Machine Problem 0

Introduction to UNIX Network Programming with TCP/IP Sockets

Due: Thursday, Jan 30th, 11:59pm

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 in Ubuntu, 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.

1. Environment Setup

1.1 Ubuntu Virtual Machines

The machine problems in this class are designed for Unix-based machines (in C programming language). Specifically, the auto-grader runs your code in VMs – 64-bit Ubuntu 18.04.1 LTS Desktop VMs. Therefore, to make sure that your submitted code can be compiled and run correctly by the auto-grader, you will want to test your programs in a 64-bit Ubuntu 18.04.1 VM of your own before submitting your code. Even if you're already running Ubuntu 18.04.1 on your personal machine, later assignments will use multiple VMs, so you might as well start using the VM now.

Images of Ubuntu 18.04.1 LTS Desktop can be downloaded at http://old-releases.ubuntu.com/releases/18.04.1/

1.2 Your GitLab Repository

You will use git to maintain your code submission. For this machine problem, each student is provided a private git repository (hosted by UIUC's GitLab service):

```
https://gitlab.engr.illinois.edu/cs438-sp2020/mp0-mp1/<netid>
```

(Please replace <*netid*> with your own netid.) Use your UIUC login credentials to access this repository. We will grade your programs that are pushed to this repository. Feel free to use this repository to maintain your code during development. Note that this repository will be used for both MP0 and MP1, so you must maintain your code in the *mp0* folder for this machine problem.

For more information about submission and grading, please refer to the Section 4 and 5.

2. Practice – Socket Oriented Programming

This step is **optional**, it requires no coding and it is intended for you to get familiar 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.

Download the zip file at the following link and unzip it:

https://courses.engr.illinois.edu/cs438/sp2020/mp/mp0-practice.zip

The unzipped folder 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 this 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?

3. 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 a 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 of 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 follows.

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.

Detailed information for 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 include in your program to use the system call. 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 byte-stream, 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/receive strings using a program called `telnet`. This, and its companion `netcat` or `nc` can be precious allies when you are debugging your code.

4. What to Submit

As mentioned in the beginning, you must submit (i.e., commit and push) your code to the private GitLab repository provided to you in Section 1.2. We will only grade your programs that are pushed to the `master` branch in that repository. In your repository, it must contain the following files:

- `/mp0/mp0client.c` # your source code
- `/mp0/Makefile` # the makefile

Your source code should be in `mp0client.c`. You must create the makefile so that execution of the command

```
make mp0client
```

compiles and generates the executable file named `mp0client`, in the same `mp0` folder. This executable 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, the auto-grader will first call `make mp0client` to compile it, and then call:

```
./mp0client cs438.cs.illinois.edu 5900 <your_netID>
```

Finally, running

```
make clean
```

should delete all executable files and any temporary files that the makefile or your program creates.

You may commit and push your code as many times as you like before the deadline. It's in fact a good practice to commit your code whenever there is a change that is worth noted. We will use the last commit you made for the repository when we grade your MP.

Your program cannot be graded if it has not been pushed to the repository. Failure to commit and push your code on time will result in your MP being considered late. It is your responsibility to ensure all your work is committed and pushed to your repository by the due date.

5. Grading

- 1.5 points: if your assignment is submitted
- 1.5 points: if your program compiles, establishes a connection, performs handshaking and prints correctly
- Late penalty: 2% of total possible score per hour

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 lose points. This will happen also if your makefile does not work properly (e.g., clean does not return the folder to its original state).

6. Epilogue

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.

7. Important Notes

1) You must use C or C++.
2) Your code must be runnable in 64-bit Ubuntu 18.04.1 LTS Desktop VMs. This is the environment in which the auto-grader runs. You program must fulfill the given assignment requirements to get said scores.
3) If you need to use a library for data structures, you MUST get the approval of the course staff. Additionally, you MUST acknowledge the source in a README in *mp0* folder. However, algorithms MUST be your own.
4) The auto-grader will test every student's current version every night, starting at 6PM. The TAs will try their best to make sure this is the case, but bugs happen. We will make an announcement of its starting date.
   - Do a git pull the next morning to see your score in *mp0/result.txt*.
   - The score you are given is hypothetical and has no effect on your final grade.
   - The late penalty is applied to the last git commit you make for anything in the folder *mp0*. If you want to make any change in *mp0* that does not change your code, please wait until 48 hours after the deadline to make it.
5) Do not make your private repository public. You will be held partially responsible for any resultant plagiarism.
6) You must work alone (for MP0 and MP1). Your code must be your own. You can discuss very general concepts with others, but if you find yourself looking at a screen/whiteboard full of pseudo code (let alone real code), you are going too far.
   - Refer to the class slides and official student handbook for academic integrity policy. In summary, the standard for guilt is "more probable than nor probable", and penalties range from warning to recommending suspension/expulsion, based entirely on the instructor's impression of the situation.
   - The College of Engineering has some guidelines for penalties that we think are reasonable, but we reserve the right to ignore them when appropriate.