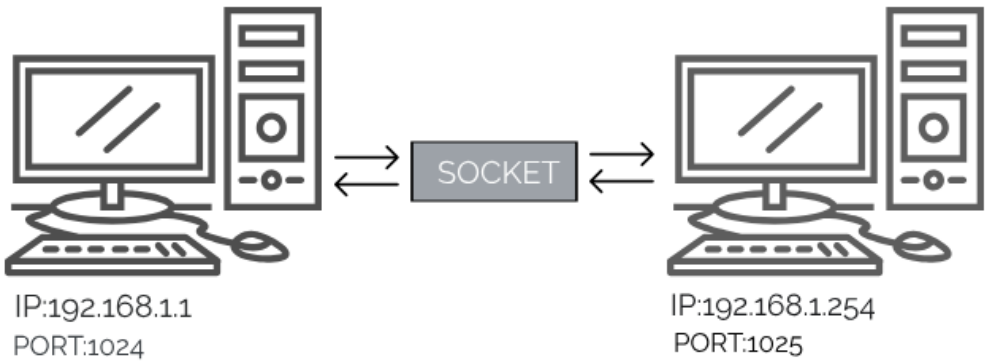
UD03: Programación de comunicaciones en red



1. Fundamentos de la programación de comunicaciones en red

La tercera acepción del diccionario de la lengua de la Real Academia Española define la comunicación como «transmisión de señales mediante un código común al emisor y al receptor». Aunque parcial, esta es una buena definición para la comunicación entre dispositivos informáticos conectados a través de una red. Por ser los sistemas informáticos deterministas e incapaces de resolver las ambigüedades propias del lenguaje, para que una comunicación sea efectiva es necesario disponer de una serie de elementos y técnicas.

Para que una comunicación entre sistemas informáticos sea posible se debe disponer de los siguientes componentes o elementos:

- Los interlocutores que participan en la comunicación.
- Un mensaje o mensajes para transmitir.
- Un canal de comunicación.
- Un lenguaje común para todos los interlocutores en el que se definen los símbolos aceptados, sus combinaciones y el significado de estas.
- Un mecanismo para poder identificar al emisor y al receptor de los mensajes de entre todos los posibles participantes de la comunicación.
- Un sistema para garantizar la integridad de los mensajes enviados y recibidos.
- Un sistema para asegurar que el orden de recepción de los mensajes es el correcto.
- Un método para confirmar la recepción de los mensajes o para notificar al emisor que alguno de los mensajes no ha llegado completamente, ya sea por pérdida o por deterioro de alguna de sus partes.

Es decir, además de los interlocutores, el canal y los mensajes en sí, se necesitan protocolos.

Aplicado a la comunicación, un protocolo es un conjunto de reglas formales que determina cualquier aspecto relacionado con las actividades propias de dicha comunicación, desde la manera en la que se identifican a los participantes hasta el mecanismo según el cual se confirma que un mensaje ha llegado de forma completa y correcta a su destinatario.

El conjunto de protocolos sobre el que se construye internet se conoce como familia de protocolos de internet. Estos protocolos abarcan todo tipo de servicios, desde la web hasta el correo electrónico. Una de sus ventajas más interesante es que son protocolos abiertos. Esto significa que no pertenecen a ninguna empresa u organización, pudiendo disponer de ellos de manera libre y gratuita. Esta característica, además de su robustez, han provocado que la familia de protocolos de internet se utilice tanto para dicha red como para redes de todo tipo y tamaño, incluidas las locales: se puede construir una red de ordenadores local basada en los protocolos de internet sin disponer de acceso a internet. El diseño es
tan versátil que sirve tanto para conectar dos dispositivos como millones de ellos.

Algunos de estos protocolos son la base sobre la que se desarrollan prácticamente todas las aplicaciones y servicios basados en comunicaciones.

En este apartado se presentan los protocolos fundamentales para la comunicación entre ordenadores. Estos son: IP, TCP y UDP. Cada uno de ellos tiene un objetivo, unas características propias y debe utilizarse en un contexto concreto. Estos protocolos son la base de las comunicaciones de internet y dan soporte al resto de protocolos de más alto nivel como HTTP o FTP.

En Java, como en prácticamente todos los lenguajes, existen clases y bibliotecas que permiten utilizar estos protocolos de manera sencilla, aceptando el desarrollo de aplicaciones capaces de comunicarse a bajo nivel utilizando un sistema de comunicación conocido como socket. Mediante el uso de socket, se puede establecer un canal de comunicación entre dos puntos (dos dispositivos) capaz de permitir el envío y recepción de datos en ambos sentidos.

> La mayoría los ejemplos presentados en esta unidad se han desarrollado y probado en un único ordenador, utilizando localhost como host cliente y servidor. Hay que recordar que, aunque ejecutándose en una única máquina física, las aplicaciones desarrolladas en Java se ejecutan sobre una máquina virtual (JVM o Java Virtual Machine), por lo que cada aplicación Java que se está ejecutando simultáneamente en un ordenador lo hace en una máquina virtual distinta. No obstante, si se dispone de una red de ordenadores, resultará interesante poder probar los ejemplos o realizar las actividades en máquinas físicas distintas.
>
> En las redes informáticas, localhost es el nombre reservado que tienen todos los ordenadores. Está asociado a la dirección IPv4 127.0.0.1 y a la dirección IPv6 ::1. Todos los ordenadores, por lo tanto, son localhost para sí mismos.

