocolo TCP/IP



Pila de protocolos TCP/IP

Lo básico del protocolo TCP/IP

- Los ordenadores en Internet utilizan alguno de los protocolos de alto nivel del **nivel de aplicación** (Application level)
 - para permitir a las aplicaciones comunicarse
 - **HTTP** (Hyper Text Transfer Protocol) / **FTP** (File Transfer Protocol) / **Telnet**
- En el **nivel de transporte** (Transport Layer) hay un protocolo de más bajo nivel utilizado para
 - determinar cómo los datos serán transportados de una máquina a la otra (establecer conexiones, cerrarlas, ...)
 - Transport Control Protocol (**TCP**) / User Datagram Protocol (**UDP**)

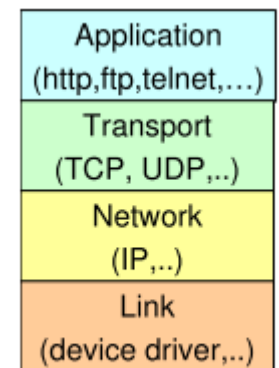


Figure 13.2: TCP/IP software stack

Lo básico del protocolo TCP/IP

- Debajo está el **nivel de red (Network Layer)** para
 - determinar cómo localizar el destino para los datos (encamina los datos)
- A un nivel más bajo está el **nivel de enlace (Link Layer)**
 - maneja la transferencia de los bits
- Toda esta familia de protocolos trabajando de forma conjunta permite el envío de datos de una máquina a otra en la red

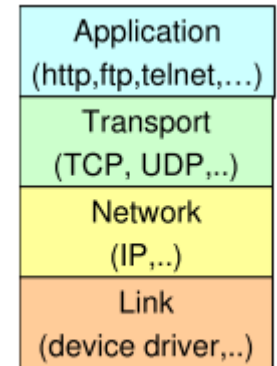
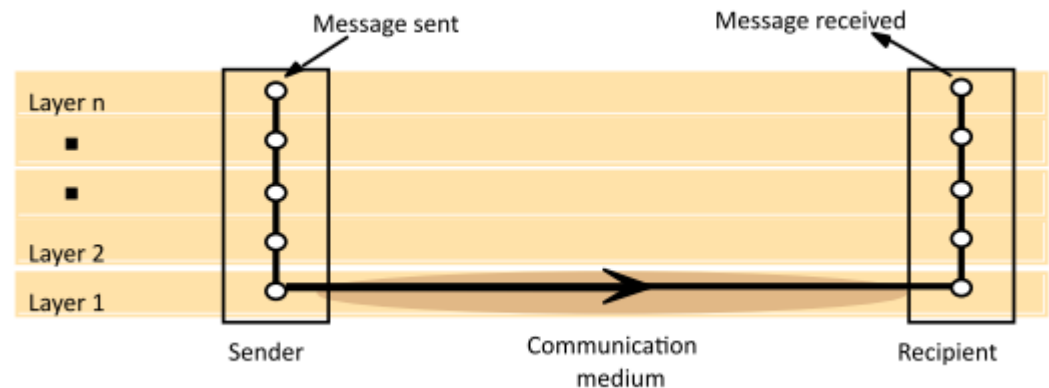
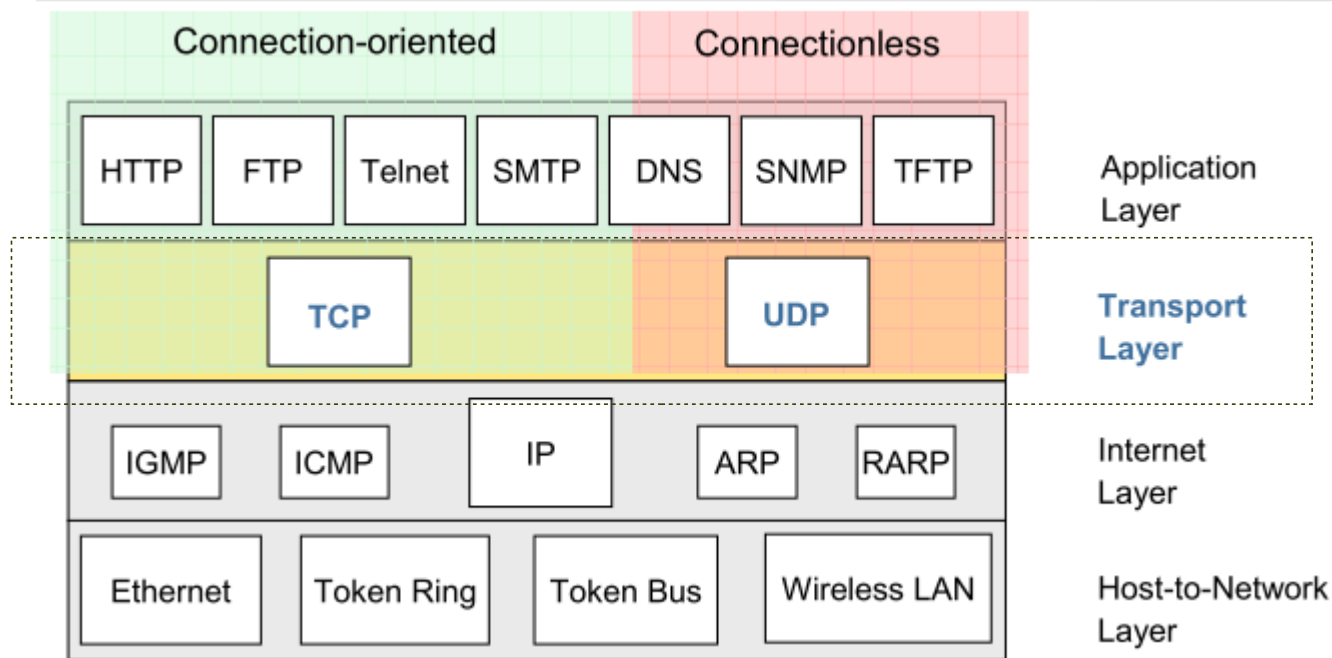


Figure 13.2: TCP/IP software stack

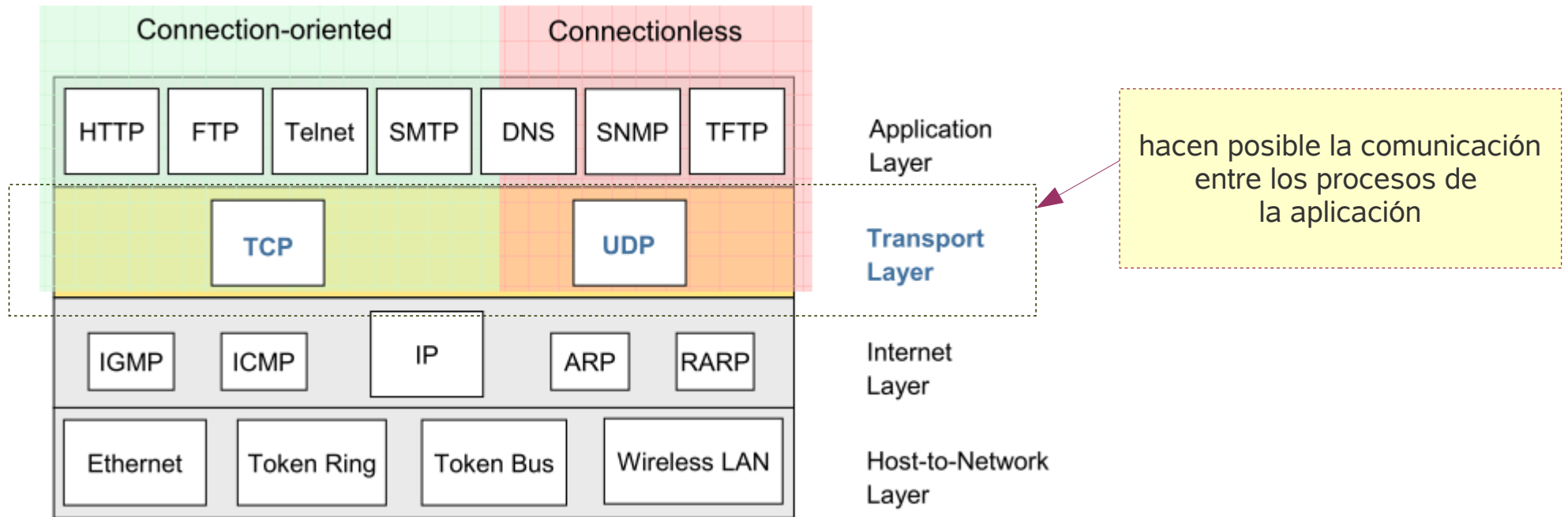


Protocolo de Transporte - TCP / UDP



- **TCP (Transmission Control Protocol)** - confiable orientado a conexión
- **UDP (User Datagram Protocol)**- no confiable, no orientado a conexión
- Java permite ambos tipos de comunicación
- Los interfaces de programación más ampliamente usados con estos protocolos son los *sockets*

