Lab-3:
Java Socket Programming – TCP Clients

1. How to be prepared for Lab-3

In the Lab-3 you will make a first contact with the socket interfaces in Java. To help you to prepare this lab, in PoliformaT (Recursos> Prácticas> CUATRIMESTREA>Práctica3) two videos are available: "Video 1: The interface of the sockets" and "Video 2: Programming of TCP clients in java". However, these videos are only in Spanish, so maybe those who are learning Spanish now can find them difficult to understand. Thus, as alternative to the videos, in this folder, you have also available a set of slides called T3- JavaSockets.pdf that explains how to work with socket interfaces.

2. Work Environment

Java is available for different operating systems and for different programming environments. The Lab-3 will be done on Linux. For the development of the Lab-3, two alternative working environments are suggested (although feel free to use any other you feel comfortable with):

a) The BlueJ work environment (www.bluej.org), of the University of Kent, and available in Windows, Linux and Mac OS.

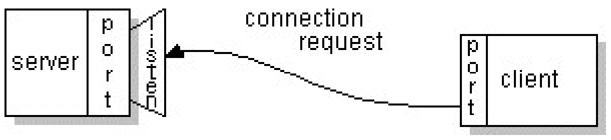
b) The "javac" compiler and the java virtual machine launcher ("java"), along with a text editor to write the program code, such as for example "gedit".

You can find information about Java classes in the online documentation provided by Oracle through its Web page (https://docs.oracle.com/en/java/javase/15/docs/api). Also, there are multiple Java tutorials on the network, for example https://openlibra.com/es/book/programacion-basica-en-java.

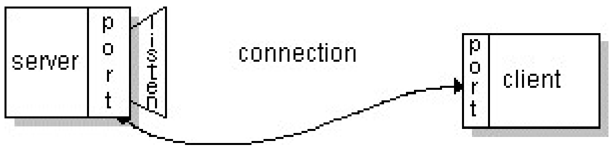
3. What Is a Socket?

Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request.

On the client-side: The client knows the hostname of the machine on which the server is running and the port number on which the server is listening. To make a connection request, the client tries to rendezvous with the server on the server's machine and port. The client also needs to identify itself to the server so it binds to a local port number that it will use during this connection. This is usually assigned by the system.



If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to the same local port and also has its remote endpoint set to the address and port of the client. It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.



On the client side, if the connection is accepted, a socket is successfully created, and the client can use the socket to communicate with the server.

The client and server can now communicate by writing to or reading from their sockets.

Thus, we can define a *socket* as one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to.

An endpoint is a combination of an IP address and a port number. Every TCP connection can be uniquely identified by its two endpoints. That way you can have multiple connections between your host and the server.

The java.net package in the Java platform provides a class, Socket, that implements one side of a two-way connection between your Java program and another program on the network. The Socket class sits on top of a platform-dependent implementation, hiding the details of any particular system from your Java program. By using the **`java.net.Socket`**[1] class instead of relying on native code, your Java programs can communicate over the network in a platform-independent fashion.

Additionally, **`java.net`** includes the **`ServerSocket`** class, which implements a socket that servers can use to listen for and accept connections to clients. This lab shows you how to use the **`Socket`** class to develop a TCP client.

4. TCP Client Program

The development of a TCP client program can be defined as:

1. Open a socket.
2. If the TCP client needs to read from the server, that is, receive data sent by the server, it is required to open an input stream to the socket
3. If the TCP client needs to write to the server, that is, send data to the server, it is required to open output stream to the socket.
4. Read from and/or write to the stream according to the server's protocol.
5. Close the socket (implicitly it closes input and output streams).

Only step 4 differs from client to client, depending on the server. The other steps remain largely the same.

[1] https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/net/Socket.html

4.1. Open a socket

There are several constructors for the java.net.Socket[2] class, we focus in the most used ones.

- `public Socket(InetAddress address, int port)throws IOException`
  - creates a stream socket and connects it to the specified port number at the specified IP address provided by an **`InetAddress`** object which corresponds to an IP address.
  - throws `IOException`[3]: it is a very general exception that occurs when an IO operations fails. It can happen because the destination port on the server is not open, there are routing problems in the network to reach the destination or simply that the specified server is off.
  - **`address`** - the IP address
    - the **`InetAddress`** class has several static methods that allow you to create **`InetAddress`** objects, initializing them. For example, the constructor:

```
public static InetAddress InetAddress.getByName (String HostName)
```

launch - in a transparent way to the programmer - a DNS resolution that translate the domain name provided in Hostame, to an IP address and store the obtained IP address. This method is used as follows:

```
InetAddress dirIP = InetAddress.getByName (“www.upv.es”);
```

- After executing this initialization, if everything has gone well, the IP address of www.upv.es will be stored in **`dirIP`**: 158.42.4.23. If name resolution fails, an `UnknownHostException` will be thrown.

