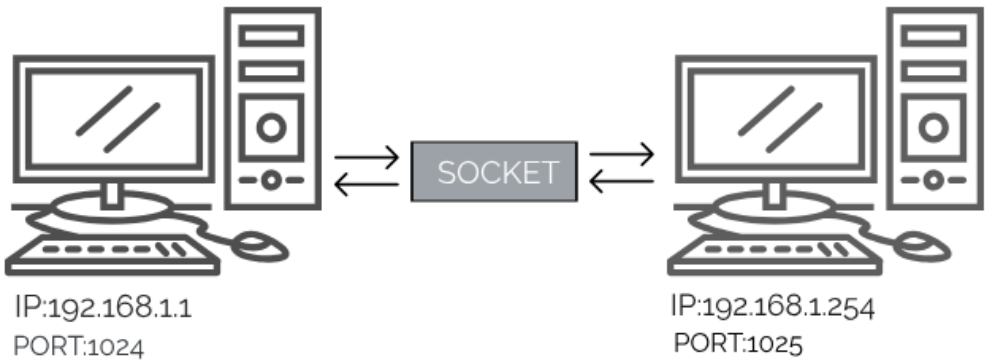
UD03: Network communications coding



1. Fundamentals of network communications programming

The third meaning of the dictionary of the language of the Royal Spanish Academy defines communication as "transmission of signals by means of a code common to the sender and the receiver". Although partial, this is a good definition for communication between computing devices connected through a network. Because computer systems are deterministic and incapable of resolving the ambiguities of language, for communication to be effective it is necessary to have a series of elements and techniques.

For communication between computer systems to be possible, the following components or elements must be available:

- The partners involved in the communication.
- A message or messages to convey.
- A communication channel.
- A common language for all interlocutors in which the accepted symbols, their combinations and their meaning are defined.
- A mechanism to be able to identify the sender and the receiver of the messages among all the possible participants of the communication.
- A system to guarantee the integrity of messages sent and received.
- A system to ensure that the order in which messages are received is correct.
- A method to confirm receipt of messages or to notify the sender that any of the messages has not fully arrived, either due to loss or damage to any of its parts.

That is, in addition to the partners, the channel and the messages themselves, protocols are needed.

Applied to communication, a protocol is a set of formal rules that determines any aspect related to the activities of said communication, from the way in which the participants are identified to the mechanism by which it is confirmed that a message has arrived. completely and correctly to its addressee.

The set of protocols on which the internet is built is known as the internet protocol family. These protocols cover all kinds of services, from the web to email. One of their most interesting advantages is that they are open protocols. This means that they do not belong to any company or organization, being able to dispose of them freely and free of charge. This characteristic, in addition to its robustness, has caused the family of Internet protocols to be used both for said network and for networks of all types and sizes, including local ones: a local computer network can be built based on Internet protocols without internet access. The design is
so versatile that it serves both to connect two devices and millions of them.

Some of these protocols are the basis on which practically all communications-based applications and services are developed.

This section presents the fundamental protocols for communication between computers. These are: IP, TCP and UDP. Each of them has an objective, its own characteristics and must be used in a specific context. These protocols are the basis of internet communications and support the rest of the higher level protocols such as HTTP or FTP.

In Java, as in practically all languages, there are classes and libraries that allow the use of these protocols in a simple way, accepting the development of applications capable of communicating at a low level using a communication system known as a socket. Through the use of socket, a communication channel can be established between two points (two devices) capable of allowing data to be sent and received in both directions.

> Most of the examples presented in this unit have been developed and tested on a single computer, using localhost as the client and server host. It must be remembered that, although running on a single physical machine, applications developed in Java run on a virtual machine (JVM or Java Virtual Machine), so each Java application that is running simultaneously on a computer does so on a different virtual machine. However, if you have a computer network, it will be interesting to be able to try the examples or carry out the activities on different physical machines.
>
> In computer networks, localhost is the reserved name that all computers have. It is associated with the IPv4 address 127.0.0.1 and the IPv6 address ::1. All computers are therefore localhost to themselves.

