UNIVERSIDADE DA CORUÑA

COMPUTER NETWORKS LAB
TUTORIAL: SOCKETS PROGRAMMING IN JAVA

Table of Contents

1. Introduction......3
2. Basic Concepts......3
   2.1. Integrated Development Environment (IDE)......3
   2.2. Data Streams......3
   2.3. Threads......4
   2.4. Exercise 1: Implementing Copy......4
   2.5. Exercise 2: Implementing Info......4
3. TCP/IP Protocol Stack......5
   3.1. TCP......5
   3.2. UDP......5
4. Sockets......6
   4.1. Using Sockets in Java......6
   4.2. Recommended Reading......6
5. UDP Sockets......7
   5.1. Exercise 3: Implementation of the UDP Echo Server......8
6. TCP Sockets......8
   6.1. Exercise 4: Implementation of the TCP Echo Server......10
7. EVALUATION......12

1. Introduction

The Computer Networks laboratory will consist of a set of **non-mandatory** practice assignments. All of them must be done **individually**.

The tasks to be performed in each practice work will be explained in a separate document including complementary information about each work. For those assignments requiring programming, **Java** will be used. For any questions about this language, it is recommended to consult the Java API documentation. For the last Java version, the documentation is available at https://docs.oracle.com/en/java/javase/19/docs/api/.

Each student will need to upload the source code files of her/his works to her/his git repository, accessible from a URL with the following structure: **https://github.com/GEI-Red-614G010172223/java-labs-<user-login>**. This repository is created automatically after having clicked on the following URL: https://classroom.github.com/a/lrkzFe6T. **It is mandatory to verify the UDC email** (https://github.com/settings/emails) **for the Github account.**

This tutorial provides an introduction to programming with sockets in Java and represents the first programming practice work of this lab (p0, from now on). Students will use the repository mentioned as base code. In particular, files contained in the **es.udc.redes.tutorial** package must be modified. Moreover, the project includes a file called **README.md**, which explains the main steps to configure the environment. **This tutorial will be evaluated in the first practical exam.**

Programs should be executed from the IDE IntelliJ IDEA.

2. Basic Concepts

There are several basic concepts.

2.1. Integrated Development Environment (IDE)

It is mandatory to use an Integrated Development Environment (IDE), IntelliJ IDEA. Moreover, it is advisable going through the following simple tutorials to become familiar with this IDE: https://www.jetbrains.com/help/idea/discover-intellij-idea.html. The objective is learning how to create, compile, run and debug a Java project.

We will use **git** as version control system and also as a way to deliver the code for evaluation. Concretely, students will follow the instructions in the README.md file available in the repository, and they will use the files contained in the git-help folder of the respository and the Github guide available at https://guides.github.com/activities/hello-world/ for more information.

For backup purposes, it is recommended uploading to Github also intermediate versions of the code, and not only the final version. Moreover, local commits should also be frequently done.

2.2. Data Streams

Software programs may need to read and/or write information from/to files and from/to the network. This information can be of any type (text, images, sound, objects…). Data streams are a programming construct that can be used for this type of read/write operations.

Java documentation includes a complete tutorial on how to use data streams (http://docs.oracle.com/javase/tutorial/essential/io/index.html). You should at least go through the following sections:

- http://docs.oracle.com/javase/tutorial/essential/io/bytestreams.html
- http://docs.oracle.com/javase/tutorial/essential/io/charstreams.html
- http://docs.oracle.com/javase/tutorial/essential/io/buffers.html

2.3. Threads

Software programs usually consist of one single execution thread that sequentially goes through the program instructions one by one. By using concurrent programming techniques, it is also possible to execute several parts of the same program concurrently.

There are two approaches to execute tasks concurrently: processes and threads. Threads within a process share resources (such as memory), while processes do not (this means that objects created in a process are not visible from other processes). Both approaches can be combined: a program may have several processes, and each one of them may have multiple threads.

Java supports multi-thread programs. Basic Java applications typically consist of a single process with a single thread, which calls the "main" method of the class being executed. For additional information, check:

The Java Tutorials. Lesson: Concurrency.

http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

We will use threads when developing an echo server using the TCP protocol.

