Asignatura 780014
Programación Avanzada

TEMA 7 –
PROGRAMACIÓN CONCURRENTE DISTRIBUIDA:
PASO DE MENSAJES Y SOCKETS

Programación distribuida: Paso de mensajes y Sockets

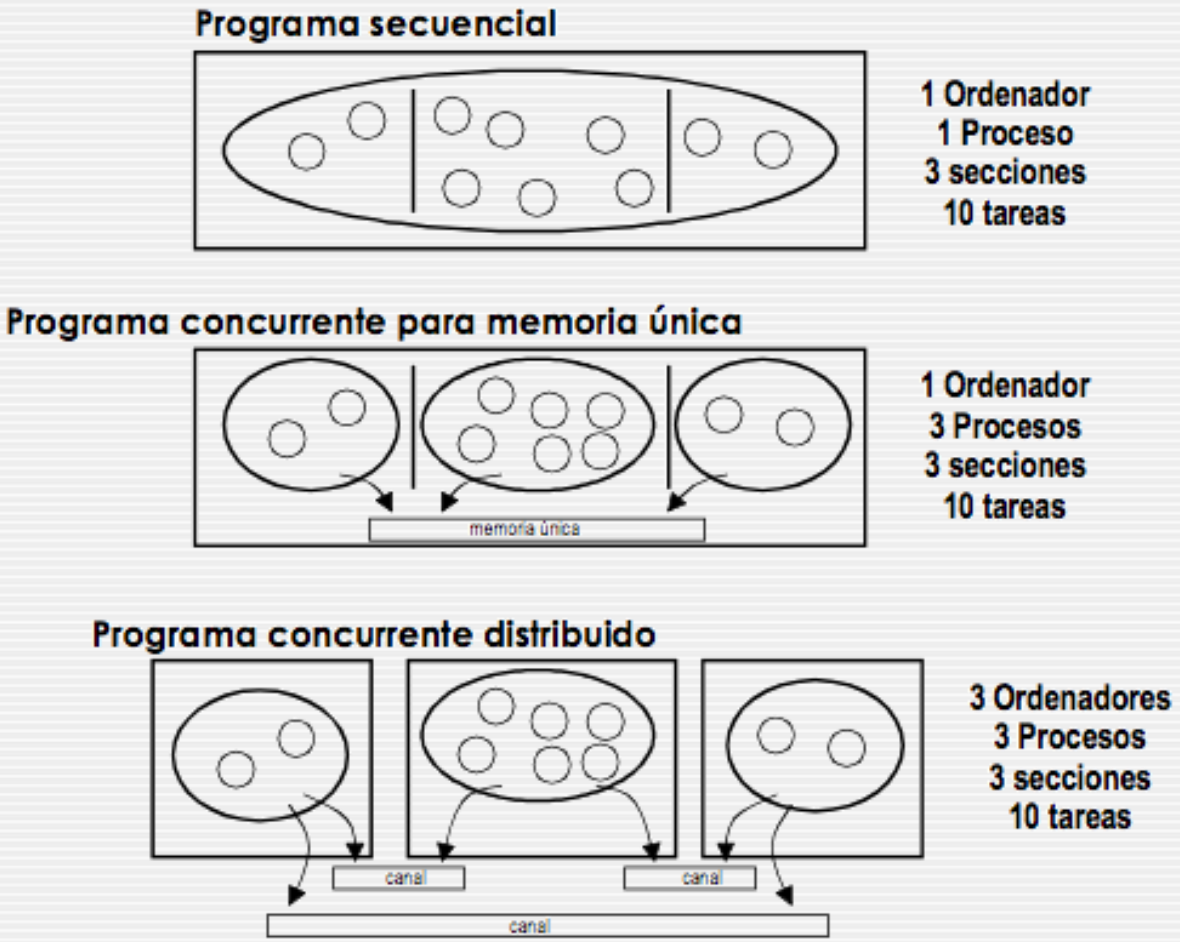
- Objetivo del tema:
  - Conocer los diferentes tipos de programación distribuida, centrándose en paso de mensajes y su implementación mediante Sockets en Java

Índice

1. Computación y sistemas distribuidos
2. Repaso de protocolos
3. Tipos de programación distribuida
   - Paso de mensajes (teórico)
   - Sockets (implementación)

Computación distribuida

- Comunicación y sincronización
  - Entre procesos **concurrentes**
  - Que se ejecutan en **distintas** máquinas
- Se **rompe** la **limitación** de la memoria única



Sistema distribuido

- Definición:
  - Solución **software**
  - Funcionalidad **repartida** entre distintas máquinas
  - Máquinas interconectadas por una **red**
  - Cada máquina con:
    - Su propio procesador (pueden ser varios)
    - Su propia memoria
    - Su propio sistema operativo

Sistema distribuido

- Ya **no hay** una **memoria común** para comunicar
  - Los procesos utilizan **líneas de comunicación** físicas
  - Concurrencia **real**: hay varios de cada HW

- Los sistemas deben ser muy **desacoplados**
  - Intercambiar la **menor** cantidad de información
    - Las líneas trabajan a velocidades inferiores

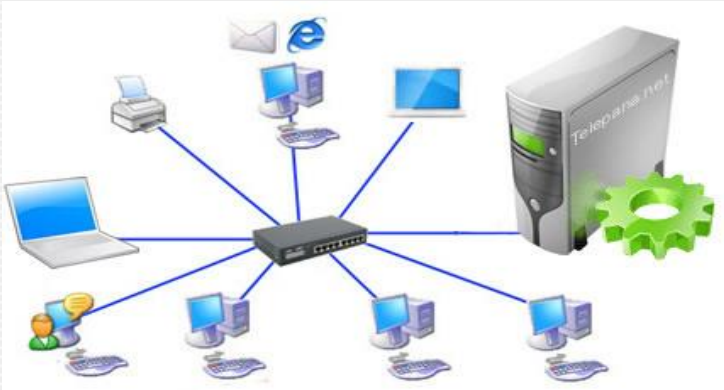
- Hay varios modelos para la creación de los programas

Ventajas y desventajas

- Ventajas:
  - Uso de la **distribución** de cálculo y datos
    - Hardware más **económico**
    - **Mayor potencia** de cálculo global
  - **Compartición** de recursos entre varios equipos
  - Mayor **fiabilidad** global del sistema ante caídas y percances
  - Posibilidad de **escalabilidad** del sistema (añadir HW)
  - Posibilidad de **sistemas abiertos** (especificaciones públicas, independencia de HW)
  - Facilidad en el **despliegue** de nuevas aplicaciones o versiones
- Desventajas:
  - **Complejidad** adicional **en la coordinación** con HW y SO distintos
  - **Necesidad de red** con una buena fiabilidad y rapidez (hoy todo está en red)
  - **Menor seguridad** que en un ordenador único, aislado y bien protegido
    - Una conexión de red es una puerta a ataques, virus, troyanos, etc.