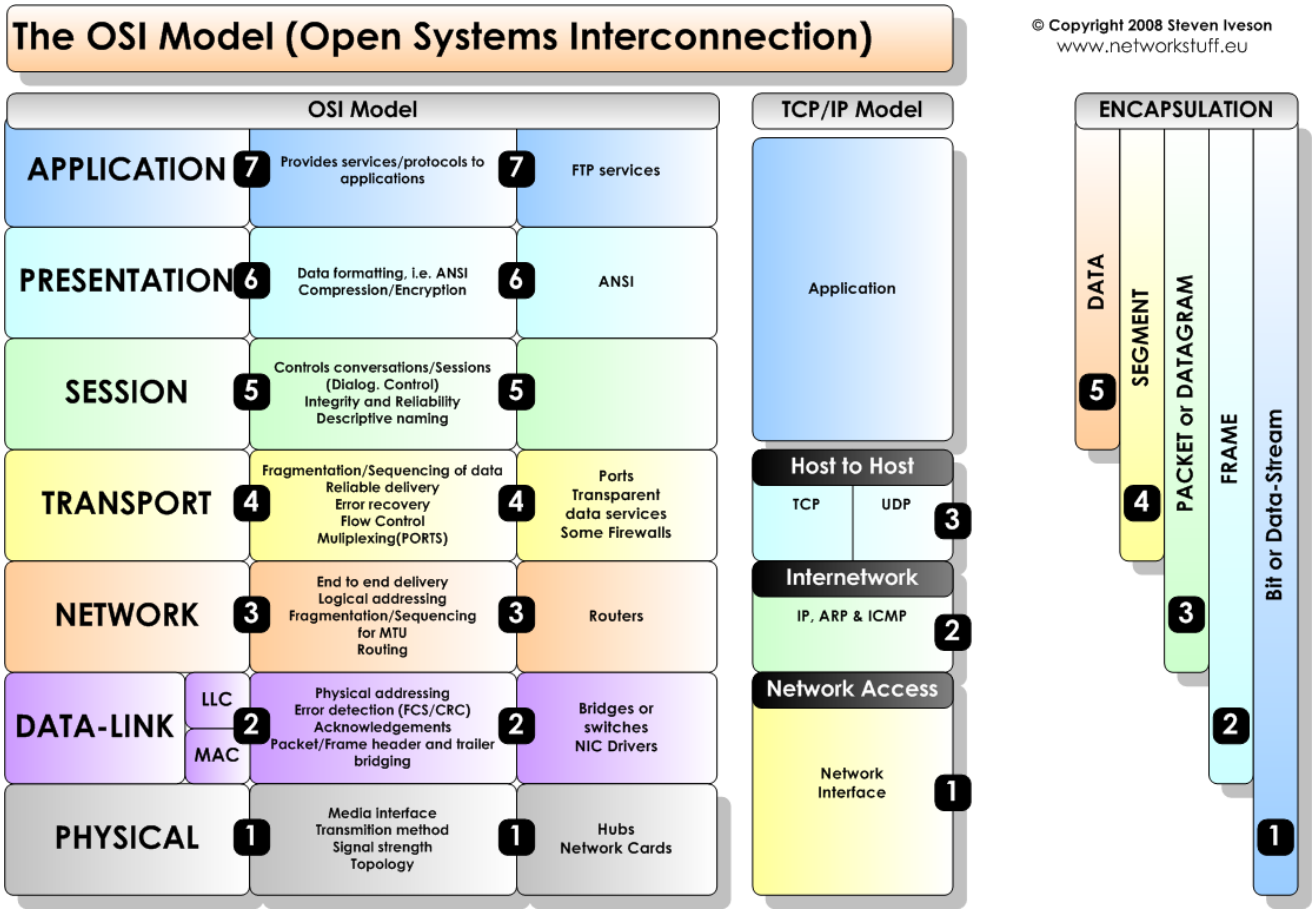
1.1. Protocols

Network communication between computer systems is a complex process. Many elements participate from different levels of abstraction. For example, at the physical level there must be a channel that allows communication, such as fiber optic cable. At a higher level, solutions are needed for device identification problems or for sending and receiving information.

The open systems interconnection model, known as the OSI model, establishes what the levels of abstraction are and what responsibility each one has in the different tasks that must be carried out in order to send and receive information through a network. It defines 7 layers and is the reference on which most network communication systems are built.

It is not the objective of this book to delve into the OSI model, since it is beyond its scope, but it is important to know its existence and the relationship it has with the Internet and its services.