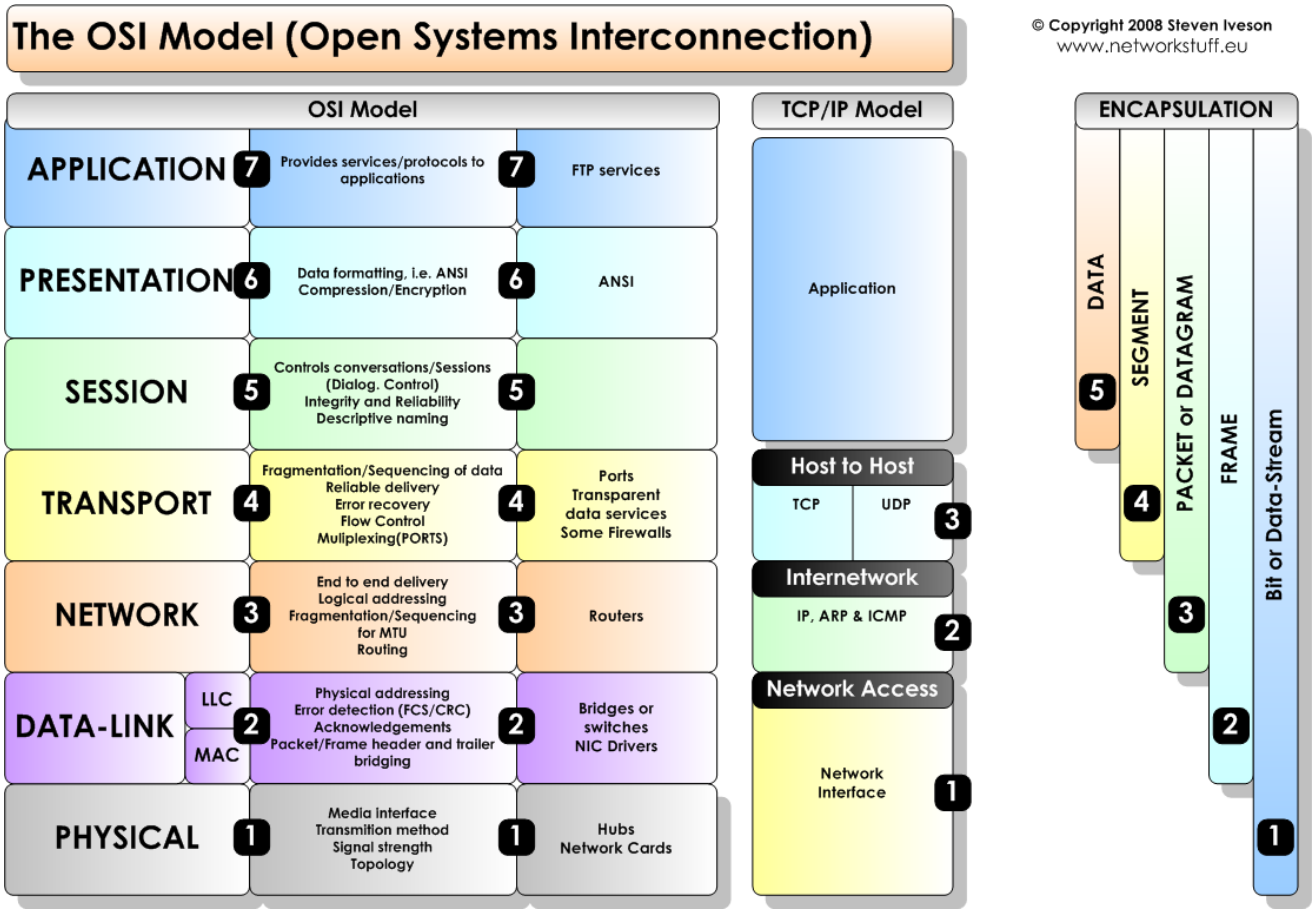
1.1. Protocolos

La comunicación en red entre sistemas informáticos es un proceso complejo. Participan muchos elementos desde diferentes niveles de abstracción. Por ejemplo, a nivel físico debe existir un canal que permita la comunicación, como puede ser el cable de fibra óptica. A más alto nivel se necesitan soluciones para los problemas de identificación de dispositivos o para el envío y recepción de la información.

El modelo de interconexión de sistemas abiertos, conocido como modelo OSI, establece cuáles son los niveles de abstracción y qué responsabilidad tiene cada uno en las diferentes tareas que deben realizarse para conseguir enviar y recibir información a través de una red. Define 7 capas y es la referencia sobre la se construyen la mayoría de los sistemas de comunicación de red.

No es el objetivo de este libro profundizar en el modelo OSI, ya que está fuera de su alcance, pero sí es importante conocer su existencia y la relación que tiene con internet y sus servicios.

El modelo OSI dice qué hay que hacer, pero no dice cómo. Cada capa de este modelo debe implementarse y eso supone dotar de protocolos a cada una de las funciones que realiza. Internet establece un modelo con menos capas que el modelo OSI, pero igual de completo. Este modelo se conoce como TCP/IP.



Desde el punto de vista de la programación es necesario conocer algunos de los protocolos que se utilizan en las capas de más alto nivel, ya que dichos protocolos van a ser utilizados para desarrollar los programas que utiliza la red para realizar comunicaciones. El resto de los protocolos son de más bajo nivel y están gestionados por el sistema operativo, por lo que su conocimiento normalmente no es necesario.