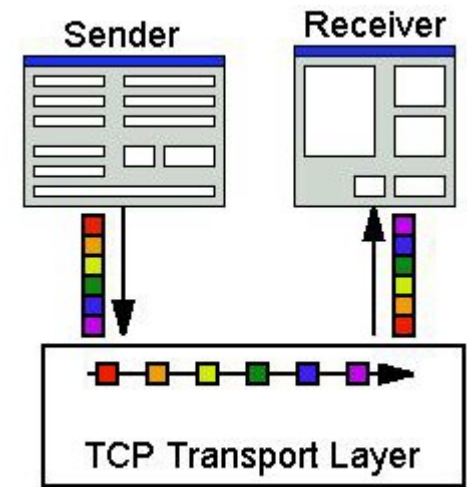
Protocolos de Transporte - TCP / UDP



- **TCP (Transmission Control Protocol)** - confiable orientado a conexión
- **UDP (User Datagram Protocol)**- no confiable, no orientado a conexión
- Java permite ambos tipos de comunicación
- Los interfaces de programación más ampliamente usados con estos protocolos son los *sockets*

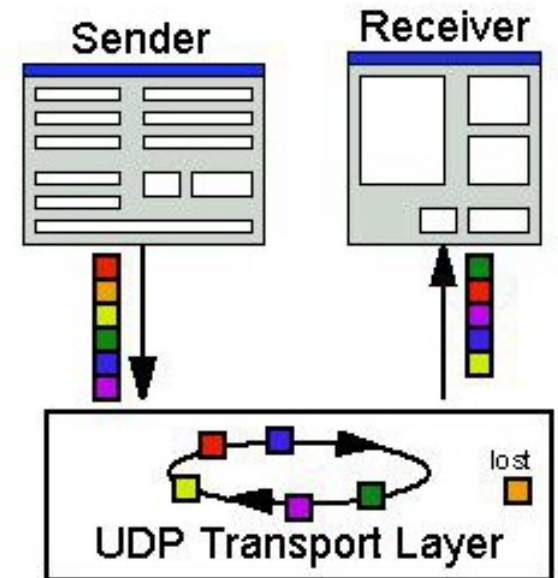
TCP (Transmission Control Protocol)

- Protocolo orientado a conexión
- Proporciona un flujo confiable de datos entre dos máquinas
 - garantiza que los datos enviados desde un punto de la conexión se reciben en el otro punto y en el mismo orden y sin pérdidas
 - similar a una llamada de teléfono
- Proporciona un canal de comunicación punto a punto para las aplicaciones
- Más lento porque hay que establecer la conexión
- HTTP, FTP, Telnet usan TCP



UDP (User Datagram Protocol)

- Envía paquetes independientes de datos – *datagramas* – de un ordenador a otro
- No hay garantía de que lleguen – protocolo no orientado a conexión
 - similar a enviar una carta
- El orden de llegada tampoco está garantizado y tampoco es importante
 - ej. Server clock / audio y vídeo
- Permite broadcast
- Más rápido, no existe la sobrecarga de establecerla conexión
- Muchos routers y firewalls están configurados para no permitir paquetes UDP
- Protocolos de nivel de aplicación que usan UDP: DNS, TFTP...



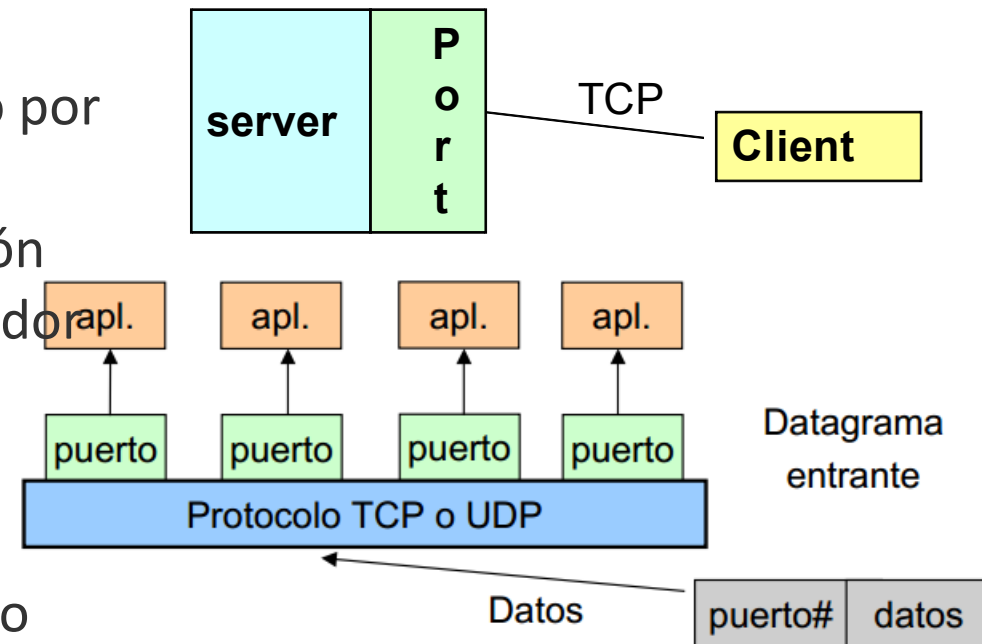
Entendiendo los puertos

- Cada ordenador en Internet (en una red) se identifica por un nº único, su dirección IP
 - entero de 32 bits escrito como 4 bytes – 128.255.56.2
 - habitualmente expresado como un nombre de dominio (www.iesmas.net) que se traduce en una IP con el servicio DNS
- Un ordenador, sin embargo
 - ofrece diferentes servicios a sus clientes (web, ftp, ...)
 - necesita atender a múltiples clientes al mismo tiempo (varias sesiones ftp, varias conexiones web, ...)
 - en definitiva, ejecuta múltiples procesos (servicios, aplicaciones)
- Cuando llegan los datos entrantes de una petición al ordenador
 - ¿cómo saber a qué aplicación corresponde?
 - ¿cómo distinguir estos servicios?
 - a través de los **puertos**

¿Qué es un puerto?

■ Puerto

- punto de acceso lógico representado por un nº de 16 bits (0 a 65535)
 - puerta de entrada a una aplicación
- cada servicio ofrecido por un ordenador está identificado por un nº único de puerto
- TCP y UDP utilizan puertos para
 - mapear datos entrantes a un proceso particular que se ejecuta en el ordenador
- Cuando se transmiten datos a un ordenador (paquetes) se indica
 - la IP del ordenador y
 - el puerto dentro del host al que van dirigidos



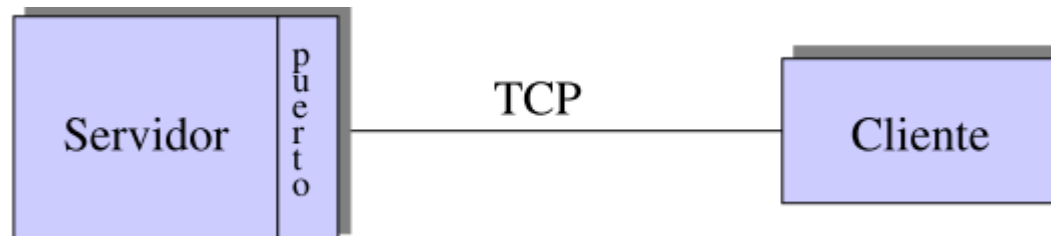
La IP – como la dirección de una casa a la que se dirige una carta
Puerto – la persona concreta dentro de la casa que recibirá la carta

¿Qué es un puerto?

