Programming Project 1

Building an ECHO Server

DUE: Sunday, September 12th

This assignment is designed to provide an introduction to network programing and the *socket* API. You will build a simple ECHO server: a program that accepts incoming connection requests and then echoes back any string that is sent. You can use the venerable Linux program **telnet** to connect to and test your program.

Required Behavior

In the next section I provide a suggested approach to the assignment. You don't have to use my template code or follow my suggestions. You will be graded against the following specifications:

Your code should be clean and easy to read with reasonable comments and variable names.

You must provide a makefile that compiles your program. It should compile without errors or warnings to a binary named "echo_s".

Your program should take one optional command line argument, -v, that invokes verbose mode causing the program to print diagnostic messages as it operates.

When run the program should create a socket and prepare it to listen for incoming TCP connections on a random port.

4.1. Once the socket is open and the port selected print a message telling the user what port it is listening on.

4.2. Begin waiting for a new connection, when a new connection is established:

4.2.1. Read a block of data from the connection, note the length of the data received (blocks can be up to 1024 bytes long)

4.2.2. Check the block to see if it starts with the letters CLOSE or QUIT.

4.2.2.1. If CLOSE, close the connection and start waiting for another connection (return to step 4.2)

4.2.2.2. If QUIT, close the connection and the listening socket and exit your program.

4.2.2.3. If the block does not contain a command, send the block back with a write() call.

4.2.3. Return to 4.2.1 waiting for the next block of data from the client.

If run with the –v flag you should print a diagnostic message at least

- When the socket is opened
- When the socket is bound.
- Before it blocks waiting for a new connection.
- When a new connection is accepted.
- Before it blocks waiting for new data.
- When you get a block of data.
- If the data included the CLOSE command,
- If the data included the QUIT command.
- Any other time you think it would be useful.

Suggested Approach

I provided skeleton code on Canvas (echo_s_skel.tgz) or from ~promig3/pub/echo_s_skel.tgz.. Remember that you may develop on any system you like but what you submit **must compile and run on isengard.mines.edu**. If you have any concerns about the portability of your code you should develop directly on Isengard.

Once you have the skeleton setup and running you should start filling in the code. You may of course do this however you like, but I would recommend the following:

- Fill in the sections of `main()` required to process command line arguments, create, bind, establish a listening queue and then close a stream type socket.

- If you run the program without the **–v** flag it should print what port was selected, then exit.
- If you run the program with the **–v** you should see each step of the socket creation process.

- Add code to accept new connection requests and as requests are received call the `processConnection()` function. At this point you can test by connect to the program with telnet.
- At this point you should have the functioning skeleton of a TCP server application. Up to this point the structure of all server applications are almost identical, regardless of the application protocol they are going to implement.
- Modify the `processConnection()` function so that it reads a block of input from the network and checks to see if it contains CLOSE or QUIT.
  - If it contains CLOSE, close the connection file descriptor and exit `processConnection()` function returning 0.
  - If it contains QUIT, close the connection file descriptor and exit `processConnection()` returning 1.
  - If it doesn't contain either command, `write()` the block back to the network. Note: you should only write the number of bytes received.

Testing with netcat

The nc (or netcat) utility is used for just about anything under the sun involving TCP, UDP, or UNIX-domain sockets. It can open TCP connections, send UDP packets, listen on arbitrary TCP and UDP ports, do port scanning, and deal with both IPv4 and IPv6. When run from the command line it makes a connection to a remote host, then reads input from the keyboard. Each time the user presses return it sends the data on the current line to the remote system. At the same time it accepts data from the remote system, displaying each line as it arrives. Hence you can use it to connect to almost any remote TCP application.

To use netcat, you simply provide the name and port of the host you are connecting to:

```
nc isengard.mines.edu 9032
```

Example

I have provided an example of my solution to the assignment which you can use to confirm your program works as I expect.

Terminal window running server	Terminal window running telnet client

`$> nc isengard.mines.edu 8518`
`Trying 138.67.209.20...`
`Connected to isengard.mines.edu.`
`Escape character is '^]'.`
`Hello World`
`Hello World`
`123456789`
`123456789`
`CLOSE`
 	
`$> nc isengard.mines.edu 8518`
`Trying 138.67.209.20...`
`Connected to isengard.mines.edu.`
`Escape character is '^]'.`
`Hello Again`
`Hello Again`
`QUIT`
 	
`$>`
	`$> ~promig3/pub/bin/echo_s -v`
	`Calling Socket() assigned file descriptor 3`
	`(echo_s.cc:90)`
	`Calling bind(3,0x7,16) (echo_s.cc:113)`
	`Using port 8518`
	`Calling listen(3,1) (echo_s.cc:129)`
	`Calling accept(3NULL,NULL). (echo_s.cc:142)`
	`We have a connection on 4 (echo_s.cc:149)`
	`Calling read(4,0x1363ea0,1024) (echo_s.cc:17)`
	`Received 13 bytes, containing the string`
	`"Hello World". (echo_s.cc:26)`
	`Calling write(4,0x1363ea0,13) (echo_s.cc:41)`
	`Wrote 13 back to client. (echo_s.cc:46)`
	`Calling read(4,0x1363ea0,1024) (echo_s.cc:17)`
	`Received 11 bytes, containing the string`
	`"123456789". (echo_s.cc:26)`
	`Calling write(4,0x1363ea0,11) (echo_s.cc:41)`
	`Wrote 11 back to client. (echo_s.cc:46)`
	`Calling read(4,0x1363ea0,1024) (echo_s.cc:17)`

What to submit.

This assignment is due by the end of the day on Sunday, September 12th. There will be a 5% per day penalty for late submissions with a maximum late penalty of 40% (but remember we need time to grade your submission before the end of the semester).

You should submit a single tarball of a single directory containing a makefile, a README.txt with your name and any information I need to compile the program and the source files needed to build your program. The single directory must be named with your username. The grader should not need to do anything other than untar your files, cd into your directory, type make and begin testing.

Do not include any core, object or binary files in the tarball. The Makefile provided with the sample code includes a target named submit that will create the tarball in the format we are looking for. All you need to do is:

$> make submit

In addition to functionality you will be graded on the quality of the code, including readability, comments and the use of proper programing practices.