



Tecnocampus
Escola Superior
Politécnica

Centre adscrit a:



Universitat
Pompeu Fabra
Barcelona

Operating Systems

PRAC 1:

Dr. Léonard Janer

Degree in Computer Engineering for Management Information Systems

Escola Superior Politècnica del TecnoCampus

COURSE 2022-2023

Introduction

Presentation

This first PRAC consists of three activities focused on how to use the libraries of the GNU/Linux operating system to:

- Create *sockets* to offer or use services over a network based on the TCP/IP protocol, such as the Internet.
- Read and write files, either in textual or binary format, to store and retrieve data persistently.

Responsible Lecturers

- Dr. Léonard Janer <leonard@tecnocampus.com>

Competencies

The specific and transversal competences that are worked on in this PRAC are those described below:

- Specific:
 - Ability to analyze a problem at the appropriate level of abstraction for each situation and apply the skills and knowledge acquired to address and solve it.
- Transversals:
 - Capacity for written communication in the academic and professional field.
 - Ability to communicate in a foreign language.
 - Ability to adapt to new technologies and environments.

Objectives

The objectives of this PRAC are:

- Learn to develop client-server applications using sockets using the C programming language and the libraries of the GNU/Linux operating system.
- Learn to develop applications that can read and write files using the C programming language and the libraries of the GNU/Linux operating system.

Resources

The following resources are available for the correct development of this PRAC:

- Basics:
 - Subject materials available on the Virtual Campus
- Complementary:
 - Devasc VM (provided on eCampus)

Evaluation criteria

The weight of each question is indicated in the statement of each exercise and/or in the rubric of the activity on campus. The clarity and justification of the answers presented will be valued.

Format and delivery date

A PDF document must be submitted with the answer to the questions clearly written, as well as **all** the programmed source files. The delivery will be made in the GIT repository that will be accepted to perform the PRAC.

The PDF report document must have a cover, with a name, title, date, an index, a section for each activity and some bibliographical references.

Screenshots should be included, with: examples how to compile all the codes, how to execute all the codes, and execution examples to show the correct features of the solution proposed. The relevant elements of the code should be explained. It is not enough to

include the code, if it is not explained in the report: you have to write a code, and write a report. The report is not just the code; the report must be a document to explain the code, with the structure of the program explained and screenshots.

Activities

To carry out this PRAC, you are asked to read and respond to the activities that are presented below, taking into account the following aspects.

Regarding the implementation of the different activities, it is requested to take into account:

- The folders with the files of each activity and a document in PDF format with the response to the activities must be delivered to the repository, which will be the explanatory and demonstrative report of each activity
- In the repository the activity source files must be inside a folder named **activity_x**, where *x* is the number of the activity, and each folder must contain the source files (`.c` and `.h`), as well as a **Makefile** file that allows you to compile the activity code using the parameterless **make** command (if used).
- In the PDF document it is necessary to include for each activity a description of the operation of the code, the process of compilation and execution of the program, and a description of the result obtained (including screenshots). You have to show (clearly, with screenshots) the procedure to obtain the executables of each activity, and you have to show (clearly, with screenshots) the procedure for executing the code. In fact, for the executions you must attach screenshots of the executions where you have to show the correct operation of the proposed solution.

Regarding the implementation of the programs that make up each activity, it is requested to take into account the following good practice tips:

- Be careful not to use *magic numbers*, so all but trivial numerical values that appear in your code should be put in a **#define** directive.
- It is recommended to initialize all variables with a default value, including vectors. In this case, you can use the **memset** function.

- To convert operating system numeric values to network numeric values you have to use the `htons` and `htonl` functions. In the reverse direction you have to use the `ntohs` and `ntohl` functions.
- At the end of the execution it is necessary to release the resources used. Specifically, it is necessary to free the file descriptors associated with the communication *socket* and the log file.
- In case of using dynamic memory it is also important to free any reservations that have been made using the call to `malloc`. This requires the call to `free`.

The practical score will be based on the campus rubric.

As can be seen, both the code and the comments are valued to understand the proposed solution, and the screenshots that show how it is compiled, and the execution of the code, where you have to show the management of input parameters and the correct operation of the code.