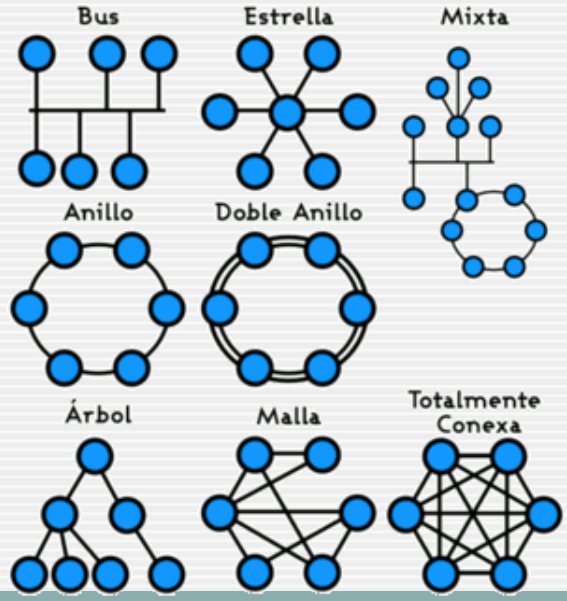
Motivación

- Motivos para el uso de la programación distribuida:
  - Compartir recursos en **tiempo real**
  - Compartir **sin** necesidad de medios removibles
  - Compartir un **recurso único**
  - Aumentar la **seguridad** en recursos
    - Mediante SW distribuido y **niveles** de acceso
  - Aumentar la **potencia** de cálculo
    - **Distribuyendo** el trabajo



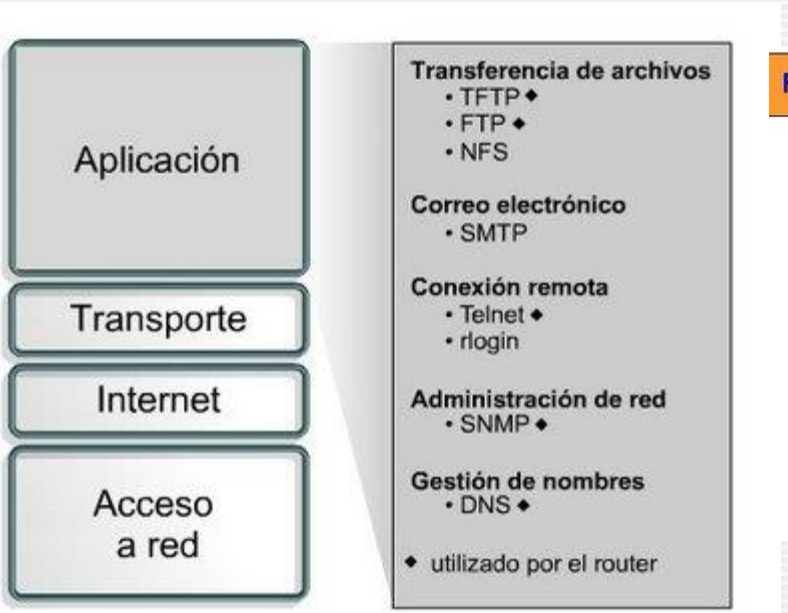
Tipos de línea física

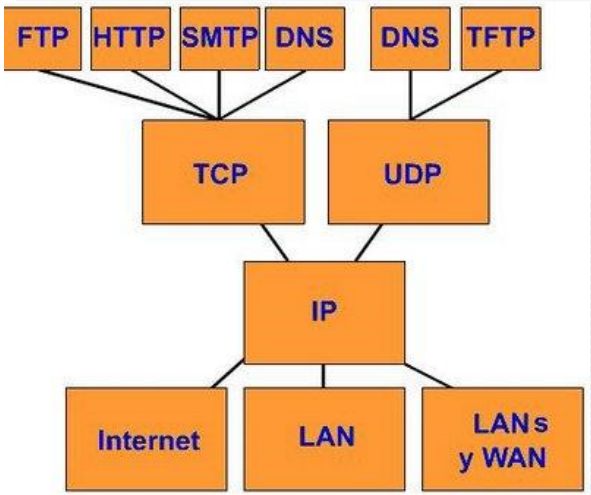
- Directa. Cable serie o infrarrojos
  - Lenta, poco fiable y sólo permite crear sistemas específicos que se basen en la existencia de la conexión
- Directa tipo red, con cable o inalámbricas, entre dos disp.
  - Más eficientes y seguras, pero el sistema seguirá estando limitado a trabajar con dos ordenadores y las ventajas de la programación distribuida no se alcanzan realmente
- Red, interna (intranet) o Internet
  - HW y SW nos aíslan del componente físico de la red
    - Necesita un **sistema de nombrado lógico**
  - **Escenario ideal**
    - Podremos obtener todas las ventajas



Canales y protocolos

- Los canales y las aplicaciones que hacemos con ellos se basan en todos los niveles de red inferiores





Repaso de protocolos

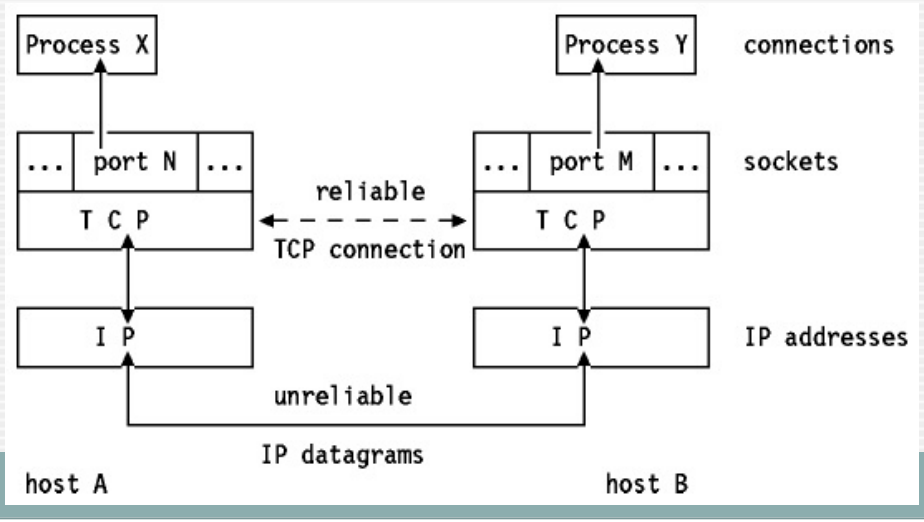
- IP (*Internet Protocol*) – Nivel de Red
  - Usado para la comunicación entre aplicaciones
    - Es el protocolo base de Internet
    - Encargado de mover datos en forma de paquetes entre un origen y un destino
  - Todo dispositivo conectado posee un identificador
    - Dirección IP de 4 bytes=32 bits (ahora más con IP v6)
    - Lo identifica unívocamente
  - La comunicación se basa en la utilización de direcciones IP