The OSI model says what to do, but it doesn't say how. Each layer of this model must be implemented and that means providing protocols for each of the functions it performs. The Internet establishes a model with fewer layers than the OSI model, but just as complete. This model is known as TCP/IP.



From the point of view of programming, it is necessary to know some of the protocols that are used in the highest level layers, since these protocols are going to be used to develop the programs that the network uses to carry out communications. The rest of the protocols are of a lower level and are managed by the operating system, so their knowledge is normally not necessary.

The protocols presented and used in this unit are:

- **TCP**: The Transmission Control Protocol is a transport layer protocol, so its job is to allow the transmission of data packets. TCP ensures that packets are delivered to the recipient in an orderly, complete, and correct manner. It is used for the transfer of information when it is necessary to guarantee the integrity of the information, such as on the web or in transferring files with FTP. This protocol is based on connections, so when a communication is established between two network nodes, a stable channel is created through which the information is sent.
- **UDP**: The User Datagram Protocol is, like TCP, a transport layer protocol that enables the transmission of data packets. Unlike TCP, UDP is not connection-based. This means that each packet is sent without an open communication channel with the receiver, so each packet can reach its destination by a different path.

  For this reason, among others, UDP data packets (known as datagrams) may not arrive at their destination in the same order in which they were sent or may even "get lost" and not reach it. This unwarranted behavior may seem conspicuous, but it has its advantages. By not having to guarantee delivery or order, communication is much faster than when the TCP protocol is used, in exchange for accepting the loss of some information packets. This feature makes UDP the most suitable protocol for some applications where speed is more important than reliability, such as real-time video or voice transmissions.

- **IP**: The Internet Protocol (Internet Protocol) belongs to the internet layer in the TCP/IP model and to the network layer in the OSI model. This protocol is lower level than the TCP and UDP protocols, which use it as the basic mechanism for sending and receiving packets or datagrams. It is a very basic protocol, not connection-oriented and that does not guarantee the reception of the transmitted packets. It allows devices connected to a network to be identified using identifiers known as IP addresses.

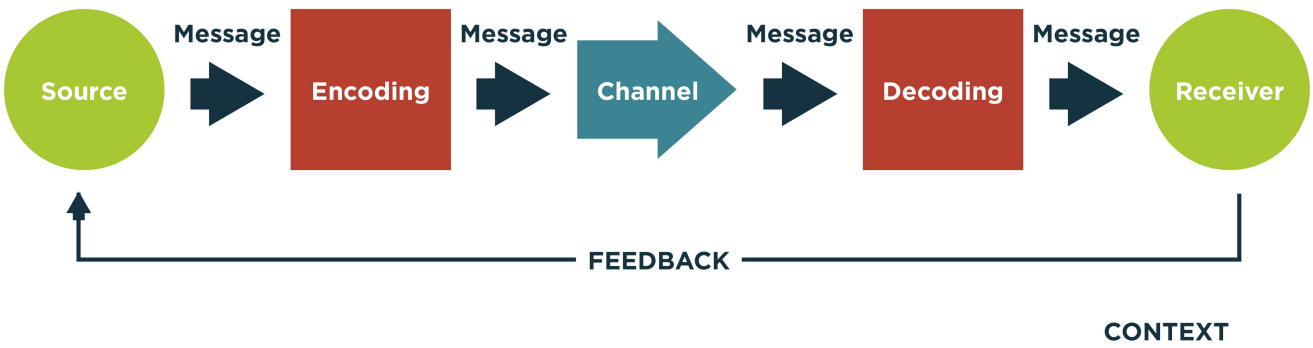
  There are currently two versions of this protocol: IPv4 and IPv6. IPv4 is the traditional internet protocol. Due to the size of the addresses, made up of 32-bit numbers, it admits a number slightly greater than four billion combinations.
  This number of possible addresses was initially considered sufficient to satisfy any demand for connected devices, an assumption that was eventually proven wrong. The IPv6 version provides 128-bit addresses, which allows for about sixteen trillion possible addresses.

1.2. Participants

There are a number of terms to be aware of when discussing computer networks and network communications programming. Some are everyday and look familiar. Others are more technical and may be novel. From the perspective of the network application programmer, at least the following terms should be known.