Activity 1

A client-server application must be generated. The server will take care of generating a random number (between 0-100) that the client must guess. The client will generate a random number in the center of the range of possibilities (50) and will send the value to the server, which will have to tell the client if the value is correct, or if the value is lower or higher. In either of these two cases, the client will update the range of possibilities. If it was greater, the new interval will be (0,50), which will generate a value in the center of the interval 25. If it was smaller, the new interval will be (50,100), which will generate a value in the center of the interval 75. And will resend the value for validation to the server. If the value is correct, the game will end.

It is requested to implement a client program, named `cli1`, and a server program, named `ser1`. The server and the client will play the number guessing game that is posed but in this case they will do it through *sockets*.

On the one hand, the **server** must open a *socket* (TCP) in which it will remain waiting for connection requests from clients. The PORT number that the TCP server will listen on will be passed as the input argument to the program: `./ser1 PORT`, where `PORT` will be the port number. If this parameter is not passed, the server will have port number 8888 defined as the default listening port.

For its counterpart, the **client** will have two input parameters: the IP and the PORT on which the server is waiting for the connection. So, we will execute '`./cli1 PORT IP`' where IP will be the IP address of the server, and PORT will be the number of the port on which the server is waiting for connection requests from the customers. If the server's IP address is not passed, the *loopback* address (127.0.0.1) will be taken by default. In the same way, if the port value is not passed, the value 8888 will be taken by default. In the event that only one input parameter is passed, it will be assumed that this is the port number (not the IP to facilitate the programming of the activity). Thus, there can be no input argument, just PORT, or both IP and PORT arguments.

Both the client and the server must check the entered input parameters and, if necessary, they must put error messages and stop their execution.

Once started, the server must be able to continuously service clients (client requests) one at a time. No creation of a new process to serve the client should be done, neither with processes nor with (*threads*). Once the connection request is accepted, the game begins. For each new connection accepted, the server generates a random number within the range 0 to 100, both numbers included. For each iteration of the game, the server receives the number proposal from the client and returns whether the chosen value is less than, equal to, or greater than the generated random number. When the value is the same, the game ends and the server closes this connection and starts to attend to a new possible connection request from another client.

For its part, the client, once the connection is accepted by the server, generates a number located in the middle of the range of the server's random numbers and will send the value to the server. When receiving the response from the server, there are three possibilities according to previous notes.

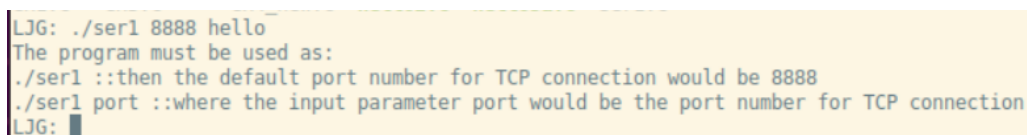
When the number is guessed the game is over, the guessed value and the number of iterations that have been needed are printed on the screen. The connection to the server is closed and the game ends by the client.

It is requested:

- Code the client (`cli1.c`) and the server (`ser1.c`)
- Indicate how to get the two executables from the client (`cli1`) and from the server (`ser1`)
- Show the different execution options of both programs, to validate the parameter check and the automatic selection of default parameters

- Show the execution of the server and at least a couple of clients
- Attach the appropriate screenshots to the report
- Attach relevant explanations of code and execution to the report
- Attach to the delivery all the necessary files for the activity.

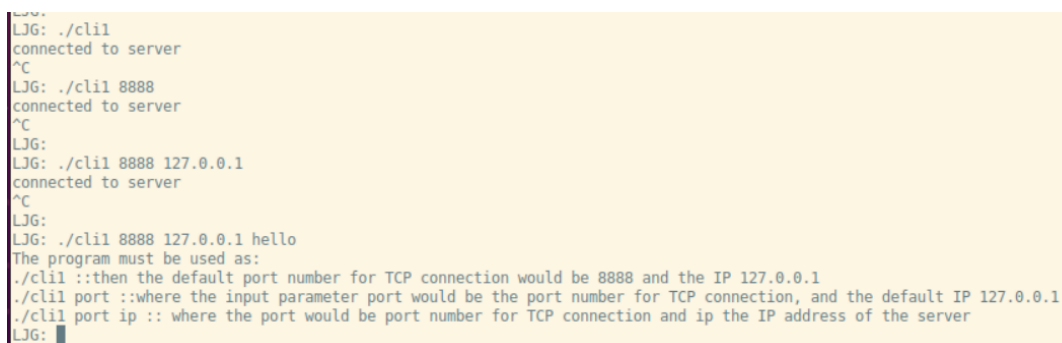
Below in Figure 1, we can see the execution of the server, and the validation of input parameters.



```
LJG: ./ser1 8888 hello
The program must be used as:
./ser1 ::then the default port number for TCP connection would be 8888
./ser1 port ::where the input parameter port would be the port number for TCP connection
LJG: █
```

