Machine Problem 0

Introduction to UNIX Network Programming with TCP/IP Sockets

Due: Sunday, Jan 26th, 11:59pm

Please read all sections of this document before you begin coding.

The purpose of this machine problem is to familiarize you with network programming in the environment to be used in the class and to acquaint you with the procedure for handing in machine problems. The problem is intended as an introductory exercise to test your background in C programming. You will write, compile and run a simple network program on the EWS workstations, then hand in some files. The extensions to the code will introduce you to one method of framing data into individual messages when using a byte stream abstraction such as TCP for communication. It will also teach the importance of the definition of a strict communication protocol in network applications.

First step – Socket oriented programming

This step is optional, it requires no coding and it is intended for you to familiarize with compiling and testing network applications on Linux machines. If you have already coded a network application in C, feel free to skip to the next section.
We recommend that you use a 64-bit EWS machine since our auto-grader program will check your MP submissions on these machines**.** Information about the workstations is available on the homepage:

Check out your SVN directory from the class repository using (remember to replace your `<netid>` into the path):

```
svn checkout https://subversion.ews.illinois.edu/svn/sp18-cs438/<netid>
```

You will find a folder named `MP0`, which contains the programs `client.c`, `server.c`, `talker.c`, and `listener.c` from Beej's Guide to Network Programming, also available at:

`http://beej.us/guide/bgnet/`

Figure out what these programs are supposed to accomplish. Reading Beej's guide itself is of course very helpful, if you can tolerate his sense of humor. Compile the files using the GNU C compiler to create the executable files `client`, `server`, `talker`, and `listener`. For example, to create the executable file `client` you'd execute:

```
gcc -o client client.c
```

Learn how to use the `make` command. The file `Makefile` in your MP0 folder is configured to compile the example code provided:

```
make client
make server
make talker
make listener
```

compile the single programs, while

```
make all
```

compiles all of them.

```
make clean
```

reverts the folder to its original state by removing any file created by previous calls to make.

Once you have compiled the code, login to two different machines connected to the network, and execute `client` on one and `server` on the other. This makes a TCP connection. Execute `talker` on one machine and `listener` on the other. This sends a UDP packet. Note that the connection oriented pair, `server` and `client`, use a different port than the datagram oriented pair, `listener` and `talker`. Try using the same port for each pair, and simultaneously run server and listener on one host, and client and talker on another. Do the pairs of programs interfere with each other? Why?

The assignment

Now it's time to get your hands on the code, and write your first networking application. Your application will connect to our server running on:

`cs438.cs.illinois.edu` port: `5900`

and perform an handshake, after which your program will receive and print some data from our server. The communication will happen using a TCP socket (the same you have seen in the *client.c / server.c* pair). The concept of *handshake* is common to many networking applications, in which an initial exchange if information is required to establish a logical connection (e.g., authenticate a user). This must not be confused with the TCP handshake, which is taken care of (thankfully!) by the system libraries. In short the procedure works as follow.

First, begin by establishing a TCP connection to the server and port specified at the beginning of this section. You can use the source code from *client.c* as a guide, but don't simply copy and paste from there, try to memorize the procedure and do it yourself. Once the socket is created, the handshake is performed with the following exchange (**s:** indicates messages sent by our server, **c:** indicates messages sent by your application, '`\n`' indicates a new line character):

```
c: HELO\n
s: 100 – OK\n
c: USERNAME <username>\n
s: 200 – Username: <username>\n
```

Where *username* identifies your client among the multiple connections that the server can handle at the same time (Will the server be confused if you run multiple clients simultaneously and specify the same username, or will it always be able to distinguish the different connections and which one to respond to? How?).
Once your client is successfully registered, it can start retrieving data from the server. To do so, repeat the RECV command, to which the server replies with a random sentence:

```
c: RECV\n
s: 300 – DATA: <some_string>\n
c: RECV\n
s: 300 – DATA: <some_string>\n
…
```

For each received line, your program should print on stdout **exactly** the following line:

```
Received: <some_string>\n
```

where of course <*some string*> is replaced by the sentence received each time. Repeat this operation 10 times, printing each sentence, and then close the connection:

```
c: BYE\n
s: 400 – Bye\n
```

Clean up your variables (close the socket, free any memory you have allocated dynamically) and exit. Congratulations, that was it!

Hints

You will need to have (or quickly acquire) a good knowledge of the ANSI C programming language, including the use of pointers, structures, typedef, and header files. If you have taken CS241 you should already have the necessary background. Get a book on the subject if necessary (the class web page has suggestions. The Beej's guide is a very useful tool in this sense.