Repaso de protocolos

- La dirección IP
  - 4 bytes = 4 cifras del 0 al 255
  - Difícil de recordar, poco natural
  - Se crea el concepto de **nombre de dominio** que es:
    - Una cadena de caracteres
    - Asociada a una dirección IP
- El **Sistema de Nombres de Dominio** (DNS)
  - Creado para la conversión entre los dominios e IP
  - De forma automática

Repaso de protocolos

- TCP (*Transfer Control Protocol*) – Nivel de Transporte
  - Incorporado al protocolo IP
    - Proporciona **fiabilidad** a la comunicación
      - **Valida** que los paquetes de un mensaje **llegan al destino**, y **en orden**, para la recomposición del mensaje original
      - Puede pedir **retransmisión** de paquetes en mal estado o perdidos



Repaso de protocolos

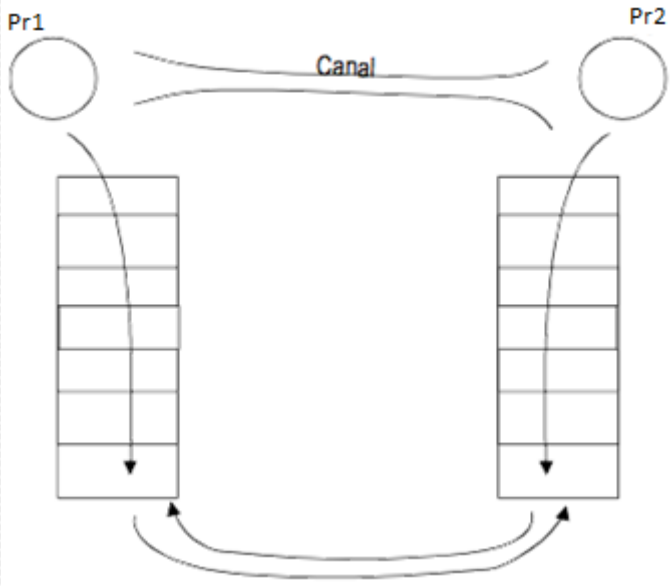
- UDP (*User Datagram Protocol*) – Nivel de Transporte
  - Se utiliza con IP, en comunicaciones donde:
    - **No** es **tan importante que lleguen** todos los mensajes a un destinatario
    - El **orden no** es **relevante**
    - Es preferible la **velocidad** a la fiabilidad en la entrega
  - Desventajas respecto TCP:
    - Menos fiable
    - Necesidad de implementar verificación de envío y sincronización
  - El funcionamiento es similar al del envío de una carta

Tipos de programación distribuida

- Vamos a ver 2 formas de crear los programas distribuidos:
  - Paso de mensajes
  - RPC

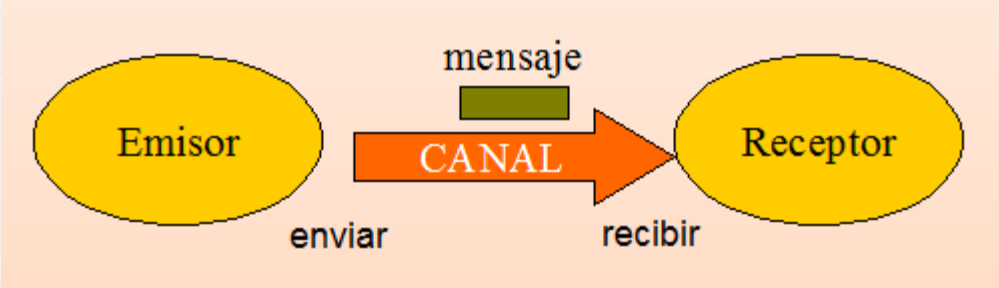
Paso de mensajes

- Canal:
  - "Conexión de red totalmente fiable y con un sistema de nombrado que nos permite crear un 'tubo' para comunicar 2 procesos en 2 máquinas distintas"
- Los canales serán bidireccionales



Paso de mensajes

- Paso de mensajes:
  - **Especificación** abstracta del uso de canales



Paso de mensajes

- Diseñar una aplicación que usa el paso de mensajes consiste en:
  - Crear los **programas** necesarios en cada uno de los ordenadores
  - Establecer los **canales** de comunicación entre ellos
  - Definir el **protocolo** de envío/recepción de datos que requiera la aplicación
- Los programas:
  - Se **crean** independientes
  - Se **compilan**
  - Y la **relación** entre ellos se establece en tiempo de **ejecución**, cuando se crean los canales y se envían los datos con las consiguientes validaciones de tipos de datos

Paso de mensajes

- Cada canal tendrá asociado un **tipo de datos** (cualquiera)
- Los canales serán de 2 tipos: **con y sin capacidad de almacenamiento**
  - Da lugar a 2 tipos de comunicación y a 2 tipos de programación: **comunicación asíncrona y síncrona**
    - **Comunicación síncrona**: los dos extremos de la comunicación tienen que estar dispuestos para que ésta se produzca. También se llama comunicación orientada a la conexión
    - **Comunicación asíncrona**: se pueden enviar datos sin necesidad de que el otro extremo los esté esperando, pudiéndolos recibir más tarde. También se llama comunicación orientada a datagramas
      - En este caso, los datagramas podrán quedar almacenados en la red o en el propio sistema receptor (en las capas inferiores)