Los protocolos presentados y utilizados en esta unidad son:

- **TCP**: El Protocolo de Control de Transmisión (Transmission Control Protocol) es un protocolo de la capa de transporte, por lo que su trabajo consiste con permitir la transmisión de paquetes de datos. TCP garantiza que los paquetes se entregan al destinatario de forma ordenada, completa y correcta. Se utiliza para la transferencia de información cuando es necesario garantizar la integridad de esta, como, por ejemplo, en la web o en la transferencia de archivos con FTP. Este protocolo está basado en conexiones, por lo que cuando se establece una comunicación entre dos nodos de la red se crea un canal estable a través del cual se envía la información.
- **UDP**: El Protocolo de Datagramas de Usuario (User Datagram Protocol) es, al igual que TCP, un protocolo de la capa de transporte que posibilita la transmisión de paquetes de datos. Al contrario que TCP, UDP no está basado en conexiones. Esto significa que cada paquete se envía sin que exista un canal de comunicación abierto con el receptor, por lo que cada paquete podrá alcanzar su destino por un camino distinto.

  Por esta razón, entre otras, los paquetes de datos UDP (conocidos como datagramas) pueden no llegar a su destino en el mismo orden en el que se enviaron o incluso pueden «perderse» y no alcanzarlo. Este comportamiento sin garantías puede resultar llamativo, pero tiene sus ventajas. Al no tener que garantizar la entrega ni el orden, la comunicación es mucho más rápida que cuando se usa el protocolo TCP, a cambio de aceptar la pérdida de algunos paquetes de información. Esta

característica hace que UDP sea el protocolo más adecuado para algunas aplicaciones en las que la rapidez prevalece sobre la fiabilidad, como las transmisiones de voz o de vídeo en tiempo real.

- **IP**: El Protocolo de Internet (Internet Protocol) pertenece a la capa de internet en el modelo TCP/IP y a la de red en el modelo OSI. Este protocolo es de más bajo nivel que los protocolos TCP y UDP, que lo utilizan como mecanismo básico de envío y recepción de paquetes o datagramas. Es un protocolo muy básico, no orientado a conexión y que no garantiza la recepción de los paquetes transmitidos. Permite identificar a los dispositivos conectados a una red mediante unos identificadores conocidos como direcciones IP.

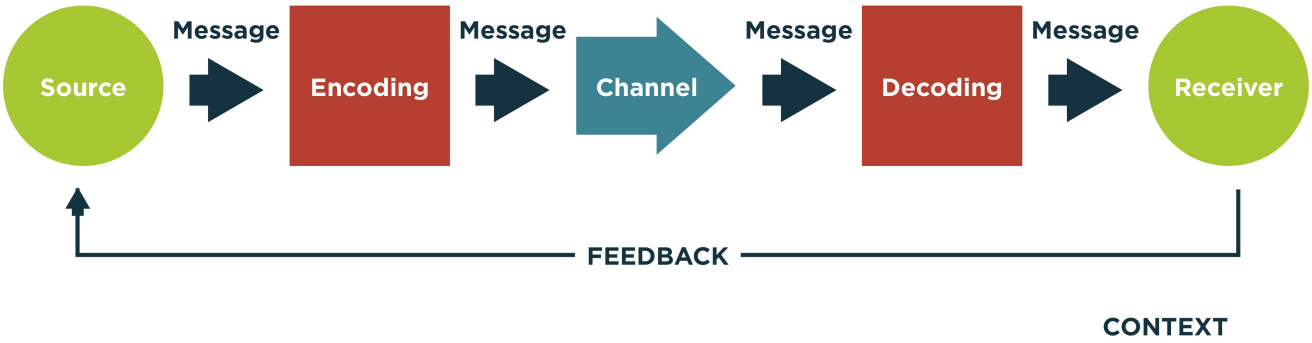
  Actualmente existen dos versiones de este protocolo: IPv4 e IPv6. IPv4 es el protocolo tradicional de internet. Debido al tamaño de las direcciones, compuesto por números de 32 bits, admite un número ligeramente superior a cuatro mil millones de combinaciones.
  Este número de posibles direcciones se consideró inicialmente suficiente para satisfacer cualquier demanda de dispositivos conectados, una suposición que con el tiempo se demostró errónea. La versión IPv6 proporciona direcciones de 128 bits, lo que permite disponer de alrededor de dieciséis trillones de direcciones posibles.

1.2. Participantes

Hay una serie de términos que se deben conocer cuando se habla de redes de ordenadores y de la programación de comunicaciones en red. Algunos son de uso cotidiano y resultan familiares. Otros son más técnicos y pueden resultar novedosos. Desde la perspectiva del programador de aplicaciones en red, al menos los siguientes términos deben ser conocidos.