Help with C and Unix system calls and commands is provided by the online manuals. For example, executing the statement `man bind` tells you about the system call `bind`, which is one of the fundamental C networking functions. The man pages for a system call tell you about the header files you need to have included in your program to use the system call, and they also specify libraries to link in when the program is compiled.

Network protocols are very strict. Make sure that newlines are added if necessary, and accounted for in received messages. Also remember that you are receiving a bytestream, not a string. What does this imply in C? Make sure that you consider this aspect in your code.

To test connectivity to our server and the protocol, you can open a TCP connection and send and receive strings using a program called `telnet`. This, and its companion `netcat` or `nc` can be precious allies when you are debugging your code.

Hand In

We will grade only the files that are submitted through SVN. Use the name `mp0client.c` for your source code. Also modify the makefile so that execution of the command `make mp0client` causes the executable code for the all the programs to be generated by the compiler. The executable files should be named `mp0client` and should be in the same `MP0` folder. Your program should run with the following command line prototype:

```
mp0client <hostname> <port> <username>
```

Where *hostname* and *port* define the parameters to connect to our server, and *username* is the one that is sent to the server during the handshake (you can use any username you'd like, we will test it with your netID). To run and test your code, we will first call `make mp0client` to compile it, and then call:

~~`./mp0client cs438.cs.illinois.edu 6000 <your_netID>`~~
`./mp0client 52.165.147.13 6500 <your_netID>`

Finally, running `make clean` should delete all executable files and any other temporary file that your makefile or programs create.

To hand in your homework, first add any new file you created and you want to submit by using the command:

```
svn add <filename>
```

This does NOT submit your MP, you need to follow the next step to commit it.
Once you are ready to submit your homework, type the following while in the directory containing the assignment:

```
svn ci –m “”
```

You may commit your MP as many times as you'd like. It's a great way to 'save' the work you've done so far. We will use the last submission you made for the MP when we grade your MP. Once a file has been committed to subversion, it has been submitted. You can verify the files on subversion by viewing your subversion though a browser by going to the following URL:

`https://subversion.ews.illinois.edu/svn/sp18-cs438/<netid>`

A file cannot be graded if it has not been committed. Failure to commit a file on time will result in your MP being considered late. It is your responsibility to ensure all your work is committed by the due date.

Grading

Complying with the instructions is extremely important when writing programs, even more so when these programs are supposed to communicate with other programs that most likely have not been written by you. For this reason in this class we will be very strict about the policies that we define. For example, if the output is not exactly the one we require, there will be a penalty. If you forget a newline, or add too many, if you don't send the username passed in via the command line args, if you do not respect upper and lower case characters, you will loose points. This will happen also if your makefile does not work properly (e.g., clean does not return the folder to its original state). In this particular MP, out of a total of 3 points, you will earn 1.5 points for submitting the assignment, 1.5 points for your program establishing a connection, performing handshake and printing statements. Late policy -2% of total possible points per hour late.

Background

This might seem a *toy* program, with a simplified structure and no real utility, but this is not entirely true. If you are interested in the topic, have a look at the SMTP protocol, to find out how your emails are sent, or the HTTP protocol to see what lies beneath your web browser main window, or the FTP protocol, and you will see that they are not that different from what we have done in this MP. In fact, if you type fast enough, and without any typo, you would be able to send an email using telnet. If you are really good at reading binary and HTML source code, you might even be able to browse a static website.

Appendix – VirtualBox VM Setup

The autograder runs your code in VMs – 64-bit Ubuntu 14.04 Server VMs, running on VirtualBox. Therefore, to test your code, you will need a 64-bit Ubuntu 14.04 VM of your own. (Even if you're already running Ubuntu 14.04 on your personal machine, later assignments will use multiple VMs, so you might as well start using the VM now.)

WARNING: COMPILATION CAN BE A LOT LESS PORTABLE THAN YOU THINK, ESPECIALLY WHEN OSX OR EWS IS INVOLVED.

Please don't assume that it will be ok after testing it only on your personal machine or EWS. (Just don't use EWS at all; it is not well suited to classes that involve networked programming assignments.)

A tutorial for installing Ubuntu on VirtualBox can be found at http://www.psychocats.net/ubuntu/virtualbox. This tutorial is for Windows, but VirtualBox works and looks the same on all OSes. The Ubuntu 14.04 image: http://www.ubuntu.com/download/alternative-downloads. You can use either the desktop or server version.
Starting from here, I'll assume that you have an Ubuntu 14.04 VM running on VirtualBox. One note, though: during the install process, there's a list of typical server programs you can choose to install; might as well take the opportunity to install an ssh server.

Use apt-get (sudo apt-get install xyz) to install any programs you'll need, like gcc, make, gdb, valgrind. I would suggest also getting iperf and tcpdump, which will be useful later.

That's it for now, but we will do more with the VMs in a later assignment.