Programming Assignment 2[†]

Due Sunday, Oct 18, 2015 at 11:59pm
Late penalty: 30% for up to 24hr; no credit afterwards

This assignment will give you practice with using transport layer sockets and developing a simple file transfer protocol.

About cooperation with other students:

This assignment is meant to be done alone. Absolutely no cooperation is allowed. If there are questions, ask the course staff; they are there to help you. For this assignment, you must do all thinking, research, and coding by yourself. Do not even discuss with anyone how far you are with this assignment

ANY HELP (EXCEPT FROM COURSE STAFF) IS PROHIBITED!!!
Individuals found guilty of violating above policy will be referred to the disciplinary committee.
(Warning: We will use software to measure the code similarity)

More Important Note:
Once again, we are giving you the responsibility to REPORT all incidences of cooperation. You MUST report to the instructor (or the TAs) such incidences. If you do not, we will consider you as guilty as those who you witnessed cooperating. All coding, debugging, web search for functions, etc. must be done by individual students!

Preamble:
You will use **Linux programming environment** for this assignment. Use your existing venus[‡] account for this purpose. As a reminder, you MUST NOT share your password with anyone including the TAs and the instructor.

You will need to submit all the required files zipped in a zip file whose name is derived from your student ID. See submission instructions (at the end) for more details.

The Assignment:
The assignment consists of three parts. The first two parts are REQUIRED and the last part is for extra credit in case you missed one or more quizzes. If you missed any quizzes, make sure to attempt the third part to get some credit for those quizzes (even if it was because of illness); note that missed work cannot be credited unless extra work is done.

Part-I:
This part demonstrates a simple TCP client server interaction. You will learn about creating client side sockets and sending/receiving data over those sockets. We are providing you with the executable server program and a reference client (available in the programming section of the class website). *You are only required to write the code for the client application.* Download these implementations to your venus account and you are set to do client/server interaction; an example interaction is also available in the programming section. Note that you must run the simpleserver at a specified port of your liking first. To see the interaction, you have a choice to either use simpleclient or simply telnet to the port on which the server is running from another putty window.

[†] Partly adapted from a course offered by Jim Kurose at UMASS.
[‡] Use venus.lums.edu.pk (inside LUMS) or 110.93.234.139 (outside). Less preferably, use mureekh.

Here is a description of what server does and what your client should do:

- The server runs at a port that you must specify at the command line (details below).
- When your client `connect()`s to the server, the server sends a welcome message which must be `recv()`d and displayed by the client.
- There is a list of commands available to the client (obtained by typing help over the established connection). Your client should be able to send, one at a time, each of these commands in a **SINGLE** `send()` message. Since the server is expected to work either with the client you coded up or with a simple telnet session (as shown in the example interaction), server is expecting a '\r\n' (newline) at the end of each command. Append these two characters at the end of your client's send stream.
- The server will send the appropriate response by using a single `send()` message which must be `recv()`d by your client. The server will terminate each message with '\r\n'.
- The commands currently available are: (case does not matter):
  `help, exit, ip, mac, os, horoscope, list, time, users`
- Assume that in either direction, server to client or client to server, each message can be completely sent in a single `send()` command and can be completely received in a single `recv()` command. For short messages like the ones we are dealing with, this should work. For larger data, multiple `send()`s and `recv()`s might be needed, and an application layer protocol (such as ftp) needs to be created.
- You should display the response sent by the simpleserver to your simpleclient.

Try out the interaction using the sample programs before starting to write your own!

You are required to provide the simpleclient.cc file (or any additional files, if needed). **Your program should do error checking to handle common error cases.** This client will be the starting point for the client you will write in Part-II.

Part-II:
In this part, you will write a simplified FTP-like client/server application. But unlike the real FTP, you will use a single TCP. The client process accepts commands from the user, and performs the requested interaction with the server, very similar to what we saw in part-I. ***You are responsible for providing code for the client as well as for the server.***

Three commands should be supported:

- the "*list*" command: sent from the client to the server and will cause the server to return a list of available files for download (possibly in current directory)
- the "*get <filename>*" command: sent from the client to the server and will cause the server to send the requested file to the client (or an error code in case the desired file does not exist)
- the "*exit*" command: sent from the client to the server, causing ***both*** to terminate

For Part-II, you can program your server so that it has only a small fixed number of files it can send (i.e., you may hardcode the number and names of the files). For this part, you can further assume that the files can only be "get" from the current directory in which the server is running. Note that simpleserver of Part-I has provision for "list" and "exit".

Your server should execute as follows:

- The server creates a TCP welcoming (or listening) socket and waits for a connection from the client. Once your server accepts the client, it enters a loop, receiving (via the new socket created by the `accept()` call) one of the three commands: *list*, *get*, and *exit*.
- If the server receives a *list* command, it returns a "200 OK" line, followed by the names of, say, three files (one file name per line) that it can send to the client. It is OK to hardcode just three file names for Part-II, and the number of bytes in each file, into your server. You should create these files in your directory and fill them with text equal to the file length.
- If the server receives an *exit* command, it closes all open sockets (which is a good programming practice) and terminates.
- If the server receives a "*get <filename>*" command, where *<filename>* is the name of one of the small number of files it has available, it returns a "200 OK <filelength>" line followed by the contents of the file, where <filelength> is the number of bytes in the file. This information is sent over the new socket created by the `accept()` call using the same port on which the server was listening. Note that this is an example of *in-band* signaling. And is in contrast with standard FTP which uses *out-of-band* signaling (using separate control TCP channel).
- If the server receives any other unknown command, or a *get* for a non-existent file, it should return a "400 Command error" line.

