Distributed Programming

Test on Network Programming of June 28, 2017   Time: 2 hours 10 minutes

The exam material is located in the folder “`exam_dp_jun2017`” within your home directory, which is referred to as `$ROOT` in this text.
For your convenience, the exam material already contains a skeleton of the C files you have to write (in the “`$ROOT/source`” folder). It is **HIGHLY RECOMMENDED** that you start writing your code by filling these files **without moving them** and that you read carefully and completely this text before starting your work!
The exam consists of writing other versions of the client and server developed for the lab exercise 2.3 (file transfer), using a slightly different protocol. According to the new protocol, the following binary encoding is used for messages, instead of the ASCII-based encoding specified in exercise 2.3:

- Message GET is made of one byte, set at integer value 0, followed by a 2-bytes unsigned integer in network byte order (L1 L2) that represents the length of the filename, followed by the ASCII characters of the filename:

0	L1	L2	…filename…

- The positive response of the server (i.e. message +OK described in exercise 2.3) is made of one byte set at integer value 1, followed by two 4-bytes unsigned integers (B1 B2 B3 B4 and T1 T2 T3 T4), followed by the file contents:

1	B1	B2	B3	B4	T1	T2	T3	T4	File content………

  B1 B2 B3 B4 and T1 T2 T3 T4 have the same meaning and encoding as in the original protocol.
- Message QUIT is made of one byte set at integer value 2.
- The negative response of the server (i.e. message -ERR) is made of one byte set at integer value 3.

Part 1 (mandatory to pass the exam, max 6 points)
Copy your server that is part of your solution of Lab exercise 2.3 into the main C template that is under the directory `$ROOT/source/server1`, and modify it so that it works according to the new version of the protocol, specified above.
The server must listen to the port specified as its only argument on the command line (as a decimal number), on all the available interfaces.
The C file(s) of the new server program must all be written in the directory `$ROOT/source/server1`, except for common files (e.g. the ones by Stevens) that you want to re-use in the other programs you have to develop in this test, which you can put into the directory `$ROOT/source` (remember that if you want to include some of these files in your sources you have to specify the ”`..`” path).
In order to test your client, you can run the command

```
./test.sh
```

from the `$ROOT` directory. Note that this test program also runs other tests (for the next parts). It will indicate if at least the mandatory tests have been passed.

Part 2 (max 6 points)
Copy your client that is part of your solution of Lab exercise 2.3 into the main C template that is under the directory `$ROOT/source/client1`, and modify it so that it works according to the new version of the protocol, specified above. If necessary, modify your client so that it receives the IPv4 address (in dotted decimal notation) and the port number (in decimal notation) of the server to connect to as the first and second argument on the command line respectively, and the name of the file to download as the third argument. For the sake of simplicity, this client does not have to request a list of files as the client of lab2.3 does; instead, it has to transfer a single file, whose name is given as the third argument on the command line, and then send the QUIT command, gracefully close the connection, and terminate.
The C file(s) of the new client program must all be written in the directory `$ROOT/source/client1`, except for common files that, as already said, have to be stored in the directory `$ROOT/source`.
The test command indicated for Part 1 will also try to test Part 2.

Part 3 (max 4 points)
Write a new version of the server developed in Part 1 (named `server2`) that behaves as `server1` but

- it is a concurrent server using pre-forking, which can serve up to 3 clients concurrently (for `server1` there was no requirement about concurrency).
- If a connection is established but no command is received within 10 seconds, the server must close the connection.
- Similarly, after having finished to send the positive response message, including the file transfer, the server must wait for another message for at most 10 seconds, and close the connection if no message arrives within that time.

The C file(s) of the `server2` program must all be written in the directory `$ROOT/source/server2`, except for common files that, as already said, have to be stored in the directory `$ROOT/source`.
The test command indicated for Part 1 will also try to test Part 3.

Further Instructions (common for all parts)
In order to pass the exam, it is enough to implement a `server1` that correctly responds to a file request according to the new protocol, **or** to implement a `client1` and a `server1` that can interact with each other as expected (i.e. at least they can transfer a small file correctly).
Your solutions will be considered valid **if and only if** they can be compiled by the following commands issued from the `source` folder (the skeletons that have been provided already compile with these commands; do not move them, just fill them!):

```
gcc -o socket_server1 server1/*.c *.c -Iserver1 -lpthread -lm
gcc -o socket_client1 client1/*.c *.c –Iclient1 -lpthread -lm

gcc -o socket_server2 server2/*.c *.c -Iserver2 -lpthread -lm
```

For your convenience, the test program also checks that your solutions can be compiled with the above commands.
Note that all the files that are necessary to compile your programs must be included in the source directory (e.g. it is possible to use files from the book by Stevens, but these files need to be included by you).
All the produced source files (`client1`, `server1`, `server2`, and common files) must be included in a single zip archive created with the following bash command (run from the `$ROOT` directory):

```
./makezip.sh
```

At the end of the exam, the zip file with your solution must be left where it has been created by the zip command.

Important: Check that the zip file has been created correctly by extracting it to an empty directory, checking its contents, and checking that the compilation commands are executed with success (or that the test program works).

Warning: the last 10 minutes of the test MUST be used to prepare the zip archive and to check it (and fix any problems). If you fail to produce a valid zip file in the last 10 minutes your exam will be considered failed!

The evaluation of your work will be based on the provided tests but also other aspects of your program (e.g. robustness) will be evaluated. Then, passing all tests does not necessarily imply getting the highest score. When developing your code, pay attention to making a good program, not just making a program that passes the tests provided here.
For your convenience, the text of Lab exercise 2.3 is reported in the next pages of this file.

Exercise 2.3 (iterative TCP server)

Develop a TCP server (listening to the port specified as first parameter of the command line) accepting file transfer requests from clients and sending the requested file.

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), to request files, and store them locally. File names to be requested must be provided to the client using the standard input, one per line. Every requested file must be saved locally and the client must print a message to the standard output about the performed file transfer, with file name, size and timestamp of last modification.

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 characters plus the characters of the file name). The server replies by sending:

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

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), 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) and then by the bytes of the requested file.

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 by sending many GET commands. When it intends to terminate the communication it sends:

Q	U	I	T	CR	LF

(6 characters) and then it closes the communication channel.

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 closes the communication channel with the client.

Save the client into a directory different from the one where the server is located. Name the client directory as the client program name.

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 for testing interoperability (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). 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.

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.

While a connection is active try to activate a second client against the same server.

Try to activate on the same node a second instance of the server on the same port.

Try to connect the client to a non-reachable address.

Try to connect the client to an existing address but on a port the server is not listening to.

Try to de-activate the server (by pressing ^C in its window) while a client is connected.