Java Socket Programming

PID_00275903

Maria Isabel March Hermo

Recommended minimum reading time: 5 hours

Universitat Oberta de Catalunya

Maria Isabel March Hermo

First edition: February 2022

Av. Tibidabo, 39-43, 08035 Barcelona
Authorship: Maria Isabel March Hermo
Production: FUOC

Contents

Introduction

The reason for the existence of the Internet is the communication between devices of any kind, connected throughout the world. Currently, there are an infinity of online applications, of very different themes: web, email, GPS navigation, playback via audio and video streaming, multi-user games, e-commerce, smart household appliances, video tutorials, social networks...

This structured, complex and heterogeneous network constantly conveys the message exchanges that occur between the processes that run these applications, transparently to the end user.

In this module, we will learn how to program one of these online applications, using the most popular application programming interface (API) for communicating remote processes: sockets.

When we are communicating two or more applications from different devices, we need a series of conventions for the network to redirect packets to both parties. Sockets are the software interface for creating this communicative framework, that is, they are a set of instructions that the application must follow to communicate with the transport level and its counterparts in the destination: thus allowing the exchange of data between local or remote processes.

There are several classifications of sockets, but we will go more deeply into the typology according to the service they offer: oriented or not to connection, or, in other words, based on the TCP or UDP transport protocol. This is a key differentiation, since it defines the nature of the sockets, and how they act and behave throughout the communication. Consequently, the sequence of operations that must be performed in one case or the other is different.

After defining the conceptual framework, the module examines the Java classes that support socket programming. The constructors, the main methods, and the exceptions that can be thrown are seen in detail, so it can be used as a reference manual in programming network applications.

Although there is no need for advanced knowledge of the Java language programming, it is recommended to know how to compile and execute files on this platform, as well as the fundamentals of structured programming (classes, constructors, *getters* and *setters* methods, etc.), the syntax of the most relevant primitive data types, and the basic instructions for manipulating them.

Objectives

The main objectives that the student shall achieve during the understanding of this module are:

1. Define the socket programming interface and situate it within the conceptual framework of the TCP/IP protocol stack, the client-server paradigm, and the different communication mechanisms that exist.

2. Differentiate the characteristics of connection-oriented and connectionless communications, to choose the ideal one according to the final functionality of the network application.

3. Draw the typical connection-oriented or connectionless client-server flowchart.

4. Know the Java classes related to socket programming, both connection-oriented and connectionless. Specifically, their constructors, main methods, and parameters.

5. Program an online application, communicating remote processes via sockets, both in the role of the clients and of the servers.

1. History

The term socket, understood as a communication mechanism between remote processes, emerged in 1971, in RFC 147 "The Definition of a Socket", written by J.M. Winett.

> "A socket is defined to be the unique identification to or from which information is transmitted in the network. The socket is specified as a 32 bit number with even sockets identifying receiving sockets and odd sockets identifying sending sockets. A socket is also identified by the host in which the sending or receiving processor is located."

This was not a public document, but used within the ARPANET network, considered to be the precursor of the networks used today. Initially, it was a military network, created by the Department of Defense of the United States in the late sixties of the last century. It ended up connecting many universities and government facilities, using conventional telephone lines. Later, when satellite or radio links were added, the systems started having problems interacting with these new networks. It became clear that a new reference architecture was needed to connect different network models. This architecture became popular as the TCP/IP reference model (initials of its two main protocols), which is the Internet model. Sockets arose from the need to connect the higher levels of this TCP/IP model and also, to communicate applications that were running on heterogeneous devices of the ARPANET network.

Today, however, the definition of sockets is far from the initial idea defined in RFC 147, even though their function is the same. Most socket implementations are based on Berkeley sockets. They are named like this, because they take as reference a distribution of the Unix operating system, in the variant implemented by the University of Berkeley, version 4.2 (UNIX BSD 4.2), in 1983. In this distribution, a series of system calls were implemented that enabled interconnectivity between processes via sockets, combined in a specific application programming interface (API sockets). These applications employed the TCP/IP protocol stack and were tested on the ARPANET network, and its successor, ARPA.

According to the Berkeley standard, sockets were implemented as a type of file descriptor, following the Unix philosophy. That is why when working with sockets, we will find similar functions to the classic `open()`, `read()`, `write()` or `close()`, more typical of the scope of files. Currently, the programming interface via sockets has been adapted to different operating systems. For example, *WinSock* is another implementation based on Berkeley sockets, used by Microsoft's operating system.

There are other implementations, similar to the sockets API, that define a set of system calls to communicate with transport protocols. One of them is the TLI interface (Transport Layer Interface), based on data flows or streams, distributed in another variant of UNIX called System V, in its version 3 (AT&T UNE System V or SVR3), in 1987. The TLI interface is based on the OSI reference model (Open System Interconnection), which became obsolete due to the TCP/IP model.

2. What is a socket?

In the previous section we have seen what motivated the need to define a programming interface via sockets, but what exactly are sockets?

The classical analogy used to understand the concept of sockets is the telephone system as a way of communicating people. The sockets would be the telephones, allowing information to be exchanged between processes, which would be the people. Thus, sockets are points of communication between agents (processes or people respectively), through which information can be sent or received.

> Sockets are communication mechanisms between local or remote processes. They allow a process to communicate bidirectionally with another process.

Another analogy is the postal service. When we want to send a letter to another person, we first have to write the content (the data) and put it in an envelope. After closing it, we have to write the recipient's full name, address and postal code, usually on the bottom right of the envelope. Then, we put a stamp on the upper right side. Finally, we take the letter to a mailbox, or a courier office. In the same way the postal service has this set of rules that must be followed by the sender of the communication to ensure that the destination receives the letter, the Internet makes use of the sockets interfaces so that the process that sends the data does so according to the established rules.

> Sockets are a software interface formed by a set of code instructions that the communicating programs must follow so that the Internet can deliver the data.

2.1. Sockets in the TCP/IP reference model

Network communication standards provide the basis for data transmission between different equipment, for the manufacturing of compatible network equipment, and for the design of routines within operating systems that facilitate remote communications.

There are two main communication models that use layer-based structuring: the OSI reference model (Open System Interconnection), which has been relegated to the theoretical field, and the TCP/IP protocol model (Transport Control Protocol/Internet Protocol), widely used, and therefore also called the Internet model or Internet protocol stack.

Remember that the Internet protocol stack is structured in five layers, as shown in the following table.

Table 1.

Application	*Data*	*DNS, HTTP, P2P, EMAIL, TELNET, FTP*
Transport	*Segments/Datagrams*	*TCP, UDP*
Net	*Packets*	*IP, ARP, ICMP*
Link	*Frames*	*ETHERNET, 802.11, ATM, PPP*
Physical	*Bit*	*RS-232, RJ-45, DSL, 100BASE-TX*

Each layer has a distinct function, which we will summarize below. In addition, in each of them we can find different protocols that respond to one of these layers' specific services, differentiated from the layer where they are located.

- **Physical**: processes the transmission of bits by the physical means of transport, such as optical fiber, coaxial, radio...

- **Link**: allows the transmission of frames between directly connected machines. The most common network is Ethernet.

- **Net**: defines the independent addressing and routing of packets over the network, so that they go from source to destination following the best route. The most commonly used protocol is IP. There is also ARP (address resolution) or ICMP (control messages).

- **Transport**: is responsible for providing services for end-to-end communication, such as reliability and security that the data reaches the destination in the correct order. The connection-oriented TCP or connectionless UDP protocol is used according to the needs of the applications.

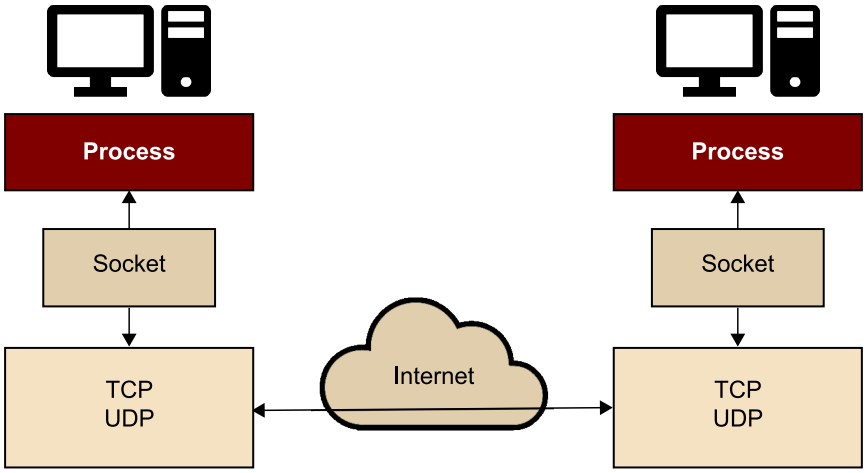
- **Application**: is the closest to the user. It is made up of applications and processes that communication networks use. The most popular protocols are HTTP (*Hypertext Transfer Protocol*), SMTP (*Simple Mail Transfer Protocol*) or SSH (*Secure Shell*), among others.

Communication between layers has two aspects:

- End-to-end communication at the same layer.

- Layer immediately below and above, in relation to the same device.

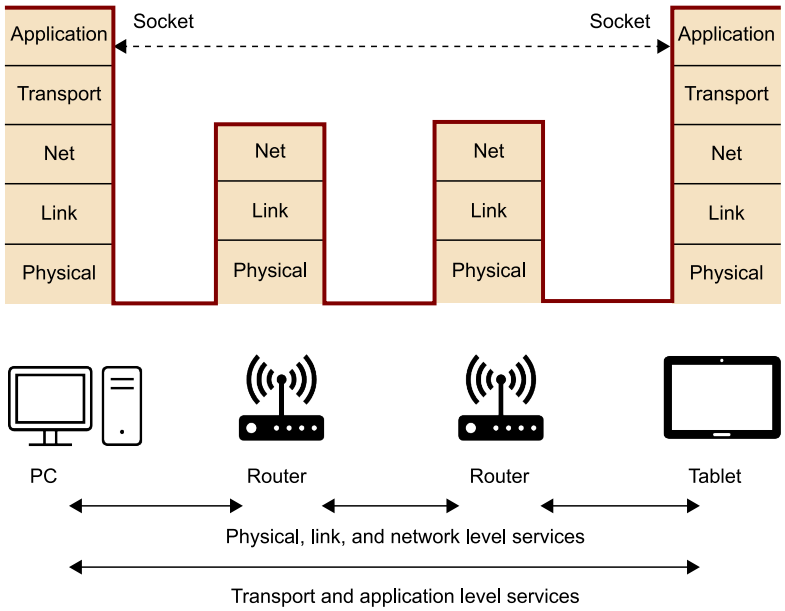
Figure 1.



As can be seen, the sockets are between the application layer and the transport layer. They are also called API (*Application Programming Interface*) between the application and the rest of the layers that will be communicated to us through the network.