- Puertos del 0 al 1023 reservados para servicios de red conocidos (*well-known ports*)
 - ftp 20/21 /tcp
 - telnet 23 /tcp
 - smtp 25 /tcp
 - http 80 /tcp
 - https – 443 /tcp
 - dns – 53 udp
- Los servicios/procesos de usuario utilizan puertos ≥ 1024

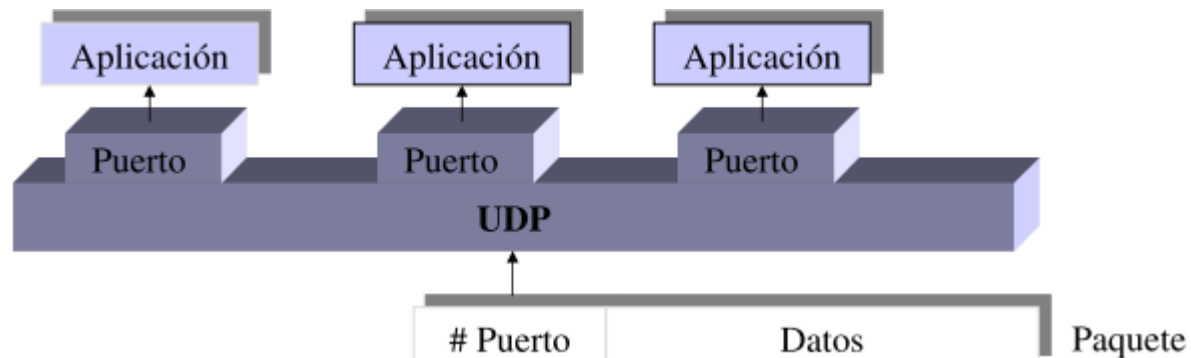
En TCP

- Un proceso (aplicación) servidor se registra en un puerto concreto
- El cliente se conecta con el servidor usando ese número de puerto
 - sólo en el establecimiento de la conexión se precisa la IP + puerto
 - el resto de paquetes TCP sólo llevan un identificador de la conexión



En UDP

- Un proceso (aplicación) servidor se registra en un puerto concreto
- El cliente envía datagramas que contienen el número de puerto del destino asociado a la aplicación servidora
 - UDP enruta hacia la aplicación adecuada
 - En cada paquete UDP va toda la información necesaria para que enrute: IP + puerto



¿Qué ofrece java.net?

- paquete Java que incluye toda la API (clases e interfaces) relacionadas con las comunicaciones en red
- **API alto nivel** – acceso a recursos de la red
 - implementan protocolos usados comúnmente – HTTP, FTP, ...
 - Clases URL, URLConnection
- **API de bajo nivel** - crear aplicaciones cliente/servidor usando protocolos TCP/UDP
 - comunicaciones basadas en flujos (*streams*) – Socket stream
 - protocolo basado en conexión – TCP
 - Socket, ServerSocket
 - comunicaciones basadas en paquetes (*packets*) – Socket UDP
 - protocolo sin conexión – UDP
 - DatagramPacket, DatagramSocket, MulticastSocket
 - clase InetAddress

La clase InetAddress

- Encapsula una dirección IP y su nombre de host asociado
- Sin constructores públicos
 - los objetos se crean usando métodos de clase (*static* - métodos *factory*)
 - ➔ **getLocalHost()** - devuelve un objeto InetAddress con la IP de la máquina sobre la que se está ejecutando el programa
 - ➔ **getByName(String nombre)** - devuelve un objeto InetAddress con la IP de la máquina cuyo nombre se pasa como parámetro
 - ✓ si no se encuentra UnknownHostException
 - ➔ **getAllByName(String nombre)** - devuelve un array de objetos InetAddress con las IP de las máquinas que tienen el nombre pasado como parámetro
 - ✓ si no se encuentra UnknownHostException

La clase InetAddress

- Otros métodos de instancia
 - **getHostName()** - devuelve String con el nombre de la máquina a la que corresponde la IP
 - **getAddress()** - array de 4 bytes con la dirección IP

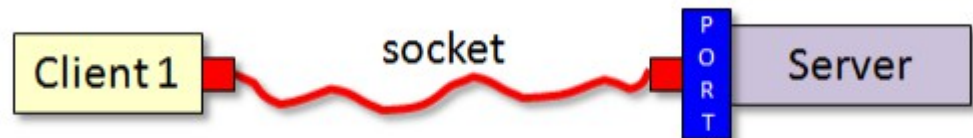
```
public class InetAddresses
{
    public static void main(String args[])
    {
        String host;
        if (args.length > 0)
            host = args[0];
        else
            host = "localhost";
    }
}
```

```
        try
        {
            InetAddress[] address =
                InetAddress.getAllByName(host);
            for (int i = 0; i < address.length; i++)
            {
                System.out.println(address[i].toString());
            }
        }
        catch (UnknownHostException e)
        {
            System.err.println(e);
        }
    }
}
```

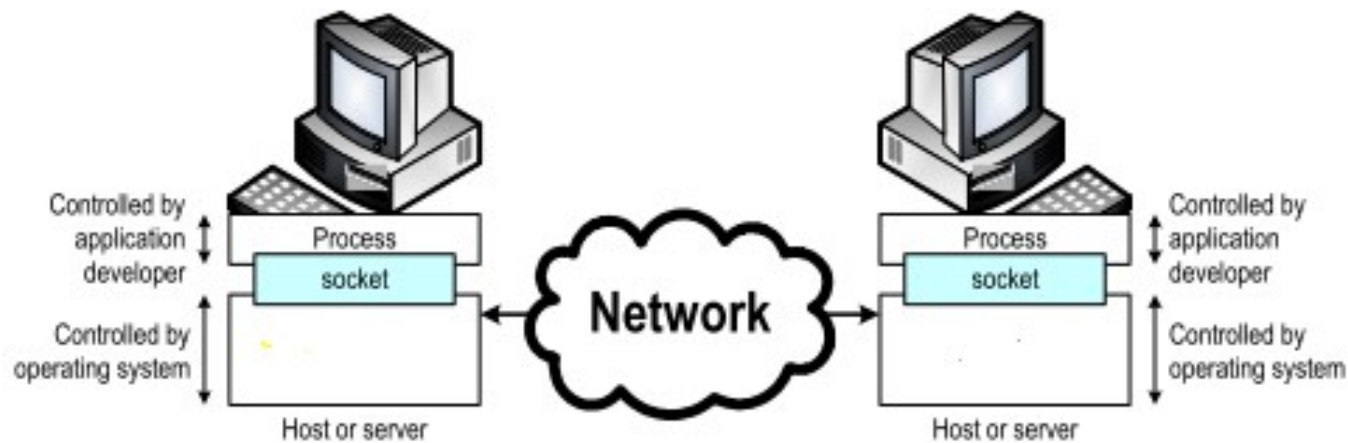
Proyecto InetAddress ejemplo 01

¿Qué es un socket?

- Es un extremo de un enlace de comunicación bidireccional entre dos programas (dos procesos) que se comunican por la red
- *Un socket se asocia a un número de puerto*
 - Se identifica por **dirección IP de la máquina + número de puerto**
- Proporciona un interfaz para programación en red a nivel de transporte
- Trata una conexión de red como un flujo de bytes que pueden ser leídos / escritos
 - comunicación con socket similar a la operaciones de IO sobre ficheros
- Existen en TCP y UDP
- Comunicación basada en socket independiente del lenguaje programación

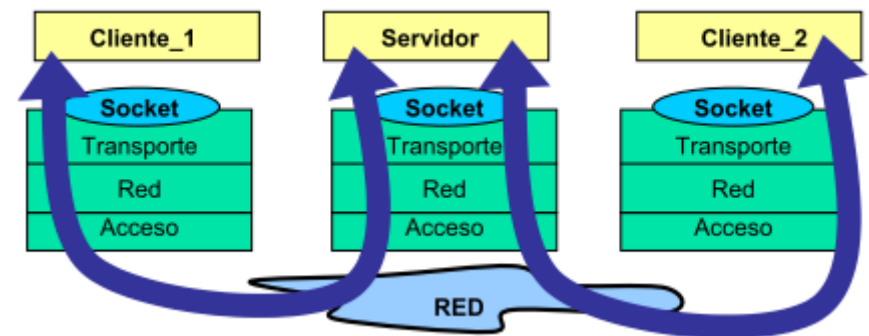


¿Qué es un socket?