- **Server**: Server is called both hardware and software that provide a service on a network. For example, web pages are provided by a web server or email service by an email server. As can be assumed, the same hardware server can host several software servers.
- **Client**: Is the consumer of a service. As with the server, it can refer to hardware (network client) or software (service client). For example, the web browser is a client of the web service, or the email program is the client of the web service.
- **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).
- **Channel**: In communication, the channel is the means by which the information is transmitted. A channel can be air, optical fiber or a copper cable.
- **Protocol**: Formal description of any of the activities that take place in the communication.
- **Message**: The message is the information that is transmitted between the interlocutors of a communication.

The Communication Process



- **Packet**: Each of the fragments of a message sent over the network is known as a network or data packet.

The fragmentation of messages has its origin in the existence of errors in communication. For example, if there are occasional interferences in a channel, by affecting small elements, it is more efficient to repeat the sending of an affected packet than of the message.
complete, since the chances of an error occurring will be lower.

- **Datagram**: A datagram is the data packet that constitutes the minimum unit of transmittable information.
- **Address**: The address is the identifier of the device or resource of a network element. For example, the IP address identifies a device, while the web address identifies a resource (usually a page) on the web.
- **Port**: In computer networks, a port is an endpoint of a communication. It is identified by a 16-bit number (the port number) that is associated with an IP address and that allows a device to have several communication channels, as many as there are ports.
  Some ports are reserved for specific services. Port numbers less than 1024 are identified with historical services and are called well-known ports. The higher port numbers between 49152 and 65535 are available for general use and are known as ephemeral ports.

  Some of the most popular known ports and associated services are:

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

  All applications that allow networking need to use one or more ports. They usually have a default port associated with them. For example, MySQL uses port 3306. Typically, however, the port to which a service is associated can be configured.

2. Classes and libraries used for network communication

Java, like virtually any modern programming language, has libraries and classes that facilitate communication over a network using various levels of abstraction.

The package in which the fundamental communication classes are found is `java.net`.

The APIs provided by this package can be divided into two groups:

- A low-level API, which provides classes to manage:
    - Addresses and network identifiers (essentially IP addresses).
    - Sockets, as a simple bidirectional communication mechanism.
    - Interfaces, which describe the network interfaces.
- A high-level API, which provides classes to manage:
    - Universal Resource Identifiers (URI).
    - Universal Resource Locators (URL).
    - Connections to resources identified by URL.

This unit introduces and explains the main low-level API classes of this package.

2.1. `InetAddress`

In Java, network device addresses are represented by the `InetAddress` class and its more specific subclasses `Inet4Address` and `Inet6Address`, which you might guess represent IPv4 and IPv6 addresses, respectively. The objects of the `InetAddress` class have an API that provides, among others, the following methods:

Method	Description
`getAddress`	Provides the `IP` address represented by the `InetAddress` object as a byte array.
`getByAddress`	Static method that provides an `InetAddress` object from an `IP` represented as a byte array.
`getByName`	static that gives the `IP` address of a host from its name.
`getHostAddress`	Provides the text-mode `IP` address of the `InetAddress` object.
`getHostName`	Provides the hostname for the address represented by the `InetAddress` object.
`getLocalHost`	Static method that provides the `IP` address of the local host.

The following code provides an object of type `InetAddress` from a host name:

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

You can also get such an object from an `IP` address represented as a `String`:

```
1 InetAddress address = InetAddress.getByName("51.68.98.141");
```

2.2. `SocketAddress`

Abstract class that represents the address of a socket.

2.3. `InetSocketAddress`

Subclass of `SocketAddress`. Represents the socket address as an IP address and port pair.

2.4. `ServerSocket`

In establishing socket connections, `ServerSocket` implements a server socket. This type of socket, when built, waits to receive requests from client processes to provide a communication socket.

The most relevant methods are presented in the following table:

Method	Description
`accept`	Receive connection establishment requests, providing a `socket`.
`bind`	Binds the `ServerSocket` to an address.
`close`	She closes the `socket`.
`isBound`	Returns the association status of the `socket`.
`isConnected`	Determines if the `socket` is connected.

2.5. Socket

Represents a client socket, known simply as a socket. It is an endpoint in the communication between two machines.

It has several constructors in which you can specify, among other properties, the IP address and the port.

