Distributed Programming

Test on Network Programming of June 22, 2016 Time: 2 hours 10 minutes

The exam material is located in the folder “`exam_dp_jun2016`” within your home directory. For your convenience, the folder already contains a skeleton of the C files you have to write (in the “`source`” subfolder). 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 another version of the client and server for the lab exercise 2.3 (file transfer), using a slightly different protocol.

Part 1 (mandatory to pass the exam, max 7 points)
Write a server (server1) that works according to the protocol specified in Lab exercise 2.3 (for your convenience the text of Lab exercise 2.3 is included in this file at the end), but with a small variation: as soon as the server finishes reading a command from the client, it computes a timestamp TS1 (using the gettimeofday function). After having sent the requested file, the server computes another timestamp TS2, and also computes the difference TS2-TS1 in microseconds. This number is encoded as a 32-bit unsigned integer in network byte order and it is sent immediately after the file contents.
The server must listen to the port specified as first argument on the command line, on all the available interfaces.
After having sent the response to the client as specified above, the server must output one line of text to the standard output with the following format:

    <prefix> <client-ip> <client-port> <filename> <TS2-TS1>

where:

- <prefix> is the fixed 9-character ASCII string "OPERATION"
- <client-ip> is the IP address of the client that made the request in dotted decimal
- <client-port> is the client port number as a decimal number
- <filename> is the file name as a sequence of ASCII characters
- <TS2-TS1> is the same value sent to the client, as a decimal number

Each pair of adjacent fields in this text line must be separated by one space.
Important: these lines of text must be the only outputs written by server1 to the standard output.

The C file(s) of the server program must be written in the directory `$ROOT/source/server1`, where <u>`$ROOT` is the exam directory</u> (exam_dp_jun2016). If you have library files (e.g. the ones by Stevens) that you want to re-use for the next parts, you can put them in 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 server 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 4 points)

Write a client (named `client`) that behaves as the one developed in Lab exercise 2.3, but with the following differences:

1. The new client must receive and use the following command line arguments: the first argument is the IP address (IPv4 or IPv6) or hostname of the server, the second argument is the port number of the server, the third argument is the name of a file to be downloaded from the server. If the first argument represents the IP address of the server, it is expressed in the standard (decimal or hex) notation, while the port number is expressed as decimal number.
2. The client must connect to the server specified on the command line, and download the file specified on the command line using the modified protocol specified in Part 1. After this, the client must output the time value received from the server as a decimal number, preceded by the 9-character fixed string "TIMESTAMP" and separated from this string by one space, and terminate. The decimal number must be the **only** output of the client to the standard output.

The C file(s) of the new client program must be written under the directory `$ROOT/source/client`, where `$ROOT` is the exam directory (exam_dp_jan2016). The test command indicated for Part 1 will also try to test Part 2.

Part 3 (max 5 points)

Write a new version of the server developed in Part 1 (server2) that can serve at least 3 clients concurrently (for server1 there was no requirement about concurrency). Moreover, server2 must gracefully close the connection with a client if no command is received from the client for 10 seconds since the start of the connection or since the last file transfer has been completed.

The C file(s) of the server2 program must be written under the directory `$ROOT/source/server2`, where `$ROOT` is the exam directory (exam_dp_jun2016). The test command indicated for Part 1 will also try to test server2.

Further Instructions (common for all parts)

In order to pass the exam it is enough to implement a server1 that sends a correctly encoded file followed by the specified timestamp information to the client and prints the expected information to the standard output, or to implement a server1 and a client that can interact with each other as expected.

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_client client/*.c *.c -Iclient -lpthread -lm

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

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).
For your convenience, the test program also checks that your solutions can be compiled with the above commands.
All the produced source files (`server1`, `server2`, `client` and common files) must be included in a single zip archive created with the following bash command (run from the `exam_dp_jun2016` 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.

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.