- **Servidor**: Se denomina servidor tanto al hardware como al software que proporcionan un servicio en una red. Por ejemplo, las páginas web son proporcionadas por un servidor web o el servicio de correo electrónico por un servidor de correo electrónico. Como se puede suponer, un mismo servidor hardware puede albergar varios servidores software.
- **Cliente**: Es el consumidor de un servicio. Al igual que ocurre con el servidor, puede referirse al hardware (cliente de la red) o al software (cliente del servicio). Por ejemplo, el navegador web es un cliente del servicio de web o el programa de correo electrónico es el cliente de dicho servicio.
- **host**: Se define como host a un equipo que funciona como extremo de una comunicación, aunque normalmente hace referencia al proveedor de un servicio (servidor).
- **Canal**: En comunicación, el canal es el medio por el que se transmite la información. Un canal puede ser el aire, la fibra Óptica o un cable de cobre.
- **Protocolo**: Descripción formal de cualquiera de las actividades que tienen lugar en la comunicación.
- **Mensaje**: El mensaje es la información que se transmite entre los interlocutores de una comunicación.

The Communication Process



- **Paquete**: Se conoce como paquete de red o de datos a cada uno de los fragmentos de un mensaje que se envían por la red.

  La fragmentación de los mensajes tiene su origen en la existencia de errores en la comunicación. Por ejemplo, si existen interferencias puntuales en un canal, al afectar a elementos pequeños es más eficiente repetir el envío de algún paquete afectado que del mensaje
  completo, ya que las probabilidades de que se produzca un error serán más bajas.

- **Datagrama**: Un datagrama es el paquete de datos que constituye la unidad mínima de información transmitible.

- **Dirección**: La dirección es el identificador del dispositivo o recurso de un elemento de una red. Por ejemplo, la dirección IP identifica a un dispositivo, mientras que la dirección web identifica un recurso (normalmente una página) en la web.

- **Puerto**: En redes de ordenadores, un puerto es un extremo (endpoint) de una comunicación. Se identifica con un número de 16 bits (el número de puerto) que se asocia a una dirección IP y que permite que un dispositivo disponga de varios canales de comunicación, tantos como puertos.
  Algunos puertos están reservados para servicios específicos. Los números de puerto inferiores a 1024 se identifican con servicios históricos y se denominan puertos conocidos. Los números de puerto más altos, entre 49152 y 65535 están disponibles para su uso general y se conocen como puertos efímeros.

  Algunos de los puertos conocidos y los servicios asociados más populares son:

  - `21` FTP.
  - `22` SSH.
  - `23` Telnet.
  - `25`: SMTP.
  - `80`: HTTP.
  - `110`: POP3.
  - `143` IMAP.
  - `443` HTTPS.

  Todas las aplicaciones que permiten trabajar en red necesitan utilizar uno o más puertos. Habitualmente tienen asociado un puerto por defecto. Por ejemplo, MySQL utiliza el puerto 3306. No obstante, lo habitual es que el puerto al que se asocia un servicio se pueda configurar.

2. Clases y librerías usadas paralacomunicación en red

Java, como prácticamente cualquier lenguaje de programación moderno, dispone de librerías y clases que facilitan la comunicación a través de una red utilizando diversos niveles de abstracción.

El paquete en el que se encuentran las clases fundamentales de comunicación es `java.net`.

Los API proporcionados por este paquete se pueden dividir en dos grupos:

- Un API de bajo nivel, que proporciona clases para gestionar:
    - Direcciones e identificadores de red (fundamentalmente direcciones IP).
    - Sockets, como mecanismo simple de comunicación bidireccional.
    - Interfaces, que describen las interfaces de red.
- Un API de alto nivel, que proporciona clases para gestionar:
    - Identificadores de Recursos Universales (URI).
    - Localizadores de Recursos Universales (URL).
    - Conexiones a recursos identificados mediante URL.

En esta unidad se presentan y explican las principales clases del API de bajo nivel de este paquete.

2.1. `InetAddress`

En Java, las direcciones de los dispositivos de red vienen representadas por la clase `InetAddress` y sus subclases más específicas `Inet4Address` e `Inet6Address`, que como se puede intuir representan a direcciones IPv4 e IPv6, respectivamente. Los objetos de la clase `InetAddress` disponen de un API que proporciona, entre otros, los métodos siguientes:

Método	Descripción
`getAddress`	Proporciona la dirección `IP` representada por el objeto `InetAddress` como un array de bytes.
`getByAddress`	Método estático que proporciona un objeto `InetAddress` a partir de una `IP` representada como un array de bytes.
`getByName`	estático que proporciona la dirección `IP` de un host a partir de su nombre.
`getHostAddress`	Proporciona la dirección `IP` en modo texto del objeto `InetAddress`.
`getHostName`	Proporciona el nombre del host para la dirección representada por el objeto `InetAddress`.
`getLocalHost`	Método estático que proporciona la dirección `IP` del host local.

El siguiente código proporciona un objeto de tipo `InetAddress` a partir del nombre de un host:

```
Inetaddress direccion = InetAddress.getByName ("www.martinezpenya.es");
```

También se puede obtener dicho objeto a partir de una dirección `IP` representada como un `String`:

```
InetAddress direccion = InetAddress.getByName ("51.68.98.141");
```

2.2. `SocketAddress`

Clase abstracta que representa la dirección de un socket.

2.3. `InetSocketAddress`

Subclase de SocketAddress. Representa la dirección del socket como un par compuesto por dirección IP y puerto.

2.4. `ServerSocket`

En el establecimiento de las conexiones con sockets, ServerSocket implementa un socket servidor. Este tipo de socket, al construirse, queda a la espera de recibir peticiones de los procesos cliente para proporcionar un socket de comunicación.