Figure 1: Example of validation of the ser1 parameters.

Below in Figure 2, we can see the execution of the client, and the validation of input parameters.



```
LJG:
LJG: ./cli1
connected to server
^C
LJG: ./cli1 8888
connected to server
^C
LJG:
LJG: ./cli1 8888 127.0.0.1
connected to server
^C
LJG:
LJG: ./cli1 8888 127.0.0.1 hello
The program must be used as:
./cli1 ::then the default port number for TCP connection would be 8888 and the IP 127.0.0.1
./cli1 port ::where the input parameter port would be the port number for TCP connection, and the default IP 127.0.0.1
./cli1 port ip :: where the port would be port number for TCP connection and ip the IP address of the server
LJG: █
```

Figure 2: Example of validation of cli1 parameters.

You can also view the result of the execution with traces to track the execution in each iteration, on the client in the Figure 4, and on the server in Figure 3.

On the server side, we can see how it has received a connection request from a client from IP=127.0.0.1 (in this case *localhost*), that the server for this game has generated the number 60, and that finally the customer has guessed the number in 6 iterations. Let us remember that, with the proposed solution, the number of iterations is quite always similar, since we use a successive approximation procedure.

On the client side, we can see how the client connects to the server, and we can see the successive number proposals until the sixth iteration guesses the number.

You should generate code that has similar behavior.

```
LJG: ./ser1
Client: 127.0.0.1
The random number Zs is 60
client has found the right number in 6 iterations
```

Figura 3: Example execution ser1.

```
LJG: ./cli1
connected to server
Selected number in the range [0-100]: 50 smaller than
your choice is lower than number: moving up interval
Selected number in the range [50-100]: 75 greater than
your choice is bigger than number: moving down interval
Selected number in the range [50-75]: 62 greater than
your choice is bigger than number: moving down interval
Selected number in the range [50-62]: 56 smaller than
your choice is lower than number: moving up interval
Selected number in the range [56-62]: 59 smaller than
your choice is lower than number: moving up interval
Selected number in the range [59-62]: 60 equal to
you have found the right number in 6 iterations
LJG: █
```

Figura 4: Example running cli1.

Activity 2

In this activity you will implement a client program, named `cli2`, and a server program, named `ser2`. The server and the client will play the game of guessing numbers.

On the one hand, the **server** must open a *socket* (TCP) with a **PORT** number that will be passed as the input argument to the program: `'./ ser2 FILE PORT'` where the **PORT** parameter will be the number of the port on which the server is waiting for connection requests from the clients, and the **FILE** parameter will be the name of a text file that will be used for the generation of random numbers. If the **PORT** parameter is not passed, the server defaults to port number 8888. In case of passing a single parameter this will be the name of the file.

On the other hand, **client** will have two input parameters: the **PORT** that the server is waiting on, and the **IP** of the server. So, the command to execute is `'./cli2 PORT IP'` where **IP** will be the IP address of the server, and **PORT** will be the number of the port on which the server is waiting for requests. of customer connections. If the port value is not passed, it will default to 8888, and if the server's IP address is not passed, it will

default to *localhost* 127.0.0.1 . In the event that only one input parameter is passed, it will be assumed that this is the port number (not the IP to facilitate the programming of the activity). Thus, there can be no input argument, just `PORT`, or both `IP` and `PORT` arguments.

Both the client and the server must check the entered input parameters and, if necessary, they must put error messages and stop their execution.

As in the previous case, the server must be able to continuously serve clients (client requests) one by one. No creation of a new process should be done to serve the customer.