- **`port`** - the port number that can range from 1 to 65,535.
- Example of use

```
InetAddress zoltar=
InetAddress.getByName("zoltar.redes.upv.es");
Socket s = new Socket(zoltar, 7);
```

[2] It is recommended at the beginning of the program to import the entire java package (**`import java.net. *;`**).

[3] The **`IOException`** class and its derived classes such as **`UnknownHostException`** require importing the io package (**`import java.io. *;`**).

- `public Socket (String host, int port)throws UnknownHostException, IOException`
    - creates a stream socket and connects it to the specified port number on the named host.
    - Throws
        - **`UnknownHostException`** when the IP address of the remote host it is trying to reach cannot be resolved, and
        - **`IOException`** (reasons indicated before)
    - **`host`** - the host name.
    - **`port`** - the port number that can range from 1 to 65,535.
    - Example of use

```
Socket s = new Socket("zoltar.redes.upv.es", 7);
```

Notice: the IP address is obtained by invoking - in a transparent way to the programmer - the service of resolution of names of the Domain Name (DNS). It is also supported the use of the IP address in format of String, like for example "158.42.180.62".

Under normal circumstances, after the creation of a socket using one of the previous constructors, a socket is connected to a machine and port. If the network is fine, the call to a socket constructor will return as soon as a connection is established, but if the remote machine is not responding, the constructor method may block for an indefinite amount of time.

4.2. Close a socket

It is important to close the connections in a consensual way. This is done using the `close ()` method of the Socket class.

Once a socket has been closed, it is not available for further networking use (i.e. it can't be reconnected or rebound). A new socket needs to be created.

Closing this socket will also close the socket's InputStream and OutputStream. Any thread currently blocked in an I/O operation upon this socket will throw a SocketException.

The connection will be permanently deleted when both processes, client and server, execute `close()` method.

The following code can be used as template for a TCP client.

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

public class TCPClient {
   public static void main(String args[]) throws
   UnknownHostException, IOException
   {
      int port = 7;    // puerto donde escucha el servidor
      String server = “zoltar.redes.upv.es”;
      Socket s=new Socket(server, port);

      // here you should include the code to develop the
      service,
      // that is, the dialog between Client and Server

      s.close(); //close the socket, the connection will be
      closed
   }
 }
```

Exercise 1:

Write a Java program called “TCPCLient” that connects to port 80 at server “www.upv.es”, and prints the message “Connected!” after establishing the connection and next closing the connection and ending. (it does not need to include exception handling code).

In Exercise 1, it has not taken into account the possibility of errors that prevent the establishment of the connection. This will happen if the specified host name or port number is incorrect.
To provoke a connection error it can be substituted the destination port by 81 in the creation of the socket, which does not offer any service in www.upv.es, or it can be change the hos name by wwww.upv.es, both cases will cause the program aborts due connection errors.

Exercise 2:

Write a Java program called “TCPClient2” that will do as Exercise 1 but now exception handling is required (using try … catch).

- ❖ Change the connected message to string “Connected again!”.
- ❖ Run it and check if it works too.
- ❖ Now change port number to 81 and try again. What happens?
- ❖ Restore port number to 80. Change the server name adding an additional “w” to the name. Try again to see what happens.
- ❖ When trying to connect a socket, when do UnknownHostException and IOException happen?

Comparing both errors, it can be seen that the exception generated is not the same in both cases. When invoking the Socket constructor with a host name as a parameter, if the name cannot be resolved, for example because the DNS server is down, the constructor will throw an exception of type UnknownHostException. In the second case we have tried to connect to a port, 81, in which the server is not offering any service. Also, other reasons that prevent the socket from connecting can also throw a ConnectException type exception. Among the causes that can produce this last exception are that the destination port on the server is not open, that there are routing problems in the network to reach the destination or simply that the specified server is turned off. The classes for the two exceptions mentioned are descended from the IOException class.

In real applications it is necessary to consider these possibilities to avoid unwanted behavior. The usual way to handle these exceptions is to use try / catch clauses that catch the exceptions that occur and allow you to indicate the appropriate measures.

Exercise 3:

Write a java program, called “ClientTCP3”, which adds the “try / catch” clauses necessary to detect the previous errors to the program in exercise 2. The program should display a message on the screen indicating whether the connection has been established or not. In case of success, it should show the message “Connected again!”, And in case of failure it will indicate the reason by showing the corresponding message: “Unknown server name” or “No connection possible”, depending on the type of exception detected .

To verify correct operation, repeat with this program the tests carried out in exercise 2, using the server “www.uv.es” in one case and the destination port 81 and in the other case try to access the wwww.upv server and port 80.

In many cases the input parameters of a socket instance, the server and port number to connect to are read from the command line using `args []` or entered by the user by keyboard. Notice that the input parameters in Java are, by default, `String`[4]. Since the port in the Socket constructor is an `int`, it is necessary to perform a type conversion. Thus, if args [1] is the parameter corresponding to the port, the conversion can be done, among other methods, by the instruction:

```
int port = Integer.parseInt(args[1])
```

5. Reception of data: open an input stream to the socket

The basic operation of the sockets emulates the files on disk, so reception will consist of performing "read" operations on the socket and, transmission performing "write" operations. To read the data that is being received through the socket we will use an object of the type `InputStream` (input byte stream) associated to the socket, which conforms well to the TCP philosophy of transmission oriented to continuous flow of data. To get the `InputStream` for a socket we will invoke the

[4] This default behavior can be changed, for example, by associating a Scanner to System.in and using the `nextInt []` method, as explained later.

getInputStream() method of the Socket class.

InputStream allows isolated byte or byte vector read operations using the read () method, but is not convenient for text-based protocols, making it difficult to read messages from the server. Therefore, it is more convenient to use some method to read a stream of characters and, if possible, complete lines of text.

The Scanner class of the java.util package allow us to read message from the server line by line, using the method nextLine(). Scanner also provides other methods to read, you can see them in https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/Scanner.html

Remember that you must include the line "import java.util.Scanner;" to use the Scanner class.

In our case, using the nextLine () method of the Scanner class on an input stream (InputStream), we can read the server responses as lines of text (String). For example, to print a line of text received through socket "s", you would use the following code:

```
Scanner read=new Scanner(s.getInputStream());
System.out.println(read.nextLine());
```

Exercise 4:

Write a Java program called "DayTimeClient" that connects to port 13 on "zoltar.redes.upv.es" server, reads the first line retuned by the server, and prints it to the standard output (stdout).

NOTE: Remember that in order to access zoltar you must connect through the VPN if you are outside the UPV.

The reception functions on the Scanner are blocking, that is, when nextLine() (or any other method) is used to read from the socket the program stops until any data is received.

The program will check is any data is received while the communication is still established. To facilitate this check, the Scanner hasNext () method returns true when there is something pending to be received, and false when no more data is expected - for example, because the associated socket has been closed at the other end - or hangs waiting for new data in another case. Thanks to this method it is possible to implement data reception loops to receive data until the other end closes the connection. For example, assuming read is a Scanner class variable, bound to the Input Stream of the socket, and that the other end will close the socket when data transmission ends, it is possible to display all lines received until the connection end using the following code:

```
while(read.hasNext())

    System.out.println(read.nextLine());
```

6. Data transmission: open an output stream to the socket

Following the previous model, to send data to the server, you have to "write" on the socket. To do it, we will use an object of the type `OutputStream` (output byte stream) associated to the socket. To get the OutputStream for a socket we will invoke the `getOutputStream()` method of the `Socket` class.

OutputStream allows isolated byte or byte vector write operations using the `write()` method, but is not convenient for text-based protocols, making it difficult to write messages to the server . Therefore, it is more convenient to use some method to write a stream of characters and, if possible, complete lines of text, also providing some buffering capability.

`PrintWriter` class of the `java.io.PrintWriter` package allows us to print formatted representations of objects to a text-output stream.

We find very useful, PrintWriter methods that allow us to write messages to the server line by line. These methods are:

- **`print(String)`**: sends the included line;

  e.g.

  ```
  print("Hello!");
  ```

  notice that no line separator is sent.

- **`println(String)`**: sends the included line adding a line separator to the string (i.e. line return, \r\n in Windows OS, and \n in Unix);

  e.g.

  ```
  println ("Hello!");
  ```

  a line separator is sent after string.

- **`printf(String, format)`**: writes a formatted string,

  e.g.