Los métodos más relevantes se presentan en la tabla siguiente:

Método	Descripción
`accept`	Recibe las peticiones de establecimiento de conexión, proporcionando un `socket`.
`bind`	Asocia el `ServerSocket` a una dirección.
`close`	Cierra el `socket`.
`isBound`	Devuelve el estado de asociación del `socket`.
`isConnected`	Determina si el `socket` está conectado.

2.5. Socket

Representa un socket cliente, lo que se conoce simplemente como socket. Es un extremo en la comunicación entre dos máquinas.

Dispone de varios constructores en los que se pueden especificar, entre otras propiedades, la dirección IP y el puerto.

Algunos de sus métodos más utilizados son:

Método	Descripción
`bind`	Asocia el `socket` a una dirección local.
`close`	Cierra el `socket`.
`connect`	Conecta este `socket` con el servidor.
`getinputStream`	Proporciona un `stream` de lectura.
`getOutputStream`	Proporciona un `stream` de escritura.
`isBound`	Devuelve el estado de asociación del `socket`.
`isClosed`	Determina si el `socket` está cerrado.
`isConnected`	Determina si el `socket` está conectado.

2.6. DatagramPacket

Representa un datagrama. Dispone de diversos constructores que admiten, entre otros parámetros, un array de bytes para determinar el contenido del paquete.

2.7. DatagramSocket

Esta clase representa un `socket` para el envío y recepción de datagramas. Los métodos más relevantes de esta clase se muestran a continuación:

Método	Descripción
`bind`	Asocia el `socket` a una dirección local.
`close`	Cierra el `socket`.
`connect`	Conecta este `socket` a una dirección remota.
`disconnect`	Desconecta el `socket`.
`isBound`	Devuelve el estado de asociación del `socket`.
`isClosed`	Determina si el `socket` está cerrado.
`isConnected`	Determina si el `socket` está conectado.
`receive`	Recibe un `datagrama` del `socket`.
`send`	Envía un `datagrama` a través del `socket`.

3. Desarrollo de sistemas de comunicación basados en sockets

Los `sockets` son un mecanismo de comunicación de bajo nivel. Permiten intercambiar información entre dos elementos de una red por medio de los protocolos TCP y UDP.

Para poder crear aplicaciones en Java basadas en `sockets` se utilizan clases del paquete `java.net` para los aspectos relacionados con la comunicación y de `java.io` para los relacionados con la lectura y escritura de bytes.

Un sistema basado en `sockets` está compuesto como mínimo por dos aplicaciones: **cliente** y **servidor**. Cada una de estas dos aplicaciones cumple una función concreta: el **servidor** inicia el proceso de escucha y espera de peticiones de conexión, mientras que el **cliente** es el elemento que establece la conexión con la dirección y el puerto en donde se ha establecido el servidor.

El desarrollo de sistemas basados en `sockets` es distinto si los `sockets` son TCP o UDP.

3.1. Sistemas basados en `sockets` TCP

Los pasos para crear un servidor basado en sockets TCP son los siguiente:

- Crear un `socket` de tipo servidor (`serverSocket`) asociado a una dirección y a un puerto concretos.
- Indicar al `socket` de tipo servidor que quede a la espera de peticiones de establecimiento de conexión por parte del cliente.
- Aceptar el establecimiento de la conexión y obtención del `socket`.
- Abrir los flujos de lectura y escritura de datos.
- Intercambiar datos con el cliente.
- Cerrar los flujos de lectura y escritura de datos.
- Cerrar de la conexión.

Por su parte, los pasos para la creación de un cliente basado en sockets TCP son los siguientes:

- Crear un socket de tipo cliente (socket) indicando la dirección IP y el puerto del servidor.
- Abrir los flujos de lectura y escritura de datos.
- Intercambiar datos con el servidor.
- Cerrar los flujos de lectura y escritura de datos.
- Cerrar la conexión.

A continuación, revisa el Ejemplo01.

3.2. Sistemas basados en `sockets` UDP

Los sockets UDP, además de tener características distintas a los sockets TCP, utilizan clases y métodos diferentes, aunque a rasgos generales el proceso de comunicación es el mismo.

Como diferencia fundamental está el hecho de que los paquetes enviados mediante UDP son independientes entre sí, por lo que las estructuras de datos que los soportan son arrays de bytes. Estos arrays deben ser dimensionados correctamente para evitar la pérdida de información o el desaprovechamiento del canal de comunicación.

El proceso desde el lado servidor para el envío y recepción de datagramas es el siguiente:

- Creación de un `socket` UDP mediante la clase `DatagramSocket` asociado a un puerto.
- Creación del array de bytes que actuará de buffer donde almacenar el mensaje recibido.
- Creación del datagrama mediante la clase `DatagramPacket`, utilizando el buffer creado en el paso anterior.
- Recepción del datagrama mediante el método receive del `socket`. En este punto el servidor queda a la espera de envíos provenientes del cliente.
- La generación y envío de la respuesta se realizará a partir de la información contenida en el datagrama recibido (host y puerto) y del uso del método send del `socket`.
- Una vez finalizada la comunicación se cierra el `socket`.

En el lado cliente, se realizan los siguientes pasos:

- Obtención de la dirección del servidor mediante el uso del método `getByName` de la clase `InetAddress`.
- Creación del `socket` UDP mediante la clase `DatagramSocket`.
- Generación del datagrama mediante la clase `DatagramPacket`, utilizando el array de bytes con el contenido que se desea enviar.
- Envío del datagrama a través del método send del `socket`.
- Para la recepción de la respuesta se debe crear un array de bytes del tamaño suficiente para almacenar la respuesta y utilizando el método receive del socket.
- Una vez finalizada la comunicación se cierra el socket.