Unlike the previous case, now the generation of random numbers on the server will be done according to the information in a text file. The server will read the content of the text file that will be passed as the input argument (`FILE`). Next, the number of letters of each of the lines of the input file will be stored in an array (module 100), in such a way that they will be numbers between 0 and 100. You can set a maximum number of lines that will be processed from the input file. input (for example 1024) and a maximum number of characters it will accept per line (for example 256), if desired (but never as magic numbers).

Once the connection request is accepted, the game will start. For each new connection, the server will generate a random number (between 0 and the number of lines -1 of the previous file). From the number obtained, the server will read the number of letters of the indicated line (which will already be stored in a variable) and will use that number (module 100) as the random value that the client must guess.

For example, if the input file has 83 lines, and the number of letters (module 100) is stored in the variable `letters`. The server will generate for each new client, a random number between 0 and 82. If the number was 34, then it would go to see the number of letters in the line (34+1) in position 34 of the `letters` array, and this value would be what the client will have to guess.

The rest of the game will be exactly the same as in Activity 1.

The following is requested:

- Code the client (`cli2.c`) and the server (`ser2.c`)
- Indicate how to get the two executables from the client (`cli2`) and from the server (`ser2`)
- Show the different execution options of both programs, to validate the parameter check and the automatic selection of default parameters

- Show the execution of the server and at least a couple of clients
- Attach the appropriate screenshots to the report
- Attach relevant explanations of code and execution to the report
- Attach to the delivery all the necessary files for the activity.

Below in Figure 5, we can see the execution of the server, and the validation of input parameters.

```
LJG: ./ser2
The program must be used as:
./ser2 FILENAME::where the input parameter FILENAME would be the file (text file) to process and the default port number
for TCP connection would be 8888
./ser2 FILENAME port ::where the input parameter port would be the port number for TCP connection
LJG: ./ser2 text1.txt
process file text1.txt
number of lines processed 147
^C
LJG: ./ser2 text1.txt 8888
process file text1.txt
number of lines processed 147
```

Figura 5: Example of validation of ser2 parameters.

Below in Figure 6, we can see the execution of the client, and the validation of input parameters.

```
LJG: ./cli2
connected to server
Selected number in the range [0-100]: 50 greater than
your choice is bigger than number: moving down interval
Selected number in the range [0-50]: 25 greater than
your choice is bigger than number: moving down interval
Selected number in the range [0-25]: 12 greater than
your choice is bigger than number: moving down interval
Selected number in the range [0-12]: 6 greater than
your choice is bigger than number: moving down interval
Selected number in the range [0-6]: 3 equal to
you have found the right number in 5 iterations
LJG: █
```

Figura 6: Example of validation of cli2 parameters.

You can also view the result of the execution with traces to track the execution in each iteration, on the client in the Figure 8, and on the server in Figure 7.

On the server side, we can see how it has received a connection request from a client from IP=127.0.0.1 (in this case *localhost*), that the server for this game has generated the number 46, and that finally the customer has guessed the number in 5 iterations. Let us

remember that, with the proposed solution, the number of iterations is always similar, since we use a successive approximation procedure.

On the client side, we can see how the client connects to the server, and we can see the successive number proposals until the fifth iteration guesses the number.

You should generate code that has similar behavior.

```
LJG: ./ser2 text1.txt
process file text1.txt
number of lines processed 147
Client: 127.0.0.1
The random number Zs is 46
client has found the right number in 5 iterations
```

Figura 7: Example execution ser2.

```
LJG: ./cli2
connected to server
Selected number in the range [0-100]: 50 greater than
your choice is bigger than number: moving down interval
Selected number in the range [0-50]: 25 smaller than
your choice is lower than number: moving up interval
Selected number in the range [25-50]: 37 smaller than
your choice is lower than number: moving up interval
Selected number in the range [37-50]: 43 smaller than
your choice is lower than number: moving up interval
Selected number in the range [43-50]: 46 equal to
you have found the right number in 5 iterations
LJG: █
```

Figura 8: Example cli2 execution.