The application layer, specifically based on the processes that are really the active articles of the communications, uses the sockets to send the data to the transport layer, which, after the relevant analysis and management, will transfer them to the lower levels. The data will be sent over the network to the host destination, where finally, after analysis and management by the lower levels, will reach the transport layer, from where they will be redirected to the corresponding process through the socket. After processing the data, it is usual for the reverse process to occur. This process is called **packet encapsulation and decapsulation** between the layers of the Internet model. We can observe this process as the strong brown line of the following figure.

Figure 2.



Sockets are network application programming interfaces (APIs) that are situated between the transport layer and the application layer, in the TCP/IP reference model.

Intermediate network elements, such as routers and switches, do not need socket implementation, since they operate at the link layer (switches) or at the network layer (routers). However, firewalls, IP address translators or proxies, do keep track of active sockets. Also, information about sockets is used in certain quality of service (QoS) policies, implemented in routers, or certain routing protocols.

2.2. Sockets in the client-server paradigm

So far, we have talked about how applications communicate with each other over the network. There are several ways to enable this end-to-end communication over the Internet. The vast majority of online applications, which use sockets to communicate, follow the client-server paradigm.

The client-server architecture is a software design model based on two profiles that communicate:

- Providers of resources or services, called **servers**.
- The applicants of these services, called **clients**.

Client programs run on hosts that typically interact with users. The processes created as a result of the execution of these programs perform requests to one or more servers to obtain certain services. This idea can be applied in a local way, but it is more common and useful in distributed systems, through a network of computers, where many users communicate with a server.

Server programs are programs that are running on a remote host and offer services to multiple potential clients, usually in an uninterrupted and continuous manner.

For example, in Web applications, a client browser exchanges messages with a domain hosted on a Web server. In a P2P file-sharing system, such as BitTorrent, a file is transferred between two hosts: the one that offers the file acts as a server, and the one that downloads the file acts as a client.

There may also be cases where a particular host acts as a client and server at once, for multiple applications or even within the same application. For example, in BitTorrent, it is also common that, from one end a file is being offered and, at the same time, it is downloading files offered by others.

In either role, sockets enable communication between both parts.

- On the **server-**process side, sockets are used for advertising services offered on Internet, and enabling communication with the clients who request them.

- On the **client-**process side, sockets are used for communicating with the servers, making requests for services and obtaining their answers.

2.3. Sockets as a communication mechanism

There are various techniques and paradigms to carry out communication between processes that are executed in a distributed, parallel or concurrent manner. The greatest exponents on which most of today's online software is based are as follows:

- **Message passing**: synchronizes two processes so that they pass a message to each other cooperatively. One of them sends it and the other receives it.

- **Remote invocation**: executes a procedure or method, typically on another device, as if it were doing it on the same machine.

In fact, both are similar, as there is an active process that sends the message or calls the execution of a method, respectively. However, remote invocation does not require the receiver to be waiting to read any requests, and also, the response is atomic.

Sockets are an implementation of **message passing**, since there are several participants who explicitly communicate bidirectionally through data passing.

2.4. Socket identification

We will recover the previous example of postal mailing for a moment. If we want to send a letter, one of the essential points is to know the full address of the recipient. Without this data it would be impossible for the transport company to make the delivery correctly. The sender of the letter is also usually put, to know who sent it, in order to be able to respond or report a possible error in delivery. These data are put on the front and back of a letter in a specific order:

- the full name,
- the address with the street and the number,
- the postal code and the town.

In the same way, the processes that want to communicate through the network need to identify themselves with some basic data. In network jargon, this identifier of processes that communicate over the Internet is the **name of the socket**, and consists of:

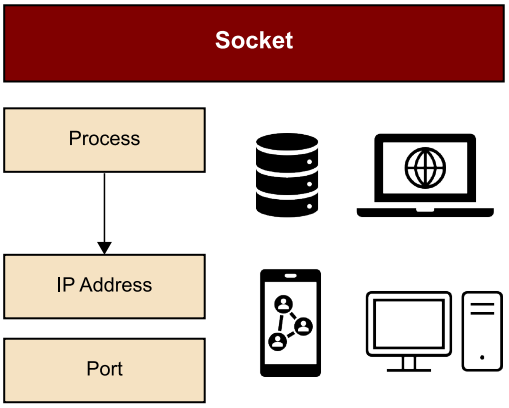
- **IP address:** identifies any terminal of any network in the world on the Internet. They are 4 bytes that are usually written in decimal notation: 4 numbers from 0 to 255 separated by points.

- **Port:** identifies a specific process on a machine. They are 2 bytes that are usually written in decimal notation (a number). Therefore, a port can be numbered from 0 to 655436.

We must not confuse the IP address with the physical address, which is the address that unequivocally identifies the machine's network card. For example, in Ethernet networks it is the MAC address, composed of 6 bytes that are often expressed in hexadecimal, such as 54:D4:6F:AD:67:E4. Unlike the IP address and the port, the programmer does not know the MAC addresses nor works with them, unless it is at a low layer, to configure some specific aspect related to the link layer.

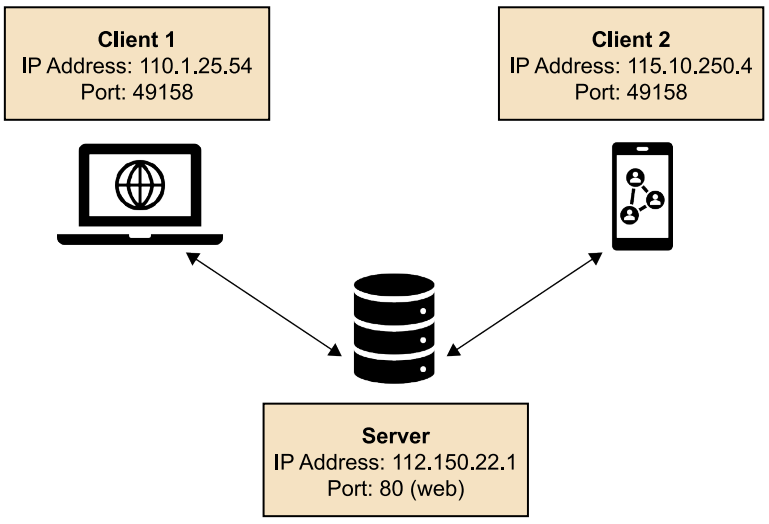
The identification of each socket by its IP address and port is unique. The IP address identifies the device within the Internet network and the port identifies the process within this device, whatever type it may be of.

Figure 3.



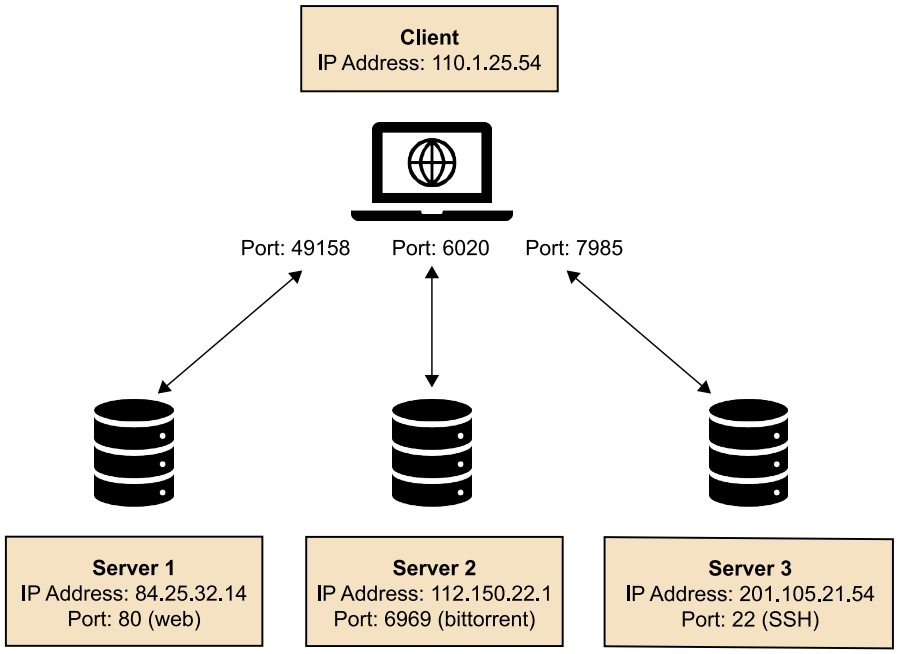
To simplify, we will take the case of a computer with a single network card. This computer is identified on the network by IP 80.55.23.69. All processes running on the same computer will have this same IP. However, there cannot be two sockets open at the same time on the same machine with the same port number. The transport layer would not be able to deliver the segments that reach one or the other process, since it would have no way to differentiate them. However, there can be two processes communicating over the network with the same port number, as long as they belong to different computers. That is, there can be two processes with the same port number but with different IP addresses. Thus, the socket is unequivocally identified by this < IP address, port >, as can be seen in the following figure.

Figure 4.



For example, on the same computer, we can have a browser open with several tabs consulting websites, and also downloading some files with the BitTorrent program, or securely uploading others with SSH. Each of these applications will have created at least one socket where data is exchanged with the relevant server, as can be seen in the following figure:

Figure 5.



A given machine will be able to have a set of active servers and connections to remote servers, thanks to port distinction.

Next, we will see in greater detail which IP addresses and ports are used for programming online applications.

IP addresses

RFC 1918 to 1996 defines the sets of IP addresses intended for internal use and public use:

- **Private addresses** are the IPv4 addresses that are intended for the creation of private networks, such as the internal network of a company that does not need to access the computers directly from the Internet.

- **Public addresses** are seen all over the Internet. Consequently, there cannot be two machines with the same IP address.

Table 2.

Class	*Prefix*	*Range*
A	10.0.0.0/8	10.0.0.0 – 10.255.255.255
B	172.16.0.0/12	172.16.0.0 – 172.31.255.255
C	192.168.0.0/16	192.168.0.0 – 192.168.255.255
All other addresses are public		

The assignment of IP addresses to hosts is usually done by our ISP or Internet provider. As the number of IP addresses is limited, ISPs rotate these addresses, which is why they are called **dynamic IP addresses**.

In addition, in order not to have to remember or write down the specific IP addresses of the servers where we want to connect, the **domain names** that we use constantly today, such as *uoc.edu* or *google.com* were devised. Using the DNS protocol (*Domain Name Server*) the translation of domain name to IP address, and vice versa, is done transparently to the user. The programmer also has conversion functions at his/her disposal, to facilitate this translation when creating a socket or connecting to a server.

Ports

At another level, the assignment of ports to processes is not arbitrary either. The IANA organization (Internet Assigned Number Authority) defined three port ranges:

- **Known ports** (0..1023): universal and known fixed ports, assigned by the IANA, associated with specific services that are standardized and regulated by an RFC.

- **Registered ports** (1024..49151): ports used by user applications.