  ```
  printf("Hello!","\r\n");
  ```

  the string and the defined line separator.

Of course, `PrintWriter` class also provides other methods to write. You can see them in

https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/io/PrintWriter.html

Remember that you must include the line "`import java.io.PrintWriter;`" to use the `PrintWriter` class.

To declare a variable of PrintWriter class, one of the most used constructors is:

`public PrintWriter(OutputStream out, boolean autoFlush)`

It is important to note that the second parameter of the constructor must be

`true`, in order to send the data immediately to the server. Next section explains us the reason of it.

7. Line flushing (*flush*)

Although the advantages of using a buffering for writing is obvious, it can also have some drawbacks if we are not careful, as we will see in the following exercise.

Exercise 5:

Write a Java program called "EchoClient", that connects to port 7 (echo service) of the server "**zoltar.redes.upv.es**", transmits the line "Hello World!", reads the first line send by the server, and prints it to stdout.

To transmit and receive data to/from the server use `println()` and `nextLine()` methods of `PrintWriter` and `Scanner` classes.

In order to observe the behaviour of the communication according to the autoFlush parameter of the PrintWrtiter constructor, this time we will declare that parameter as `false`.

Even if your program doesn't work, keep reading ….

This echo client should send a message to the echo server in zoltar and receive its response, but it does not receive anything. Why? Because he does not send anything to the server either. To improve efficiency, the PrintWriter tries to fill its buffer as much as possible before sending the data, but as the client has no more data to send (so far) their request never is sent.

The solution to this problem is given by the `flush()` method of the `PrintWriter` class. This method forces the data to be sent, even if the buffer is not already full. In case of doubt about whether or not it is necessary to use it, it is better to use it, since to make an unnecessary flush consumes few resources, but not to use it when it is necessary can cause blockages in the program.

Exercise 6:

Modify your "EchoClient" program to use the `flush()` method and verify that it is working correctly now.

We can automatically make the line flushing when we write to the buffer (without having to invoke the flush method explicitly). To do it, we need two things:

• The constructor of the PrintWriter class must be used as shown above, with a second additional parameter (autoFlush) to `true`.

• In writing, the end of the line must be explicitly indicated, by use of the `println()` method the line separator is included.

- Another possibility to force the emptying of the buffer automatically is to use the `printf ()` method, passing a line ending with the characters `\ r \ n` as a parameter. Important!! This possibility does not work with the `print()` method.

8. Line Separator

As previously stated, `println()` method, terminates the current line by writing the line separator string, that in Linux (and systems of type Unix ass Mac OS) is defined as \n, (\n ASCII code 10). However, most of the Application protocols that sends text messages, defines as a line separator the characters \r\n (\r ASCII code13). Usually it is not a problem because servers accept line separators that doesn't completely match the defined standard. However, a client, that doesn't consider it, can find problems with servers that follow exactly the protocol. Fortunately, it is easy to define the line separator string as the application protocol specifies using the system property `line.separator`:

```
System.setProperty ("line.separator","\r\n");
```

The line separator definition above must be done before declaring the PrintWiter variable.

Exercise 7:

Write a Java program called "SMTPClient", that connects to port 25 (smtp service) of the server "**smtp.upv.es**", reads the first line send by the server, and prints it to stdout.

To develop a smtp client, add the code necessary to send to the server using println the request "HELO upv.es". Remember to use the System.setProperty statement mentioned earlier. Add the code to receive and display the following line from the server. After that it closes theconnection.

Start the Wireshark and use "port 25" as capture filter (SMTP protocol). Run your java SMTPClient program and analyse the captured packets (if no response is received, consider whether emptying buffer has been taken into account). What line separator characters are transmitted over the network?

To check the influence of the line separator, comment on your program the System.setProperty line code and run your program again. Capture the traffic with wireshark. What line separator characters are transmitted over the network? Are they the same as before?

Important! If you use the BlueJ programming environment, once modified the separator remains changed until you exit the environment (even if you comment the code line in the program) or modify it again using the corresponding instruction.

9. How to get information about the established connection

The Socket class has several methods to obtain information about the established connection between the client and the server:

• **`public int getPort()`:** returns the remote port to which the socket is connected. Match the port provided when building the socket.

• **`public InetAddress getInetAddress()`**: returns the remote IP address to which the socket is connected. Corresponds to the server provided (by name or in the form of InetAddress) as a parameter in the socket constructor.

• **`public int getLocalPort()`:** returns the local port to which the socket is attached. In clients, it is usually selected by the operating system. Although there are Socket class constructors that allow you to set the local port, they are rarely used.

• **`public InetAddress getLocalAddress()`:** Returns the local IP address to which the socket is bound. A host has at least one IP address, but can have several if you have multiple network adapters (real cards, USB and WiFi adapters, or even virtual adapters) of which the operating system will usually select one. There are also constructors that allow you to specify it.

Exercise 8:

Modify your exercise 3 program "TCPClient3" to show information about the established connection (local and remote IP addresses and port numbers). Run it four times in a row, connecting to the server www.upv.es on port 80 and check which values are modified. What has and has not changed? Why?

•

10. Another real example

As a synthesis of all these concepts, we will develop a basic HTTP 1.0 client

Write a Java program called "HTTPClient", that connects to port 80 (World Wide Web service) of the server "**www.upv.es**", transmits the request **`"GET / HTTP/1.0\r\n\r\n"`** and reads the lines send by the server, and prints them to standard output (stdout), until the connection is closed.