Paso de mensajes

- Los programas deberán:
  - **Definir el canal**. Necesitará:
    - Identificador para el canal
    - Dirección de destino del canal
    - Tipo de canal (síncrono o asíncrono)
    - Tipo de datos a enviar
    - Este canal existirá durante toda la ejecución del programa
  - **Enviar datos**. Necesitará:
    - Identificador del canal de envío
    - Datos a enviar (coherentes con el tipo del canal)
    - El resultado de esta operación dirá si se ha llevado a cabo o ha fallado
  - **Recibir datos**. Necesitará:
    - Identificador del canal
    - La variable donde guardar los datos recibidos
    - El resultado de esta operación dirá si se ha llevado a cabo o ha fallado

Paso de mensajes

- El **comportamiento** de las operaciones será distinto dependiendo del **tipo de canal** y del **estado de ocupación** del mismo:
  - ***Send* en canales asíncronos**, siempre enviará el dato y continuará la ejecución con la siguiente instrucción
  - ***Send* en canales síncronos**, sólo enviará el dato y continuará su ejecución si el receptor está preparado para recibir. Si no, esperará (espera no activa) hasta que el destino ejecute *receive*
  - ***Receive* en canales asíncronos**, recibirá el dato y continuará la ejecución con la siguiente instrucción. Si no hubiera dato pendiente en el canal, quedará a la espera de que se produzca un envío
  - ***Receive* en canales síncronos**, sólo recibirá el dato y continuará su ejecución si hay un envío bloqueado a la espera. En caso contrario se bloqueará (sin espera activa) hasta que el envío se produzca

Paso de mensajes

- Ejemplo de productor-consumidor, donde la comunicación se hace en un solo sentido:
  - Un programa Pr1 lee un dato *x* y se lo pasa a otro programa Pr2 que lo escribe. Esta acción la realiza 10 veces

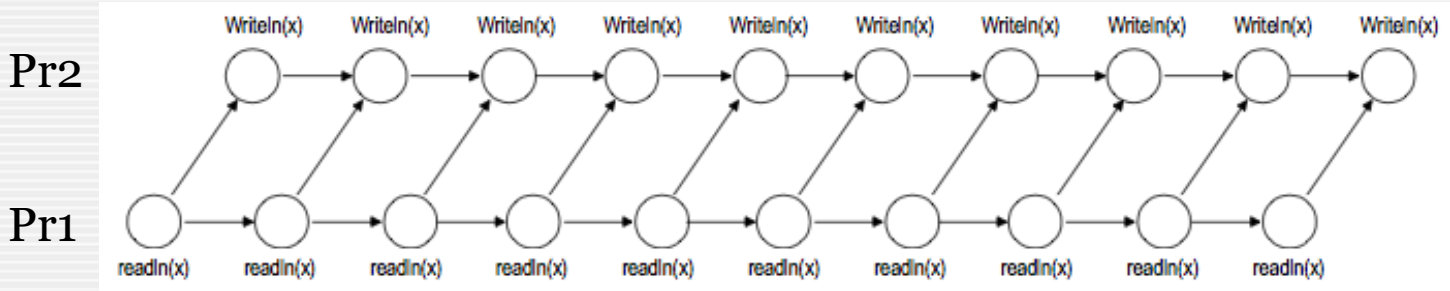
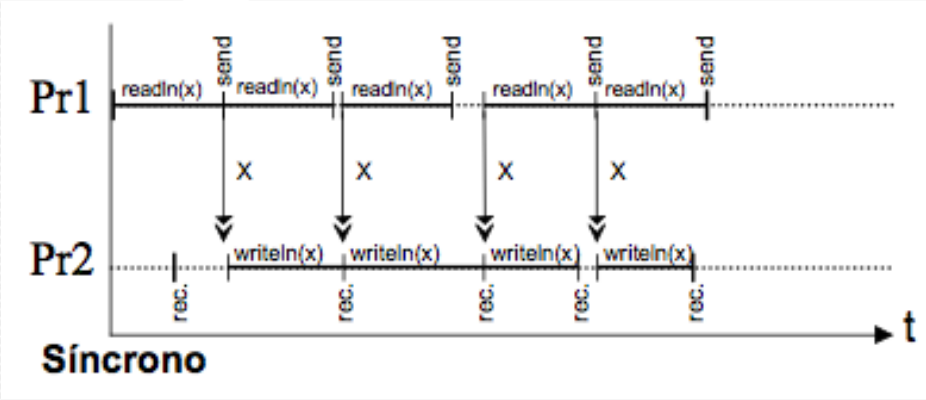


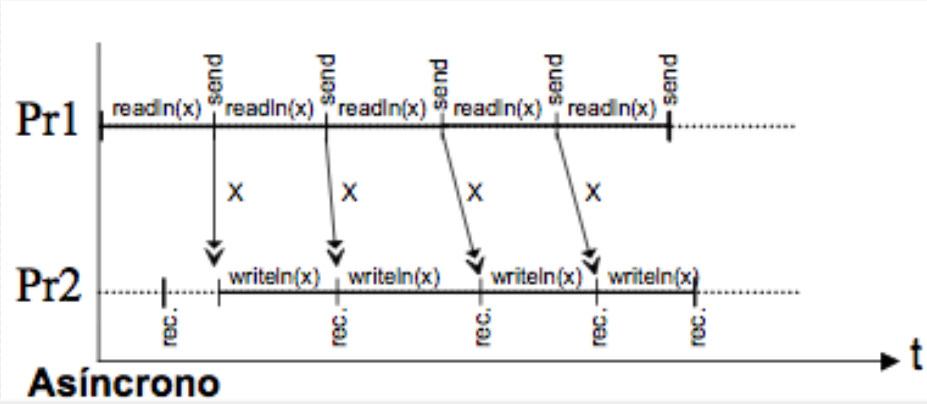
Diagrama de precedencia

Paso de mensajes

- Solución síncrona:

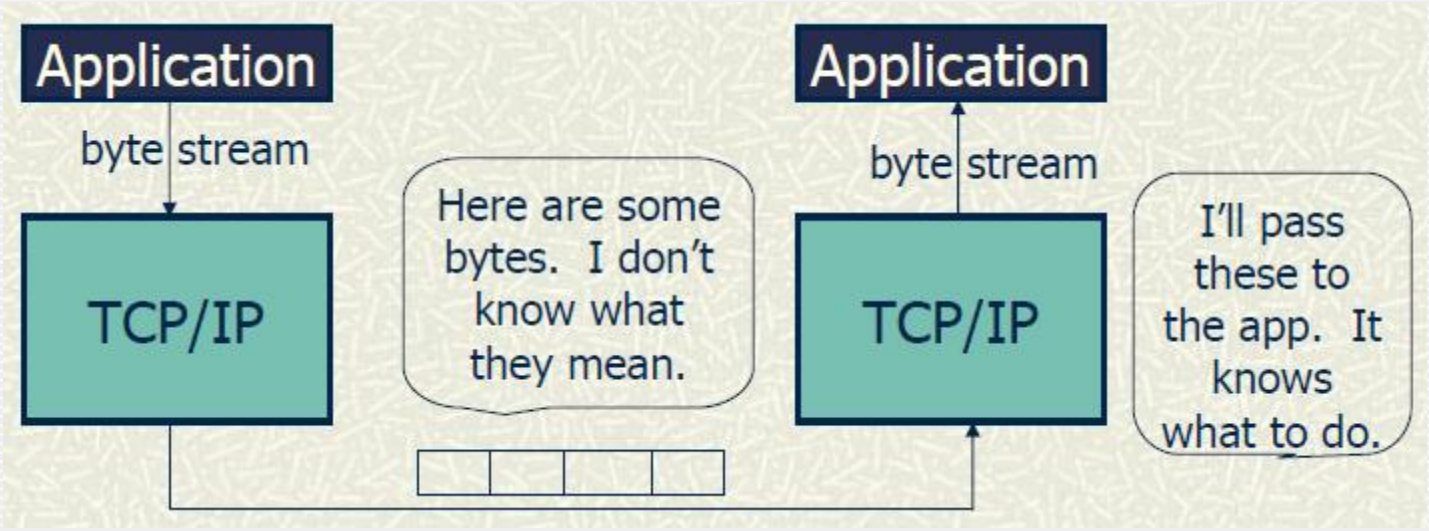


- Solución asíncrona:
  - El productor, si es más rápido, no necesita esperar al consumidor



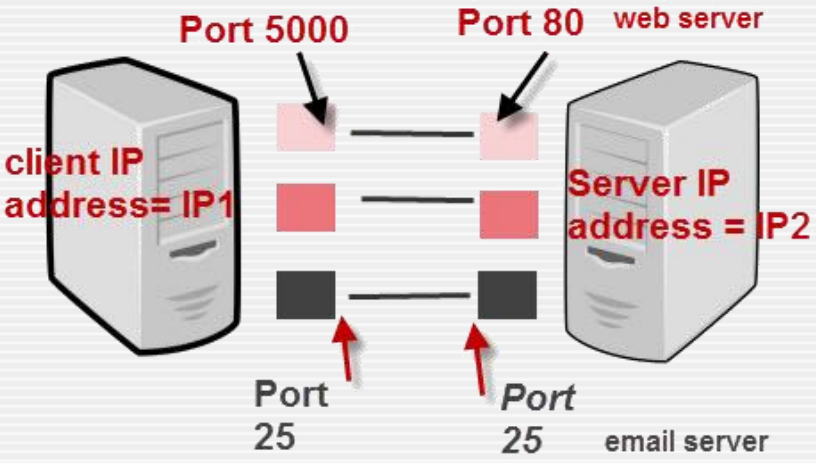
Paso de mensajes

- El protocolo TCP/IP transporta **bytes**
- El protocolo de Aplicación proporciona la **semántica**



Sockets

- Sockets:
  - **Implementación** de paso de mensajes
- Para establecer una comunicación entre dos programas, se crea **un socket en cada máquina**
- Podemos imaginar un cable enchufado en cada extremo al socket correspondiente

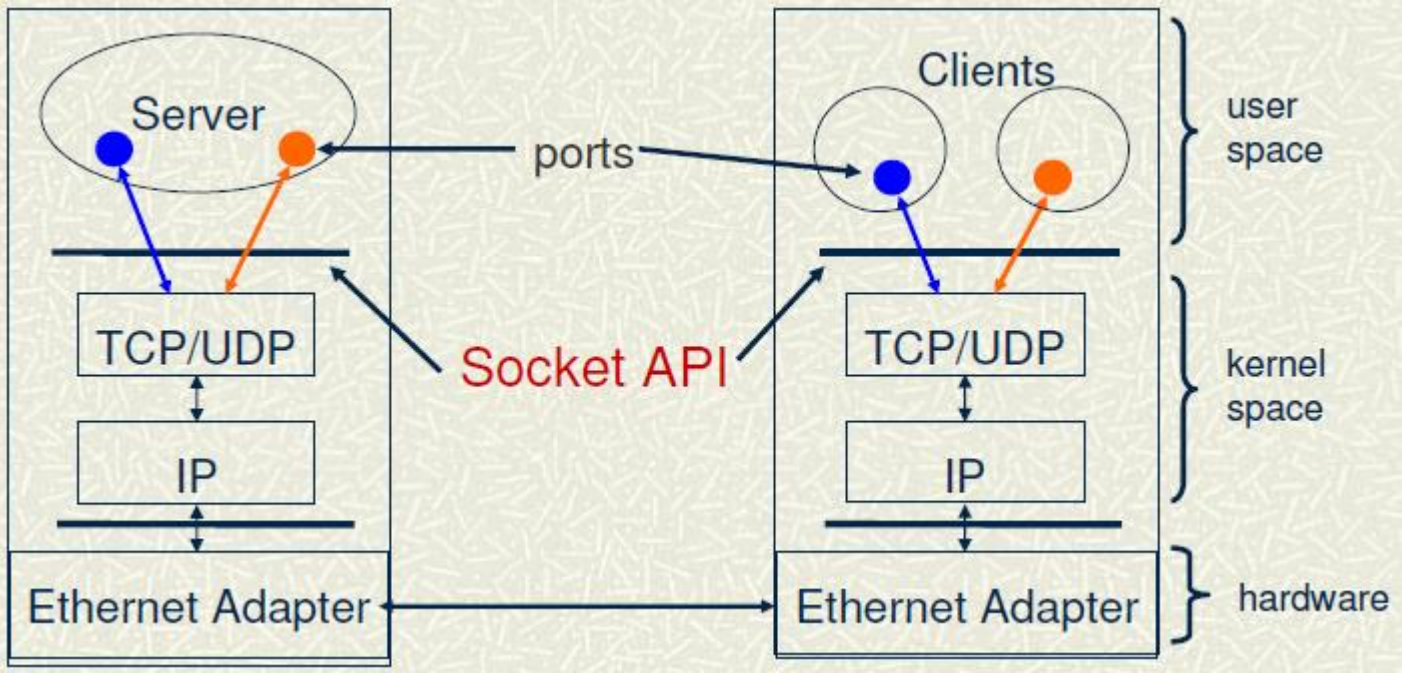


Sockets

- Son una abstracción del sistema operativo (no HW)
  - Su funcionamiento está controlado por el sistema operativo
  - Las aplicaciones los **crean**, los **utilizan** y los **cierran** cuando ya no son necesarios
- Permiten **comunicación** entre procesos
  - Los procesos envían/reciben mensajes a través de su socket
  - Los sockets se comunican entre ellos
- La comunicación en Internet es de socket a socket
  - El **proceso** que está comunicándose se identifica por medio de su **socket**
  - El socket tiene un **identificador**
    - Identificador = dir. IP del ordenador + núm. puerto

