Lab 5: Iterative TCP Servers

The Lab-5 aim is to learn how to program iterative TCP servers. An iterative server only can provide its service to one client at time. On the other hand, a concurrent server can provide its service to more than one client at time. The development of concurrent server is more complex and will be learn in later labs.

Lab-5 aims

At the end of the lab-5 you should be able to program a basic iterative TCP server in Java. In particular you will be able to:

• Use the `ServerSocket` class to program an iterative TCP server that listens on a port and establish a TCP connection with a client.

• Differentiate the characteristics of the sockets created by a client, using `Socket` class, and the sockets created by a server, using `ServerSocket` class.

• Send and receive information of text type through a TCP connected socket.

1. Java Classes to create TCP Servers

In Lab-5, we will transfer messages from the application layer using the TCP transport layer protocol. Therefore, in this section we will see the basic classes for communicating a client and a server using TCP sockets.

At least we need two classes belonging to the java.net.* package, the `Socket` and `ServerSocket` classes:

1. `Socket` class allows you to establish a TCP connection between a client and a server. The client creates a socket and initiates the connection to the server. Once they are connected they can send and receive information using the connected socket.
2. `ServerSocket` class creates a server socket, bound to the specified port for waiting a TCP connection request. When the TCP connection request is accepted, a new socket (a new instance of `Socket` class) is created, and the TCP connection is established. This new connected socket is used to exchange information between both ends (client and server).

Servers wait for a client connection request using the method `accept()` of the `ServerSocket` class. The `accept()`method blocks the execution of the program while waiting for a connection request from a client. Once a connection request is received, `accept()`creates a new socket (an instance of the `Socket` class). This new socket will be used during the rest of the communication with that client.

The use of the `accept()` method, or the creation of a `ServerSocket` object can generate an `IOException` exception, so either it will have to be captured by a `try-catch` clause, as done previous labs, or will be threw to the program or function that called the method that generated the exception. If we use this last option, we will not require programming a try clause. In our case, since it is the Java virtual machine which called the main method, this exception will be thrown at it.

The skeleton of an iterative TCP server could be as follow:

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

class TCPServer {
public static void main(String args[]) {
   try{
         ServerSocket ss=new ServerSocket(port);
         while(true){
               Socket s=ss.accept(); // wait for a client request
               //code to provide the service to the client
               s.close();
         }
        }
   catch(IOException e) { System.out.println(e); }
   }
  }
```

The `while(true)` block allows the server to connect to new clients, once it has finished giving the service to the client in progress. Remember to close the connected socket (`s.close()`) at the end of the loop.

2. Input / Output Management

Once the server has been connected to a client, in order to handle the transfer of information through the newly created socket, the `Socket` class has two methods: `getInputStream ()` and `getOutputStream ()`, which provides an input and output stream, respectively. As we saw in previous Labs, it is more convenient to exchange text lines using `Scanner` or `PrintWriter` classes, than to handle these flows directly.

The code below is a brief example of using these classes and some of their methods:

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

...
Scanner receive=new Scanner(client.getInputStream());
receive.nextLine();
...

PrintWriter send=new PrintWriter(client.getOutputStream());
send.printf("message to send");
send.flush();
```

being `client` the connected socket to the client.

Exercise 1:

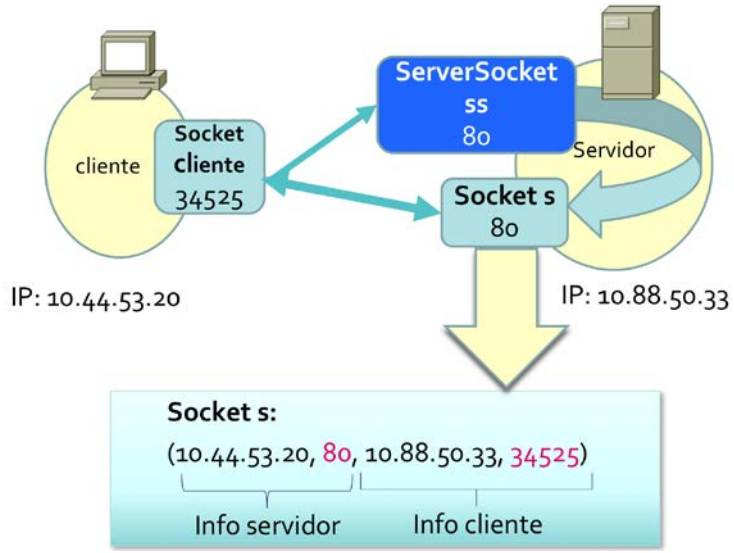
Write an Echo server that listens on port 7777. After accepting the client connection the server:

- returns the first text line received through the socket,
- prints to the screen the message: "A client has been connected to the server", and
- closes the socket connected to the client.

You can test your Echo server with the command "`nc localhost 7777`".

3. Differences between the two types of used sockets

There is an important difference between the two types of sockets that we are using in Lab-5. While the initial socket, generated with the `ServerSocket` class, is an unconnected socket, the socket that is generated as a result of the `accept()` method (an instance of Socket class) is connected to the client. Both sockets (ServerSocket and connected sockets) use the same local IP address and the same local port. However, the connected socket to the client will also have the remote values associated with the IP address and port of the client.



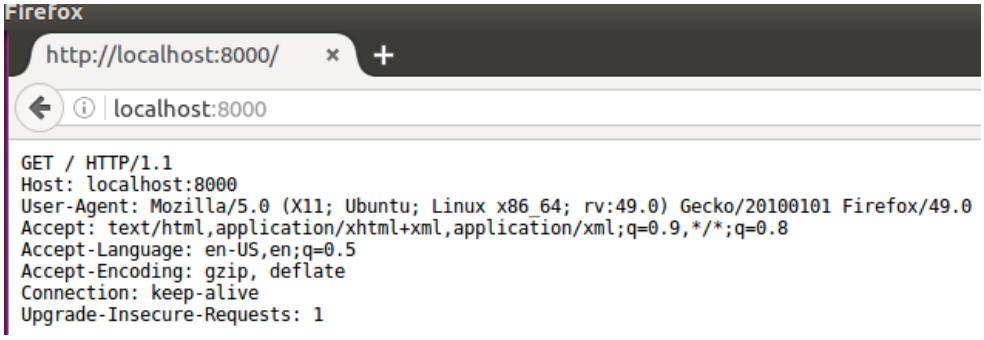
Remember that the Socket class gives us the `getLocalAddress()`, `getLocalPort()`, `getInetAddress()`, and `getPort()` methods that allow us to get those 4 values associated with the `Socket` and to identify the established TCP connection. On the other hand, the `ServerSocket` class offers only two of these methods, those associated with local values: `getInetAddress()`, `getLocalPort()`. Notice that the `getInetAddress()` method has different meaning depending on whether it is `Socket` or `ServerSocket` objects.

Exercise 2:

Modify the server of exercise 1 to display on screen the IP address and port values used by the program sockets, indicating whether they correspond to the object of type `ServerSocket` or `Socket`.

An exercise that can be useful is to program a server that returns the control information that a browser sends when it connects to the server. This server will allow us to see the initial line and headers sent by different browsers. Notice that this server, although it will allow us to see the HTTP request made by a browser, is NOT a web server.

In the figure below you can see the result of the client execution:



Exercise 3:

Write a HTTP headers mirror server. The server will listen on port 8000. Once it receives the request from the browser, the server will return to the client a "web page" containing the request line and the headers that the client sent to it in its request. That is, the control information sent by the client is the content that the server will return as the body of your message. Therefore, the server response will be as follows:

Response	**Comments**
`HTTP/1.0 200 OK \r\n`	1st line, status line
`Content-Type: text/plain \r\n`	2nd line, response header
`\r\n`	Blank line
1st line of the answer body`\r\n`	The answer body will contain all the lines of the original request sent from the browser.
2nd line of the answer body `\r\n`	
3rd line of the answer body `\r\n`	

…

After sending the response, the server will close the connection and will wait for the arrival of a new client request.

To check how your server works, open a browser and enter as URL `http://localhost:8000`.

Why a blank line should be included? When your server works correctly you can check what happens if you delete the blank line in the response to the browser.