- **Dynamic or ephemeral ports** (49151..65535): ports used temporarily so that a client process can connect to a server process.

Normally, if we want to access a known server, we will have to create a client socket that makes requests to a server socket, indicating the domain name of this server and the known port according to the table below.

Table 3.

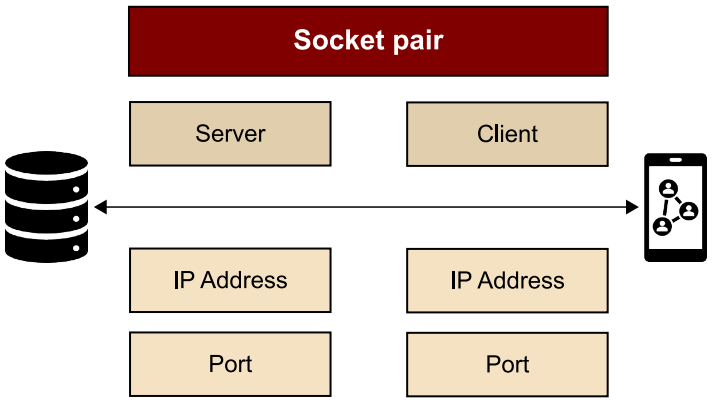
Service	*Port*	*TCP*	*UDP*
DayTime	13	√	√
FTP-Data	20	√	
FTP	21	√	

Service	*Port*	*TCP*	*UDP*
SSH	22	√	
Telnet	23	√	
SMTP	25	√	
DNS	53	√	√
HTTP	80	√	
POP3	110	√	
IMAP	143	√	
SNMP	161		√
HTTPS	443	√	
YES	5060		√

2.5. Types of sockets

The link between the two sockets, the server process socket and the client process socket, allows a two-way communication; thus, both ends of the communication can write and read at the same time. This is what we call a ***socket pair***, and unequivocally identifies a connection. As the name suggests, it is formed by the sockets corresponding to the two ends of the communication.

Figure 6.



This is a characteristic of sockets which differentiates them from other communication mechanisms, such as pipes.

Table 4.

Sockets	**Pipes**
Bidirectional channels	Unidirectional channels
Communication between remote processes	Communication between local processes
Client-server philosophy	Simple data exchange

A socket pair is a pair of sockets, typically formed by the client socket and the server socket, which communicate with each other in a bidirectional manner.

If we consider the role of sockets, we can distinguish between:

- **Active sockets**: those that initiate the connection with the other end.
- **Passive sockets**: those waiting to receive connections.

For example, in a typical client-server communication, the server will be waiting for connections from potential clients through a passive socket. By contrast, the client will attempt to connect to the server by using an active socket.

If we look at the type of applications that use sockets, we can have:

- **Standard applications**. Those that intend to communicate through a standard protocol according to the rules established in an RFC. For example, we want to program a client to communicate with any web server. The client's communication via sockets will have to follow the rules established in RFC 2616 of the HTTP (Web) protocol, writing requests and interpreting responses as dictated by the protocol. Otherwise, the web server will not understand what the client requests, and will return an error.

- **Proprietary applications**. Another option is to develop our own client and server, according to rules that we devise expressly for such communication, without following any standard. This option is common if we want to offer a specific service, which is not yet regulated, such as an online game, for example.

However, the most important classification lies in the communication services offered by the socket, as it determines the type of exchange that may take place:

- Connection-oriented sockets.
- Connectionless sockets.

Given its relevance in the programming of communications via sockets, we will see it in greater detail in the following sections.

3. Communication services

Continuing with the postal service analogy, we can now focus on the fact that all delivery companies usually offer more than one service to their clients for the delivery of letters or packages: express delivery, receipt confirmation, shipment tracking, etc. Similarly, the Internet provides multiple services for its online applications.

In particular, when we are developing a program that will be communicated by network, we need to choose which transport protocol we are most interested in depending on the services we require for monitoring the end-to-end data transmission: TCP or UDP.

This choice conditions the programming in all kinds of sockets, since it defines their nature. Also, the type of communication generated between the clients and the servers must be the same, so the pair of client-server sockets must be TCP or UDP, since the transport protocol is unique for each type.

Let us briefly review the different services offered by these two transport layer protocols:

UDP (*User Datagram Protocol*) is a connectionless protocol:

- There is no concept of connection established between the two ends of communication.
- Each datagram that is sent is independent of the previous and the next.
- There is no fragmentation of the datagrams.
- There is no guarantee about the receipt order of the datagrams, or even if they have been received correctly. These errors have to be detected by the application, and the datagrams must be retransmitted, if deemed necessary.

These characteristics make it a very fast protocol, widely used in specific requests and response queries to a server, and in applications in which the speed of communication prevails over the accuracy of the data. It is widely used in real-time applications. An example is the online broadcast of a bicycle race. The user prefers to follow the race even with a few bits missing, or without having a high quality image. It is also used in fast request-response protocols, like DNS, used to resolve the IP address of a domain name.

TCP (*Transport Datagram Protocol*) is a connection-oriented protocol that provides us with:

- Reliability in the transmission of the segments.
- Fragmentation of segments.
- Maintenance of the order of the segments.
- Use of a checksum to detect errors.
- A negotiation to establish the initial connection (handshake).

These characteristics make it an optimum protocol for applications such as the Web (HTTP) or SSH, where we need the data transmitted over the network to reach the recipient in full.

> The series of code instructions used for programming the sockets will be different if we are implementing a service that requires one transport protocol or the other.

Maintaining this duality according to the transport protocol, two main types of sockets are defined:

- **UDP or *datagram sockets***: for communications in non-connected mode, with the sending of datagrams of limited size (telegram type).

- **TCP or *stream sockets*** (data flow): for reliable communications in connected, two-way mode and with variable size of data messages.

In TCP or stream sockets, the socket interface clearly distinguishes between the client role and the server role which establish a permanent, bidirectional, univocal and reliable connection for the duration of the communication. By contrast, in UDP or datagram sockets there is no such concept of connection between the two ends of communication, nor is there any kind of error control of datagrams. To offer a service with a certain degree of reliability, it would have to be programmed explicitly.

There are other types of more specific sockets, such as ***raw sockets***, which allow access to lower-layer protocols, such as IP (network level). Raw sockets provide expert access to communications, and are used for configuring certain lower layer parameters of the TCP/IP protocol stack, varying their standard operation.

3.1. UDP or connectionless sockets

Next, we will see how sockets that use UDP as the transport protocol works. Remember that, being a connectionless protocol, it does not offer reliability, and therefore, there will be no control of the data flow nor fragmentation. The application layer will be responsible for giving the relevant instructions, if a certain degree of reliability is desired.

These fast communications, without prior and continuous communication establishment, imply that each datagram has to incorporate address information in its header.

> In connectionless communications, each datagram must include the socket pair, that is, the IP address and port to which the datagram is addressed, and the sender's IP address and port.

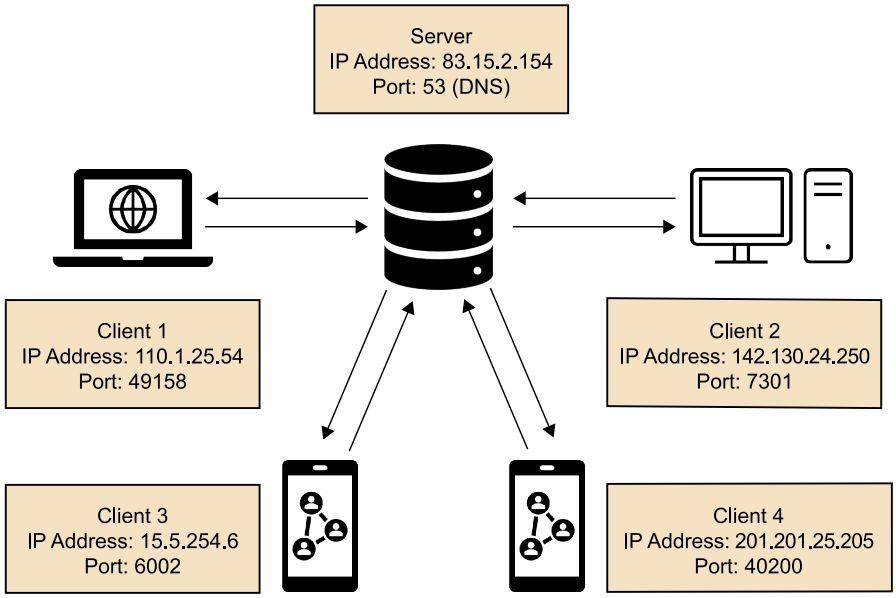
In the absence of such information, the communication endpoints would not know who they are communicating with, because there is no established and maintained connection for the data to travel. Each received datagram is treated individually.

In fact, if we recover the analogy with the postal service, it would simply be the data we put about the recipient and sender of a letter. If the recipient's name is not written down, the postal service cannot deliver it. And if the sender's address is not written down, the person who receives the letter will not have the address to direct the response, and will not be able to answer it.

> If we look at the client-server paradigm, clients and server are guided by a request-response mechanism.

As there is no dedicated connection between the two ends of the communication, the server will have to alternate attention to the multiple clients that make requests. That is, the server will deal with the requests that arrive, regardless of the client that makes them, according to the order of arrival. It will analyze and process them and respond to the recipient client, indicated in the same request.

Figure 7.



In connectionless communications, there is a unique socket of known name on the server, which is waiting for requests from new clients, and when there are some, it responds to them, writing on the same socket.

Client processes will also have to create a socket, without a known name, which they will use for making requests directly to the server, without establishing a previous connection.

> The socket pair between client and server is, therefore, not dedicated. It varies in the course of the communication and is maintained only for the treatment of the request and the preparation of the response to a certain client.

Let's look at this process in more detail:

- The server creates a listening socket, with a known name (IP address and port), through which the server waits for requests from potential clients.

- Each client creates an unnamed socket, to launch requests to the server.

- The client writes the data through the created socket, indicating the request made and the socket pair, that is, the IP address and port of the server, on the one hand, and the IP address and port (of the client), on the other. By putting the data of the sender of the communication, the server will know where to answer.

- The server receives the request from the client, reading from the listening socket. The server analyzes and processes it, and writes the response data

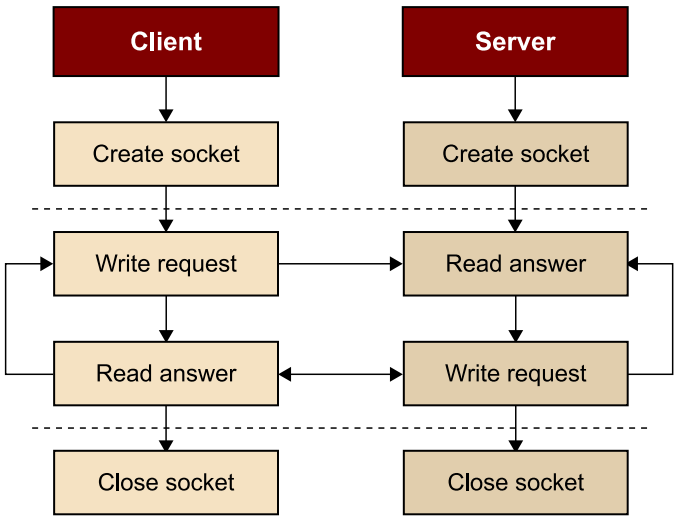
in the same socket. This response also carries the addresses and ports of the socket pair.