2.4. Exercise 1: Implementing Copy

Using IntelliJ IDEA, modify the Java file **es.udc.redes.tutorial.copy.Copy.java** contained in **java-labs** repository in order to implement an application for copying a file (with either text or binary content) in the local disk using data streams. The application will receive two input parameters as shown below:

```
java    es.udc.redes.tutorial.copy.Copy    <source    file>
<destination file>
```

2.5. Exercise 2: Implementing Info

Using IntelliJ IDEA, modify the Java file **es.udc.redes.tutorial.info.Info.java** contained in **java-labs** repository in order to implement an application for obtaining the main properties of a file: size, last modification date, name, extension, file type (image, text, directory, unknown), absolute path. The application will receive as input parameter the relative path of a file with respect to p0-files, as shown below:

```
java es.udc.redes.tutorial.info.Info <relative path>
```

For additional information, check the following link:

https://docs.oracle.com/javase/tutorial/essential/io/fileio.html

3. TCP/IP Protocol Stack

The TCP/IP protocol stack enables data transmission in computer networks. The stack is composed of the layers shown below:

Application (http, ftp, ssh, telnet…)
Transport (tcp. udp)
Network (ip, icmp, arp)
Link (ethernet)
Physical

When writing Java programs, we will typically work at the application level, using the protocols of the transport layer. That is why we need to review the main differences between the two main protocols used at the transport layer: TCP (Transmission Control Protocol) and UDP (User Datagram Protocol).

3.1. TCP

- It is connection-oriented
- It provides a reliable data stream between two computers. It guarantees data is received correctly at the other end, and that bytes are received in order. It also implements flow and congestion control.
- Some application-level protocols using TCP are: telnet, HTTP, FTP, SMTP...

3.2. UDP

- It is **not** connection-oriented
- Sends individual data packets (datagrams), without any delivery guarantees
- It allows broadcast and multicast communications
- Some application-level protocols using UDP are: DNS, TFTP...

4. Sockets

What do we need to communicate two processes running on two computers connected through a computer network? First, we need a way of uniquely identifying each process.

NOTE: We will use the term ‘server’ to refer to a process offering a service through the network. We will call clients to the processes consuming such service.

To identify a process in a computer network running on TCP/IP, we need first to identify the computer running it. All computers in a TCP/IP network have a unique IP address to identify them (IPv4 addresses are 32 bits long, IPv6 addresses are 128 bits long). But since we may have several processes using the network in the same computer, we need something else to uniquely identify each of these processes. For doing this, TCP/IP associates each process using the network to a **port number**.

Therefore, every server process needs to register to a port. Client processes will direct their requests to the IP address of the computer running the server, and to the port number associated to the server.

Port numbers:

- are independent for TCP and UDP
- are 16 bits long (therefore, their range goes from 0 to 65535)
- may be “reserved” in the case of ports used for frequently used servers such as HTTP, FTP, etc that should not be used by user applications. Reserved port numbers are in the range 0-1023.

A **socket** is defined as a bidirectional communication endpoint between two processes which communicate through a TCP/IP network.

- A socket is identified by an IP address and a port number
- Both TCP and UDP use sockets

4.1. Using Sockets in Java

Java includes the `java.net` package to write programs using sockets. Both TCP and UDP sockets are supported. We will use this library in all the remaining exercises of this tutorial, so you will need to import it in the code of all of them.

4.2. Recommended Reading

The Java Tutorials. Lesson: All About Sockets.

http://docs.oracle.com/javase/tutorial/networking/sockets/index.html

5. UDP Sockets

UDP sockets are **not** connection-oriented. Every data packet sent will be delivered independently. There is not any delivery guarantee and packets may arrive unordered to the destination. This type of data packets are usually called **datagrams**.

We will start implementing an UDP echo client. The format to invoke the client should be:

```
java  es.udc.redes.tutorial.udp.client.UdpClient  <server_
address>  <server_port> <message>
```

Where,

- `server_address` is the name or IP address of the computer running the echo UDP server.
- `server_port` is the port number where the echo UDP server is listening for requests.
- `message` is the text to send.

The source code of the UDP echo client has been implemented in the **es.udc.redes.tutorial.udp.client.UdpClient.java** file.

