LAB1-C

OPERATING SYSTEMS-DR. LÉONARD JANER

2021-2022

JANUARY 2022

TecnoCampus
Escola Superior
Politècnica

Centre adscrit a la

Universitat
Pompeu Fabra
Barcelona

LAB1-C

Objectives

Test your learnings on socket communications on a LINUX environment.

Background / Scenario

In this lab, you will work with sockets to share informacion between different programs/processes/applications

Required Resources

- DEVASC VM, properly configured

Instructions

Part 1: **QUOTE OF THE DAY**

The current LAB will work based on the **Quote of the Day**, **Request for Comments**. These documents managed by the **Internet Engineering Task Force** are the way to explain, document and manage protocols used over Internet.

The RFC 865 (https://datatracker.ietf.org/doc/html/rfc865) defines the **Quote of the Day Protocol**.

Read carefully this document to understand the protocol. It is necessary for the development of the lab.

Part 2: **QUOTE OF THE DAY CONNECTION ORIENTED CLIENT**

Implement a program named **part2.c** to implement a QOTD client using TCP protocol based on the information from RFC. The client will ask to a QOTD connection oriented server to send a QOTD message (according to the RFC), will receive the message from the server, and then will print on the terminal screen the message, and will finish the execution.

Some details for the implementation:

- The IP address of the QOTD server must be an input argument of the program through argc/argv parameters (the calling syntax for the program must be "**./part2 IP"** where the **IP** would be the IPv4 address of the server).
- The port of the server could be a second input parameter, then the calling syntax would be **"./part2 IP PORT"** where the **PORT** would be the TCP PORT of the server. If this input parameter is not provided the program will use the default port number 17 for the server.
- The program has to check the number of input parameters are the expected. In case of error the pertinent error message must be written on terminal screen and the program execution must finish.
- The provided IP address for the server must be checked before using it. If the provided IP address is not correct (format) then the pertinent error message must be written on terminal screen and the program execution must finish.
- You have to set the maximum number of characters the received message from the server could be according to RFC.
- You have to identify available QOTD servers to test your client. There are different sites on Internet with QOTD server implemented.

Part 3: **QUOTE OF THE DAY NON-CONNECTION ORIENTED CLIENT**

Implement a program named **part3.c** to implement a QOTD client using UDP protocol based on the information from RFC. The client will ask to a QOTD non-connection oriented server to send a QOTD message (according to the RFC), will receive the message from the server, and then will print on the terminal screen the message, and will finish the execution.

Some details for the implementation:

- The IP address of the QOTD server must be an input argument of the program through argc/argv parameters (the calling syntax for the program must be "**./part3 IP"** where the **IP** would be the IPv4 address of the server).
- The port of the server could be a second input parameter, then the calling syntax would be **"./part3 IP PORT"** where the **PORT** would be the UDP PORT of the server. If this input parameter is not provided the program will use the default port number 17 for the server.
- The program has to check the number of input parameters are the expected. In case of error the pertinent error message must be written on terminal screen and the program execution must finish.
- The provided IP address for the server must be checked before using it. If the provided IP address is not correct (format) then the pertinent error message must be written on terminal screen and the program execution must finish.
- You have to set the maximum number of characters the received message from the server could be according to RFC.
- You have to identify available QOTD servers to test your client. There are different sites on Internet with QOTD server implemented.

Part 4: **QUOTE OF THE DAY SERVER IMPLEMENTATION STEP 1**

This will be the first step to implement a QOTD server. The program named **part4.c** has to process the information form a QOTD file, and process some information from the file:

- The file (an example is provided on the repository named **quotes.txt**) has the following structure:
  - The maximum number of messages in a file will be **1024**.
  - The maximum number of characters per message in a file will be **512**.
  - A QOTD message is written on two consecutive lines. The first one with the sentence and the second one with the author of the sentence.
  - The consecutive messages are separated on the file, with a line that has only the "**%**" character. That character could not be used anywhere apart from the circumstance to split messages.
- The name of the file with the quotes must be an input argument of the program through argc/argv parameters (the calling syntax for the program must be "**./part4 filename"** where the **filename** would be the file name with all the quotes).
- The program has to check the number of input parameters are the expected. In case of error the pertinent error message must be written on terminal screen and the program execution must finish.
- The program must write on the terminal screen for each message: the number of the message (a sequential value from 1 to the total number of messages), the sentence, the author, and the number of characters of the sentence. The number of characters of the sentence must be the shown characters. Be careful with **Line Feed** and **Carriage Return** characters that must not be considered (not computed).
- Once all the sentences have been shown, the program must write a final message with the **total number of messages** and the **total number of characters** (for all the messages).