Your client should operate as follows:

- The client creates a socket and makes a TCP connection to the server. Your client can be hardwired so that it connects to just your server (i.e., it is OK to hardwire your server's hostname, and port number but do not hardcode the IP address). Don't forget to use `#define` or `static consts` even if you are hardcoding!
- Your client then waits for commands from the user: *list*, *get <filename>*, and *exit*. For Part-II, if any other command is entered, a local error message should be displayed (without contacting the server), and the client should loop back and wait again for user input.
- If a valid command is entered, the client should send the command to the server. In the case of "*list*" or "*get*", the client should then read the response from the server. Your client should read the status code sent by the server and print a message out indicating whether or not the command was successful. The client should then display the contents returned by the server (i.e., everything following the status line). In the case of the *get* command, you should locally store the file in the current directory where the client is running and append '*client_*' to the start of the original filename. AFTER finishing this task, the client then loops back and waits for more user input. In the case of *exit*, the client **should send the exit command to the server** and should close its socket before exiting.

A client-server interaction might thus look like the following:

<client enters the *list* command>

c: list
s: 200 OK
s: file1.txt
s: file2.txt
s: file3.txt
<client enters *get file1.txt* command>
c: get file1.txt
s: 200 OK 76
    <... the 76 bytes constituting the contents of file1.txt ....>
<client enters *get file4.txt* command, for a file4.txt that does not exits>
c: get file4.txt
s: 400 Command error
<client enters *exit* command>
c: exit

Part-III:
Credit for quizzes missed **due to valid reasons** in the past will be given only if you complete Part-III. For this Part, you have to modify your server to implement dynamic directory lookup rather than hard-coding the filenames – basically, implementing the *ls* command. The client and server exchange commands and status as before, but now when the server receives a *list* command, instead of sending back a hardcoded list of files, it will search its current directory to find out what files are present at the moment and then reply back with this list (it would only consist of files **NOT** folders). You'll need to do some extra reading to figure out how to do this. For a start, try searching on how *ls* command works in UNIX. In the resources section, there is a handout on Unix Programming Tools (outdated) which will help but will not solve this. You will have to look somewhere else! (You are **NOT** allowed to use the Unix/Linux `exec` command and other similar commands)

Part-IV: (optional)
This Part is optional and wouldn't give you any credit but it is fun. I will buy you coffee (at JJ) or something of similar value on campus (your choice!) **IF** you complete this part successfully, in addition to the previous three parts. In this part, you should try to implement an additional "cd" command used to change the directory. The command is issued at the client and the directory is changed at the server end. Of course, the other commands should work as before. Using system calls such as `exec` is not allowed.

Important Programming Notes:

- What ports to use? Except for Part-I in which you are only writing a client, you MUST use only ONE port in your server which is XYabc where 20XY-??-0abc is your roll number. Example, if your roll no is 14100055 (2014-10-0055), your required port number is 14055. At the client side, let the OS/program choose a port for you (randomly).
- We suggest that everyone begin by writing a client and server that execute the *list* and *exit* commands. Then program the *get* command. Finally, write the dynamic directory listing server, if you want to do Part-III of the assignment.

- Make sure you close every socket that you use in your program. If you abort your program, the socket may still hang around and the next time you try and bind a new socket to the port ID you previously used (but never closed), you may get an error – but in this case the unclosed port would become available again after a little time, so in such a situation just use a different port in the range 8000-9000 only temporarily for a short time.
- Should error checking (for bad commands) be done on the client or server side? You want it on **BOTH** sides. The server never knows if the client is correctly written, so it needs to check the client input. Similarly the client can't know if the user is going to input the correct information, so it should check; client should also check return codes from the server (in case there is a problem there as well).
- As the instructions say: To print out the contents of file, you are to assume that all downloadable files are filled with text. For initial debugging purposes use small files with few lines of text and then for a final run just try on bigger files composed of 25+ lines of text.

Where do I start and where do I get help?
Make sure you have read the Beej's guide to Unix Socket Programming. It **IS** useful. Or talk to the TAs or the instructor if you have any questions. You should also consult man pages on the venus machine which is available by simply typing: *man gethostbyname* on the command prompt.

What is the BIGGEST piece of advice?
Start early. This assignment will require you to debug a lot of client/server code and will take considerable amount of time. Keep the entire weekend for the assignment and add another several days here and there! The only other advice is to get help from the course staff and/or the gdb debugger. Learn to debug programs using *cout*/*printf* as well.

What and Where to submit?
1. Very importantly, **strictly** stick to the following submission guidelines.
2. Use the names of the files specified in this handout.
(a) For Part-I, we need simpleclient.cc file
(b) For Part-II, we need myftpserver.cc and myftpclient.cc files
(c) For Part-III, we need the same files as in (b) but with extended "*ls*" functionality.
(d) For Part-IV, we need the same files as in (c) but with extended "*cd*" functionality.
3. Zip all these files together in one file named <your_student_num>.zip where <your_student_num> is an **8-digit** number. Example, if your roll no is 14100055, your submission file will be 14100055.zip. Be very careful with this, our script **might** throw away zip files that do not follow this convention. No contest will be allowed in that case.
4. **Do NOT email your assignment to us.**
5. Do just one submission of your zip file through LMS system (use the ***Assignment link*** in your portfolio, and **NOT** the Dropbox).