Sockets

- La API Socket:
  - Las máquinas intercambian mensajes a través de la red mediante la API Socket
  - Permite a las aplicaciones utilizar los protocolos de la pila TCP/IP
  - Define las operaciones permitidas y sus argumentos



Sockets en Java

- La clase **InetAddress**
  - Permite la manipulación y conocimiento de direcciones y dominios
  - Representa una dirección IP
  - Métodos:
    - `byte[] getAddress()` //Devuelve la dirección IP de un objeto InetAddress
    - `static InetAdress getByName(String host)` //Devuelve un objeto InetAddress representando el host que se le pasa como parámetro
    - `static InetAdress[] getAllByName(String host)`//Devuelve un array de objetos InetAddress correspondientes a todas las direcciones IP asignadas al host
    - `static InetAddress getByAddress(byte[] addr)` //Devuelve un objeto InetAddress, dada una dirección IP
    - `static InetAddress getByAddress(String host, byte[] addr)` //Devuelve un objeto InetAddress a partir del host y la dirección IP dada
    - `static InetAdress getLocalHost()` //Devuelve un objeto InetAddress representando el ordenador local en el que se ejecuta la aplicación

Sockets en Java

- Ejemplo de uso:

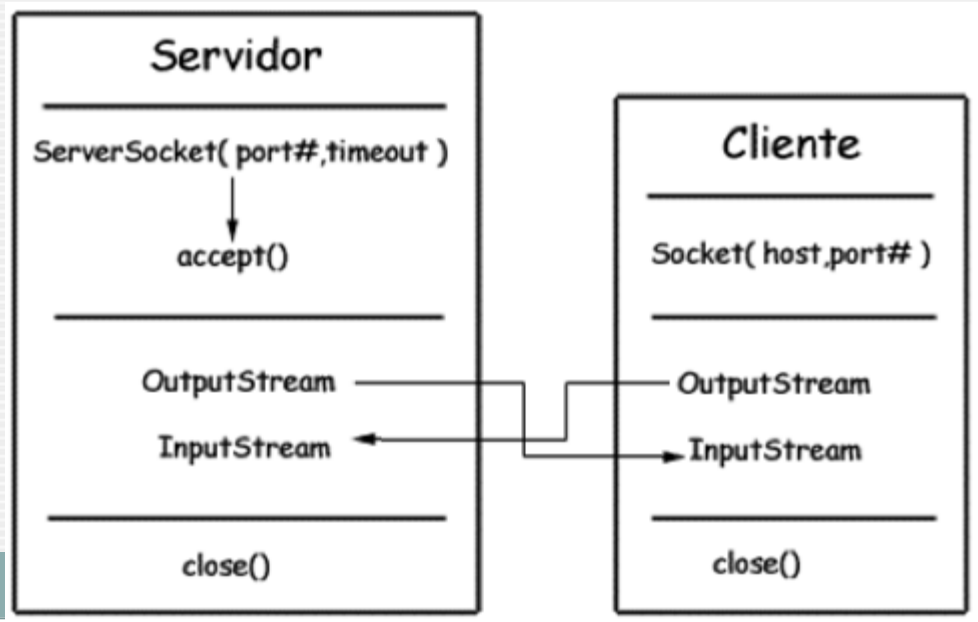
```
public class DireccionesIP {
    public static void main(String[] args) {
        byte[] direccionLocal = {127, 0, 0, 1}; //Dirección IP del Localhost
        InetAddress equipo;
        try {
            // Métodos estáticos de InetAddress para obtener el objeto equipo:
            equipo = InetAddress.getLocalHost();  // Creamos el objeto equipo de la clase InetAddress
            System.out.println("Mi equipo es: "+equipo);
            System.out.println("Su dirección IP es: "+equipo.getHostAddress());
            System.out.println("Su nombre es: "+equipo.getHostName());
            System.out.println("Y su nombre canónico: "+equipo.getCanonicalHostName());
            // Obtenemos el equipo a partir del nombre:
            equipo = InetAddress.getByName("www.google.com");
            System.out.println("el equipo www.google.com es: "+equipo);
            System.out.println("Su dirección IP es: "+equipo.getHostAddress());
            System.out.println("Su nombre es: "+equipo.getHostName());
            System.out.println("Y su nombre canónico: "+equipo.getCanonicalHostName());
            // Obtenemos el equipo a partir de su dirección IP:
            equipo = InetAddress.getByAddress(direccionLocal);
            System.out.println("Mi equipo es: "+equipo);
            System.out.println("Su dirección IP es: "+equipo.getHostAddress());
        } catch(Exception e){}
    }
}
```

Sockets en Java

- Representan cada extremo del canal de comunicación
- La conexión entre sockets es full-dúplex
- Dos tipos:
  - **Sockets Stream (Sockets TCP):**
    - Los datos son transmitidos en **bytes** (no son empaquetados en registros o paquetes)
    - Para establecer la comunicación, utilizan el protocolo **TCP**
    - La **conexión** empezaría una vez que los **dos** sockets estén **conectados**
    - Para crear aplicaciones con este socket en Java: clases **Socket** y **ServerSocket**
  - **Sockets Datagrama (Sockets UDP):**
    - Los datos son enviados y recibidos en paquetes denominados **datagramas**
    - Para establecer la comunicación entre estos sockets se usará el protocolo **UDP**
    - Aunque se deben enviar datos adicionales, este socket es **más eficiente** que el anterior, pero **menos fiable**
    - Para crear aplicaciones con este socket en Java: clase **DatagramSocket**
- La utilización de los sockets es muy similar a la utilización de ficheros

Sockets TCP en Java

- El **servidor establece un puerto y espera** durante un cierto tiempo (*timeout* segundos) a que el cliente establezca la conexión
- Cuando el **cliente solicite** una conexión, el servidor abrirá la conexión socket con el método *accept()*
- El cliente establece una **conexión** con la máquina *host* a través del puerto que se designe en *port#*
- El cliente y el servidor se comunican con manejadores **InputStream** y **OutputStream**