- When the communication is completed, the client closes the socket, to avoid unnecessary consumption of resources.

- The server continues responding to requests from clients.

- The server will close its socket when it stops, that is, when it no longer wants to act as such.

Figure 8.



In connectionless communications, wires or threads can also be used on the server. However, being a protocol oriented to treatment of requests and sending quick responses, it is not so essential.

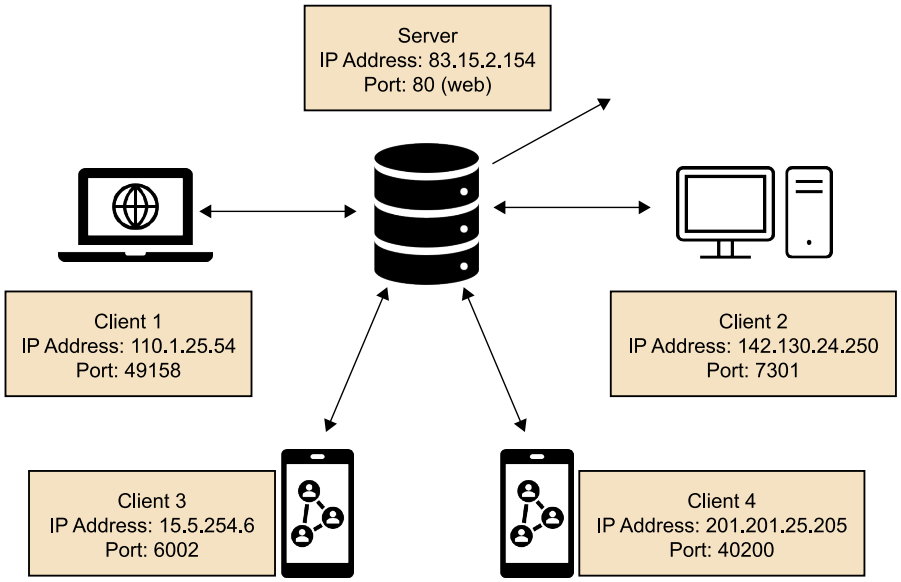
3.2. TCP or connection-oriented sockets

We are now going to specify what happens during communication between processes through connection-oriented sockets. Remember that this type of socket uses TCP as the transport protocol, and it is this protocol that is in charge of delivering the segments in a reliable manner. This reliability translates into the fact that the byte stream that is generated in the source process is delivered without errors to the destination process. In addition, the protocol fragments the data flow from the application layer into smaller messages.

Following the client-server paradigm, a process will act as a server process by creating a socket with a known name, and offering a service. This socket is called a listening socket. It will allow potential clients to connect in order to obtain the services offered.

Similarly, client processes also have to create a socket, without a known name, which they will use it to communicate with the server they connect to. This socket will establish a connection with the server. As a result, a new socket with no name will be created on the server, and it will connect only to the connected client.

Figure 9.



In connection-oriented communications, we will have:

- A listening socket on the server, waiting for potential clients.
- A socket on the server for each client connected to it.
- A socket on the client that connects it to the server in question.

These last two sockets define the socket pair, which will be the vehicle for the exchange of messages between clients and server.

There can be as many socket pairs on the servers as there are connected clients. However, there is usually only one listening socket waiting for connections from potential clients.

Now that we have a general idea of it, we will look at the specific functioning:

- The server creates a listening socket, with a known name (IP address and port), through which the server waits for connections from potential clients.

- Each client creates an unnamed socket, to launch the connection requests to the server.

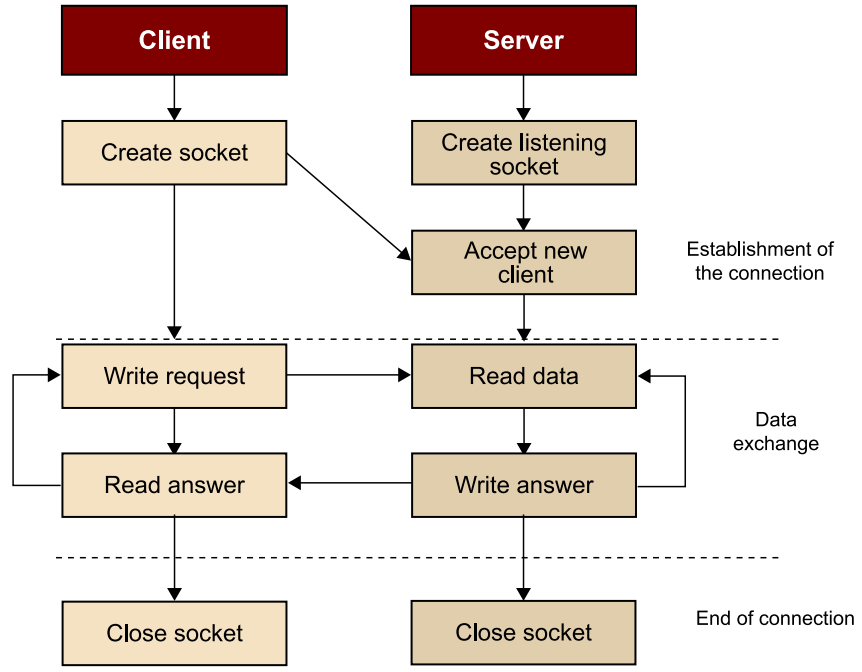
- The server process accepts this connection by creating a new socket. This will be the one that really maintains a two-way communication with the client, through which they will exchange requests and responses. The mentioned socket, created initially, is usually kept waiting for new connections from other clients.

- Once the connection is established, it is treated as a typical input and output stream, writing data and reading it, both on the client and on the server.

- When they have finished communicating, both the client socket and the server socket, both without a known name, must be closed to terminate the communication and thus avoid the unnecessary consumption of system resources.

- The server will continue waiting for connection requests from new clients through the listening socket.

Figure 10.



This is the typical process followed on a sequential server. However, in network communications, it is very common to use wires or threads on the server, that is, a father process creates other children processes to serve in parallel the clients who are connecting. Thus, a client does not have to wait for the server to finish attending to the client that was previously connected, in order to communicate the consequent waiting time with them. This results in multiple bidirectional and simultaneous communications that occur in parallel

between the children of the server and the clients. This fact implies that there are multiple sockets created on the same server, one for listening and one for each connected client.

4. Java Programming

4.1. Basic concepts

Now that we have finished the presentation of the conceptual framework, we will put it into practice and create our first network application.

Java is a programming language created in 1995 by Sun Microsystems. The great feature of this free platform is that it is a multiplatform, that is, it was designed with the idea that programmers write the program regardless of the operating system or device on which it is run. This fact is possible thanks to the Java virtual machine (JVM), which takes care of this portability, making it transparent to programmers and end users. The speed, robustness, security and reliability make it ideal for writing web applications easily. In addition, it has a very large community on the network, to consult basic or specific topics of any aspect of programming. As a support, Java developers can consult the official online documentation of their APIs, also called Javadoc.

Like any other programming language, Java has its own rules and syntax, derived from C and object-oriented programming (OOP). Java is structured in projects, which contain classes. Each class has a series of methods and variables, among others, that dictate its operation. Each of these files is named with the .java extension. After compilation, we will get the corresponding .class files, which can already be executed by the Java virtual machine (JVM).

First, to program any application, we will have to install a Java development kit (JDK), which contains, among other things, the compiler and the general utility class libraries. This must not be confused with JRE (Java Runtime Environment), which includes components required for running programs in Java.

Normally, to program applications in the Java programming language, an IDE is used, that is, a graphical interface that facilitates programming, error detection, compilation and execution of applications. There are many, but probably Eclipse and Netbeans are the most popular.

Java is based on **structured programming**. We will usually have a class that will match the name of the file containing the program code. The code execution must be started in the `main()` method of this class. This function is considered the entry point of the application, where the process begins. Con-

sequently, the instructions contained in the `main()` method must be executed in sequential order and following the code breaks, as indicated by the programmer.

Let us now look at a basic skeleton of what would be a program written in this programming language. In this case, it should be saved in a file called *test.java*.

```
import java.io.*; //libraries
public class test {  //class test
    public static void main(String argv[]) {
        //first method to be executed
        //program code
    }
}
```

4.2. Java exceptions

Java has its own exception or error handling system, based on an event scheduling mechanism.

A process executes the code instructions as dictated by the program it executes. If the programmer does not handle the exceptions and any of these instructions fail, the process ends due to an unexpected error. However, if the programmer handles the exceptions, the process continues its execution and is redirected to a series of instructions, as programmed.

Exception handling is very common and necessary when programming online applications, since errors are common and do not have to involve the program aborting unexpectedly. For example, let us suppose that a server is down or that it cannot serve us because there is an overload of clients. The scheduled client could look for an alternative server or wait for the situation to be resolved and try again, communicating it to the user with an informative message. A good programmer not only has to write the instructions so that the process runs when there are no errors; on the contrary, he/she must foresee all the possibilities and propose alternative code paths, when the functions executed do not respond as expected.

The words reserved for capturing and handling exceptions in Java are:

- ***Try***: within this block, we will put the code of the program that we want to "test". The code exceptions inside the *try* block will be captured.

- ***Catch***: this block lists the code instructions that will be executed when an error occurs in some part of the previous block, that is, it contains the error handling.

- ***Finally***: this part contains a block of code instructions that are always executed, whether there is an error or not.

Let's look at a simple example. Before continuing, try to understand the code and think about what the result of this program would be.

```
import java.io.*;
public class test {
    public static void main (String [] args) {
        try {
            System.out.println("Test1");
            int num = Integer.parseInt("E");
            System.out.println("Test2");
        } catch (Exception e) {
            System.out.println("Test3");
        } finally {
            System.out.println("Test4");
        }
    }
}
```

Notice that the line of code referring to the function fails, because we are passing a character as a parameter when it expects an integer. The result of running the above code would be:

```
Test 1
Test 3
Test 4
```

4.3. Java sockets classes

After this rough sketch of Java, we will now delve into the programming of network applications using sockets. In the java.net packet, Java provides the following classes to facilitate socket programming, according to their nature:

- ***DatagramSocket***: Class responsible for carrying out unreliable, connectionless communications (UDP protocol). The sending unit is a datagram, symbolized in an instance of the DatagramPacket object.