Revisa el código del Ejemplo02

> Los sockets UDP no están orientados a conexión, por lo que el servidor no se detiene porque el cliente cierre la conexión, ya que esta no existe. La parada del servidor debe ser realizada como consecuencia de una condición cumplida.

4. Comunicación multihilo con sockets

El servidor de un sistema en red basado en sockets habitualmente deberá dar servicio a varios clientes. Si dicho servidor atendiera a las peticiones de manera secuencial provocaría un cuello de botella y la ralentización de todo el sistema.

Para poder atender a varias peticiones simultáneamente es necesario utilizar threads para realizar los procesos solicitados sin bloquear el servidor.
El mecanismo es el siguiente: cuando el servidor recibe una petición de conexión, crea un socket y se lo asigna a un hilo que se encarga de atender la petición de comunicación, quedando el socket servidor a la espera de la siguiente petición. A grandes rasgos, los pasos que sigue la ejecución de este tipo de servidores de sockets son los siguientes:

- El servidor queda a la espera de peticiones de los clientes mediante el método accept de la clase ServerSocket. Cada una de estas peticiones se traduce en un socket TCP entre el cliente y el servidor.
- Cuando el servidor recibe una petición, crea un thread (en el ejemplo que se muestra a continuación mediante la clase GestorProcesos) y le proporciona el socket.
- El thread se encarga de realizar el proceso, liberando al servidor para atender a la siguiente petición.

En el Ejemplo03 ejemplo se muestra el código asociado a un servidor multihilo que realiza un proceso que genera un número aleatorio, espera tantos segundos como indique dicho número y lo devuelve al cliente.

5. Ejemplos

5.1. Ejemplo01

A continuación se muestra el código correspondiente a un servidor y un cliente basados en sockets TCP. En este ejemplo hay intercambio muy básico de información: el cliente envía un entero con el valor 100 al servidor y este le responde a su vez con otro entero con el valor 200. Al utilizar el método `read()` solo se podrán leer números enteros entre 0 y 255 (un byte).

Las dos clases deben contemplarse como dos sistemas software distintos, debiéndose ejecutar en primer lugar el servidor, que quedará a la espera de las peticiones, y en segundo lugar el cliente.

`SocketTCPServer.java`

```
1  import java.io.IOException;
2  import java.io.InputStream;
3  import java.io.OutputStream;
4  import java.net.ServerSocket;
5  import java.net.Socket;
6
7  public class SocketTCPServer {
8
9      private ServerSocket serverSocket;
10     private Socket socket;
11     private InputStream is;
12     private OutputStream os;
13
14     public SocketTCPServer(int puerto) throws IOException {
15         serverSocket = new ServerSocket(puerto);
16     }
17
18     public void start() throws IOException {
19         System.out.println("(Server) Waiting connections...");
20
21         socket = serverSocket.accept();
22         is = socket.getInputStream();
23         os = socket.getOutputStream();
24         System.out.println("(Server) Connection stablished.");
25     }
26
27     public void stop() throws IOException {
28
29         System.out.println("(Server) Closing connections...");
30         is.close();
31         os.close();
32         socket.close();
33         serverSocket.close();
34         System.out.println("(Server) Connections closed.");
35     }
```

```
36
37      public static void main(String[] args) {
38          try {
39              SocketTCPServer server = new SocketTCPServer(49171);
40              server.start();
41              System.out.println("Message from the client:" + server.is.read());
42              server.os.write(200);
43              server.stop();
44          } catch (IOException ioe) {
45              ioe.printStackTrace();
46          }
47      }
48  }
```

En el constructor de la clase se crea un socket de tipo `ServerSocket` asociado a la dirección `IP` del host (por defecto se asigna la dirección `IP` de la máquina, siendo en este ejemplo 192.168.10.10) y a un puerto (en este ejemplo, el 49171).
En el método `start()` se indica al `socket` que quede a la espera de peticiones mediante el método `accept()` del objeto `ServerSocket` creado en el constructor. En este punto la ejecución quedará detenida hasta que llegue una petición de conexión por parte de un cliente. Una vez recibida la petición de conexión se crea el objeto de la clase `Socket` que representa la conexión entre servidor y cliente.

Una vez establecida se crean los flujos de entrada y de salida de datos. En este ejemplo, los flujos de comunicación se establecen a través de objetos de las clases abstractas `InputStream` y `OutputStream`, que básicamente permiten leer y escribir bytes.

El servidor, lee del `InputStream` el número entero enviado por el cliente mediante el método `read()` y envía el número entero 200 mediante el método `write` del `OutputStream`.

