This assignment will give you more 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 five parts. The first four parts are

REQUIRED and the last part is for extra credit in case you missed one or more quizzes. (First four parts are 25 marks each.) If you missed any quizzes, make sure to attempt the fifth 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.

---

**VERY IMPORTANT NOTE:**

**For each part, you must submit a SEPARATE set of client/server files, as**

**required, even if it means repeating some code from previous parts.**

1

## 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.

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.

- The server is expected to handle no more than one client, ever. The server process terminates when the client exits. *Try out the interaction using the sample programs before starting to write your own!* For this part, 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. From Part-II to Part-V, you will need to provide both pieces of code: for the client and for the server. The client can send requests to the server to store, get or modify files. For simplicity, the file system can be assumed to be flat (i.e. no directories). You do have to handle file sizes of up to 10 MB. (Although if programmed correctly the system can easily work for file sizes greater than 10 MB.) *You are responsible for providing code for the client as well as for the server SEPARATELY for each part onwards.*

## Part-II:

In this part you have to create a simple server program which listens on a certain port. The hostname and the port number are passed as command-line arguments. *For example*: `myserver localhost 2090` where 'localhost' is the hostname of your own machine and '2090' is the port number. When the client connects to the server, the server should send a **'Hello' message** to the client (Check the reference implementation for Part-II to see the 'Hello' message).

Along with the server, you need to create a simple client program which just connects to the server listening on a certain port. For *example*: `myclient localhost 2090` where '2090' is the port number where the server is running.

On the client side if the user types 'EXIT', the client application should close the connection with the server and terminate. A message should pop up at the server side that the client has disconnected and then it should *start listening on the port again* for any other client to connect again. Note that this server behavior of handling the repeat clients is different from the server you were provided in Part-I.

**You do not need to handle the case where the server unexpectedly ends.**

Also, you need to do error handling in case the user input is incorrect. For example, if a user types myserver 12 12 a message can pop up, "Usage: myserver <hostname> <port number between 1024 and 65535>: Success". You need to do this error handling for both the client and the server.

By the end of this part you'll have a working server and client program.

**For remaining parts, it is important to make sure the client and server programs are located in separate directories in your file system (within your Linux account).**

**Part-III:**

In this part you will start to add the functionality to transfer files between the client and the server. For this functionality you'll first need to add **CREATE** and **LIST** commands.

1. LIST: This command should list the contents of the current directory at either the server or the client (depending on the argument). When the user types “LIST client” the contents of the directory at the client end should be listed and if the user types “LIST server” the contents of the directory at the server end should be listed.

Example:

> LIST server  > Server: File List

> file1.txt  > john.mp4  > assignment1.txt > random

2. CREATE:

PA # 2

a. CREATE server: When the client sends the 'CREATE server <filename>' command to the server, the server should create an empty file with the filename that the client has specified. After the server creates the file the server should send back a message. If the file already exists on the server side, the server should send back an error message.

Example:

> CREATE server assignment1.txt  > Server: The file 'assignment1.txt' has been created. > CREATE server assignment1.txt  > Server: The file 'assignment1.txt' already exists.

b. CREATE client: When the user enters the 'CREATE client <filename>' an empty file with the filename and extension should be created in the client's directory. If the file already exists an error should pop up.

Example:

> CREATE client assignment1.txt  > Client: The file 'assignment1.txt' has been created. > CREATE client assignment1.txt  > Client: The file 'assignment1.txt' already exists.

**For the CREATE command any type of file can be asked to be created: txt file, mp4 files etc. Also, you can assume that no illegal file name would be provided.**

P.S: You need to add error handling in the case the client sends something incorrect or the commands have an incorrect number of arguments etc. The client / server program should not unexpectedly end at any point. **(You are not allowed to use system calls such as exec and any similar calls.)**

## Part-IV:

This is an extension of Part-III and now you will be adding the 'SEND', 'RECEIVE' and ‘DELETE’ commands.

1. **SEND:** When the client sends the 'SEND <filename>' command to the server the specific file present in the client's directory should be sent to the server (you see, why it is important to have the server running in a different directory). If the file does not exist on

the client side, an error message should pop up (also at the client end). If the file is already present at the server then the server should send back an error message. (In this part you should be able to handle text, audio video files under 10 MB.)

Example:

> SEND assignment1.txt  > Server: File 'assignment1.txt' received.  > SEND assignment1.txt  > Server: File 'assignment1.txt' already exists on the server.

2. **RECEIVE:** When the client sends the 'RECEIVE <filename>' command to the server, the server should send back the specific file. If the file does not exist at the server, then the server should send back an error message (which the client should display). If the file already exists at the client, then the client program should ask the user if the file should be replaced or the older file should be kept.

Example:

> RECEIVE assignment2.txt  > Server: File 'assignment2.txt' does not exist.  > CREATE server 'assignment2.txt'  > Server: The file 'assignment2.txt' has been created.  > RECEIVE assignment2.txt  > Server: The file 'assignment2.txt' has been sent.  > RECEIVE assignment2.txt  > Client: The file 'assignment2.txt' already exists on the client. Please choose to either Replace (R) or Keep (K) older? R/K?

3. **DELETE:** This command should delete a file either at the server or the client end (Depends on the argument.) When the user types "DELETE client <filename>" the file should be deleted at the client end and if the user types "DELETE server <filename>" the file should be deleted at the server end.

Example:

> DELETE client assignment2.txt  > Client: File 'assignment2.txt' does

not exist.  > CREATE client assignment2.txt  > Client: The file 'assignment2.txt' has been created. > DELETE client assignment2.txt  > Client: File 'assignment2.txt' has been deleted.  > DELETE server assignment3.txt  > Server: File 'assignment3.txt' has been deleted.

**Part-V (Optional):**

In this part we will relax the assumption that the file system is flat. Now you are going to add the functionality to change the current directory for both the client and the server. To add this functionality, you are required to add the 'CD' command. **Again, you are not allowed to use system calls such as exec and any similar calls.**

1. CD: If the user types 'CD client random' the current directory at the client end should become 'random' and vice versa if the user types 'CD server random' the current directory at the server should become 'random'. (**Assuming that a directory named 'random' is present**). In this you will also have to handle one special case as below:

a. CD client .. / CD server .. : In this case the parent directory at either the client or server end should become the current directory.

Example:

> CD server ..  > Server: Directory 'assignment' has become the current directory. > CD server ..  > Server: Directory 'networks' has become the current directory.  > CD server assignment  > Server: Directory 'assignment' has become the current directory.

Now you have a client server program with filesystem commands that actually replicate commands that are present in LINUX (cd - CD, touch - CREATE, rm - DELETE).

Try to start this assignment early. because although this may look easy, this could take a lot of time. We have also uploaded sample program for each task so you know what you have to make. **You're allowed to use any C++ libraries in this assignment (You'll even be compiling this**

**using g++).** Try to use functions because a lot of work in these parts are similar. So you don't want to retype or copy paste code.

**You have to submit a client and server program for each part except for Part-I in which you just have to submit a client file. (Essentially 7 files. 1 file for Part-I and 2 each for the rest. If you do the optional Part-V then 9 files.) Make sure all the files are working perfectly.**

**Important Programming Notes:**

• What ports to use? 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).

• 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 un-closed 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 error message from the server (in case there is a problem there as well).

**Where do I start and where do I get help?**

Make sure you have read the Beej's guide (or another) 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(s) 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.