Distributed Programming I (03MQPOV)

Laboratory exercise n.2

Exercise 2.1 (perseverant UDP client)

Modify the UDP client of exercise 1.4 so that - if it does not receive any reply from the server in 3 seconds - it re-transmits the request (up to a maximum of 5 times) then it terminates by reporting if it has received the reply or not.
Perform the same tests of exercise 1.4

Exercise 2.2 (binary UDP protocol)

Write a UDP server that waits for datagrams from clients and responds to the received datagrams by sending a response datagram. The received datagram must contain only one 32-bit unsigned integer in network byte order and the response datagram must contain only two 32-bit unsigned integers. The first one is the same integer that was received, while the second one is the current time, expressed as the number of seconds since the start of epoch (it can be obtained by calling the time() function). If the server receives a datagram which is not 32-bits long, the server simply has to ignore it.
Write a client that sends a UDP datagram to a server with address and port number specified by the first and second parameter of the command line. The UDP datagram must contain the current time, expressed as the number of seconds since the start of epoch, encoded as a 32-bit unsigned integer in network byte order. After having sent this datagram, the client must wait for the response datagram. If the response datagram is received, the client must print its contents (which should be two 32-bit unsigned integers in network byte order) to the standard output (as decimal integers). If the received response is not 64-bits long, or if no response arrives within 5 seconds after sending the request, the client must just print an error message.

Exercise 2.3 (iterative file transfer TCP server)

FOR EXEMPTION, this exercise has to be submitted by May 20, 2019, 11:59am
Develop a TCP sequential server (listening to the port specified as the first parameter of the command line, as a decimal integer) that, after having established a TCP connection with a client, accepts file transfer requests from the client and sends the requested files back to the client, following the protocol specified below. The files available for being sent by the server are the ones accessible in the server file system from the working directory of the server.
Develop a client that can connect to a TCP server (to the address and port number specified as first and second command-line parameters, respectively). After having established the connection, the client requests the transfer of the files whose names are specified on the command line as third and subsequent parameters, and stores them locally in its working directory. After having transferred and saved locally a file, the client must print a message to the standard output about the performed file transfer, including the file name, followed by the file size (in bytes, as a decimal number) and timestamp of last modification (as a decimal number).
Any timeouts used by client and server to avoid infinite waiting should be set to 15 seconds.

The protocol for file transfer works as follows: to request a file the client sends to the server the three ASCII characters “GET” followed by the ASCII space character and the ASCII characters of the file name, terminated by the ASCII carriage return (CR) and line feed (LF):

G	E	T		…filename…	CR	LF

(Note: the command includes a total of 6 ASCII characters, i.e. 6 bytes, plus the characters of the file name). The server responds by sending:

+	O	K	CR	LF	B1	B2	B3	B4	File contents………	T1	T2	T3	T4

Note that this message is composed of 5 characters followed by the number of bytes of the requested file (a 32-bit unsigned integer in network byte order - bytes B1 B2 B3 B4 in the figure), followed by the bytes of the requested file contents, and then by the timestamp of the last file modification (Unix time, i.e. number of seconds since the start of epoch, represented as a 32-bit unsigned integer in network byte order - bytes T1 T2 T3 T4 in the figure).

To obtain the timestamp of the last file modification of the file, refer to the syscalls *stat or* f*stat.*

The client can request more files using the same TCP connection, by sending several GET commands, one after the other. When it has finished sending commands on the connection, it starts the procedure for closing the connection. Under normal conditions, the connection should be closed gracefully, i.e. the last requested file should be transferred completely before the closing procedure terminates.
In case of error (e.g. illegal command, non-existing file) the server always replies with:

-	E	R	R	CR	LF

(6 characters) and then it starts the procedure for gracefully closing the connection with the client.

When implementing your client and server, use the directory structure included in the zip file provided with this text. After having unzipped the archive, you will find a directory named `lab2.3`, which includes a `source` folder with a nested `server1` subfolder where you will write the server and a `client1` subfolder where you will write the client. Skeleton empty files (one for the client and one for the server) are already present. Simply fill these files with your programs without moving them. You can use libraries of functions (e.g. the ones provided by Stevens). The C sources of such libraries must be copied into the `source` folder (do not put them into the `client1` or `server1` subdirectories, as such directories must contain only your own code!). Also, remember that if you want to include some of these files in your sources you have to specify the ”`..`” path in the include directive).
Your code is considered valid if it can be compiled with the following commands, issued from the `source` folder:

```
gcc -std=gnu99 -o server server1/*.c *.c -Iserver1 -lpthread -lm

gcc -std=gnu99 -o client client1/*.c *.c -Iclient1 -lpthread -lm
```

The `lab2.3` folder also contains a subfolder named `tools`, which includes some testing tools, among which you can find the executable files of a reference client and server that behave according to the specified protocol and that you can use for interoperability tests. Try to connect your client with the reference server, and the reference client with your server for testing interoperability (note that the executable files are provided for both 32bit and 64bit architectures. Files with the _32 suffix are compiled for and run on 32bit Linux systems. Files without that suffix are for 64bit systems. Labinf computers are 64bit systems). 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 reference client and server.

Try the transfer of a large binary file (100MB) and check that the received copy of the file is identical to the original one (using diff) and that the implementation you developed is efficient in transferring the file in terms of transfer time.

Try also your client and server under erroneous conditions:

- While a connection is active try to activate a second client against the same server. You may notice that further clients connect to the server anyway using the TCP 3-way handshake even if the server has not yet called the accept() function. This is a standard behaviour of the Linux kernel to improve the server response time. However, until the server has not called accept() it cannot use the connection, i.e., commands are not received. In case the server does not call accept() for a while, the connection opened by the kernel will be automatically closed.
- Try to activate a second instance of the server on the same node and on the same port.
- Try to connect the client to a non-reachable address.
- Try to connect the client to a reachable address but on a port the server is not listening to.
- Try to stop the server (by pressing ^C in its window) while a client is connected.

When you have finished testing your client and server, you can use the test script provided in the `lab2.3` folder in order to perform a final check that your client and server conform to the essential requirements and pass the mandatory tests necessary for submission. In order to run the script, just give the following command from the `lab2.3` folder

```
./test.sh
```

The script will tell you if your solution is acceptable. If not, fix the errors and retry until you pass the tests. The same acceptance tests will be executed on our server when you will submit your solution. Additional aspects of your solution will be checked after submission closing, in order to decide about your exemption and to assign you a mark.
If you want to run the acceptance tests on a 32-bit system you have first to overwrite the 64-bit version executables under `tools` with their respective 32-bit versions. For example:

```
 mv server_tcp_2.3_32 server_tcp_2.3
```

You will have to possibility to submit your solution to be considered for exemption **together with the solution of another exercise that will be assigned with lab3**. Instructions about how to submit will be given with lab3.