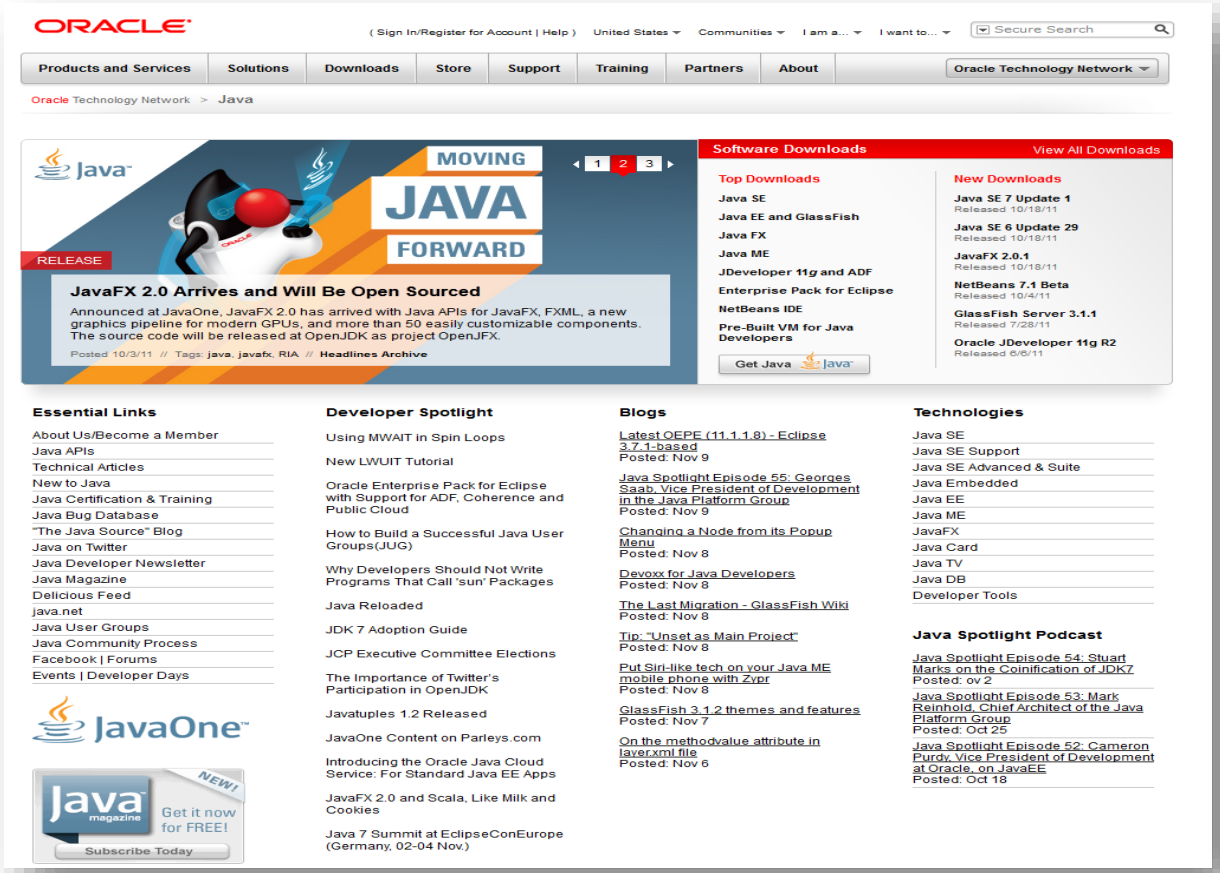
Tema 3: Java y los Sockets

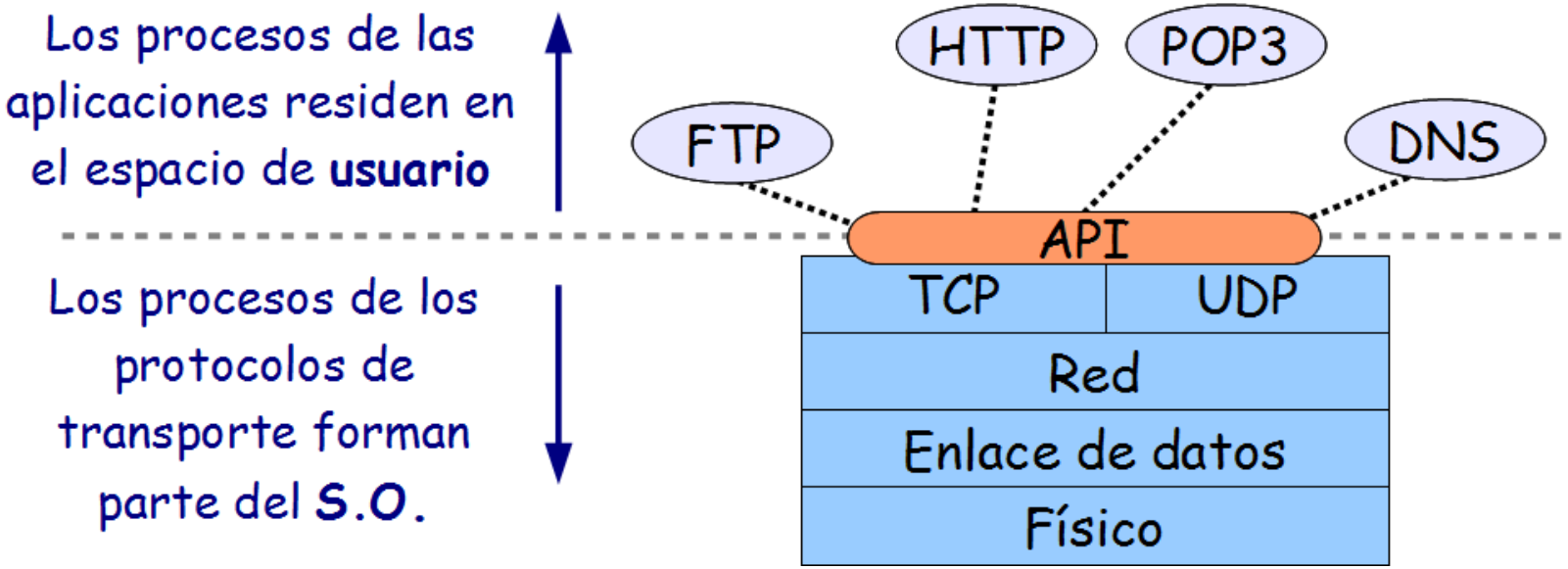


Bibliografía:

- [Kurose10] Apartados 2.1, 2.7, 2.8
- https://docs.oracle.com/en/java/javase/15/docs/api/index.html