Some of its most used methods are:

Method	Description
`bind`	Binds the `socket` to a local address.
`close`	She closes the `socket`.
`connect`	Connect this `socket` to the server.
`getinputStream`	Provides a read `stream`.
`getOutputStream`	Provides a write stream.
`isBound`	Returns the association status of the `socket`.
`isClosed`	Determines if the `socket` is closed.
`isConnected`	Determines if the `socket` is connected.

2.6. `DatagramPacket`

Represents a datagram. It has several constructors that allow, among other parameters, an array of bytes to determine the content of the packet.

2.7. `DatagramSocket`

This class represents a `socket` for sending and receiving datagrams. The most relevant methods of this class are shown below:

Method	Description
`bind`	Binds the `socket` to a local address.
`close`	She closes the `socket`.
`connect`	Connect this `socket` to a remote address.
`disconnect`	Disconnect the `socket`.
`isBound`	Returns the association status of the `socket`.
`isClosed`	Determines if the `socket` is closed.
`isConnected`	Determines if the `socket` is connected.
`receive`	Receive a `datagram` from the `socket`.
`send`	Send a `datagram` through the `socket`.

3. Development of communication systems based on sockets

`Sockets` are a low-level communication mechanism. They allow the exchange of information between two elements of a network through the TCP and UDP protocols.

In order to create Java applications based on `sockets`, classes from the `java.net` package are used for aspects related to communication and from `java.io` for those related to reading and writing bytes.

A `sockets`-based system is made up of at least two applications: **client** and **server**. Each of these two applications fulfills a specific function: the **server** starts the process of listening and waiting for connection requests, while the **client** is the element that establishes the connection with the address and port where the server has been established.

The development of systems based on `sockets` is different if the `sockets` are TCP or UDP.

3.1. TCP `sockets` based systems

The steps to create a server based on TCP sockets are as follows:

- Create a server-type `socket` (`serverSocket`) associated with a specific address and port.
- Tell the server-type `socket` to wait for connection establishment requests from the client.
- Accept the establishment of the connection and obtain the `socket`.
- Open data read and write streams.
- Exchange data with the client.
- Close the data reading and writing flows.
- Close the connection.

For its part, the steps to create a client based on TCP sockets are as follows:

- Create a client-type socket (socket) indicating the IP address and port of the server.
- Open data read and write streams.
- Exchange data with the server.
- Close the data reading and writing flows.
- Close the connection.

Next, review Example01.

3.2. UDP `sockets` based systems

UDP sockets, in addition to having different characteristics from TCP sockets, use different classes and methods, although in general terms the communication process is the same.

One fundamental difference is the fact that the packets sent via UDP are independent of each other, so the data structures that support them are arrays of bytes. These arrays must be correctly sized to avoid loss of information or waste of the communication channel.

The process from the server side for sending and receiving datagrams is as follows:

- Creation of a UDP `socket` using the `DatagramSocket` class associated with a port.
- Creation of the byte array that will act as a buffer to store the received message.
- Creation of the datagram through the `DatagramPacket` class, using the buffer created in the previous step.
- Reception of the datagram through the receive method of the `socket`. At this point the server is waiting for submissions from the client.
- The generation and sending of the response will be carried out based on the information contained in the received datagram (host and port) and the use of the send method of the `socket`.
- Once the communication is finished, the `socket` is closed.

On the client side, the following steps are performed:

- Obtaining the server address by using the `getByName` method of the `InetAddress` class.
- Creation of the UDP `socket` using the `DatagramSocket` class.
- Generation of the datagram through the `DatagramPacket` class, using the byte array with the content to be sent.
- Sending the datagram through the send method of the `socket`.
- To receive the response, an array of bytes large enough to store the response must be created using the receive method of the socket.
- Once the communication is finished, the socket is closed.

Check the code in Example02

> UDP sockets are connectionless, so the server doesn't stop because the client closes the connection, since the connection doesn't exist. The server stop must be performed as a consequence of a fulfilled condition.

4. Multithread communication with sockets

The server in a socket-based network system will typically have to serve multiple clients. If said server were to attend to the requests sequentially, it would cause a bottleneck and slow down the entire system.

In order to attend to several requests simultaneously, it is necessary to use threads to carry out the requested processes without blocking the server.
The mechanism is as follows: when the server receives a connection request, it creates a socket and assigns it to a thread that is responsible for meeting the communication request, leaving the server socket waiting for the next request. Broadly speaking, the steps involved in executing this type of socket server are as follows:

- The server waits for requests from clients through the accept method of the ServerSocket class. Each of these requests is translated into a TCP socket between the client and the server.
- When the server receives a request, it creates a thread (in the example shown below using the Process Manager class) and provides it with the socket.
- The thread is responsible for carrying out the process, freeing the server to attend to the next request.