These are the steps followed by the UDP client:

1. It creates a non connection-oriented socket. It is important to notice that we do not need to specify a port number when invoking the constructor, since it will automatically choose a port among the ones currently available (this is typically called choosing an "ephemeral port"). See the Javadoc documentation of the `DatagramSocket` class (especially the section about the constructors) for more information.
2. It sets a maximum timeout for the socket. If nothing is received before the timeout expires, the corresponding exception will be thrown.
3. It obtains the IP address of the computer running the server from the first argument received from the command line. The `InetAddress` class represents the concept of an IP address in Java. This class includes the method `getByName()`, which obtains the IP address from the `String` value received as parameter. This value may be either a hostname or an IP address.
4. It obtains the port number where the server is running from the second argument received from the command line.
5. It obtains the message from the third argument received from the command line. If the message has several words, we need to use double quotes (e.g. "This is a test").
6. It prepares the datagram to be sent by indicating: the message (as a byte array), number of bytes to send, IP address of the server and server port. See the documentation of the class `DatagramPacket` for details.
7. Sends the datagram by invoking the `send()` method of the socket created in the first step.
8. Prepares the datagram to receive the answer from the server. For this, we first need to create a byte array to store the response. To create the datagram, we need to indicate the array where the response will be stored and the maximum number of bytes that can be stored in this array.

9. Receives the datagram by using the `receive()` method of the socket. This call is blocking, meaning that the program will wait at this point until it either receives a response from the server, or the timeout expires.
10. Finally, it closes the socket.

15.1. Exercise 3: Implementation of the UDP Echo Server

The goal for this exercise is implementing a UDP echo server able to respond the requests sent by the client shown in the previous section, modifying the **es.udc.redes.tutorial.udp.server.UdpServer.java** file provided.

The server will implement the following steps:

1 Create a `DatagramSocket` associated to a specific port number.

2 Set a maximum timeout for the socket.

3 Create an infinite loop executing the following steps:

  3.1 Prepare a datagram to receive the message from the client. It is recommended to create a new `DatagramPacket` object for each message

  3.2 Receive a message

  3.3 Prepare the datagram to answer to the client. Remind that the received datagram contains the IP address and port number for the client.

  3.4 Send the response message

Once the UDP echo server is implemented, you can check it is working correctly by executing it from the IDE:

```
java es.udc.redes.tutorial.udp.server.UdpServer 5000
```

The server will wait for incoming requests (if the timeout expires without receiving any request, the server will end execution).

Now, from the IDE, run the UDP client:

```
java es.udc.redes.tutorial.udp.client.UdpClient localhost 5000
"Testing my UDP server"
```

What should happen is: 1) the server receives the message from the client, 2) responds to the client with the same message, 3) the client receives the server response:

Messages from server terminal:

```
SERVER: Received Testing my UDP server from /127.0.0.1:35286

SERVER: Sending Testing my UDP server to /127.0.0.1:35286
```

Messages from client terminal:

```
CLIENT: Sending Testing my UDP server to
localhost/127.0.0.1:5000

CLIENT: Received Testing my UDP server from /127.0.0.1:5000
```

6. TCP Sockets

TCP sockets are connection-oriented and provide reliable communication. This implies that before sending and receiving data, it is needed to establish a connection between the client and the server. Once the connection is established, the TCP protocol guarantees data is received without errors and in the same order as it was sent.

First, we provide you a TCP echo client already implemented. The format to invoke the client will be:

```
java        es.udc.redes.tutorial.tcp.client.TcpClient
<server_address>    <server_port> <message>
```

Where,

- `server_address` is the name (or IP address) of the host where the TCP echo server is running.
- `server_port` is the port number where the TCP echo server is listening.
- `message` is the message to send.

Now, we show the source code for the TCP echo client:

These are the steps executed in the client:

1. It obtains the IP address of the host running the server from the first argument received from the command line.
2. It obtains the port number where the server is running from the second argument received from the command line.
3. It obtains the message from the third argument received from the command line. If the message has several words, we need to use double quotes (e.g. “This is a test”)
4. It creates a connection-oriented socket. No port number is specified for the client since an available port will be automatically chosen (“ephemeral port”). Therefore, the constructor invocation only specifies the server IP address and the server port. The connection is established immediately after the socket creation. You can check the Javadoc documentation of the `Socket` class, specially the section about constructors.
5. It sets a maximum timeout for the socket. If nothing is received before the timeout expires, the corresponding exception will be thrown.
6. It creates the input channel (input stream) to receive data from the server. To read the data from the server, it uses the `BufferedReader` class, which is obtained through the `getInputStream()` method of the client socket. This class includes the `readLine()` method to read data one line at a time.
7. It creates the output channel (output stream) for sending data to the server. It uses the class `PrintWriter`, which is obtained from the `getOutputStream()` method of the client socket. This class includes the methods `println()` and `print()`, which are similar to the ones commonly used to print data in the screen.
8. Sends the message to the server by invoking the `println()` method of the output stream.
9. Waits for the server response and reads it when it arrives, by invoking the `readLine()` method of the input stream.
10. Finally, it closes the socket.