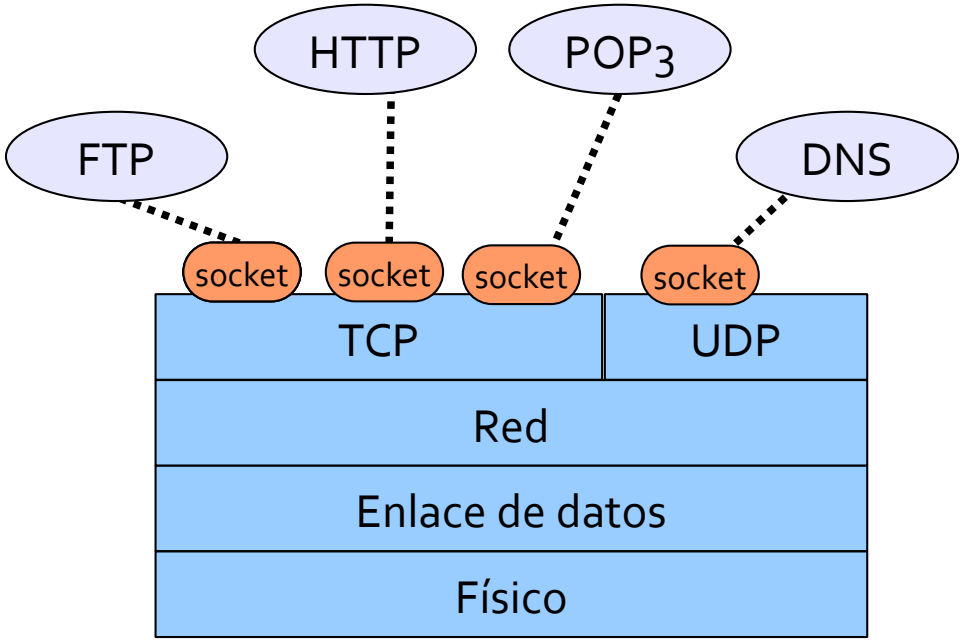
La abstracción de los sockets

- Clientes y servidores utilizan protocolos de transporte
- Se necesita un mecanismo para ponerlos en contacto
  - API (*Application Programming Interface*)



API socket

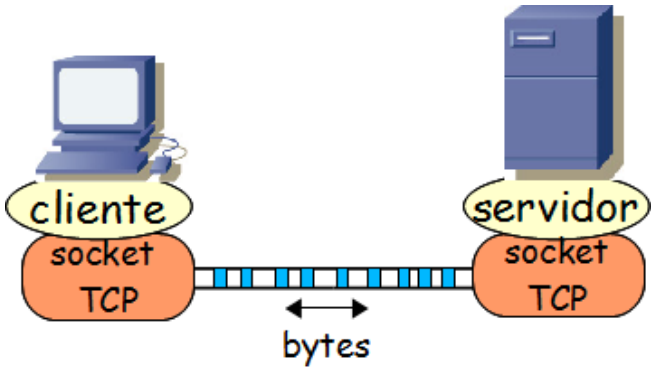
- Originalmente diseñado en BSD Unix
  - Permite a las aplicaciones utilizar los protocolos de la pila TCP/IP
- Define las operaciones permitidas y sus argumentos
  - Parecido a la forma de acceder a los ficheros en Unix
  - Operaciones: open, read, write, close
  - Cuando se abre un fichero obtenemos un descriptor
- Es un estándar de facto



Tipos de sockets

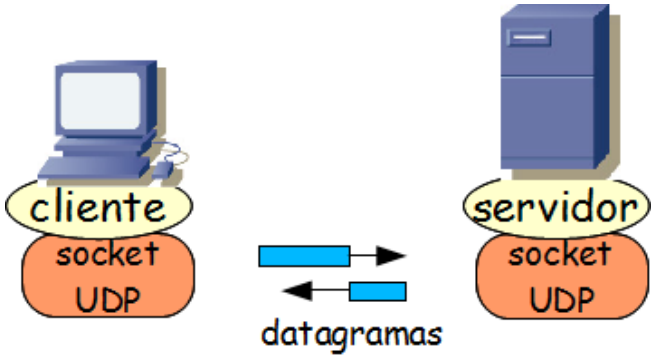
Sockets TCP

- Las aplicaciones piden al S.O. una comunicación controlada por TCP:
  - Orientada a la conexión
  - Comunicación fiable y ordenada
- También se denominan sockets de tipo Stream



Sockets UDP

- Las aplicaciones piden al S.O. una comunicación controlada por UDP:
  - Transferencia de bloques de datos
  - Sin conexión ni fiabilidad ni entrega ordenada
  - Permite difusiones
- También se denominan sockets de tipo Datagram



- Dentro del paquete java.net existen tres clases de sockets:
  - `Socket` Cliente TCP
  - `ServerSocket` Servidor TCP
  - `DatagramSocket` Cliente/Servidor UDP

