Distributed Programming

Test on Network Programming of February 3, 2015 Time: 2 hours and 10 minutes

The exam material is located in the folder "`exam_dp_feb2015`" 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 a client and a server aimed at testing data transfer on a TCP/IP connection, according to the specifications given below. In order to pass the exam, writing a server that behaves according to the specifications is mandatory.

Part 1 (**mandatory** to pass the exam, max 8 points)
Write a server (server1) that receives two arguments on the command line. The first argument is the (decimal) TCP port number the server listens to, while the second argument is another decimal integer that specifies how many Megabytes (i.e. 2^{20} bytes = 1048576 bytes) the server has to receive on each established connection. As soon as a connection is established, the server starts receiving the data sent by the client on the connection. While receiving the data, the server computes a 32-bit hash code of the received bytes (see below how to do this). As soon as the total amount of Megabytes specified in the second argument on the command line has been received, the server sends the 32-bit hash code (4 bytes) to the client in network byte order, prints the hash code as decimal integer in the standard output, and then gracefully closes the connection.
The 32-bit hash code has to be computed by means of the `hashCode` function whose source code is included in the `utils.c` file (in the `source` folder). In order to compute the hash code chunk by chunk, proceed in the following way: call `hashCode(data,n,1)`, where `data` is the first chunk of data and `n` is its length. Then, for each subsequent new chunk of data, call `hashCode(data,n,hc)`, where `data` and `n` are the new chunk of data and its length, while `hc` is the result obtained in the previous call to `hashCode`. The integer returned by the last call to `hashCode` is the overall code that has to be sent by the server.
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_feb2015). 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 "`../source`").
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 test has been passed.

Part 2 (max 4 points)
Write a client (named `client`) that receives exactly 4 arguments on the command line: the first argument is the server IP address in dotted decimal notation, the second argument is the (decimal) server TCP port number, the third argument is the decimal integer that specifies

how many Megabytes the client has to send, and the last argument is the name of a local file. At start up, the client has to fill a 1Megabyte memory buffer with the first Megabyte of data read from the local file whose name is specified in the last command line argument. If the file does not exist or its content is less than 1 Megabyte long, the client must exit with code 1. Then the client must continuously send the content of the 1Megabyte memory buffer for n times (where n is the integer specified in the third argument on the command line), i.e. n copies of the buffer must be sent to the server. Finally, the client must receive the hash code sent by the server (4 bytes in network byte order), print it as a decimal integer in the standard output, and terminate.
The C file(s) of the client program must be written under the directory `$ROOT/source/client`, where $ROOT is the exam directory (exam_dp_feb2015).

Part 3 (max 4 points)
Write a new version of the server developed in Part 1 (server2) according to the following extra specifications.
The server must be able to serve at least 3 clients concurrently (for server1 there was no requirement about concurrency).
The server must be able to detect unexpected interruptions of the data flow from the client: if the server still expects data but no more data are received, after a period of inactivity of 10 seconds the server must close the connection.
The C file(s) of the server program must be written under the directory `$ROOT/source/server2`, where $ROOT is the exam directory (exam_dp_feb2015). The test command indicated for Part 1 will also try to test server2.

Further Instructions (common for all parts)
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_feb2015` directory):

```
zip socket.zip source/client/*.[ch] source/server[12]/*.[ch] source/*.[ch]
```

The zip file with your solution must be left where it has been created by the zip command at the end of the test.

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!