6.1. Exercise 4: Implementation of the TCP Echo Server

The goal for this exercise is implementing a multithread TCP echo server **(es.udc.redes.tutorial.tcp.server.TcpServer.java)**. TCP servers are usually multithread to allow processing multiple connections simultaneously. To make the exercise easier, first a monothread version of the server will be implemented to check if it works correctly with the TCP echo client. Once the monothread version is working, it can be modified to include multithread support.

The monothread server implementation should follow the next steps:

1 Create a `ServerSocket`, associated to a specific port number.

2 Set a maximum timeout for the socket.

3 Create an infinite loop executing the following steps:

  3.1 Invoke the `accept()` method of the server socket. This method waits until a new connection request is received from a client. Once the connection is established, it returns a new socket that will be used for communicating with that particular client.

  3.2 Set the input and output streams for the new socket.

  3.3 Receive the message from the client.

  3.4 Send the message back to the client.

  3.5 Close the streams and the connection associated to the socket created in the `accept()` method.

Once the monothread server is implemented, you can check it works as expected by running in the IDE:

```
java es.udc.redes.tutorial.tcp.server.MonoThreadTcpServer 5000
```

The server will wait for incoming requests (if the timeout expires without receiving any request, the server will end execution).

Now, from the IDE, run the TCP client:

```
java es.udc.redes.tutorial.tcp.client.TcpClient localhost 5000
"Testing my TCP server"
```

What should happen is: 1) the server receives the message from the client, 2) responds to the client with the same message, 3) the client receives the server response:

Messages in the server terminal

```
SERVER: Received Testing my TCP server from /127.0.0.1:36725

SERVER: Sending Testing my TCP server to /127.0.0.1:36725
```

Messages in the client terminal

```
CLIENT: Sending Testing my TCP server to
localhost/127.0.0.1:5000

CLIENT: Received Testing my TCP server from /127.0.0.1:5000
```

To transform the server from monothread to multithread, you need to create a new class (`ServerThread`) which extends the `Thread` class. This class will be in charge of processing each connection request. Now, the steps that need to be performed by the TCP server are:

1 Create a `ServerSocket`, associated to a port number.

2 Set a maximum timeout for the server socket.

3 Create an infinite loop executing the following steps:

3.1 Invoke the `accept()` method of the server socket. When a connection from a client is established, this method returns a new socket which will be used for communication with that particular client.

3.2 Create a new `ServerThread` object passing the new socket as parameter. This way the connection with that client will be processed in an independent thread using the new socket, while the server socket is ready to receive new client connections.

3.3 Start the execution of the thread using the `start()` method of `ServerThread`. Important Note: if you invoke the method `run()` instead of `start()`, execution will be sequential instead of multithread.

The steps that need to be perfomed by the `ServerThread` class are (these steps need to be implemented in the `run()` method):

1 Prepare the input and output streams.

2 Read the message from the client.

3 Send the message back to the client.

4 Close the streams and the connection of the socket obtained with the `accept()` method.

Once the multithread server is implemented, you can check it works as expected by running in the IDE:

```
java es.udc.redes.tutorial.tcp.server.TcpServer 5000
```

The server will wait for incoming requests (if the timeout expires without receiving any request, the server will end execution).

Now, in other terminal, run a `nc` command to connect with the server, and leave the connection open:

```
nc localhost 5000
```

Now, in the IDE, run the TCP client:

```
java es.udc.redes.tutorial.tcp.client.TcpClient localhost 5000
"Testing my TCP server"
```

In this case, the server will start a thread to attend the connection initiated by the `nc` command. Since the server is multithread, it will start a different thread to attend the request initiated by the TCP echo client. What should happen is: 1) the server receives the message from the client, 2) responds to the client with the same message, 3) the client receives the server response, just like in the monothread TCP server case. Finally, to finalize the `nc` connection you can simply type something and press ENTER: that will send a message to the server, which will respond with the echo and will close the connection.

7. EVALUATION

The student will upload the code implemented to her/his github repository: https://github.com/GEI-Red-614G010172122/java-labs-<user-login>.

Once the last changes have been done, the student should tag the last commit in order to identify the final version and then the tag p0 should be published. The git commands to use are:

1. `git tag -a p0 -m “p0”`
2. `git push origin p0`

This tutorial will be evaluated in the first practical exam, which has an overall weight of up to 0,75 points in the final qualification of this subject. This exam will contain questions about the programming assignments (p0 and p1).