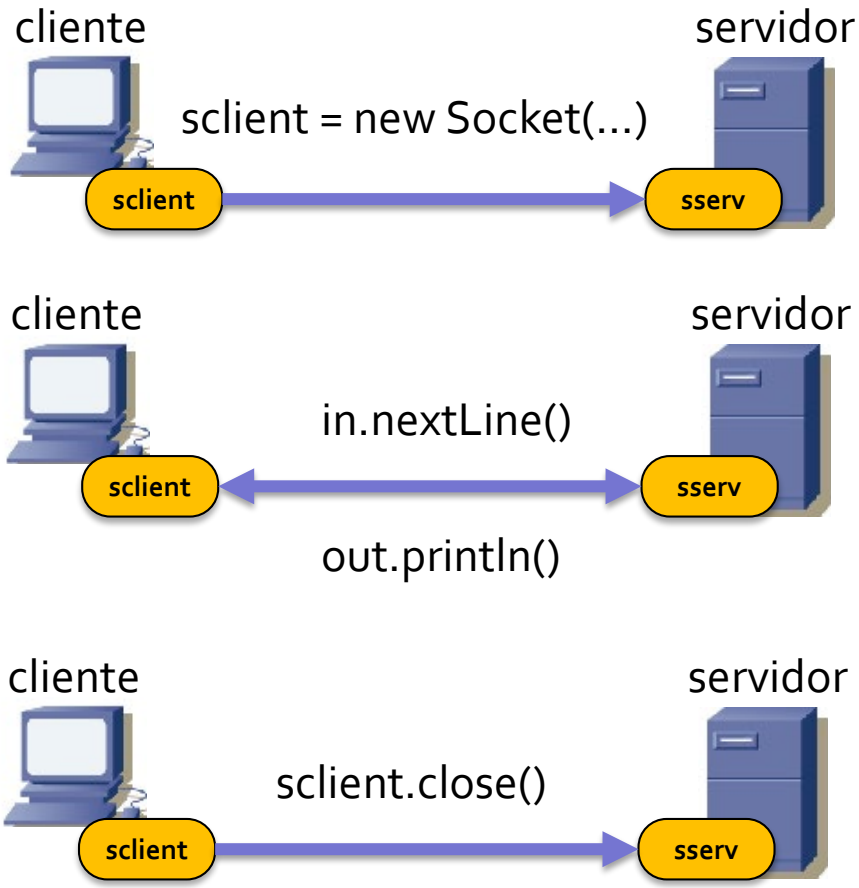
- También hay otras clases auxiliares que facilitan la programación de aplicaciones en red
  - Ver referencias

Servidor:

- Ha de estar en ejecución
- Debe haber creado un socket (por ejemplo `sserv`) donde recibir a los clientes que conectan con él

Cliente:

- Crea un socket (por ejemplo `sclient`) y lo conecta con el del servidor
- Transfiere información
- Cierra el socket y la conexión (a veces lo hace el servidor)



- Clase `Socket`
- Constructores más frecuentes
  - Crean un socket y lo conectan con el servidor y el puerto indicados
  - `Socket(String nombre, int puerto)`
  - `Socket(InetAddress dirIP, int puerto)`
  - Ejemplos de uso de los constructores:
    - `Socket s = new Socket("www.upv.es",80);`
    - `Socket s = new Socket(dirIP,pto_http);`
      - `dirIP se supone un objeto de tipo InetAddress`
      - `pto_http se supone un int`

- Algunos métodos importantes de la clase **`Socket`**
  - **`close()`**: Cierra el socket
  - **`InputStream getInputStream()`**
    - Proporciona un descriptor para leer del socket
    - **`InputStream`** proporciona un flujo de bytes
      - Se puede leer un byte: **`read()`**
      - O un grupo de bytes: **`read(byte[] b)`**
  - **`OutputStream getOutputStream()`**
    - Proporciona un descriptor para escribir en el socket
    - **`OutputStream`** admite un flujo de bytes
      - Se puede escribir un byte: **`write(int b)`**
      - O un grupo de bytes: **`write(byte[] b)`**

Gestión de los flujos de entrada con Scanner

- **`Scanner`** facilita la lectura de un flujo de bytes
  - Se puede leer la palabra siguiente (String): **`next()`**
  - el siguiente entero: **`nextInt()`**
  - el siguiente número en coma flotante: **`nextFloat()`**
  - o la siguiente línea: **`nextLine()`**
- Ejemplo:

```
import java.util.Scanner;
...

Scanner entrada=new Scanner(s.getInputStream());
entrada.nextLine();
```