Interface entre el protocolo de transporte y las aplicaciones de usuario

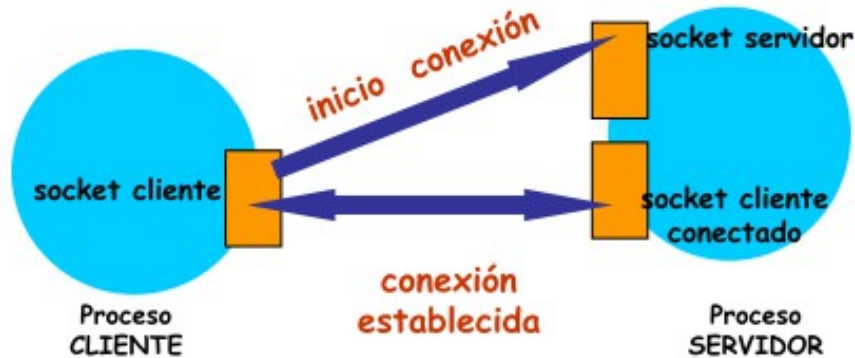
Comunicación inter-procesos



Socket TCP

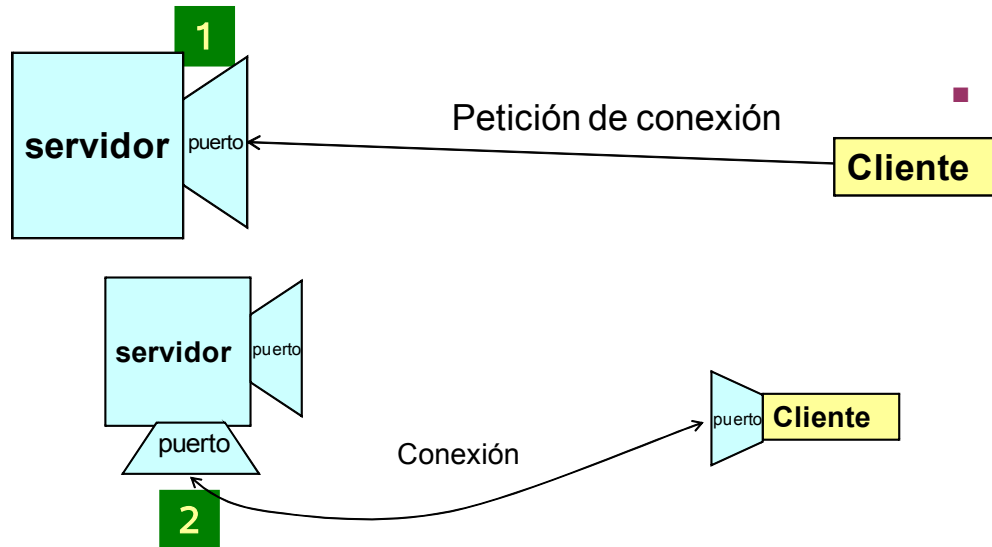
- Orientados a flujos (*streams*)
- Orientado a conexión
 - Antes del intercambio de información hay una fase de establecimiento de conexión
- Una vez establecida la conexión
 - canal punto a punto dedicado entre dos procesos (cliente y servidor)
 - los procesos intercambian información mediante flujos (leer / escribir flujos de datos)
- Los datos se reciben en el mismo orden en que se enviaron
 - conexión confiable

Programación de sockets en TCP

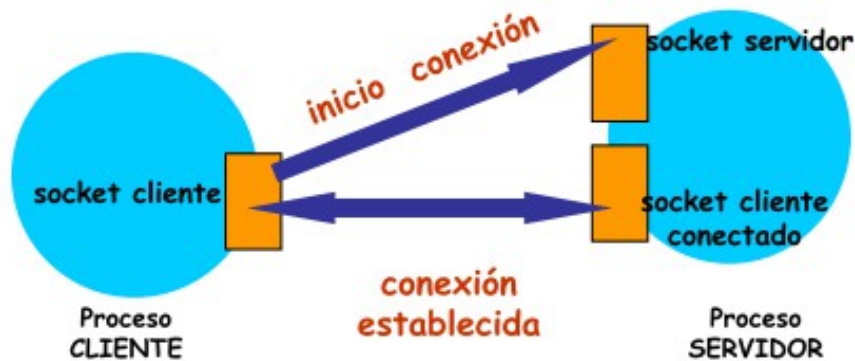


■ Parte servidor

- el proceso servidor tiene que ejecutarse primero
- el servidor crea un socket orientado a conexión asociado a un puerto específico
 - servidor espera, escuchando por ese socket, a que los clientes **1** hagan peticiones de conexión
- aceptada la conexión, el servidor crea un nuevo socket para comunicarse con el cliente conectado **2**
 - a partir de ahora leer/escribir flujos, intercambio de información entre cliente y servidor
 - servidor sigue escuchando nuevas peticiones

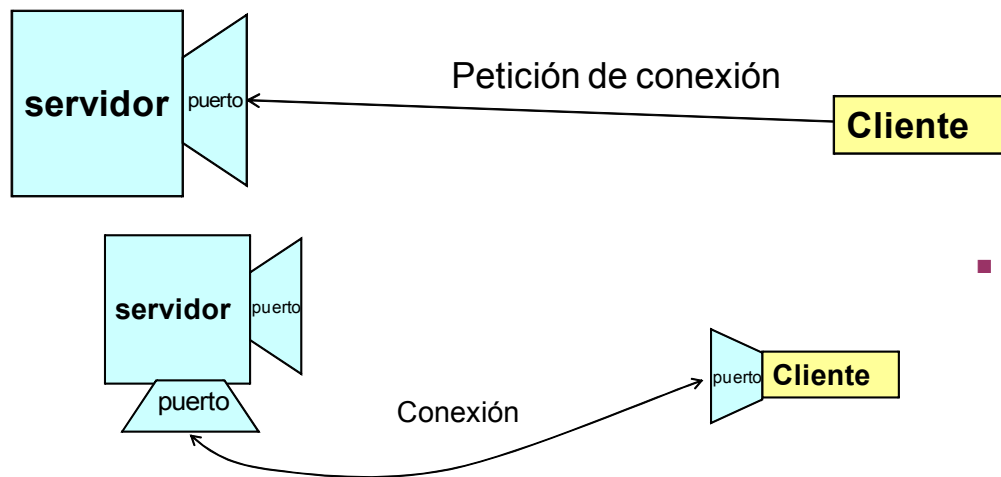


Programación de sockets en TCP

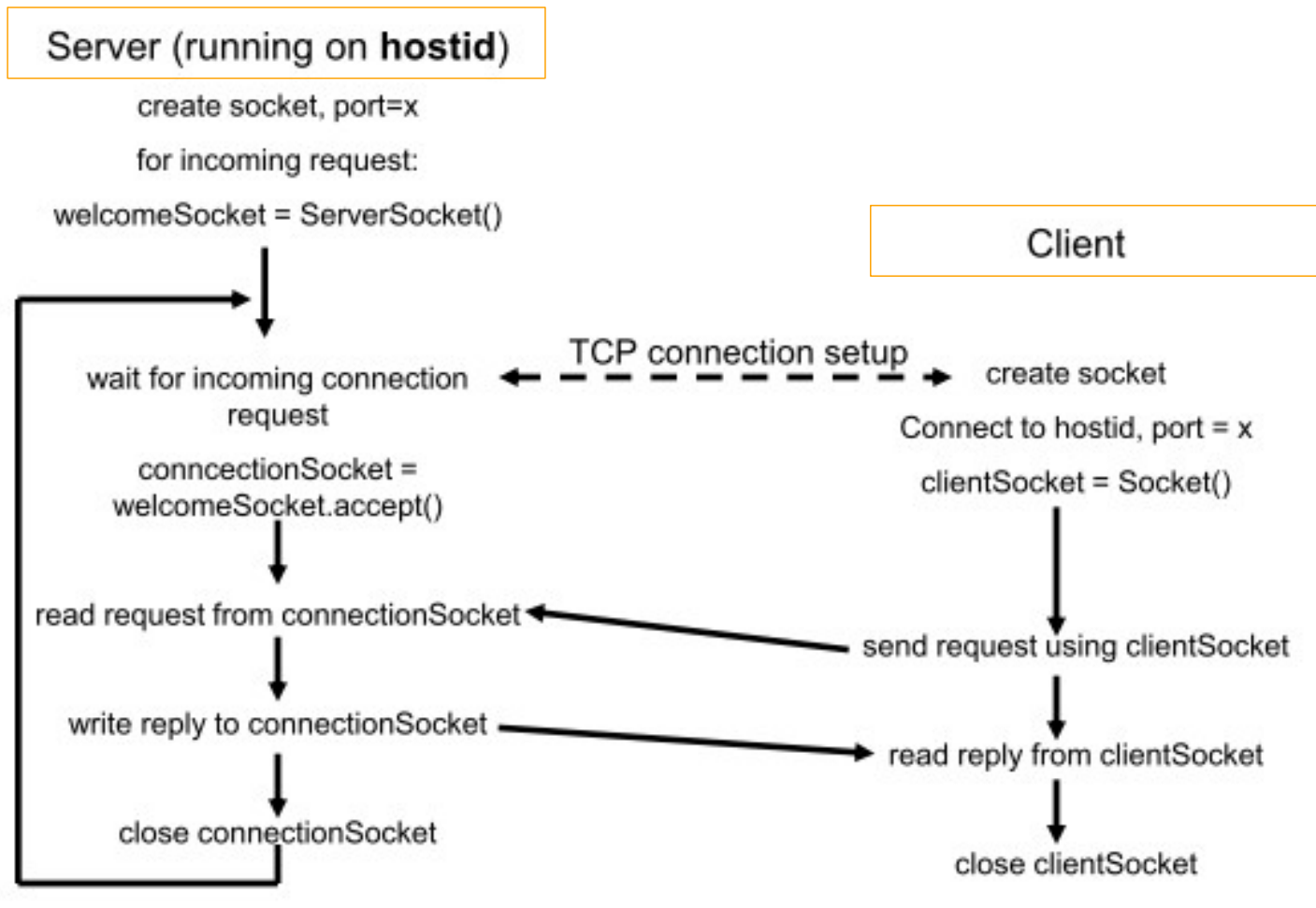


■ Parte cliente

- cliente conoce la IP del servidor y el puerto sobre el que escucha
- cliente crea un socket y realiza una petición de conexión al servidor enviando la IP y puerto del servidor
- cliente tiene que identificarse ante el servidor
 - al cliente se le asigna un puerto en su máquina, que será utilizado a lo largo de la conexión (la realiza el sistema)
- si conexión aceptada, se obtiene un socket conectado con el servidor
 - a partir de ahí, cliente y servidor se comunican escribiendo y leyendo por sus respectivos sockets