- ***Socket***: Basic object in a network communication. It represents a socket, that is, one of the ends of the connection or the socket pair, in reliable connection-oriented communications (using TCP as the transport protocol). Clients will connect to a particular server by using this socket object. Also, both clients and servers will write and read the data that is exchanged.

- ***ServerSocket***: Class responsible for implementing the Server connection in reliable connection-oriented communications (TCP protocol). It repre-

sents the listening socket, whereby the server is waiting for connection requests from potential clients. When a client's connection request is accepted, an instance of the socket class is returned, where the communication with the client will really take place.

- ***SocketImpl***: Abstract class for defining any type of communication, that is, to enable configuring the sockets without having to stick to the standard properties of TCP or UDP sockets described above. If we instantiate a subclass of SocketImpl, we can redefine its services to have full control of communication. For example, if we wanted to implement a firewall, we would have to redefine the `accept()` method, adding the necessary security controls. In fact, all the listed classes are instances of the SocketImpl class.

Another class often used when programming online applications is:

- ***InetAddress***: Class responsible for implementing an IP address.

4.4. Connectionless socket programming

As we have already seen in the "UDP or connectionless sockets" section, UDP is an unreliable protocol, that is, it allows sending datagrams through a network, without previously establishing a connection. The communication endpoints exchange datagrams. The datagram's header incorporates enough addressing information to know the sender and the recipient.

The sequence of programming instructions of a typical **sequential server** serving multiple clients is as follows:

- Create a connectionless socket that waits for client requests on the configured port.
- Read the datagram, related to the request sent by a client, through this listening socket.
- Respond to this request, writing the response datagram through the same socket.
- Repeat steps 2 and 3 as many times as necessary.
- Close the socket to free up system resources.

In connectionless communications, remember that the datagrams exchanged by clients and servers contain the socket pair, that is, the IP addresses and ports of the two communication endpoints.

Another point to highlight in this type of communication is that the server can receive requests from any client. As there is no establishment of communication, nor a single two-way channel between server and client, there is

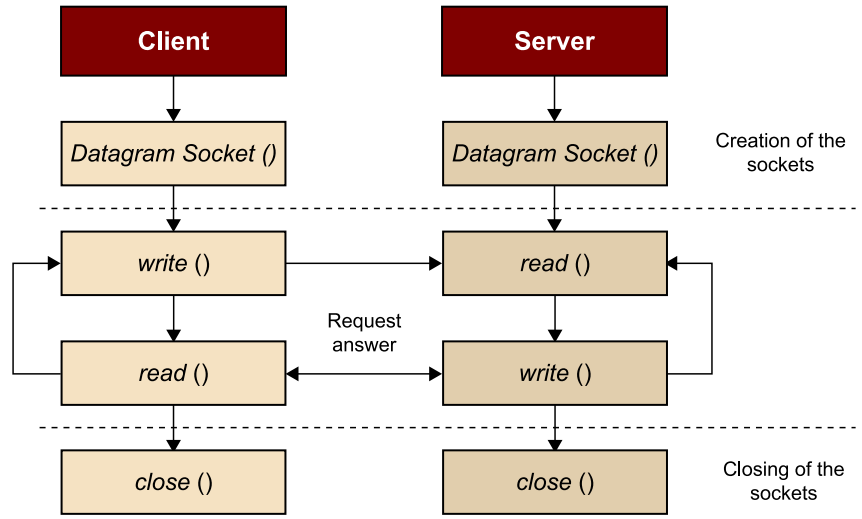
only one socket where requests are received and answered. This implies that we can insert requests from several clients at the same time, since there is no dedicated connection.

By contrast, the **client** will employ a similar mechanism, but being the active part of the communication:

- Create a connectionless socket to communicate with the server.
- Send the service request to the server, writing the datagram in the created socket, indicating the socket (the IP address and port) of the server to which it is addressed.
- Read the datagram related to the response sent by the server.
- Repeat steps 2 and 3 as many times as necessary.
- Close the socket to free up system resources.

In connectionless communications, both client and server use the same classes with their methods, so we will explain them generically and then see a specific example for each of these two roles. The generic graph of the methods involved in the above process is:

Figure 11.



4.4.1. Create a socket

To create a new UDP socket, either server or client, the first thing we have to do is create a new object of the `DatagramSocket()` class which will be used for sending and receiving datagrams. One of the following constructors can be used:

- `DatagramSocket ()`: creates a UDP socket, assigning it an available port number of the local machine.
- `DatagramSocket (int port)`: creates a socket and assigns it to the port passed as a parameter.

- `DatagramSocket (SocketAddress binaddr)`: creates a UDP socket, assigning it the address passed as a parameter.
- `DatagramSocket (int port, InetAddress addr)`: creates a socket and assigns it the address and port passed as a parameter.

The exceptions that can result from the execution of these calls are:

- `SocketException`: if the socket cannot be opened.
- `SecurityException`: if the device's security manager does not allow the socket to be created.

Normally, in the case of clients, we will use the first of the listed constructors, leaving the system to choose to create the socket with the IP address of the machine and one of the free ports. For servers, it is common to use the constructor described in the second point of the list, because a socket with a known name has to be created, and we need to know beforehand the port where it will be offering services so that clients can request it. For example, the following connectionless server listens for client requests on port 6000.

```
try {
    DatagramSocket se = new DatagramSocket (6000);
} catch (SocketException ex) {
    System.err.println(ex);
}
```

4.4.2. The datagram

In the course of communication between connectionless clients and servers, the unit of the sent packet is the datagram. In Java, the class that represents the datagram is `DatagramPacket()`. As we have mentioned, apart from the data itself, it has to include the recipient's address in order to direct the package. The most relevant constructors of this class are:

- *`DatagramPacket`*`(byte[] buf, int len, InetAddress address, int port)`
- *`DatagramPacket`*`(byte[] buf, int len, SocketAddress address)`

In both cases, a datagram-type package is constructed, containing:

- The data passed as the first parameter, of length `len`.
- The recipient socket identified by an IP address and port.

Given an object of the datagram type, we can always query or configure the previous attributes, with the methods:

- `InetAddress getAddress()`
- `byte[] getData()`

- `int getLength()`
- `int getPort()`
- `SocketAddress getSocketAddress()`
- `setAddress(InetAddress iaddr)`
- `setData(byte[] buf)`
- `setLength(int length)`
- `setPort(int iport)`
- `setSocketAddress(SocketAddress address)`

4.4.3. Send data

As we have seen, connection-oriented communications are based on a request-response protocol. Both to send the request from the client to the server and to send the response from the server to the client, the ideal call of the `DatagramSocket()` class is `send()`:

- `send(DatagramPacket p)`: sends a datagram over the socket. This datagram passed as a parameter contains the data to be sent, the IP address and port of the recipient, as specified in the previous section.

The `send()` method can mainly cause the following exceptions:

- `IOException`: if an input/output error occurs.
- `SecurityException`: if the security manager of the device does not allow the sending of datagrams via the socket.
- `PortUnreachableException`: if the destination socket cannot be found. There is no certainty that this exception will always be launched.

The following example constructs a datagram that is addressed to a local server that is running on port 6000 and sends it a message.

```
import java.net.*;
import java.io.*;
public class clienttcp {
    public static void main(String argv[]) {
        InetAddress adr;
        String message = "";
        byte[] message_bytes = new byte[256];
        DatagramPacket packet;
        try {
            DatagramSocket socket = new DatagramSocket (6001);
            adr = InetAddress.getByName("localhost");
            message = "Request to send to the server: ";
            message_bytes = message.getBytes();
            packet = new DatagramPacket(message_bytes, message.length(), adr, 6000);
            socket.send(packet);
        } catch (IOException ex) {
```

```
            System.err.println(ex);
        }
    }
}
```

4.4.4. Read data

Reading datagrams that reach us through a connectionless socket can come either from client requests on the server or from server responses on clients. The method to read from the socket in either case is `receive()`:

- `receive(DatagramPacket p)`: read a datagram from the socket. This datagram shall contain data, the IP address and the port of the sender, as specified in section 4.4.2.

We can highlight the following exceptions to this method:

- `IOException`: if an input/output error occurs.
- `PortUnreachableException`: if the destination socket cannot be found. There is no certainty that this exception will always be launched.
- `SocketTimeoutException`: if the time that has been configured for receiving the datagram runs out.

The `receive()` method is blocking, that is, the process will wait at this instruction until a datagram is received. If we want to configure this waiting time, we can use the following functions:

- `setSoTimeout (int timeout)`: enables or disables SO_TIMEOUT with the time specified as a parameter, in milliseconds. If it is 0, the `receive` call will be blocking until a datagram is received.
- `int getSoTimeout ()`: to get the current value of SO_TIMEOUT.

In case of running out of time while the process is waiting in the `receive()` call, the *SocketTimeoutException* exception is launched, which we can handle with the instructions deemed appropriate, in the code error treatment section.

Next, we will see a simple example of reading a datagram. Once received, it's content is displayed on the screen.

```
import java.net.*;
import java.io.*;
public class test {
    public static void main(String argv[]) {
        try {
            DatagramSocket socket= new DatagramSocket (6000);
            byte[] message_bytes = new byte[256];
            DatagramPacket packet = new DatagramPacket(message_bytes, 256);
```

```
            socket.receive(packet);
            System.out.println("Sender IP address: " + packet.getAddress());
            System.out.println("Sender port: " + packet.getPort());
            String message = new String(packet.getData());
            System.out.println("Received data: " + message);
            System.out.println("Length of data received: " +
            packet.getLength());
            socket.close();
        } catch (IOException ex) {
            System.err.println(ex);
        }
    }
}
```

4.4.5. Close a connection

To close a socket, the `close()` method must be called on the socket instance that explicitly communicates us with the client:

- `close()`: closes a socket.

The exception that this call can throw is:

- `IOException`: if an input/output error occurs.

An example would be:

```
sc.close();
```

4.4.6. Example of a client and a server

The following is a complete and functional example of a connectionless client and server. The general operation is as follows:

- The client sends to the server everything that the user enters by keyboard, until he/she receives the word *end*.
- The server reads everything that the client sends and displays it on the screen.