The Example03 example shows the code associated with a multithreaded server that performs a process that generates a random number, waits as many seconds as that number indicates, and returns it to the client.

5. Examples

5.1. Example01

Below is the code for a server and client based on TCP sockets. In this example there is a very basic exchange of information: the client sends an integer with the value 100 to the server and the server responds with another integer with the value 200. When using the `read()` method, only numbers can be read integers between 0 and 255 (one byte).

The two classes must be considered as two different software systems, the server must be executed first, which will wait for the requests, and the client second.

`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  }
```

In the class constructor, a socket of type `ServerSocket` associated with the `IP` address of the host is created (by default the `IP` address of the machine is assigned, in this example being 192.168.10.10) and a port (in this example, 49171).
In the `start()` method, the `socket` is told to wait for requests via the `accept()` method of the `ServerSocket` object created in the constructor. At this point the execution will be stopped until a connection request arrives from a client. Once the connection request is received, the `Socket` class object is created that represents the connection between server and client.

Once established, the input and output data streams are created. In this example, the communication streams are established through objects of the abstract classes `InputStream` and `OutputStream`, which basically allow reading and writing bytes.

The server reads from the `InputStream` the integer sent by the client via the `read()` method and sends the integer 200 via the `write` method of the `OutputStream`.

`SocketTCPClient.java`

```
1  import java.io.IOException;
2  import java.io.InputStream;
3  import java.io.OutputStream;
4  import java.net.Socket;
5  import java.net.UnknownHostException;
6
7  public class SocketTCPClient {
8
9      private String serverIP;
10     private int serverPort;
11     private Socket socket;
12     private InputStream is;
13     private OutputStream os;
14
15     public SocketTCPClient(String serverIP, int serverPort) {
16         this.serverIP = serverIP;
17         this.serverPort = serverPort;
18     }
19
```

```
    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();
        }
    }
}
```

The execution of the server would be:

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

And the client's:

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

For its part, the client creates a socket indicating the host name (in this example localhost) or the IP address. Whether the host name or IP address is used to identify the server, the data corresponding to the computer hosting said server must be indicated. If the client and server are running on the same machine, localhost can be used as the hostname, as in this case.

In addition to the IP address, the port on which the server is listening is indicated, in this case being 49471.

Once the connection is established, the communication streams represented by objects of the `InputStream` and `OutputStream` classes are obtained to proceed to write the message that you want to send to the server and read the one that it sends as a response.

As can be seen, the level of abstraction is low, as there is nothing more than the existing methods in the `InputStream` and `OutputStream` classes, but these objects can be wrapped with higher level classes to allow the exchange of other types. of data.

The `InputStream` class's parameterless `read` method reads only one byte. If you want to read groups of bytes, you will have to use one of the variants of this method. (See the reading and writing information topic)

> We can establish the connection between different Operating Systems, we can run the server on Windows and the client on Linux or vice versa. We could even connect from a Java client to a Server programmed in Python (or another language) or vice versa.

5.2. Example02

Below is a server and client exchanging messages over UDP `sockets`.

`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();
```

```
            DatagramPacket outputDatagram = new DatagramPacket(sentMessage,
                    sentMessage.length, inputDatagram.getAddress(),
                    inputDatagram.getPort());
            socket.send(outputDatagram);
            System.out.println("(Server): Closing socket...");
            socket.close();
            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();
        }
    }
}
```

The execution of the server would be:

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

And the client's:

```
(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. Example03

`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`

```
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();
        }
    }
}
```

To run this example on a single computer, you can open multiple terminals, run the server on one of them, and the clients on the rest. In this way, it is verified how the connections are established and the requests are attended to simultaneously. By arranging the class that performs the process of a timer, you can check how it works correctly.

Next we see the execution of the server:

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

And the executions of the four clients in terminals:

```
(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.
```

All the connections are established and the processes are started without the need to have previously terminated any of the other client processes.

Of course, if the number of concurrent requests were very high, a job queue would have to be established to store them while resources are released, in order not to overload the system.

6. Information sources

- 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