Distributed Programming

Test on Network Programming of June 29, 2015 Time: 2 hours 10 minutes

The exam material is located in the folder "`exam_dp_jun2015`" 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 variations of the client and server for the lab exercise 2.3 (file transfer). In order to pass the exam, writing a server that behaves according to the specifications given in part 1 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 every how many seconds the server has to check and update the file contents on each established connection.As soon as a connection is established, the server starts behaving according to the protocol specified in Lab exercise 2.3, with the followingdifferences:

1. In the positive response to the GET command, the server also sends a 32-bit unsigned integer in network byte order that represents the last modification time of the file that will be transferred, expressed as Unix time (i.e. number of seconds since the start of epoch, i.e. since 1 January 1970 at 00.00.00 UTC). This number, which can be obtained by the server as specified below, is transmitted just before the integer representing the number of bytes to be sent. Hence, in summary, the positive response to the GET command is made up of the 5 ASCII characters "**+OK** `CR LF`", followed by the last modification time of the file (a 32-bit unsigned integer in network byte order), followed by the number of bytes of the file (a 32-bit unsigned integer in network byte order), followed by the bytes of the file contents.
2. After having sent the file contents, while the client keeps the connection open, every N seconds, where N is the value of the second argument on the command line, the server checks the last modification time of the last file requested by the client on the local file system and, if it is greater than the last modification time previously transmitted to the client, the server sends an update of the file contents to the client. The update starts with the 5 ASCII characters "**UPD** `CR LF`", followed by the new last modification time, followed by the number of bytes of the file, followed by the bytes of the file. The encoding of these data is the same as in the positive response to the GET command. If the client requests a new file with a new GET command, the updating of the previous requested file is interrupted. In that case, the server will send the new requested file and then it will start the updating process for the new file. In case of errors in accessing the file during an update, the server will close the connection.

The last modification time of a file can be read by calling the stat function. An example of call is given here (for more information, look at the stat man pages):

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
```

```
#include <sys/stat.h>

char * fname      /* the file name */
struct stat buf;  /* a buffer where file information is read */
…
if (stat(fname, &buf)!=0) {
// error
} else {
unsignedintlastmodtime = buf.st_mtime; // read last modification time
}
```

The C file(s) of the server program must be writtenin the directory `$ROOT/source/server1`, where `$ROOT` is the exam directory (exam_dp_jun2015).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.
Note that in order to pass the exam it is enough to implement the variation specified in point 1. For this reason, you are suggested to first implement point 1, thencheck that at least the mandatory test passes,save the solution using the zip command given later, and finally implement point 2.

Part 2 (max 4 points)
Write a client (named `client`) that receives exactly 3 arguments on the command line: the first argument is the server IPv4 addressin dotted decimal notation, the second argument is the (decimal) server TCP port number, and the third argument is a file name. At start up, the client has to connect to the server specified by the first two command line arguments and then it has to request the file whose filename is specified as the third command line argument. Then, the client has to receive the file, save a copy in the current client directory (the one from which the client has been run), and set,for the local copy,the same last modification time that was specified by the server (see below the instructions about how to set the last modification time of a file). After having completed the reception of the file, the client must keep waiting for updates from the server. When the server sends an update, the client has to receive the new file contents and update the local file copy, along with the last modification time. In case of errors found in the data coming from the server, or in case the server closes the connection, the client has to print an error message to the console and terminate.
The last modification time of a file can be set by calling the utime function. An example of call is given here (for more information, look at the utime man pages):

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <utime.h>

char * fname            /* the file name */
```

```
structutimbufbuf; /* a buffer where file information is read */
…
buf.actime = … ;  // set access time
buf.modtime= … ;  // set last modification time

if (utime(fname, &buf)!=0) {
    // error
} else {
    // success
}
```

Important note: call utime after fclose, otherwise fclose will overwrite the modification time! 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_jun2015).

Part 3 (max 4 points)
Write a new version of the server developed in the first part(server2) that can serve at least 3 clients concurrently (for server1 there was no requirement about concurrency).
If server1 already had this capability, you can just copy the source code of server1 to server2.
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_jun2015). 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 commandsissued 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 programsmust 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_jun2015`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!

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.