It is advisable to copy the server code in a file called serverudp.java, and the client code in a file called clientudp.java. They can be compiled and run, to test the interaction between client and server. The execution would be similar to the following screen:

```
Writing
to
the server
end
END
```

```
/** CONNECTIONLESS SERVER **/
/** Habitual libraries when working with sockets **/
import java.net.*;
import java.io.*;
/** A new Java class called UDP Server and a new main function
that governs its operation are created. In the first place, two variables are declared,
one of UDP socket server type and another message to manage the data that
we will exchange with the potential clients **/
public class serverudp {
    public static void main(String argv[]) {
        DatagramSocket socket = null;
        String message = "";
/** A server socket is created in UDP called socket by port 6000 that expects connections
    of potential clients. **/
        try {
            socket = new DatagramSocket(6000);
/** The variables that will read the data sent to us by the client are declared and initialized
using the socket we have created with it. **/
            byte[] message_bytes = new byte[256];
            DatagramPacket packet = new DatagramPacket(message_bytes, 256);
/*The message is initiated by a loop that reads what the client sends and
    shows it on the screen until the client sends the word “END” **/
            do {
                socket.receive(packet);
                message = new String(message_bytes);
                System.out.println(message);
            } while (!message.startsWith("END"));
/** We close the socket that allowed us to communicate with the client **/
            socket.close();
        } catch(IOException e1){
         e1.printStackTrace();
            System.err.println(e1);
        }
    }
}
```

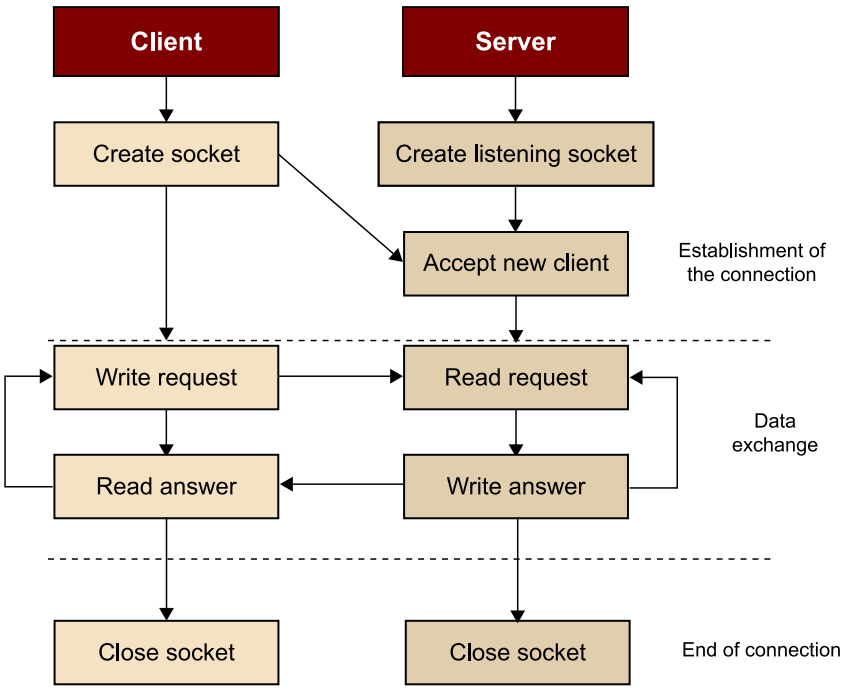
```
/** CONNECTIONLESS CLIENT **/
/** Habitual libraries when working with sockets **/
import java.net.*;
import java.io.*;
/** A new Java class called UDP Client and a new main function that governs its operation are created. First
a UDP client socket; the other called adr to manage the address of the server to which we want
```

```
to connect;
finally the message variable is used to manage the data that we will exchange with
the server **/
public class clientudp {
    public static void main(String argv[]) {
        DatagramSocket socket = null;
        InetAddress adr;
        String message = "";
        byte[] message_bytes = new byte[256];
        DatagramPacket packet;
        BufferedReader input = new BufferedReader(new InputStreamReader(System.in));
        message_bytes = message.getBytes();
/** A client socket called socket that will connect to the UDP server is created. In the adr variable
    we also put the address of the server that we have passed as parameter of the program **/
        try {
            socket = new DatagramSocket();
            adr = InetAddress.getByName(argv[0]);
/** Message exchange is initiated by a loop that reads what the user enters by keyboard,
    and sends it to the server, until we write the word  "END" **/
            do {
                message = input.readLine();
                message_bytes = message.getBytes();
                packet = new DatagramPacket(message_bytes, message.length(), adr, 6000);
                socket.send(packet);
            } while (!message.startsWith("end"));
/** We close the client socket that allowed us to communicate with the server **/
            socket.close();
        } catch(IOException e1){
         e1.printStackTrace();
            System.err.println(e1);
        }
    }
}
```

5. Connection-oriented socket programming

TCP is a connection-oriented protocol which allows the reliable delivery of segments, as explained in subsection 3.2. Before any data can be transmitted, it is necessary to establish a connection between the two ends that want to communicate, through which the messages are exchanged. Next, we will see the generic flowchart of client and server.

Figure 12.



5.1. The TCP server

In connection-oriented communications, the usual sequence of operations that a **sequential server** performs in the course of communication with clients is the following:

- Create a server socket and assign it a certain known port number so that potential clients can connect to it.
- Listen to new client connections and accept them. As a result, a new socket that will communicate the client and server is created. The listening socket will keep waiting for new connections.
- Read data that the client sends to us through the established socket pair.
- Send data to the client using the socket pair.
- Repeat steps 3 and 4 as many times as necessary.
- Close the connection with the client.

- Repeat the steps from step 2 onwards.

Next, we will take a detailed look at the Java ServerSocket class of the java.net packet, to see the methods that implement these steps of a typical sequential server. Later, we will see the exchange of data between client and server, as the calls are the same for the two endpoints of the communication.

5.1.1. Create a socket

To create a new server socket that listens for new connection requests from clients, the first thing we have to do is create a new object of the `ServerSocket` class, using one of the following constructors:

- `ServerSocket (int port)`: creates a server socket linked to the port passed as a parameter. If the port is 0, a random port is assigned, usually a number within the range of ephemeral ports. The maximum number of clients that are requesting to connect to the server and are waiting in the queue is 50. From this number onwards, new clients are discarded.

- `ServerSocket (int port, int backlog)`: creates a server socket linked to the port passed as a parameter. The maximum number of connections in the queue is as specified in the `backlog` parameter.

- `ServerSocket (int port, int backlog, InetAddress bindAddr)`: creates a server socket linked to the port and IP address passed as a parameter, with the maximum number of connections in the queue specified by the `backlog` parameter. Specifying the IP address is useful when we have several network interfaces on the machine where the server runs: wifi, ethernet... It can also be used when we want the server to only accept clients from a certain address.

Of the listed options, the most used is the first, for its ease of use. Note that, as we are on the server, we always have to indicate a port number that will be known by the rest of potential clients who want to connect to it. Without the IP address and port of this server, which are the ones that identify the socket, it would be impossible for clients to know where they need to connect.

The above calls may throw the following exceptions:

- `IOException`: if any error occurs when we open the socket.

- `SecurityException`: if there is a security manager, such as a firewall, that does not allow the operation performed.

- `IllegalArgumentException`: if the parameters are not correct. For example, if we put a port number that is outside the range of valid ports, from 0 to 65535 included.

For example, the following code creates a new server socket in the variable code, and displays a message on the screen in case of error:

```
try {
    ServerSocket se = new ServerSocket(80);
} catch (IOException ex) {
    System.err.println(ex);
}
```

5.1.2. Accept a connection

Once we have created a `Serversocket` object instance, the next step is to start listening for requests from new clients who want to connect to the server and accept them, using the `accept()` method. This method blocks the ongoing process until a connection with the client is performed. That is, we will not move on to the next code statement until a new client connects to the server.

- `Socket accept()`: the socket is kept waiting, listening for a new request from a client, and when it arrives, accepts it. It returns the created socket, which will be part of the socket pair that will link it to the client end, and through which bidirectional communication will be routed.

The above calls may throw the following exceptions:

- `IOException`: if an error occurs when we are waiting for a new connection from a client.

- `SecurityException`: if there is a security manager, such as a firewall, that does not allow the operation performed.

- `IllegalBlockingModeException`: if a socket has an associated channel, the channel is in non-blocking mode and there is no connection ready to be accepted.

Continuing with the example of the previous section, we would have:

```
try {
    ServerSocket se = new ServerSocket(80);
    Socket sc = se.accept();
} catch (IOException ex) {
    System.err.println(ex);
}
```

Note that the `accept` method returns a new `socket (sc)` object. It has been created to communicate explicitly with the client whose request has been accepted. Here is where we will read the clients' requests and send the server's reponses. So, the `server's sc socket`, together with the socket created in the client part form the socket pair through which bidirectional communication between client and server is produced. However, the `socket code` is a socket which is kept in standby mode, to listen for new client requests.

Also, there is the possibility of limiting the time during which the `accept()` method is blocked waiting for a new client:

- `setSoTimeout (int timeout)`: sets the time in milliseconds that the server is waiting for new client connections, that is, the time that the process is blocked in the `accept()` call. When the specified time expires, a `SocketTimeoutException` is thrown, which we will have to deal with appropriately.

- `int getSoTimeout ()`: related to the previous call, it returns time set to SO_TIMEOUT.

5.1.3. Close a connection

To close a socket, the `close()` method must be called on the socket instance that communicates explicitly with the client or that is listening to requests from new clients:

- `close()`: closes a socket.

The previous call may throw the exception:

- `IOException`: if an input or output error occurs when the socket is closed.

When we have finished the communication with a certain client, we can close the socket that was the local endpoint of the server in the socket pair. However, a server must always be active, waiting for connections from new clients, in order to serve them. If we want to stop it for any reason, we can use the previous close() method on the listening socket.

Calling this method is especially relevant when programming and testing, since otherwise, the process is left in an inconsistent state, and problems occur to reuse the same connection (same port number on the same computer).

The following code only accepts the client connection and ends up closing the created sockets.

```
try {
    ServerSocket se = new ServerSocket(8000);
    Socket sc = se.accept();
    sc.close();
    se.close();
} catch (IOException ex) {
    System.err.println(ex);
}
```

5.2. The TCP client

Now that we have seen the main classes and their methods at the server side, we will look at the client part in detail. The sequence of operations is as follows:

- Create a client socket, indicating the IP address and port of the server to which it wants to connect.
- Send data to the server, using the socket pair established with it.
- Read data sent to us by the server, through the established socket pair.
- Repeat steps 2 and 3 as many times as wanted.
- Close the connection with the server.

Next, we will look in detail at the socket class of the java.net packet, which is the class that implements a client socket and, remember, also the server socket that communicates with the client, forming the socket pair.

5.2.1. Establish a connection

To connect to a server, we have to create a new socket object, using one of the following constructors:

- `Socket (InetAddress address, int port)`: creates a client socket that connects to the server that has the IP address and port indicated as parameters.

- `Socket (String host, int port)`: creates a client socket that connects to the server, which has the domain name and port indicated as parameters.

- `Socket (InetAddress address, int port, InetAddress localAddr, int localPort)`: creates a client socket from the last two parameters, which connects to the server indicated in the first two parameters.

- `Socket (String host, int port, InetAddress interface, int localPort)`: the same as the previous one, but we explicitly indicate the network interface on which we will work. It is used in equipment with two or more Internet connections, for example Wifi and Ethernet.

The most used option is the second, for its ease of use. For example, the following line of code attempts to connect to port 80 in the uoc.edu domain, creating a new client socket in variable *c*:

```
Socket c = new Socket("uoc.edu", 80);
```

The client port is automatically assigned by the operating system within the range of ephemeral ports (from 49151 to 65535). The client-side application assigns the port number automatically and transparently, while on the server-side, the listening socket has to be known.