Sockets TCP en Java

- Las **solicitudes** de los clientes se hacen siempre **al mismo puerto** del servidor

- Una vez aceptada la conexión, se establece el **canal** de **socket a socket**

Sockets TCP en Java

- Clases:
  - **Socket**: es el objeto básico en toda **comunicación** a través de Internet, bajo el protocolo **TCP**
    - Proporciona métodos para la entrada/salida a través de *streams* que hacen la lectura y escritura a través de sockets muy sencilla
  - **ServerSocket**: es un objeto utilizado en las aplicaciones servidor para **escuchar las peticiones** que realicen los clientes conectados a ese servidor
    - Este objeto no realiza el servicio, sino que **crea un objeto Socket** en función del cliente para realizar toda la comunicación a través de él

Sockets TCP en Java

- Operaciones:
  - 1º. Apertura de sockets
  - 2º. Creación de flujos de entrada y de salida
  - 3º. Recepción y envío de información
  - 4º. Cierre de flujos
  - 5º. Cierre de sockets

Sockets TCP en Java

- **Apertura** de Sockets (en la parte del **cliente**):

```
Socket miCliente;
miCliente = new Socket("maquina", numeroPuerto);
```

  - *maquina*: nombre de la máquina en donde estamos intentando abrir la conexión
  - *numeroPuerto*: puerto del servidor que está corriendo sobre el cual nos queremos conectar. Los puertos en el rango 0-1023 están reservados. Para las aplicaciones que se desarrollen, asegurarse de seleccionar un **puerto por encima del 1023**

  - Mismo ejemplo usando excepciones:

```
Socket miCliente;
try {
    miCliente = new Socket("maquina", numeroPuerto);
} catch(IOException e) {
    System.out.println(e);
}
```

Sockets TCP en Java

- **Apertura** de Sockets (en la parte del **servidor**):
  - Creamos un objeto **ServerSocket** para que esté atento a las conexiones de clientes potenciales:

```
ServerSocket miServicio;
try {
    miServicio = new ServerSocket(numeroPuerto);
} catch(IOException e) {
    System.out.println(e);
}
```

  - El ServerSocket esperará la petición de un cliente, aceptará la conexión y **crearemos un objeto Socket** para poder enviar y recibir datos:

```
Socket socketServicio = null;
try {
    socketServicio = miServicio.accept(); //Se crea desde el ServerSocket
} catch(IOException e) {
    System.out.println(e);
}
```

Sockets TCP en Java

- **Creación de *streams* de entrada (cliente)**
  - Se puede utilizar la clase **DataInputStream** para recibir las respuestas del servidor:

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

  - La clase DataInputStream:
    - Permite la **lectura de tipos de datos primitivos** de Java de un modo altamente portable
    - Dispone de métodos para leer todos esos tipos como: `readChar()`, `readInt()`, `readDouble()`, `readLong()`, etc.
    - Deberemos utilizar el método que creamos necesario **dependiendo del tipo de dato que esperemos recibir** del servidor

Sockets TCP en Java

- **Creación de *streams* de entrada (servidor)**
  - En el lado del servidor, también usaremos **DataInputStream**, pero en este caso para recibir las entradas que se produzcan de los clientes que se hayan conectado:

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

Sockets TCP en Java

- **Creación de *streams* de salida (cliente)**
  - Podemos crear un *stream* de salida para enviar información al socket del servidor utilizando la clase **DataOutputStream**:

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

  - La clase DataOutputStream:
    - Permite la **escritura de tipos de datos primitivos** de Java de un modo altamente portable
    - Dispone de métodos para escribir todos esos tipos como: `writeInt(i)`, `writeDouble(d)`, `writeLong(l)`, etc.
    - Deberemos utilizar el método adecuado **dependiendo del tipo de dato que vayamos a enviar** al servidor

Sockets TCP en Java

- **Creación de *streams* de salida (servidor)**
  - En el servidor también podemos utilizar la clase **DataOutputStream** para enviar información al cliente:

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

Sockets TCP en Java

- **Recepción y envío de información:**
  - Las clases DataInputStream y DataOutputStream ofrecen métodos para recibir y enviar información, respectivamente:
    - `boolean readBoolean()` / `void writeBoolean(b)`
    - `double readDouble()` / `void writeDouble(d)`
    - `float readFloat()` / `void writeFloat(f)`
    - `String readUTF()` / `void writeUTF(s)`
    - …
  - Ejemplo de uso:
    - `double d = entrada.readDouble();`
    - `salida.writeDouble(d);`
  - Se debe **usar el método adecuado en cada caso**, dependiendo del tipo de datos que se desee enviar y recibir en cada momento

Sockets TCP en Java

- **Cierre de *streams* y de sockets**
  - Hay que cerrar los canales de I/O abiertos por la aplicación
  - **El orden de cierre es relevante**: cerrar **primero los *streams*** relacionados con un socket **antes que el propio socket**, ya que de esta forma evitamos posibles errores de escrituras o lecturas sobre descriptores ya cerrados

En la parte del cliente:

```
try {
     salida.close();
     entrada.close();
     miCliente.close();
} catch(IOException e) {
     System.out.println(e);
}
```

En la parte del servidor:

```
try {
     salida.close();
     entrada.close();
     socketServicio.close();
     miServicio.close();
} catch(IOException e) {
     System.out.println(e);
}
```

Sockets TCP en Java

- La clase **Socket**:
  - Constructor (entre otros):
    - `Socket(InetAddress address, int port)` //Crea un socket stream y lo conecta al canal (ip+puerto)
  - Métodos:
    - `Socket s;  InputStream ent;  OutputStream sal;`
    - `ent=s.getInputStream()` //Obtiene un *stream* para leer datos del socket
    - `sal=s.getOutputStream()` //Obtiene un *stream* para escribir en el socket
    - `InetAddress getInetAddress()` //Obtiene la dirección remota
    - `InetAddress getLocalAddress()` //Obtiene la dirección local
    - `int getPort()` //Obtiene el puerto remoto
    - `int getLocalPort()` //Obtiene el puerto local
    - `void close()` //Cierra el socket

Sockets TCP en Java

- La clase **ServerSocket**:
  - Constructor (entre otros):
    - `ServerSocket(int puerto)` //Abre un **socket** en modo escucha en el **puerto** indicado. Si **puerto=0,** elige un puerto cualquiera
  - Métodos:
    - `ServerSocket ss;`
    - `Socket s;`
    - `s = ss.accept();` //Acepta una conexión de un cliente y devuelve un socket asociado a ella. La operación se **bloquea** hasta que se establece la conexión
    - `ss.close();` //Cierra el socket servidor