Comunicación cliente / servidor TCP



Implementación de un servidor TCP

- Clases implicadas

- `java.net.ServerSocket`
- `java.net.Socket`

- Pasos a seguir

1. crear un socket servidor

```
ServerSocket servidor = new ServerSocket( PUERTO );
```

2. mientras dure la ejecución del servidor (*while (true)*)


- esperar conexión de un cliente

```
Socket socket = servidor.accept();
```

- crear (abrir) flujos de E/S para comunicación con el cliente

```
DataInputStream is = new DataInputStream(socket.getInputStream());
```

```
DataOutputStream os = new DataOutputStream(socket.getOutputStream());
```



Servidor se
ejecuta
continuamente

Implementación de un servidor TCP

- intercambio información con el cliente

Recibir desde el cliente – `String linea = is.readLine();`

Enviar al cliente – `os.writeUTF("Hola desde el servidor");`

- cerrar flujos de E/S
- cerrar socket cliente

`socket.close();`

3. cerrar socket servidor
`servidor.close();`

java.net.ServerSocket / java.net.Socket

■ ServerSocket

- constructor – `ServerSocket(int puerto)`
 - se especifica el puerto por el que escuchará peticiones el servidor
 - cualquier puerto que no esté actualmente en uso
- método `Socket accept()`
 - espera hasta que un cliente conecta al socket de servidor
 - devuelve un `Socket` conectado al cliente que realizó la conexión
- `void close()`
 - cerrar socket servidor y cliente

en la parte servidor

Implementación de un cliente TCP

- Clases implicadas
 - `java.net.Socket`
- Pasos a seguir
 1. crear un socket y establecer conexión con el servidor
`Socket cliente = new Socket(SERVIDOR, PUERTO);`
 2. crear (abrir) flujos de E/S para comunicación con el servidor
`DataInputStream is = new DataInputStream(cliente.getInputStream());`
`DataOutputStream os = new DataOutputStream(cliente.getOutputStream());`
 3. efectuar la comunicación con el servidor
Recibir desde el servidor – `String linea = is.readLine();`
Enviar al servidor – `os.writeUTF("Hola desde el cliente");`
 4. cerrar flujos
 5. cerrar socket
`cliente.close();`

Ejemplo Servidor TCP

```
public class ServidorSimple
{
    public static void main(String[] args) throws IOException
    {
        final int PUERTO = 1234;
        // registrar el servicio en el puerto 1234
        ServerSocket servidor = new ServerSocket(PUERTO);
        System.out.println("Servidor conectado");
        Socket socket = servidor.accept(); // esperar y aceptar conexión
        // obtener flujo de escritura asociado al socket
        OutputStream os = socket.getOutputStream();
        DataOutputStream out = new DataOutputStream(os);
        // enviar saludo al cliente
        out.writeUTF("Saludo desde el servidor");
        out.close();
        socket.close();
        servidor.close();
    }
}
```

Proyecto Servidor Simple

Ejemplo Cliente TCP

```
public class ClienteSimple
{
    public static void main(String[] args) throws IOException
    {
        final int PUERTO = 1234;
        // crear socket y conectar con el servidor
        Socket cliente = new Socket("localhost", PUERTO);
        // obtener flujo de entrada y leer los datos recibidos desde el servidor
        InputStream is = cliente.getInputStream();
        DataInputStream input = new DataInputStream(is);
        String strSaludo = input.readUTF();
        System.out.println("Recibido - " + strSaludo);
        // cerra flujos y conexión
        is.close();
        input.close();
        cliente.close();
    }
}
```

Proyecto Cliente Simple

java.net.Socket

■ Socket

- creación y conexión del socket
- constructores
 - `Socket(InetAddress dir, int puerto)`
 - `Socket(String nombreServidor, int puerto)`

```
Socket cliente = new Socket("130.254.204.36", 8000); // IP
Socket cliente = new Socket("drake.armstrong.edu", 8000);
// nombre dominio
Socket cliente = new Socket("localhost", 8000);
```

en la parte cliente

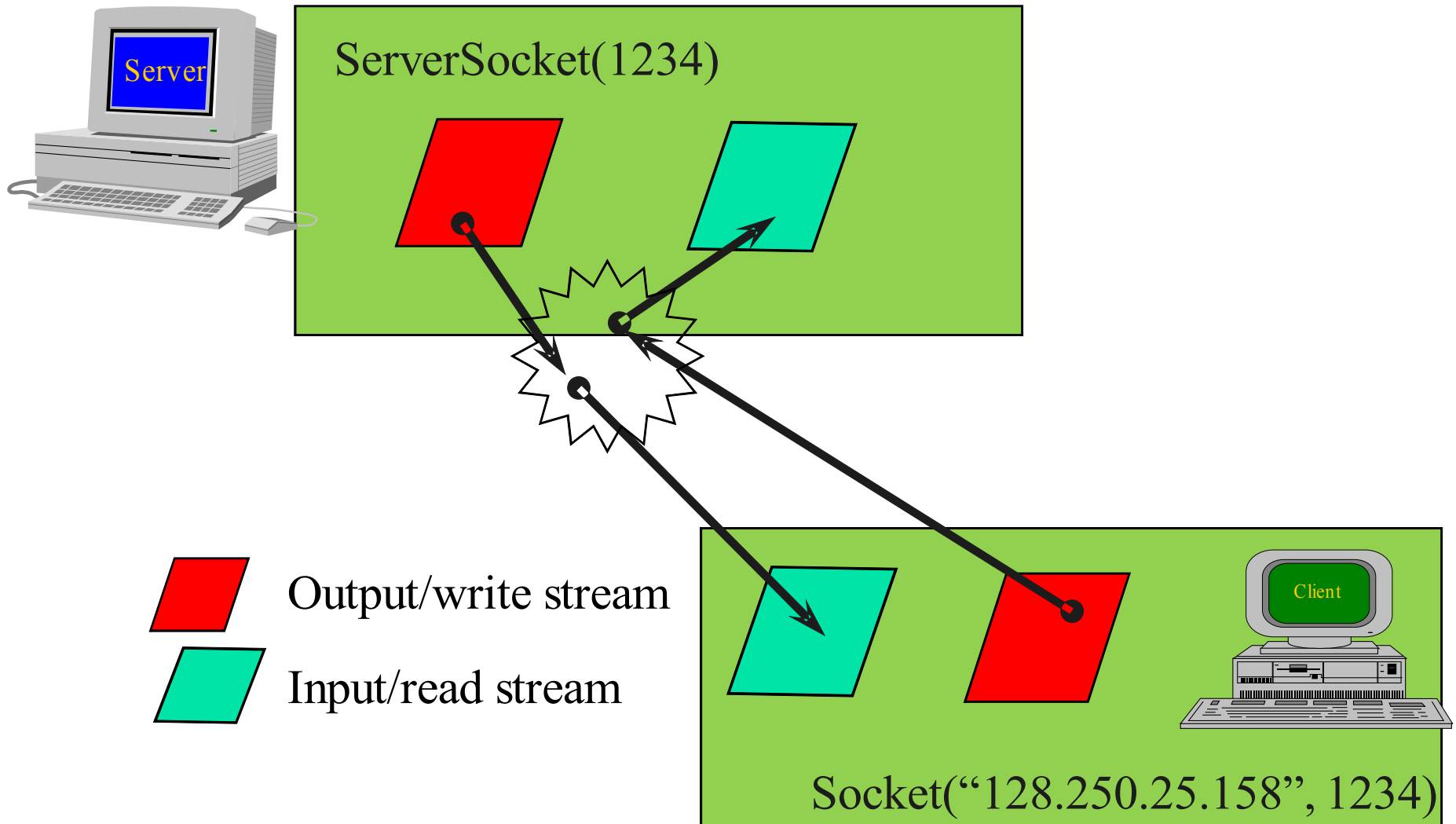
Creación de los flujos de lectura / escritura