These constructors can primarily throw the following exceptions:

- `IOException`: if any input or output error occurs when we create the socket.

- `UnknownHostException`: if the IP address of the device cannot be resolved (in calls where we pass a domain name).

- `SecurityException`: if there is a security manager, such as a firewall, that does not allow the performed operation.

- `IllegalArgumentException`: if the parameters are not correct. For example, if we put a port number that is outside the range of valid ports, from 0 to 65535 included.

As we have seen before, we can handle these exceptions when we create a socket instance*:*

```
try {
    Socket c = new Socket("uoc.edu", 80);
} catch(IOException e1){
    e1.printStackTrace();
    System.err.println(e1);
}
```

5.2.2. Close a connection

To close a socket, the `close()` method must be invoked on the socket instance that communicates us explicitly with the client. The operation is the same as we saw on the server:

- `close()`: closes a socket.

The previous call may throw the exception:

- `IOException`: if an input or output error occurs when the socket is closed.

The following code is a very simple example of a client connecting to a server and closing the connection:

```
try {
    Socket c = new Socket("uoc.edu", 80);
    c.close();
} catch (IOException ex) {
    System.err.println(ex);
}
```

5.3. Communication flows in TCP

When we are programming sockets in Java, apart from using the classes offered by the java.net packet, it is usual to work together with the java.io packet, which contains a set of input and output channels that we can use to read and write data easily. Specifically, abstractions of the read and write operations of the data flows or streams are used. This abstraction allows the same model to be used regardless of the type of data exchanged and the device that performs these operations. That is, the same classes and methods are used to manage files, a device's screen, or the sockets. This fact offers great flexibility in programming.

A flow or stream is a sequence of data exchanged between a source and a destination of the communication. Java defines two main types of flows: byte streams and character streams.

Flows or byte streams manage byte input and output channels, for example, when reading and writing binary data, being especially useful when working with files. The following classes work on byte streams:

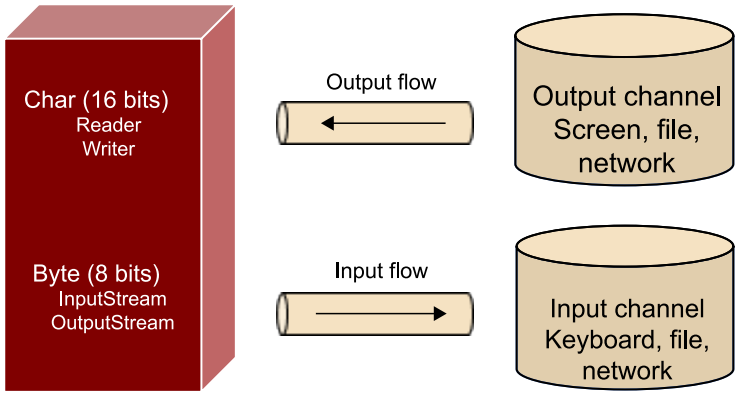
- *InputStream*: class for reading.
- *OutputStream*: class for writing.

Flows or character streams are another type, to manage the input and output of Unicode text and, therefore, can be internationalized:

- *Reader*: class for reading.
- *Writer*: class for writing.

The streams of bytes and characters **can be combined** on the input and output channel in question as it suits us, and both can be used at the same time, with the corresponding instance. For example, we can have a communication protocol that uses a set of characters for the header, and a binary file for the content. In such a case, we would use a character flow for the header and a binary type flow for the file. We could exchange data by using the methods that each type of flow offers us, using the same channel.

Figure 13.

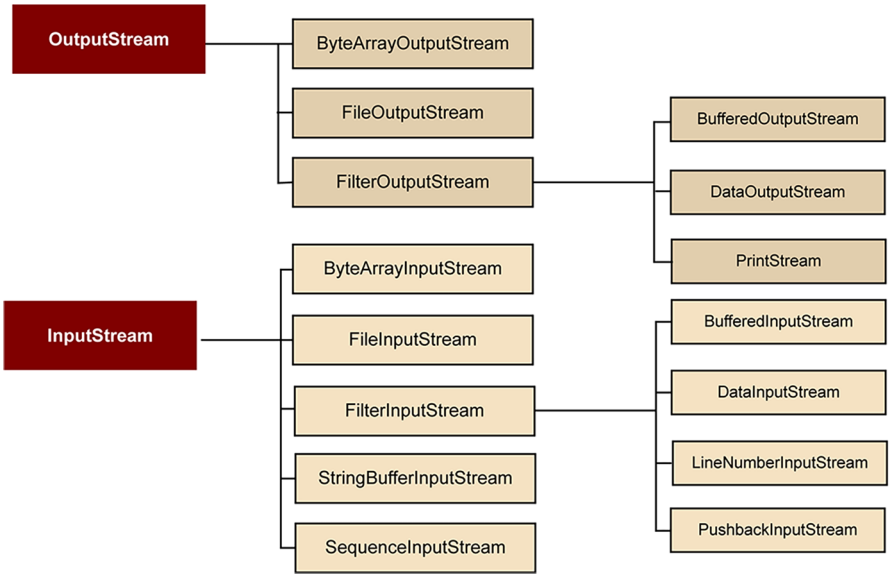


It can also be **translated** or switched from a byte stream to a Unicode character stream and vice versa, encoding the data and using the following classes:

- *InputStreamReader*: reads bytes and decodes them as characters.
- *OutputStreamWriter*: receives characters and encodes them as bytes.

Both classes shown, for working with a binary type flow or for working with a character type flow, cannot be used directly, because they are abstract classes. These classes will have to be extended by others on which we can work and, thus, perform the corresponding input and output operations on objects. This **wrapper** is common in Java, and provides a set of classes that offer methods for manipulating abstract class objects. The *InputStream* and *OutputStream* subclass hierarchy for implementing specific types of input and output channels is extensive, so we will only highlight the following, for being of popular use when programming sockets.

Figure 14.



If they are **binary flows**, it is common to work reading and writing data, through the classes:

- *DataInputStream*: for the input channel (read).
- *DataOutputStream*: for the output channel (write).

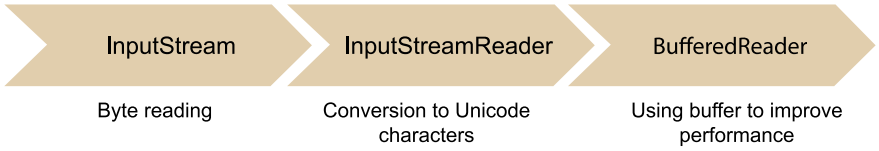
If we are working with **character flows**, this option does not exist directly, but the *PrintWriter* class for writing formatted data does exist.

It is also common to use the technique of ***buffers,*** for its efficiency when reading and writing. We use an internal buffer where the characters are stored, so we can choose the reading rate if it is not constant, optimizing the performance of the process. The classes that provide this functionality are:

- *BufferedReader*: for the input channel (read).
- *BufferedWriter*: for the output channel (write).

Next, we will work on these concepts with a practical example: the keyboard. The Java *System* class represents the standard input and output channel. The keyboard is represented as *System.in*. It is of *InputStream* type, that is, a byte flow. As we normally use characters, and not bytes, when working with the keyboard, we will create an *InputStreamReader* object from *System.in*. Now that we have the input channel, we can wrap it with another one, to provide it with the functionalities that suit us, such as, for example, the *BufferedReader*.

Figure 15.



Once the class that represents the input or output channel of bytes or characters has been defined, the usual functions to read and write data can be used, based on `read()` and `write()`. Each class offers a set of functions of its own, which differ slightly from each other, depending on the nuances they introduce. For example, reading or writing bytes, characters, lines, etc.

```
import java.io.*;
public class eco {
    public static void main (String[] args){
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String msg;
        System.out.println("Introduce a sentence: ");
        msg = br.readLine();
        System.out.println("The sentence is: " + msg);
    }
}
```

5.3.1. Send data

The reading and writing mechanism for client and server are identical. Once the socket between client and server has been established, we can manage byte flows by using the `InputStream` and `OutputStream` superclasses.

We will look at the writing first. To send binary data at a low level (byte array), an instance of the `OutputStream` class is used. The main functions are:

- `OutputStream()`: class constructor.
- `write(int b)`: writes the indicated byte on the output stream.
- `write(byte[] b)`: writes all bytes passed as a parameter.
- `write(byte[] b, int off, int len)`: writes *len* bytes of parameter b, starting from off position (offset). The first element is *b[off]* and the last *b[off+len-1]*.
- `flush()`: forces the writing of any pending byte on the buffer, that is, everything that had been indicated is immediately written in the output stream.
- `close()`: closes the output flow and releases the associated system resources.

The most relevant exception is:

- `IOException`: if any input or output error occurs.

A code example would be:

```
OutputStream out = socket.getOutputStream();
byte[] data = {0x5b, 0x42, 0x40, 0x34};
out.write(data);
```

If we want to send data in text format, we can convert the output stream into other classes, which will inherit the essential behaviour but add particular functionalities. One of these wrapping classes is `DataOutputStream` which allows writing Java primitive data types. Apart from the methods explained above, we can highlight:

- `DataOutputStream(OutputStream out)`: class constructor.
- `writeByte(int v)`: writes a byte in the output stream.
- `writeChar(int v)`: writes a character in the output stream, that is, two bytes, high byte first.
- `writeBytes(String s)`: writes a sequence of bytes, character by character, as a single byte.
- `writeChars(String s)`: the same as the previous one, but each character is written as two bytes.
- `writeUTF(String s)`: writes the string using UTF-8 encoding, creating machine independence.
- `writeInt(int v)`: writes an integer to the output stream, that is, four bytes, the high byte first.
- `writeLong`, `writeFloat`, `writeDouble`, ...

As we can see, by using the **`DataOutputStream`** class, the richness of methods offered is greater, making the task of the programmer easier. An example for typing a character or string of characters would be:

```
DataOutputStream out = new DataOutputStream(socket.getOutputStream());
out.write('a');
out.writeUTF("This is a message for a server.");
```

Another class that can be useful when writing via sockets and is also a wrapping of the `OutputStream` superclass is the **`PrintWriter`** class, responsible for printing formatted representations of objects to a flow of characters. The most important methods are:

- `PrintWriter (OutputStream out)`: class constructor. It will implicitly convert characters to bytes before typing.

- `PrintWriter(Writer out)`: class constructor. Both in this option and in the previous one, we can pass a Boolean as a second parameter to configure

the autoflush, that is, force the writing of everything that is in the buffer. Otherwise, it is not done.

- `print (boolean b)`, `print (char c)`, `print (int i)`, etc. Characters are converted to bytes according to the default encoding system, and are written by the corresponding output stream. The same variants are for the `println()` call that adds the current line by typing a line ending.