Activity 3

In this last activity you will have to code three programs; a program `file3.c` that will be in charge of reading the file and obtaining the random number, a program with the game server `ser3.c` and, finally, another for the game client `cli3.c`

First, the random number generator `file3` will have two possible input parameters. The name of the file `FILE` which will be required, and a port number `PORT` which will

be optional. Therefore, its execution will be `./file3 FILE PORT`. This program will implement a server (UDP) that will open a port at the value given as the input parameter PORT. If this (optional) value is not entered, the default value will be 9999.

Once the execution starts, the program will read the file indicated by the FILE parameter, obtain the total number of lines, and store the number of characters in each line (module 100) in an array for later use. Finally, the program will wait for requests from the 'serv3' program and, when it receives a request, it will send the data in two communications. In the first communication it will send you the number of lines of the file. In the second communication (second request from the server) the program will send it a line number, and the information on the number of characters of the line in question will be sent modulo 100. It is important to note that for each new request from the server, the program file3 will perform the same operation.

- UDP server receives a message from UDP client
- UDP server sends to the client the total number of lines of the file, to limit the random number generated by the client
- the UDP client will generate a number (the line number of the file to know the characters) and will send it to the server
- The UDP server will read the number of characters of the requested line to the client (be careful with the number of line and the index of the variable). And will send it to the server back

For its part, the server program 'serv3' will start its execution by making a UDP request to 'file3' to obtain the number of lines in the file. It will then wait for a connection request from a new client. For each new client, the program will generate a random number (according to the maximum lines of the file it has received), and send that number to 'file3' to send it the number of characters in this line of the file (module 100). This will be the value that the client must guess.

Thus, the server 'ser3' must have the following input parameters: PORT_UDP for the value of the UDP port to make requests to 'file3', PORT_TCP for the configuration of its TCP port by the connection requests of the 'cli3' client, and IP with the value of the address where 'file3' is running. Unlike the previous activities, all three parameters will be forced. The call will always be `./ser3 PORT_UDP IP PORT_TCP`.

Finally, the 'cli3' client will have the same arguments and will implement the same functionality as in the case of Activity 2.

Therefore, we will have:

- Let's imagine that the file FILE1.TXT has 4 lines with 10,23,45,145 characters
- We execute '`./file3 FILE1.TXT 700`'
- The file FILE1.TXT is read and the values of the number of characters of each line (module 100) are stored in a variable: 10,23,45,46
- '`file3`' open PORT 700 to receive UDP requests
- We execute '`./ser3 700 127.0.0.1 888`'. It will send a UDP message to port 700 of IP 127.0.0.1. As a response you will receive a 4 (number of lines of the file) It prepares its TCP *socket* to receive connection requests from the client, on port 888.
- Run '`./cli3 127.0.0.1 888`' which will make a connection request to the server '`ser3`' to start the game.
- Upon receiving the request, the server will generate a random number between 0-3, which it will send to '`file3`'. If for example you send the value 2, you will receive from '`file3`' the number of characters on line 3, which is 45
- From now on the game begins, and the customer must guess that number.

The rest of the game will be exactly the same as in Activity 2.

The following is requested:

- Encode the three programs (`file3.c`, `cli3.c` and `ser3.c`)
- Indicate how to get the three executables (`file3`, `cli3` and `ser3`)
- Show the different execution options of the three programs, to validate the parameter check and the automatic selection of default parameters
- Show the execution of the three programs, with at least a couple of clients
- Attach the appropriate screenshots to the report
- Attach relevant explanations of code and execution to the report
- Attach to the delivery all the necessary files for the activity.


```
LJG: ./file3
The program must be used as:
./file3 FILENAME::where the input parameter FILENAME would be the file (text file) to process and the default port number for UDP connection would be 9999
./file3 FILENAME port ::where the input parameter port would be the port number for UDP connection
LJG: ./file3 text1.txt 9999
process file text1.txt
number of lines processed 147
```

Figura 9: Example of validation of the parameters file3.

Below in Figure 9, we can see the execution of the UDP server, and the validation of input parameters.

Below in Figure 10, we can see the execution of the TCP server, and the validation of input parameters.

```
LJG: ./ser3
The program must be used as:
./ser3 PORT_UDP IP PORT_TCP:: where PORT_UDP is the port number and IP the ip address of the UDP server, and PORT_TCP is the TCP port
LJG: ./ser3 9999 127.0.0.1 8888
```

Figura 10: Example of validation of the ser3 parameters.