Gestión de los flujos de entrada con Scanner

- Si el servidor (S) envía un **mensaje de texto de varias líneas** y a continuación **cierra el socket**
  - El cierre del socket provoca el cierre de la conexión TCP en los dos extremos C y S
  - Se puede leer en el cliente las líneas recibidas mediante el Scanner **`entrada`** conectado al flujo de entrada del socket **`s`** y el método **`hasNext()`**
  - **¡¡OJO!! Sólo sirve si el servidor cierra la conexión tras enviar los datos**

```
Scanner entrada = new Scanner(s.getInputStream());
...
while (entrada.hasNext()){
   System.out.println(entrada.nextLine());
}
```

- **Mucho cuidado con el ejemplo anterior:**

```
Scanner entrada = new Scanner(s.getInputStream());
...
while (entrada.hasNext()){
   System.out.println(entrada.nextLine());
}
```

- **El cliente se bloqueará en el bucle si el servidor no cierra la conexión.**

- Mejor evitar bucles de este estilo si no sabemos lo que estamos haciendo

Gestión de los flujos de entrada con Scanner

- A veces también es interesante leer de la entrada estándar (teclado)
- Ejemplo:

```
import java.util.Scanner;
...

Scanner leeTeclado=new Scanner(System.in);
System.out.println("Nombre del servidor destino: ");
String servidor = leeTeclado.nextLine();
System.out.println("Introduce el puerto destino: ");
int pto = leeTeclado.nextInt();
```

Gestión de los flujos de salida

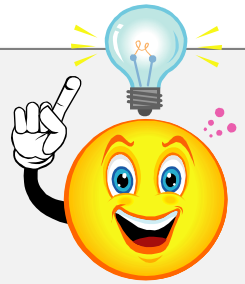
- **`PrintWriter`** permite enviar texto (caracteres)
  - Tiene métodos que permiten escribir texto:
    `print(String s)`, **`println(String s)`**, **`printf(String s)`**
- Ejemplo:

```
PrintWriter salida = new PrintWriter(s.getOutputStream());
salida.printf("GET / HTTP/1.0" + "\r\n");
salida.flush();
```

- Ejemplo:

```
PrintWriter salida = new PrintWriter(s.getOutputStream(), true);
salida.println("GET / HTTP/1.0" + "\r\n");
// no hace falta salida.flush()- sólo con println + true
```

Cliente TCP básico



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

public class ClienteTCP0 {
    public static void main(String args[])
                    throws UnknownHostException, IOException {
      Socket s=new Socket("zoltar.redes.upv.es", 7);
      System.out.println("Conectado");
      PrintWriter salida = new PrintWriter(s.getOutputStream());
      salida.printf("Hola Mundo!\r\n");
      salida.flush();
      Scanner entrada=new Scanner(s.getInputStream());
      System.out.println(entrada.nextLine());
      s.close();
      System.out.println("Desconectado");
    }
}
```

- Este cliente se conecta al servidor ECHO (puerto 7), envía y recibe una línea y después cierra la conexión

Excepciones de la clase Socket

- Al intentar conectar un socket con un servidor hay dos problemas típicos, que generan excepciones en Java:
  - No se puede resolver el nombre del servidor
    - Se genera una excepción **`UnknownHostException`**
  - Se ha resuelto el nombre pero no se puede establecer la conexión
    - Se genera una excepción **`ConnectException`**
- Al intentar cerrar el socket puede generarse una excepción **`IOException`**
- Estas excepciones se pueden manejar mediante