Sockets TCP en Java

- Ejemplo (servidor):

```
public class Servidor {
    public static void main(String args[]) {
        ServerSocket servidor;
        Socket conexion;
        DataOutputStream salida;
        DataInputStream entrada;
        int num = 0;
        try {
            servidor = new ServerSocket(5000); //Creamos un ServerSocket en el Puerto 5000
            System.out.println("Servidor Arrancado....");
            while (true) {
                conexion = servidor.accept(); //Esperamos una conexión
                num++;
                System.out.println("Conexión n."+num+" desde: "+conexion.getInetAddress().getHostName());
                entrada = new DataInputStream(conexion.getInputStream()); //Abrimos los canales de E/S
                salida = new DataOutputStream(conexion.getOutputStream());
                String mensaje = entrada.readUTF(); //Leemos el mensaje del cliente
                System.out.println("Conexión n."+num+" mensaje: "+mensaje);
                salida.writeUTF("Buenos días " + mensaje); //Le respondemos
                entrada.close(); //Cerramos los flujos de entrada y salida
                salida.close();
                conexion.close(); //Y cerramos la conexión
            }
        } catch (IOException e) { }
    }
}
```

Sockets TCP en Java

- Ejemplo (cliente):

```
public class Cliente {
    public static void main(String args[]) {
        Socket cliente;
        DataInputStream entrada;
        DataOutputStream salida;
        String mensaje, respuesta;
        try {
            cliente = new Socket(InetAddress.getLocalHost(), 5000); //Creamos el socket para conectarnos
                                                                    //al puerto 5000 del servidor
            entrada = new DataInputStream(cliente.getInputStream()); //Creamos los canales de E/S
            salida = new DataOutputStream(cliente.getOutputStream());
            mensaje="Miguel Sánchez";
            salida.writeUTF(mensaje); //Enviamos un mensaje al servidor
            respuesta = entrada.readUTF(); //Leemos la respuesta
            System.out.println("Mi mensaje: "+mensaje);
            System.out.println("Respuesta del Servidor: "+respuesta);
            entrada.close(); //Cerramos los flujos de entrada y salida
            salida.close();
            cliente.close(); //Cerramos la conexión
        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

//Dirección del servidor

Sockets TCP en Java

- Salida del programa anterior:

Servidor:

```
run:
Servidor Arrancado....
Conexión n.1 desde: DESKTOP-8386928
Conexión n.1 mensaje: Miguel Sánchez
BUILD STOPPED (total time: 4 seconds)
```

Cliente:

```
run:
Mi mensaje: Miguel Sánchez
Respuesta del Servidor: Buenos días Miguel Sánchez
BUILD SUCCESSFUL (total time: 0 seconds)
```

Sockets UDP en Java

- **No** se establece una **conexión** previa al envío/recepción de datos
- Los datos se envían en bloques (**datagramas**)
  - Los datagramas son objetos de la clase **DatagramPacket**
  - Cada datagrama contiene:
    - Una **cabecera**: dirección origen y destino, puerto, longitud paquete, checksum, etc.
    - **Cuerpo**: los datos del paquete
  - Los datagramas pueden no llegar en el mismo orden en el que se enviaron (o no llegar nunca)
- Permiten difusiones (broadcast y multicast)
- Para enviar o recibir datagramas (DatagramPacket) se utilizan los objetos **DatagramSocket** o **MulticastSocket**
  - Se diferencian por el tipo de canal:
    - **DatagramSocket**: Punto a punto
    - **MulticastSocket**: Multipunto

Sockets UDP en Java

- Clases:
  - **DatagramSocket**: utilizada para implementar sockets UDP para transferir datagramas
    - La **comunicación** por estos sockets es **muy rápida** porque **no hay** que perder tiempo estableciendo la **conexión** entre cliente y servidor
  - **DatagramPacket**: representa un paquete datagrama
    - Se usará para enviar y recibir información a través de sockets UDP

Sockets UDP en Java

- Ejemplo (servidor):

```
public class Servidor {
     public static void main(String[] args) throws IOException {
        DatagramSocket socket = new DatagramSocket(4445);
        String recibido=null;
        do {
             try {
                byte[] buf = new byte[128];
                DatagramPacket paquete = new DatagramPacket(buf, buf.length);
                socket.receive(paquete);
                recibido = new String(paquete.getData()); //Pasamos datos a String
                String mensaje = "Eco: " + recibido;
                buf = mensaje.getBytes(); //Para enviarlo necesitamos pasarlo a array de bytes
                InetAddress destino = paquete.getAddress(); //El destino lo sacamos del paquete recibido
                int puerto = paquete.getPort(); //Ídem con el puerto
                paquete = new DatagramPacket(buf, buf.length, destino, puerto);
                socket.send(paquete);
             } catch (IOException e) { }
        } while(!recibido.equals("ciao"));
        socket.close();
    }
}
```

Sockets UDP en Java

- Ejemplo (cliente):

```
public class Cliente {
   public static void main(String[] args) throws IOException {
        DatagramSocket socket = new DatagramSocket();
        InetAddress destino = InetAddress.getByName("localhost");
        BufferedReader consola = new BufferedReader(new InputStreamReader(System.in));
        String entradaConsola = null;
        do {
             entradaConsola = consola.readLine();
             byte[] buf = entradaConsola.getBytes(); //Para enviar tiene que ser byte[]
             DatagramPacket paquete = new DatagramPacket(buf, buf.length, destino, 4445);
             socket.send(paquete);
             buf = new byte[128];
             paquete = new DatagramPacket(buf, buf.length);
             socket.receive(paquete);
             String recibido = new String(paquete.getData()); //Pasamos datos a String
             System.out.println(recibido);
        } while (!entradaConsola.equals("ciao"));
        socket.close();
   }
}
```

Ejercicios

- 1.- Ejecutar los códigos utilizados en esta presentación.
- 2.- Crear una aplicación distribuida con sockets TCP, a la que se le envíen dos números y nos devuelva como resultado la multiplicación de estos.
- 3.- Crear una aplicación distribuida con sockets TCP, a la que enviemos nuestra fecha de nacimiento y nos devuelva como resultado la edad que tenemos.
- 4.- Repetir los ejercicios 2 y 3, pero con sockets UDP.