Distributed Programming (03NQVOC)
Distributed Programming I (03MQPOV)

Laboratory exercise n.1

Important preliminary note. For the network programming part of the course we use the Linux OS that is installed in the PCs at LABINF. In that installation, “wireshark” and “tshark” are installed and configured so that they can capture traffic on the various network interfaces even if the program is run by a normal user (no super-user, nor root). You are invited to debug your applications by running them locally and by using the above mentioned tools to capture the packets sent and received by your applications on the loopback interface (“lo”). This use of the traffic analysis tools, limited to the “lo” interface, is admitted. However, we remind you that capturing traffic on **other** interfaces, where traffic **not belonging to your applications** is transmitted, is not allowed. In particular, capturing authentication credentials (e.g. passwords) in order to get unauthorized access to computers is an offense with criminal consequences. For this reason, it is strictly forbidden.
Those who will be discovered capturing or trying to capture credentials or similar will be immediately expelled from the lab and deferred to the disciplinary commissions of the University, besides being subject to administrative and criminal penalty according to law.

Exercise 1.1 (test server)

In Linux environment, look at the provided test server (it will be listening to the specified port number, and it will calculate the sum of the two numbers received as input) and compile it using the compilation command:

gcc -Wall -DTRACE -o server_test server_test.c errlib.c sockwrap.c

Then run it using:

./server_test format port

where **format** is an option that specifies the data format used by the server to communicate with the client:

- **-a** to represent data in ASCII format
- **-x** to represent data in XDR format

and **port** is the port number on which the server will be listening to (e.g. 1500).

Test the server by using the -a option and the **telnet** command as client application. Telnet takes as input parameters the server address and the port number. Once connected you are expected to type two decimal numbers followed by RETURN.

Note: to interrupt the connection inside the telnet application press CTRL + ALT GR +] then Q, followed by RETURN.

Note 2: the test server code uses the functions provided by sockwrap.c (Socket, Connect, etc) and errlib.c. They are a selection of the functions provided in the book by Stevens. As explained

in the lectures, uppercase functions are an exact copy of the socket API functions but they also automatically check the function return values in order to signal and report errors and exit the program in case of error. You can modify the code in those files to adapt it to your needs (e.g. you can add more functions). This may be useful for you in the final test.

Note 3: **IMPORTANT**: if you want, you can use an integrated development environment (e.g. Eclipse, XCode, etc.). In that case, you are strongly recommended to set the -Wall option for compilation, so that all potential errors and warnings are notified. In any case, you are strongly suggested to make practice with command-line compilation which is the only one that guarantees correct functioning at the exam.

Exercise 1.2 (connection)

Write a client that connects to a TCP server with the address and port number specified as first and second command-line parameters, respectively. The client will then notify the user if it has managed to perform the connection or not. Finally it will close the connection, and terminate.

Exercise 1.3 (ASCII data)

Develop a client that connects to a TCP server like the provided test server, to the address and port number specified as first and second command-line parameters, respectively. The client then reads two unsigned decimal integers from the standard input and sends them (encoded as strings of ASCII characters, see below for details)to the server. Finally, the client receives the answer (sum, or error) from the server and prints it to the standard output. In order to test the program, it is possible to use the server compiled in exercise 1.1.
Note that the integers are represented locally in the client and in the server as 16-bit unsigned integers, while they are represented using ASCII characters on the TCP connection (see details below). The server also handles the overflow case, by signalling it with a specific error (read the example below).
The client sends the two numbers to the server represented in the decimal notation and encoded as a sequence of ASCII characters. The ASCII strings of the two numbers must be separated by a single space and transmission must be terminated by CR-LF (carriage return and line feed, that is, the two bytes that corresponds to the two hexadecimal values 0x0d 0x0a, '\r' and '\n' in C notation). The server returns the result (a single number) by expressing it through ASCII characters ijn the decimal notation, without spaces, and terminated by CR-LF. In case of error, the server returns a single error message (sequence of ASCII characters), also terminated by CR-LF. The error message can be distinguished from the successful result because it does not start with a digit (see examples). In your C code you can assume that the char type uses ASCII.

Examples (every element represents a single byte sent in ASCII code):

(client -> server) `1 2 3 4 5  3 CR LF`

(server -> client) `1 2 3 4 8 CR LF`

or, in case of error:

(server -> client) `o v e r f l o w CR LF`

or:

(server -> client) `i n c o r r e c t  o p e r a n d s CR LF`

Exercise 1.4 (client-server UDP base)

Write a client that can send to a UDP server (to the address and the port number specified by the first and second parameter of the command line) a datagram containing a name (max 31 characters) specified as third parameter on the command line. The client then awaits any reply datagram from the server. The client terminates by showing the content (ASCII text) of the received datagram or by signalling that it didn't receive any reply from the server within a certain timespan.
Develop a UDP server (listening to the port specified as first parameter on the command line) that replies to any received datagram by answering with a datagram containing the same name provided by the received packet.
Try to perform datagram exchanges between client and server with the following configurations:

- client sends a datagram to the same port the server is listening to
- client sends a datagram to a port the server is NOT listening to
- client sends a datagram to an unreachable address (es. 10.0.0.1)

In the provided lab material you can find the executable version of a client and of a server that behave as expected (note that the executable files are provided for both 32bit and 64bit architectures. Files with the suffix _32 are compiled for and run on 32bit Linux systems. Files without that suffix are for 64bit systems. Labinf computers are 64bit systems).
Try to connect your client with the server included in the provided lab material, and the client included in the provided lab material with your server, in order to test interoperability of your code. If fixes are needed in your client or server, make sure that your client and server can communicate correctly with each other after the modifications. Finally you should have a client and a server that can communicate with each other and that can interoperate with the client and the server provided in the lab material.