■ Socket

- InputStream is = socket.getInputStream();
- OutputStream os = socket.getOutputStream();
- flujos binarios para leer/escribir bytes
- Se recomienda si se transmite texto
 - BufferedReader / InputStreamReader / (lectura) / Scanner
 - PrintWriter (escritura)
- Para leer / escribir tipos primitivos
 - DataInputStream / DataOutputStream

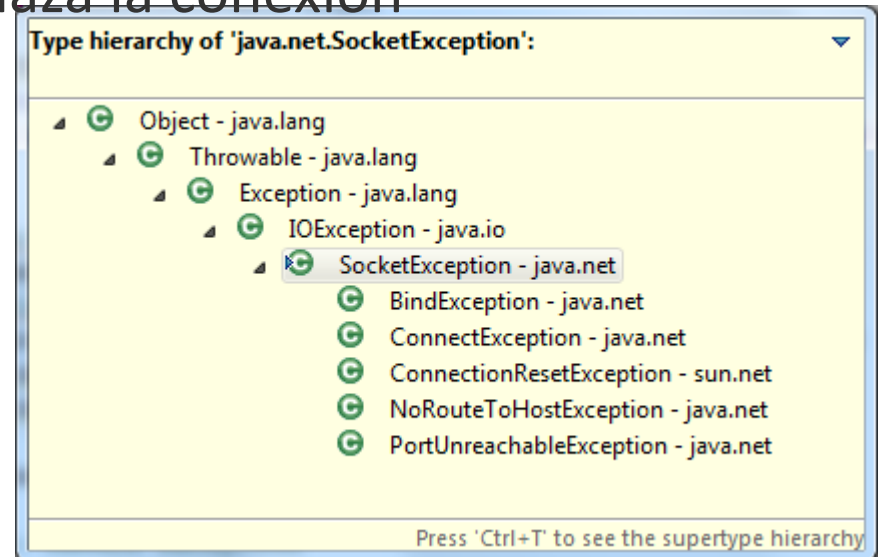
en la parte servidor y
cliente

Flujos de lectura / escritura en una conexión



Socket TCP y Excepciones

- **BindException**
 - error cuando se intenta iniciar el servidor en un puerto ya en uso
- **UnknownHostException**
 - el nombre del ordenador con el que se quiere establecer la conexión no se encuentra
- **ConnectException**
 - ocurre cuando el host remoto rechaza la conexión
- **SocketException**
 - generada cuando hay un error en el socket
- **NoRouteToHostException**
 - cuando ha expirado el tiempo de establecimiento de la conexión
- **IOException**



Ejemplo servidor iterativo

```
public class ServidorIterativoMayusculas
{
    /**
     * Servidor recibe un string del cliente y lo devuelve en mayúsculas
     */
    public static void main(String[] args) throws IOException
    {
        final int PUERTO = 6789;
        // registrar el servicio en el puerto 6789
        ServerSocket servidor = new ServerSocket(PUERTO);
        System.out.println("Servidor conectado");
        try
        {
            while (true)
            {
                Socket socketConexion = servidor.accept(); // esperar y aceptar
                                                            conexión

                try
                {
                    // obtener flujo de entrada asociado al socket
                    BufferedReader br = new BufferedReader(new
                        InputStreamReader(socketConexion.getInputStream()));
```

Servidor iterativo
Atiende a múltiples
clientes de
forma secuencial,
cuando acaba con
uno empieza con
el siguiente

Proyecto Servidor iterativo mayúsculas

Ejemplo servidor iterativo

```
        // obtener flujo de salida asociado al socket
        PrintWriter pw = new PrintWriter(new
        OutputStreamWriter(socketConexion.getOutputStream()), true);
        // leemos una línea desde el socket
        String lineaRecibida = br.readLine();
        // convertir a mayúsculas y enviar al cliente
        pw.println(lineaRecibida.toUpperCase());
        pw.close();
        br.close();
    }
    finally
    {
        socketConexion.close();
    }
}
}
finally
{
    servidor.close();
}
}
}
```

Otro ejemplo de cliente

```
public class ClienteMayusculas
{
    public static void main(String[] args) throws IOException
    {
        if (args.length != 1)
            System.exit(0);
        String mensaje = args[0];
        final int PUERTO = 6789;
        // crear socket y conectar con el servidor
        Socket cliente = new Socket("localhost", PUERTO);
        // obtener flujo de salida asociado al socket
        PrintWriter pw = new PrintWriter(new BufferedWriter(new
            OutputStreamWriter(cliente.getOutputStream())),
            true);
        // obtener flujo de entrada asociado al socket
        BufferedReader br = new BufferedReader(new
            InputStreamReader(cliente.getInputStream()));
        // enviar cadena al servidor
        pw.println(mensaje);
        // leer cadena devuelta por el servidor
        String resul = br.readLine();
        System.out.println("Recibida desde el servidor - " + resul);
        pw.close();
        br.close();
        cliente.close();
    }
}
```

la cadena que se envía al servidor se recibe desde línea de comandos

Proyecto cliente mayúsculas

Servidor TCP concurrente

Proyecto ServidorConcurrenteMayúsculas

```
public class ServidorConcurrenteMayusculas
{
    public static void main(String[] args) throws IOException
    {
        final int PUERTO = 6789;
        // registrar el servicio en el puerto 6789
        ServerSocket servidor = new ServerSocket(PUERTO);
        System.out.println("Servidor conectado");
        int numCliente = 0;
        try
        {
            while (true) // servidor ejecutándose indefinidamente
            {
                numCliente++;
                // esperar y aceptar conexión
                Socket socketConexion = servidor.accept();
                new ManejadorCliente(numCliente, socketConexion);
            }
        }
        finally
        {
            servidor.close();
        }
    }
}
```

Servidor concurrente

Atiende a múltiples clientes a la vez,
concurrentemente, creando un hilo para cada
nueva conexión de cliente

Servidor TCP concurrente

```
public class ManejadorCliente implements Runnable
{
    private int numCliente;
    private Socket socketConexion;

    public ManejadorCliente(int numCliente, Socket socketConexion)
    {
        this.numCliente = numCliente;
        this.socketConexion = socketConexion;
        Thread th = new Thread(this);
        th.start();
    }
    public void run()
    {
        System.out.println("Aceptada conexión Cliente " + numCliente);
        try
        {
            // obtener flujo de entrada asociado al socket
            BufferedReader br = new BufferedReader(new
                InputStreamReader(socketConexion.getInputStream()));
            // obtener flujo de salida asociado al socket
            PrintWriter pw = new PrintWriter(new
                OutputStreamWriter(socketConexion.getOutputStream()), true);
        }
    }
}
```

Servidor TCP concurrente

```
// leemos una línea desde el socket
String lineaRecibida = br.readLine();
pw.println(lineaRecibida.toUpperCase());
pw.close();
br.close();
}
catch (IOException e)
{
}
finally
{
    try
    {
        socketConexion.close();
    }
    catch (Exception e)
    {
    }
}
}
}
```

Resumen de flujos a utilizar

■ Asumimos

- `Socket socket = new Socket(HOST, PUERTO);` // en parte cliente
- `Socket socket = serverSocket.accept();` // en parte servidor

■ Obtener flujos de entrada

- a) `BufferedReader entrada = new BufferedReader(new
InputStreamReader(socket.getInputStream()));`
 - `entrada.readLine();`
- b) `Scanner entrada = new Scanner(socket.getInputStream());`
 - `String linea = entrada.nextLine();`
 - `int valor = entrada.nextInt();`
- c) `DataInputStream entrada = new
DataInputStream(socket.getInputStream());`
 - `String linea = entrada.readUTF();`
 - `int valor = entrada.readInt();`

Resumen de flujos a utilizar

- Asumimos

- `Socket socket = new Socket(HOST, PUERTO);` // en parte cliente
- `Socket socket = serverSocket.accept();` // en parte servidor