`SocketTCPClient.java`

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
import java.net.UnknownHostException;

public class SocketTCPClient {      private String serverIP;     private int serverPort;
private Socket socket;     private InputStream is;     private OutputStream os;      public
SocketTCPClient(String serverIP, int serverPort) {         this.serverIP = serverIP;
this.serverPort = serverPort;     }      public void start() throws IOException {
System.out.println("(Client) Starting connection...");         socket = new Socket(serverIP,
serverPort);         os = socket.getOutputStream();         is = socket.getInputStream();
    System.out.println("(Client) Connection stablished.");     }      public void stop()
throws IOException {          System.out.println("(Client) Closing connections...");
is.close();         os.close();         socket.close();         System.out.println("(Client)
Connections closed.");     }      public static void main(String[] args) {
SocketTCPClient client = new SocketTCPClient("127.0.0.1", 49171);         try {
client.start();             client.os.write(100);             System.out.println("Message from
the server:" + client.is.read());             client.stop();         } catch
(UnknownHostException e) {             e.printStackTrace();         } catch (IOException e) {
            e.printStackTrace();         }     }
}
```

La ejecución del servidor seria:

```
(Server) Waiting connections...
(Server) Connection stablished.
Message from the client:100
(Server) Closing connections...
(Server) Connections closed.
```

Y la del cliente:

```
(Client) Starting connection...
(Client) Connection stablished.
Message from the server:200
(Client) Closing connections...
(Client) Connections closed.
```

Por su parte, el cliente crea un socket indicando nombre de host (en este ejemplo localhost) o la dirección IP. Tanto si se usan el nombre del host o la dirección IP para identificar al servidor, deberá indicarse el dato correspondiente al ordenador que albergue dicho servidor. Si se ejecutasen el cliente y el servidor en la misma máquina se podrá utilizar localhost como nombre del host, como en este caso.

Además de la dirección IP, se indica el puerto en el que se encuentra a la escucha el servidor, siendo en este caso el 49471.

Una vez establecida la conexión se obtienen los flujos de comunicación representados por objetos de las clases `InputStream` y `OutputStream` para proceder a escribir el mensaje que se quiere enviar al servidor y leer el que este envía como respuesta.

Como se puede observar el nivel de abstracción es bajo, al disponer nada más que de los métodos existentes en las clases `InputStream` y `OutputStream`, pero estos objetos se pueden envolver con clases de más alto nivel para permitir el intercambio de otro tipo de datos.

El método `read` sin parámetros de la clase `InputStream` solo lee un byte. Si se quieren leer grupos de bytes habrá utilizar alguna de las variantes de dicho método. (Consulta el tema de lectura y escritura de información)

> Podemos establecer la conexión entre diferentes Sistemas Operativos, podemos ejecutar el servidor en Windows y el cliente en Linux o viceversa. Incluso podriamos conectar desde un cliente Java con un Server programado en Python (u otro lenguaje) o viceversa.

5.2. Ejemplo02

A continuación se muestran un servidor y un cliente que se intercambian mensajes a través de `sockets` UDP.

`SocketUDPServer.java`

```
1  import java.io.IOException;
2  import java.net.DatagramPacket;
3  import java.net.DatagramSocket;
4  import java.net.SocketException;
5
6  public class SocketUDPServer {
7
8      public static void main(String[] args) {
9          DatagramSocket socket;
10         try {
11             System.out.println("(Server): Creating socket...");
12             socket = new DatagramSocket(49171);
13             System.out.println("(Server): Receiving datagram...");
14             byte[] readBuffer = new byte[64];
15             DatagramPacket inputDatagram = new DatagramPacket(readBuffer,
16                     readBuffer.length);
17             socket.receive(inputDatagram);
18             System.out.println("(Server): Message received: " + new String(
19                     readBuffer));
20             System.out.println("(Server): Sending datagram...");
21             byte[] sentMessage = "Message sent from the server".getBytes();
22             DatagramPacket outputDatagram = new DatagramPacket(sentMessage,
23                     sentMessage.length, inputDatagram.getAddress(),
24                     inputDatagram.getPort());
25             socket.send(outputDatagram);
26             System.out.println("(Server): Closing socket...");
27             socket.close();
28             System.out.println("(Server): Socket closed.");
```

```
        } catch (SocketException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

`SocketUDPClient.java`

```
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketException;
import java.net.UnknownHostException;

public class SocketUDPClient {      public static void main(String[] args) {         String strMessage = "Message sent from the client";         DatagramSocket socketUDP;         try {          System.out.println("(Client): Stablishing connection parameters");  InetAddress serverHost = InetAddress.getByName("localhost");             int serverPort = 49171;             System.out.println("(Client): Creating socket...");             socketUDP = new DatagramSocket();             System.out.println("(Client): Sending datagram...");     byte[] message = strMessage.getBytes();             DatagramPacket peticion = new DatagramPacket(message,                     message.length, serverHost, serverPort);   socketUDP.send(peticion);             System.out.println("(Client): Receiving datagram...");            byte[] buffer = new byte[64];             DatagramPacket respuesta = new DatagramPacket(buffer, buffer.length,                     serverHost, serverPort);  socketUDP.receive(respuesta);             System.out.println("(Client): Message received: " + new String(                     buffer));             System.out.println("(Client): Closing socket...");             socketUDP.close();             System.out.println("(Client): Socket closed.");         } catch (SocketException e) {             e.printStackTrace();         } catch (UnknownHostException e) {             e.printStackTrace();         } catch (IOException e) {             e.printStackTrace();         }     }
}
```

La ejecución del servidor seria:

```
(Server): Creating socket...
(Server): Receiving datagram...
(Server): Message received: Message sent from the client
(Server): Sending datagram...
(Server): Closing socket...
(Server): Socket closed.
```

Y la del cliente:

```
(Client): Stablishing connection parameters
(Client): Creating socket...
(Client): Sending datagram...
(Client): Receiving datagram...
(Client): Message received: Message sent from the server
(Client): Closing socket...
(Client): Socket closed.
```

