Distributed Programming

Test on Network Programming of February 18, 2020 Time: 2 hours 10 minutes

The exam material is located in the folder "`exam_dp_feb2020`" 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 another version of the client and of the server developed for lab exercise 2.3 (file transfer), using a slightly different version of the protocol. According to the modified protocol, the GET command sent by the client is modified as it follows:

G	E	T		offset		…filename…	CR	LF

i.e., between the GET keyword and the filename, the client has to put the requested offset, which is a positive integer represented in decimal notation as a sequence of ASCII characters. The offset and the filename are separated by a space ASCII character, as well as the GET keyword and the offset. The meaning of this request is that the client wants to get the part of the specified file contents that starts at the specified offset, where an offset of 0 means the beginning of the file. The offset must not exceed 2^{32}-1 for obvious reasons. If the client uses 0 for the offset, it means the full contents of the file are requested.
The positive response from the server is modified as it follows:

+	O	K	CR	LF	B1	B2	B3	B4	O1	O2	O3	O4	File contents……	T1	T2	T3	T4

which is the same format that was used in the original protocol, with the addition of bytes O1 O2 O3 O4. These bytes are interpreted as a 32-bit unsigned integer in network byte order, which is the offset that has been requested by the client. The file contents field is just the last part of the file, starting at the requested offset, while the value represented by bytes B1 B2 B3 B4 represents the full size of the file, as it was in the original protocol. This implies that the length of the file contents field can be computed as the difference between the full file size (bytes B1 B2 B3 B4) and the offset (bytes O1 O2 O3 O4).
If the server receives an invalid offset (e.g. greater than the maximum allowed value or containing characters that are not decimal digits), it must respond with the error message. If instead the offset is valid but it is greater than or equal to the size of the file, the server must respond with a +OK message including an empty (i.e. 0 bytes long) file contents field.
The server can move the file pointer to the right position for reading, by calling fseek(fp, offset, SEEK_SET). See man page if other details are needed.

Part 1 (mandatory to pass the exam, max 6 points)
Copy your server that is part of your solution of Lab exercise 2.3 (iterative file transfer TCP server) into the main C template that is under the directory `$ROOT/source/server1`, and modify it so that it works according to the modified version of the protocol, specified above.

Apart from what specified above, all the other features of server1 are the same as in the original version.
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 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 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 `$ROOT/source/client1`, and modify it so that it works according to the new version of the protocol, specified above. The new client must receive the IP address and the port number of the server, as first and second argument on the command line, as it was in the original exercise, while the next arguments on the command line are the names of the files the client has to request, each preceded by the requested offset, written as decimal number. That is, the third argument on the command line is the offset of the first requested file, the fourth argument is the name of the first requested file, and so on for the other files.
Of course, for each requested file, the client has to store locally the file contents it received from the server (i.e. the last part of the file, starting at the offset). The message the client has to output to the standard output when each file has been received is now made of the file name, followed by the file size and the offset (in bytes, as decimal numbers) and the timestamp of the last modification (as a decimal number).
Apart from these specifications, all the other features of client1 are the same as in the original version.
The C file(s) of the new client program must all be written in the directory `$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 serves clients concurrently, using creation of new processes on demand.
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.
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 -std=gnu99 -o socket_server1 server1/*.c *.c -Iserver1 -lpthread -lm

gcc -std=gnu99 -o socket_client1 client1/*.c *.c –Iclient1 -lpthread -lm

gcc -std=gnu99 -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 test, 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 file transfer TCP server)

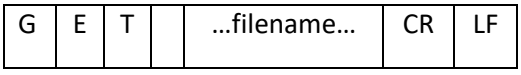
FOR EXEMPTION, this exercise has to be submitted by May 20, 2019, 11:59am

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.

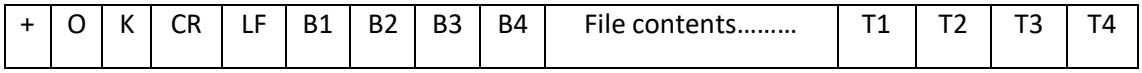
Develop a client that can connect to a TCP server (to the address and port number specified as first and second command-line parameters, respectively). After having established the connection, the client requests the transfer of the files whose names are specified on the command line as third and subsequent 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):



(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:



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.