```
try{...}
catch(Exception e) {instrucciones al producirse
la excepción}
```

- Hay que capturar primero las excepciones más específicas
  - **`UnknownHostException`**, **`ConnectException`**

IOException y sus subclases

➔ https://docs.oracle.com/javase/7/docs/api/java/io/IOException.html

java.io

Class IOException

java.lang.Object
  └ java.lang.Throwable
    └ java.lang.Exception
      └ java.io.IOException

All Implemented Interfaces:

Serializable

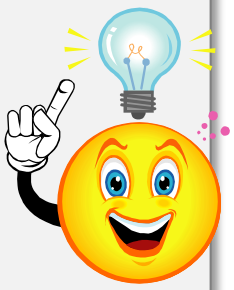
Direct Known Subclasses:

ChangedCharSetException, CharacterCodingException, CharConversionException, ClosedChannelException, EOFException, FileLockInterruptionException, FileNotFoundException, IIOException, InterruptedIOException, MalformedURLException, ObjectStreamException, ProtocolException, RemoteException, SocketException, SSLException, SyncFailedException, UnknownHostException, UnknownServiceException, UnsupportedEncodingException, UTFDataFormatException, ZipException

* ConnectException es una subclase de SocketException

Cliente TCP modificado

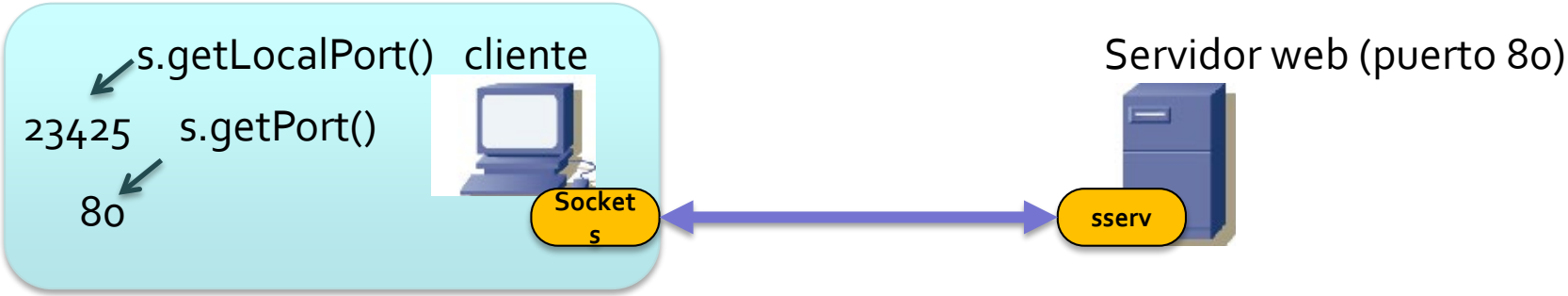
```
public class ClienteTCP1 {
    public static void main(String args[]) {
    try{
          Socket s=new Socket("noexiste.redes.upv.es", 7);
          System.out.println("Conectado");
          PrintWriter salida = new PrintWriter(s.getOutputStream());
          salida.print("Hola Mundo!\r\n");
          salida.flush();
          Scanner entrada=new Scanner(s.getInputStream());
          System.out.println(entrada.nextLine());
          s.close();
          System.out.println("Desconectado");
        } catch (UnknownHostException e) {
            System.out.println("Host desconocido");
            System.out.println(e);
        } catch (ConnectException e) {
            System.out.println("No se puede conectar");
            System.out.println(e);
        } catch (IOException e) {
            System.out.println(”Fallo al cerrar el socket");
            System.out.println(e);
        }

    }
}
```



Obteniendo información de la conexión

- Métodos de la clase **`Socket`** para obtener información de una conexión ya establecida
  - Información local
    - **`int getLocalPort()`**
    - **`InetAddress getLocalAddress()`**
  - Información remota
    - **`int getPort()`**
    - **`InetAddress getInetAddres()`**

InetAddress es la clase que se utiliza para almacenar direcciones IP en Java



Cliente TCP con información de la conexión

```
public class EjemploInfo {
    public static void main(String args[]) {
    try{
         Socket s=new Socket("zoltar.redes.upv.es", 7);
        System.out.println("Conectado");

        System.out.print("IP local:");
        System.out.println(s.getLocalAddress());
        System.out.print("Puerto local:");
        System.out.println (s.getLocalPort());
        System.out.print("IP remota:");
        System.out.println(s.getInetAddress());
        System.out.print("Puerto remoto:");
        System.out.println(s.getPort());
        s.close();
        System.out.println("Desconectado");
        } catch (UnknownHostException e) {
            System.out.println("Host desconocido");
            System.out.println(e);
       } catch (ConnectException e) {
            System.out.println("No se puede conectar");
            System.out.println(e);
        } catch (IOException e) {
            System.out.println("Fallo al cerrar el socket");
            System.out.println(e);
        }

    } // main
} // class
```