Besides, as in the previous case, we have the `write()` call on integers, characters, and character strings *(string)*, `flush()`, `close()`...

```
PrintWriter pw = new PrintWriter(socket.getOutputStream(),true);
pw.println("This is the message that we send to the server.");
```

5.3.2. Read data

In parallel, we will briefly summarize the main classes and methods involved in reading data, both on clients and servers, in connection-oriented communications. To read binary data at a low level (*byte array*), an instance of the `InputStream` superclass is used. The main functions are:

- `InputStream()`: class constructor.

- `int read()`: reads the next byte from the data stream (0..255). If no byte is available because we are at the end of the stream, it returns the value -1. This method is blocking, that is, the process will stay in this instruction waiting for a new datum.

- `int read(byte[] b)`: reads a number of bytes determined by the length of the variable that is passed as a parameter. They are stored in position *b[0]* to *b[len-1]*. It returns the number of bytes that have been read.

- `int available()`: returns the number of bytes that are available to be read in the input stream.

- `close()`: closes the input stream and releases the associated system resources.

The most relevant exception is:

- `IOException`: if any input or output error occurs.

A code example would be:

```
InputStream input = socket.getInputStream();
byte[] data = new byte[10];
```

```
int l = input.read(dades);
```

If we want to read data at a higher level (characters or strings) we can use a wrapper of the `InputStream` superclass, and convert it to a **`DataInputStream`** object, similar to how we have explained it for the writing. This class allows reading primitive data types from an input stream, inherited from the `DataInput` class, which represents strings of characters encoded in a Unicode format, that is a slight modification of UTF-8. Apart from the reading methods discussed in the previous superclass, we can highlight:

- `DataInputStream(InputStream out)`: class constructor.
- `readChar() byte`: reads 2 bytes from the input stream and converts them to a character.
- `String readUTF()`: reads in a Unicode string, encoded according to the modified UTF-8 format.
- `int readInt()`: reads 4 bytes of the input stream and converts them to an integer.
- `readDouble`, `readFloat`, `readLong` ...

Another reading class is **`InputStreamReader`** which is interesting, because it converts the bytes, read in the input flow, into characters. For this, it uses specified encoding, or that has been configured by default on the machine. It does this in such a way that is transparent to the user. The most relevant methods are:

- `InputStreamReader (InputStream out)`: class constructor.

- `InputStreamReader (InputStream out, String charsetName)`: class constructor, where we specify the encoding that will be used in the process of reading bytes from the input stream.

- `int read ()`: single character reading.

- `int read (char[] cbuf, int offset, int len)`: reading a string of characters of len length, stored from the *offset* position of the first parameter.

- `close()`: closes the input stream and releases the associated system resources.

An example of reading a single character would be:

```
InputStream input = socket.getInputStream();
InputStreamReader reader = new InputStreamReader(input);
int c = reader.read();
System.out.println("Received data " + (char)c);
```

If we use the `InputStreamReader` class and we read byte by byte, the conversion process would make our application inefficient. Therefore, it is common to use a new wrapping as the **`BufferedReader`** class, which uses *buffers* of reading to store what is available, and consume it as requested, using the FIFO technique. Apart from the reading methods described in the previous class we have:

- `BufferedReader (Reader in)` : class constructor. The size of the buffer can be indicated as a second parameter.

- `String readLine()` : reads a line of text, considering \n, \r or \r\n as the end of the line.

An example would be the following:

```
InputStream input = socket.getInputStream();
BufferedReader reader = new BufferedReader(new InputStreamReader(input));
String line = reader.readLine();
System.out.println("Received data " + line);
```

5.4. Example of a client and a server

The following is a complete and functional example of a connection-oriented client and server. The general operation is as follows:

- The client sends to the server everything that the user enters by keyboard, until it receives the word *END*.

- The server reads everything sent by the first connected client and displays it on the screen. When it receives the END message from this client, the execution ends.

It is recommended that you copy the server code into a *servertcp.java* file and the client code in a *clienttcp.java* file. You can compile and run them to test the interaction between client and server. An execution on both client and server would be:

```
Writing
to
the server
end
END
```

```
/** CONNECTION ORIENTED SERVER **/
/** Habitual libraries when working with sockets **/
import java.net.*;
```

```
import java.io.*;
/** A new Java class called TCP Server a new main function that governs
    its operation are created. In the first place, two variables are declared, one of TCP socket type  and
    another message to manage the data we exchange with potential
    clients **/
public class servertcp {
    public static void main(String argv[]) throws IOException {
        ServerSocket socket = null;
        String message;
/** A TCP server socket called socket is created by port 6001 that waits for connections from potential
    clients. When a particular client connects to the server, it is accepted by creating
    a new socket called socket_cli, through which bidirectional communication will be established
    between server and client **/
        socket = new ServerSocket(6001);
        Socket socket_cli = socket.accept();
/** A variable called input is declared and initialized through which the data that the
client sends us is read through the socket we created with it. **/
DataInputStream entry = new DataInputStream(socket_cli.getInputStream());



/** Message exchange is initiated by a loop that reads what the client sends and displays
    it on the screen until the client sends the word “END” **/
        do {
            message = input.readUTF();
            System.out.println(message);
        } while (!message.startsWith("END"));
/** We close the socket that allowed us to communicate with the client and the socket that was waiting
    for new connections from other clients **/
        socket_cli.close();
        socket.close();
    }
}
```

```
/** CONNECTION ORIENTED CLIENT **/
/** Habitual libraries when working with sockets **/
import java.net.*;
import java.io.*;
/** A new Java class called TCP Client and a new main function that governs
    its operation are created. First of all, the variables are declared, one of socket type
    that will represent a TCP client socket; the other called adr to manage the address of the
    server to which we want to connect; finally the message variable is used to manage the data
    that we will exchange with the server **/
public class clienttcp {
    public static void main(String argv[]) throws IOException {
        Socket socket = null;
        InetAddress adr;
        String message = "";
```

```
/** A client socket called socket is created that will connect to the server, that we pass to it
    as a program parameter (argv[0]) and port 6001. **/
        adr = InetAddress.getByName(argv[0]);
        socket = new Socket(adr, 6001);
/** A variable called input is declared and initialized that reads what the user is writing
    by keyboard and the output variable that will be responsible for writing the messages
  to the server. **/
        BufferedReader input = new BufferedReader(new InputStreamReader(System.in));
        DataOutputStream output = new DataOutputStream(socket.getOutputStream());
/** Message exchange is initiated by a loop that reads what the user enters
    by keyboard, and sends it to the server, until we write the word “END” **/
        do {
            message = input.readLine();
            output.writeUTF(message);
        } while (!message.startsWith("END"));
/** We close the client socket that allowed us to communicate with the server **/
        socket.close();
    }
}
```

6. Other operations

Some auxiliary operations to obtain information from the socket are briefly summarised below. Some are unique to servers, others can be used through both clients and servers.

Both in the socket class, as in the *ServerSocket* class, we highlight:

- `InetAddressgetInetAddress()`: returns the local address of the socket.
- `int getLocalPort()`: returns the port number to which the socket is listening.
- `SocketAddressgetLocalSocketAddress()`: returns the address of the local socket.
- `setReuseAddress(boolean on)`: enable or disable the OS_REUSEADDR option. If it is not enabled, when a TCP connection is closed, the connection is in a *timeout* state for some time (known as the TIME_WAIT state), making the same address and port unable to be reused. In fact, for applications that use known addresses or ports, this reuse probably would not be possible.
- `setReceiveBufferSize(int size)`: specifies the size of the buffer of clients waiting to be accepted by the server, according to the OS_RCVBUF option.
- `int getReceiveBufferSize()`: gets the previous time.
- `boolean isClosed()`: returns the status of the socket, whether it is closed or not.

Exclusive to the socket class (and consequently, they cannot be used in the server listening socket), we highlight the following functions:

- `int getPort()`: returns the remote port number we are communicating with.
- `SocketAddressgetRemoteSocketAddress()`: returns the address of the remote socket.

Here there is an example with some of these functions. Basically, a client socket is created which tries to connect to port 80 (web) of the server passed as a parameter and displays all the data of the connection on the screen:

```
import java.io.*;
import java.net.*;
public class informationSocket {
    public static void main(String[] args) {
        try {
            Socket sc = new Socket(args[0], 80);
            System.out.println("Connected to the host " + args[0]
                + " with IP address " + sc.getInetAddress().getHostAddress()
                + " and port " + sc.getPort()
                + ", from my socket with address " + sc.getLocalAddress()
                + " and port " + sc.getLocalPort());
        }
        catch (UnknownHostException ex) {
            System.err.println("Unknown host error " + args[0]);
        } catch (SocketException ex) {
            System.err.println("Error when connecting in the host " + args[0]);
        } catch (IOException ex) {
            System.err.println(ex);
        }
    }
}
```

The expected result, if no exception is thrown during the creation of the socket, is:

```
Connected to the host uoc.edu with IP address 213.73.40.242 and port 80, from my socket
with address /192.168.1.137 and port  55102
```

Summary

Sockets are a software interface formed by a set of code instructions, which the programs that communicate online have to follow, so that the Internet can deliver the data. Then, the main classes and methods for programming an online application in the Java programming language are summarized.

In connectionless communications, a single socket on the sequential server and a socket in each of the clients are created. Both are of the same type, created by the `DatagramSocket` class. The data exchange unit is the datagram and is represented via the `DatagramPacket` class. To send and receive data by the client and server socket, the `send()` and `receive()` methods are used, always indicating the data of the socket pair.

In connection-oriented communications, the sequential server will create two types of sockets. On the one hand, a listening socket with a known name is created, represented by the *ServerSocket* class, and where clients will be accepted by using the `accept()` method. On the other hand, a socket will also be created on the server, represented by the Socket class, and without a known name, to serve each client that connects. The client creates the same type of socket of the socket class. Data flows are exchanged by means of the two-way and univocal communication channel that the socket pair forms between the server and each client, and which remains active throughout the communication. To send and receive data, we will use the `read()` and `write()` methods, or similar, to the `InputStream` and `OutputStream` superclasses. To facilitate the programming and processing of data, it is common to use wrappers and their methods, such as `DataInputStream`, `BufferedReader` or `DataOutputStreamPrintWriter`.

In any of the types of sockets, it is always advisable to close the socket by using the `close()` method, when it no longer has to be used. In this way, we will free up unnecessary system resources and can reuse connections.

Bibliography

Kurose, J.; Ross, K. (2000). *Computer Networking: A Top-Down Approach*. Pearson.

Webography

<https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_71/rzab6/howdosockets.htm>.

<https://docs.oracle.com/javase/tutorial/networking/overview/networking.html>.

<http://web.mit.edu/6.031/www/sp19/classes/23-sockets-networking/>.

<https://ioc.xtec.cat/materials/FP/Materials/2201_SMX/SMX_2201_M05/web/html/index.html**>.**

<https://www3.uji.es/~belfern/libroJava.pdf>.