- Obtener flujos de salida

a) `PrintWriter salida = new PrintWriter(socket.getOutputStream());`

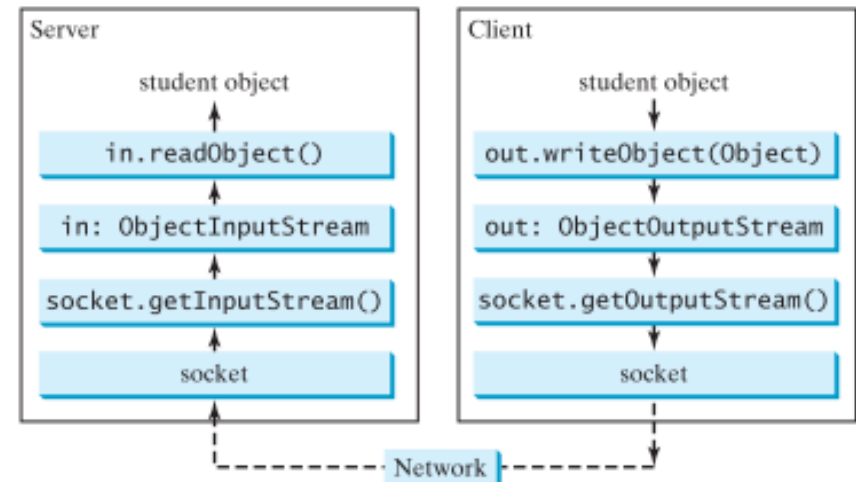
- `salida.println();`
- `salida.flush();`

b) `DataOutputStream salida = new
DataOutputStream(socket.getOutputStream());`

- `salida.writeUTF();`
- `salida.flush();`
- `salida.writeInt();`
- `salida.flush();`

Enviando / Recibiendo objetos - ObjectInputStream / ObjectOutputStream

- Podemos enviar / recibir objetos a través de un socket mediante ObjectOutputStream / ObjectInputStream
- Los objetos a enviar deben ser serializables



- clases implementan el interfaz Serializable
- ```
ObjectInputStream in = new
ObjectInputStream(socket.getInputStream());
```
- ```
ObjectOutputStream out = new  
ObjectOutputStream(socket.getOutputStream());
```

Enviando / Recibiendo objetos - ObjectInputStream / ObjectOutputStream

- Al crear flujos de objetos a ambos lados de una conexión se intercambian y verifican cabeceras de información
 - hay que tener cuidado con el orden de creación de esos flujos
 - si en el servidor, por ej
 - `ObjectOutputStream salida = new ObjectOutputStream(socket.getOutputStream());`
 - `ObjectInputStream entrada = new ObjectInputStream(socket.getInputStream());`
 - entonces en el cliente
 - `ObjectInputStream entrada = new ObjectInputStream(socket.getInputStream());`
 - `ObjectOutputStream salida = new ObjectOutputStream(socket.getOutputStream());`

Ejemplo

ObjectInputStream / ObjectOutputStream

```
.....
public void iniciar()
{
    try
    {
        ServerSocket server = new ServerSocket(PUERTO);
        System.out.println("Servidor funcionando ...");
        while (true)
        {
            Socket socket = server.accept();
            String cliente = socket.getInetAddress().getHostName();
            System.out.println("Conectado al cliente " + cliente);
            // crear flujos
            ObjectOutputStream salida = new ObjectOutputStream(socket.getOutputStream());
            ObjectInputStream entrada = new ObjectInputStream(socket.getInputStream());
            salida.writeUTF("Nº coches que desea, entre 1 y 4");
            salida.flush();
            int numCoches = entrada.readInt();
            enviarCoches(numCoches, salida);
            System.out.println("Enviados " + numCoches);
            socket.close();
        }
    }
}
```

parte servidor – un servidor de coches que se comunica con los clientes siguiendo un protocolo

obtener flujos

Proyecto Ejemplo Servidor Cliente Coche ObjectInput

Ejemplo

ObjectInputStream / ObjectOutputStream

```
private void enviarCoches(int numCoches, ObjectOutputStream salida) throws IOException
{
    for (int i = 1; i <= numCoches; i++)
    {
        int n = (int) (Math.random() * 4); // un coche aleatorio
        Coche coche = coches.get(n);
        salida.writeObject(coche);
        salida.flush();
    }
}
```

.....

public class Coche implements Serializable

Ejemplo

ObjectInputStream / ObjectOutputStream

```
public void iniciar()
{
    try
    {
        Socket socket = new Socket(host, PUERTO);
        String servidor = socket.getInetAddress().getHostName();
        System.out.println("Conectado al servidor " + servidor);
        // obtener flujos, observar el orden
        ObjectInputStream entrada = new ObjectInputStream(socket.getInputStream());
        ObjectOutputStream salida = new ObjectOutputStream(socket.getOutputStream());
        String inicio = entrada.readUTF();
        System.out.println(inicio);
        int num = (int) (Math.random() * 4 + 1);
        salida.writeInt(num);
        salida.flush();
        // recibimos los coches
        recibirCoches(num, entrada);
        socket.close();
    }
    catch (IOException e)
    {
    }
}
```

.....

parte cliente

obtener flujos

Ejemplo

ObjectInputStream / ObjectOutputStream

```
private void recibirCoches(int numCoches, ObjectInputStream entrada) throws IOException
{
    try
    {
        System.out.println("Recibiendo " + numCoches + " coches");
        for (int i = 1; i <= numCoches; i++)
        {
            Coche coche = (Coche) entrada.readObject();
            System.out.println("Recibido\n" + coche.toString());
        }
    }
    catch (ClassNotFoundException e)
    {
        System.out.println(e.toString());
    }
}
```

.....



Ejercicios

Socket Datagrama (Socket UDP)

- No orientados a conexión
- Más eficientes y rápidos que los sockets TCP pero no confiables
- Utiliza el protocolo de la capa de transporte UDP
 - no hay conexión directa entre el cliente y el servidor
- Cuando el cliente hace una petición o el servidor responde
 - los mensajes se dividen en paquetes o **datagramas UDP**
 - mensajes independientes enviados a través de la red
 - no está garantizada la recepción de todos los paquetes ni el orden de llegada
 - los paquetes pueden perderse e incluso duplicarse

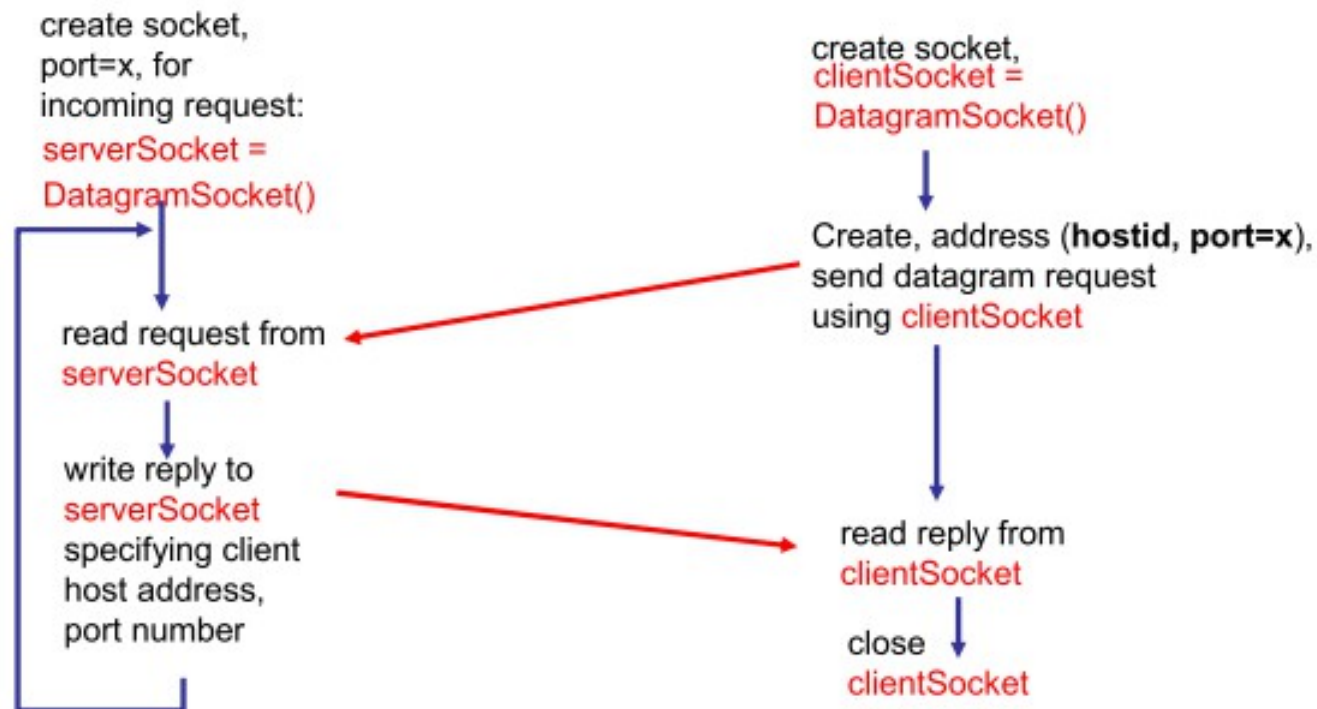
Socket Datagrama (Socket UDP)