Below in Figure 11, we can see the execution of the client, and the validation of input parameters.

You can also view the result of the execution with traces to track the execution in each iteration, on the client in the Figure 14, and on the two servers in Figure 12 and Figure 13.

On the UDP server side, we can see how it has received a request from IP 127.0.0.1, has read a file of 147 lines, has received the request for line 79, and has returned the number of characters in the line (33). as a number to guess.

On the TCP server side, we can see how it has received a connection request from a client from IP=127.0.0.1 (in this case *localhost*), that the server has received the number of lines from the UDP server (147) has asked for the number of characters in line 79 and has received the number 33 as the number to be guessed by the client. And finally, the client has guessed the number in 7 iterations. Let us remember that with the proposed solution, the number of iterations is always similar, since we make a procedure of successive approximations.

On the client side, we can see how the client connects to the server, and we can see the successive number proposals until the seventh iteration guesses the number.

```
you have found the right number in 7 iterations
LJG: ./cli3 8888
connected to server
Selected number in the range [0-100]: 50 greater than
your choice is bigger than number: moving down interval
Selected number in the range [0-50]: 25 greater than
your choice is bigger than number: moving down interval
Selected number in the range [0-25]: 12 smaller than
your choice is lower than number: moving up interval
Selected number in the range [12-25]: 18 smaller than
your choice is lower than number: moving up interval
Selected number in the range [18-25]: 21 smaller than
your choice is lower than number: moving up interval
Selected number in the range [21-25]: 23 smaller than
your choice is lower than number: moving up interval
Selected number in the range [23-25]: 24 equal to
you have found the right number in 7 iterations
LJG: ./cli3
connected to server
Selected number in the range [0-100]: 50 greater than
your choice is bigger than number: moving down interval
Selected number in the range [0-50]: 25 greater than
your choice is bigger than number: moving down interval
Selected number in the range [0-25]: 12 smaller than
your choice is lower than number: moving up interval
Selected number in the range [12-25]: 18 smaller than
your choice is lower than number: moving up interval
Selected number in the range [18-25]: 21 greater than
your choice is bigger than number: moving down interval
Selected number in the range [18-21]: 19 smaller than
your choice is lower than number: moving up interval
Selected number in the range [19-21]: 20 equal to
you have found the right number in 7 iterations
LJG: ./cli3 8888 127.0.0.1 extra
The program must be used as:
./cli3 ::then the default port number for TCP connection would be 8888 and the I
P 127.0.0.1
./cli3 port ::where the input parameter port would be the port number for TCP co
nnection, and the default IP 127.0.0.1
./cli3 port ip :: where the port would be port number for TCP connection and ip
the IP address of the server
LJG: █
```

Figura 11: Example of validation of cli3 parameters.

You should generate code that has similar behavior.

```
LJG: ./file3 text1.txt 9999
process file text1.txt
number of lines processed 147
REQUEST FROM SERVER UDP: 127.0.0.1
Number of lines of file: 147
REQUEST FROM SERVER UDP: 79
Random number: 33
```

Figura 12: Example running file3.

```
LJG: ./ser3 9999 127.0.0.1 8888
Client: 127.0.0.1
NUMBER OF LINES FROM SERVER UDP: 147
the RANDOM line number is 79
RANDOM NUMBER FROM SERVER UDP: 33
client has found the right number in 7 iterations
```

Figura 13: Example running ser3.

```
LJG: ./cli3
connected to server
Selected number in the range [0-100]: 50 greater than
your choice is bigger than number: moving down interval
Selected number in the range [0-50]: 25 smaller than
your choice is lower than number: moving up interval
Selected number in the range [25-50]: 37 greater than
your choice is bigger than number: moving down interval
Selected number in the range [25-37]: 31 smaller than
your choice is lower than number: moving up interval
Selected number in the range [31-37]: 34 greater than
your choice is bigger than number: moving down interval
Selected number in the range [31-34]: 32 smaller than
your choice is lower than number: moving up interval
Selected number in the range [32-34]: 33 equal to
you have found the right number in 7 iterations
LJG: █
```

Figura 14: Example running cli3.