5.3. Ejemplo03

`ProcessManager.java`

```java
import java.util.concurrent.TimeUnit;
import java.io.IOException;
import java.io.OutputStream;
import java.net.Socket;
import java.util.Random;

public class ProcessManager extends Thread {

    private Socket socket;
    private OutputStream os;

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

    @Override
    public void run() {
        try {
            doProcess();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void doProcess() throws IOException {
        os = this.socket.getOutputStream();
        Random randomNumberGenerator = new Random();
        int waitTime = randomNumberGenerator.nextInt(60);
        try {
            TimeUnit.SECONDS.sleep(waitTime);
            os.write(waitTime);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            os.close();
        }
    }
}
```

`SocketTCPServer.java`

```java
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class SocketTCPServer {

    private ServerSocket serverSocket;

    public SocketTCPServer(int puerto) throws IOException {

        serverSocket = new ServerSocket(puerto);
        while (true) {
            Socket socket = serverSocket.accept();
            System.out.println("(Server) Connection stablished");
            new ProcessManager(socket).start();
        }
    }

    public void stop() throws IOException {
        serverSocket.close();
    }

    public static void main(String[] args) {
        try {
            SocketTCPServer server = new SocketTCPServer(49171);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

`SocketTCPClient.java`

```java
import java.io.IOException;
import java.io.InputStream;
import java.net.Socket;
import java.net.UnknownHostException;

public class SocketTCPClient {

    private String serverIP;
    private int serverPort;
    private Socket socket;
    private InputStream is;

    public SocketTCPClient(String serverIP, int serverPort) {
        this.serverIP = serverIP;
        this.serverPort = serverPort;
```

```
    }

    public void start() throws UnknownHostException, IOException {
        System.out.println("(Client) Stablishing connection...");
        socket = new Socket(serverIP, serverPort);
        is = socket.getInputStream();
        System.out.println("(Client) Connection stablished.");
    }

    public void stop() throws IOException {
        System.out.println("(Client) Closing connections...");
        is.close();
        socket.close();
        System.out.println("(Client) Connections closed.");
    }

    public static void main(String[] args) {
        SocketTCPClient client = new SocketTCPClient("localhost", 49171);
        try {
            client.start();
            System.out.println("Message from the server:" + client.is.read());
            client.stop();
        } catch (UnknownHostException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Para ejecutar este ejemplo en un único ordenador, se pueden abrir varios terminales, ejecutar el servidor en uno de ellos y los clientes en el resto. De esta manera se comprueba cómo se establecen las conexiones y se van atendiendo las peticiones de forma simultánea. Al disponer la clase que realiza el proceso de un temporizador, se puede comprobar cómo el funcionamiento es correcto.

A continuación vemos la ejecución del servidor:

```
(Server) Connection stablished
(Server) Connection stablished
(Server) Connection stablished
(Server) Connection stablished
```

Y las ejecuciones de los cuatro clientes en terminales:

```
(Client) Stablishing connection...
(Client) Connection stablished.
Message from the server:27
(Client) Closing connections...
(Client) Connections closed.
```

```
(Client) Stablishing connection...
(Client) Connection stablished.
Message from the server:16
(Client) Closing connections...
(Client) Connections closed.
```

```
(Client) Stablishing connection...
(Client) Connection stablished.
Message from the server:2
(Client) Closing connections...
(Client) Connections closed.
```

```
(Client) Stablishing connection...
(Client) Connection stablished.
Message from the server:3
(Client) Closing connections...
(Client) Connections closed.
```

Todas las conexiones se establecen y los procesos se comienzan a tratar sin la necesidad de haber terminado ninguno de los procesos de los demás clientes previamente.

Por supuesto, si el número de peticiones concurrentes fuese muy elevado habría que establecer una cola de trabajos para almacenarlas mientras se liberan recursos, con el fin de no sobrecargar en exceso al sistema.

6. Fuentes de información

- Wikipedia
- Programación de servicios y procesos - FERNANDO PANIAGUA MARTÍN [Paraninfo]
- Programación de Servicios y Procesos - ALBERTO SÁNCHEZ CAMPOS [Ra-ma]
- Programación de Servicios y Procesos - Mª JESÚS RAMOS MARTÍN - [Garceta] (1ª y 2ª Edición)
- Programación de servicios y procesos - CARLOS ALBERTO CORTIJO BON [Sintesis]
- Programació de serveis i processos - JOAR ARNEDO MORENO, JOSEP CAÑELLAS BORNAS i JOSÉ ANTONIO LEO MEGÍAS [IOC]
- GitHub repositories:
    - https://github.com/ajcpro/psp
    - https://oscarmaestre.github.io/servicios/index.html
    - https://github.com/juanro49/DAM/tree/master/DAM2/PSP
    - https://github.com/pablohs1986/dam_psp2021
    - https://github.com/Perju/DAM
    - https://github.com/eldiegoch/DAM
    - https://github.com/eldiegoch/2dam-psp-public
    - https://github.com/franlu/DAM-PSP
    - https://github.com/ProgProcesosYServicios
    - https://github.com/joseluisgs
    - https://github.com/oscarnovillo/dam2_2122
    - https://github.com/PacoPortillo/DAM_PSP_Tarea02_La-Cena-de-los-Filosofos