Part 5: **QUOTE OF THE DAY SERVER IMPLEMENTATION STEP 2**

This will be the second step to implement a QOTD server. The program named **part5.c** has to implement a TCP connection oriented QOTD server:

- The port where the server will be running must be an input parameter of the program through argc/argv parameters. The calling syntax of the program must be "**./part5 PORT"** where **PORT** would be the TCP port of the server. If this input parameter is not provided the program will use the default port number 17 for the server.
- The program has to check the number of input parameters are the expected. In case of error the pertinent error message must be written on terminal screen and the program execution must finish.
- Once a connection requested by a client has been accepted the server should send a message with all the ASCII characters from letter '**a'** to letter '**z**', and the author of the message must be the full name of the members of the group (student names).
- The server must be waiting forever for client communications and must finish the execution only when the process is stopped (by keyboard with the CTRL+C keys or using the **kill** tool with the pertinent **PID**).

You have to test your server by two ways:

- Using the previously implemented client.
- Using **netcat** tool.

Part 6: **QUOTE OF THE DAY SERVER IMPLEMENTATION FINAL IMPLEMENTATION**

This will be the final implementation of a QOTD server. The program named **part6.c** has to us information from PART4 and PART5 programs:

- The program named **part6.c** has to implement a TCP connection oriented QOTD server
- The program has to process information from an input file (an example is provided on the repository named **quotes.txt**) with the structure defined and explained previously.
- The name of the file with the quotes must be an input argument of the program through argc/argv parameters (the calling syntax for the program must be "**./part6 filename"** where the **filename** would be the file name with all the quotes).
- The port where the server will be running could be an input parameter of the program through argc/argv parameters. The calling syntax of the program must be "**./part6 filename PORT"** where **PORT** would be the TCP port of the server. If the port number is not provided the program must use the default port number **17**.
- The server must first of all read and store the information from the quotes file on memory.
- The server must then prepared the socket to received client requests.
- Once a connection request has been accepted, the server must generate a random number (according to the number of read messages from the quotes file), and send to the client that message from the file. Remember that a seed must be used to guarantee that the random numbers are different from execution to execution of your program.
- A log file named **last.log** must be written. The file must be persistent from execution to execution. For each new connection, the server must log onto this file, a text line with the following information:
    - **TIMESTAMP: CLIENT-IP: CLIENT-PORT**
    - Where **TIMESTAMP** must be a temporal information of the connection request.
    - Where **CLIENT-IP** must be the IP of the client (in a text format, not a integer value) (something like A.B.C.D).
    - Where **CLIENT-PORT** must be the PORT of the client in a text format.
- The server must be waiting forever for client communications and must finish the execution only when the process is stopped (by keyboard with the CTRL+C keys or using the **kill** tool with the pertinent **PID**).

Part 7: **GENERAL RECOMMENDATIONS FOR ALL THE PROGRAMS**

Some recommendations:

- Do not use **magic numbers** on your code. Every numerical and/or static value on your code must be defined using the convenient **#define** directive.
- It is a good practice to initialize (always) all the variables, with a default value (event vectors, arrays, …) before you use them. One solution could be to use **memset** when convenient.
- Remember the endianness for your Operating System and TCP/IP could be different, and you can not presuppose the type of endianness of the operating system/microprocessor where your program will be executed. So you must use the convenient functions **htonl, htons, ntohl, ntohs**.
- Once your program / process is finished you must freeze the previously allocated resources. If you are using dynamic memory allocation on your program, the **free** system call must be used before exiting your program.

Part 8: **HOW TO ISSUE THE SOLUTION**

For every PART, you have to provide:

- Your code on the repository.
- On the report you must explain and issue screenshots showing the compilation process to obtain the executable files. If you are using make tool for that process the MAKEFILE must be provided. If the compilation process is not well documented, the program could not be tested.
- Screenshots and explanation of the compiling process, on the report section. The code must be explained on the report. It is not enough to have a code with comments embedded, you have to explain the main structure of the solution, the functions, the parameters. It is as much important the way you write documentation than the way you code.
- Screenshots and explanation of the execution process (more than one example, just as a kind of proof), on the report section.
- Comments on the relevant parts of the code. What you considered must be explained, on the report section.
- On the repository you have to have only one directory. All the files, programs, everything must be on the same directory.