- Cada paquete que se envía contiene
 - los datos del mensaje
 - longitud del mensaje (nº bytes)
 - dirección del destino (una InetAddress)
 - puerto del destino
 - dirección y puerto del emisor
- Clases Java implicadas en parte cliente y parte servidor
 - `java.net.DatagramSocket`
 - `java.net.DatagramPacket`

Comunicación cliente / servidor UDP

Server (on host **hostid**)

Client



Implementación de un cliente UDP

■ Pasos a seguir

- crear un socket no orientado a conexión

```
DatagramSocket clienteSocket = new DatagramSocket();
```

- obtener IP, puerto del servidor y mensaje a enviar
- preparar el datagrama a enviar

- mensaje a enviar (como array de bytes)
- nº bytes a enviar
- IP destinatario
- puerto destinatario

```
DatagramPacket paquete = new DatagramPacket(datosEnviados,  
                                             datosEnviados.length, ipServidor, puerto);
```

- enviar el datagrama

- método **send()**

```
clienteSocket.send(paquete);
```

Implementación de un cliente UDP

- preparar datagrama para recibir respuesta del servidor
 - crear un nuevo array de bytes para la respuesta

```
DatagramPacket paqueteRecibido = new DatagramPacket(datosRecibidos,  
datosRecibidos.length);
```

- recibir datagrama
 - método **receive()** bloqueante, el programa se queda esperando hasta recibir algo o
 - si se ha establecido un timeout hasta que venza

```
clienteSocket.receive(paqueteRecibido);
```

- cerrar socket

Implementación de un cliente UDP

```
public class ClienteUdp
{
    public static void main(String[] args)
    {
        if (args.length != 3)
        {
            System.out.println("Error argumentos: nombreServidor
                               puertoServidor mensaje");
            System.exit(0);
        }
        String nombreHost = args[0];
        int puerto = Integer.parseInt(args[1]); // puerto del servidor
        String mensaje = args[2]; // mensaje a enviar
        try
        {
            // crear socket cliente
            DatagramSocket clienteSocket = new DatagramSocket();
            // trasladar nombre servidor en su IP
            InetAddress ipServidor = InetAddress.getByName(nombreHost);
            byte[] datosEnviados = new byte[1024];
            byte[] datosRecibidos = new byte[1024];
            datosEnviados = mensaje.getBytes();
```

nombre servidor, puerto servidor
y mensaje a enviar tomados
desde argumentos main() en
este ejemplo

Proyecto ClienteUdp

Implementación de un cliente UDP

```
// crear datagrama
DatagramPacket paquete = new DatagramPacket(datosEnviados,
      datosEnviados.length, ipServidor, puerto);
// enviar datagrama
clienteSocket.send(paquete);
// leer datagrama del servidor
DatagramPacket paqueteRecibido = new
      DatagramPacket(datosRecibidos,
      datosRecibidos.length);
clienteSocket.receive(paqueteRecibido);
String mensajeModificado = new String(paqueteRecibido.getData());
System.out.println("Desde servidor: " + mensajeModificado);
clienteSocket.close();

}
catch (IOException e) { }
}
}
```

Implementación de un servidor UDP

■ Pasos a seguir

- crear un socket asociado a un puerto específico

```
DatagramSocket serverSocket = new DatagramSocket (PUERTO);
```

- crear un bucle infinito (*while (true)* - servidor se ejecuta siempre)

- preparar el datagrama para recibir paquetes de clientes

```
DatagramPacket paqueteRecibido = new DatagramPacket (datosRecibidos,  
                                                    datosRecibidos.length);
```

- recibir datagrama del cliente y extraer información de cliente

```
serverSocket.receive(paqueteRecibido);  
String mensaje = new String(paqueteRecibido.getData());  
// obtener IP y puerto del cliente  
InetAddress dirCliente = paqueteRecibido.getAddress();  
int puertoCliente = paqueteRecibido.getPort();
```

- preparar el datagrama de respuesta para enviar al cliente

```
DatagramPacket paqueteEnviado = new  
    DatagramPacket (datosEnviados, datosEnviados.length,  
                    dirCliente, puertoCliente);
```

- enviar datagrama de respuesta

```
serverSocket.send(paqueteEnviado);
```

Implementación de un servidor UDP

Proyecto ServidorUdp

```
public class ServidorUdp
{
    public static void main(String[] args) throws SocketException
    {
        final int PUERTO = 8090;
        DatagramSocket serverSocket = new DatagramSocket(PUERTO);
        System.out.println("SERVER online");
        byte[] datosRecibidos = new byte[1024];
        byte[] datosEnviados = new byte[1024];
        try
        {
            while (true)
            {
                // esperar a que lleguen peticiones de clientes
                // buffer para datagramas recibidos
                DatagramPacket paqueteRecibido = new
                    DatagramPacket(datosRecibidos, datosRecibidos.length);
                // recibir datagrama paqueteRecibido
                serverSocket.receive(paqueteRecibido);
                String mensaje = new String(paqueteRecibido.getData());
                // obtener IP y puerto del cliente
                InetAddress dirCliente = paqueteRecibido.getAddress();
                int puertoCliente = paqueteRecibido.getPort();
                String aMayusculas = mensaje.toUpperCase();
            }
        }
    }
}
```

Implementación de un servidor UDP

```
        datosEnviados = aMayusculas.getBytes();
        // crear datagrama para el cliente
        DatagramPacket paqueteEnviado = new
            DatagramPacket(datosEnviados,
                datosEnviados.length, dirCliente, puertoCliente);
        // escribir datagrama en el socket
        serverSocket.send(paqueteEnviado);
    }
} catch (IOException e)
{
    e.printStackTrace();
}

}
```

java.net.DatagramSocket

Descripción	Representa un socket para enviar y recibir paquetes datagrama. Punto de envío y recepción para el servicio de paquetes. Cada paquete enviado/recibido sobre un socket datagrama se direcciona y rutea individualmente
Constructores	<code>DatagramSocket(int puerto)</code> – crea un socket datagrama enlazado al puerto especificado sobre la máquina local <code>throws SocketException</code> – si el socket no puede ser abierto o no puede ser asociado al puerto indicado <code>DatagramSocket()</code> - crea un socket datagrama enlazado a cualquier puerto disponible de la máquina local <code>throws SocketException</code> – si el socket no puede ser abierto o no puede ser asociado al puerto indicado
Métodos	<code>void send(DatagramPacket paquete)</code> – envía un paquete datagrama desde este socket <code>throws IOException</code> – si ocurre un error de IO <code>void receive(DatagramPacket paquete)</code> – recibe un paquete datagrama desde este socket. Este método bloquea hasta que el datagrama se recibe <code>throws IOException</code> – si ocurre un error de IO <code>void close()</code> – cierra el socket datagrama <code>throws SocketException</code> –cualquier hilo bloqueado por el correspondiente <code>receive()</code> lanzará esta excepción

java.net.DatagramPacket

Descripción	Representa un paquete datagrama. Usado para implementar el servicio de envío de paquetes sin conexión. Cada mensaje se rutea de una máquina a otra basándose en la información contenida en el paquete
Constructores	<p><code>DatagramPacket(byte[] buf, int longitud, InetAddress host, int puerto)</code> – crea un paquete datagrama en el array de bytes <i>buf</i> de <i>longitud</i> especificada. Se especifica el <i>host</i> y <i>puerto</i> al que el paquete se enviará. Usado este constructor para crear un paquete para envío.</p> <p><code>DatagramPacket(byte[] buf, int longitud)</code> - crea un paquete datagrama en el array de bytes <i>buf</i> de <i>longitud</i> especificada. Usado este constructor para crear un paquete para recepción.</p>
Métodos	<p><code>byte[] getData()</code> – devuelve el buffer de datos</p> <p><code>void setData(byte[] buf)</code> – establece el buffer de datos para el paquete</p> <p><code>InetAddress getAddress()</code> – devuelve la IP de la máquina a la que el datagrama será enviado o desde la que se ha recibido el datagrama</p> <p><code>void setAddress(InetAddress dir)</code> – establece IP de la máquina a la que el datagrama se enviará</p> <p><code>int getPort()</code> – devuelve nº puerto del host remoto al que el datagrama se enviará o desde el que se recibe</p> <p><code>void setPort(int puerto)</code> – establece puerto del host remoto al que el datagrama se enviará</p>



Ejercicios