ECE 4122/6122 Hmk #4

(100 pts)

Section	Due Date
89313 - ECE 4122 - A	Oct 20th, 2019 by 11:59 PM
89314 - ECE 6122 - A	Oct 20th, 2019 by 11:59 PM
89340 - ECE 6122 - Q	Oct 23th, 2019 by 11:59 PM
89706 - ECE 6122 - QSZ	Oct 23th, 2019 by 11:59 PM

ECE 4122 students need to do Problem 1. **ECE 6122** students need to do Problem 2.

Notes:

You can write, debug and test your code locally on your personal computer. However, the code you submit must compile and run correctly on the PACE-ICE server.

In this homework you will need to develop ***two*** executable programs. One program for the server and one for the client.

Submitting the Assignment:

See Appendix C.

Grading Rubric

AUTOMATIC GRADING POINT DEDUCTIONS PER PROBLEM:

Element	Percentage Deduction	Details
Does Not Compile	40%	Code does not compile on PACE-ICE!
Does Not Match Output	10%-90%	The code compiles but does not produce correct outputs.
Clear Self-Documenting Coding Styles	10%-25%	This can include incorrect indentation, using unclear variable names, unclear/missing comments, or compiling with warnings. (See Appendix A)

LATE POLICY

Element	Percentage Deduction	Details
Late Deduction Function	score - (20/24)*H	H = number of hours (ceiling function) passed deadline note : Sat/Sun count as one day; therefore $H = 0.5*H_{weekend}$

Problem #1 TCP Sockets

Server-70 points

Write a console program that takes as a **command line argument** the **port number** on which the **TCP Server** will listen for connection requests. A separate thread shall be created to handle the data received from each remote client and the remote clients can continue to send and receive data on the connections until either the server or the client closes the connection. The TCP server needs to maintain a list of all connected clients so that it can send out the appropriate messages. The TCP server needs to be able to receive data from clients without blocking the main application thread. The program needs to respond to user input while handling socket communications at the same time.

1) The program should continuously prompt the user for commands to execute like so:

Please enter command: 0 - prints to the console the last message received (if any)

Last Message: *last message received*

Please enter command: 1 - prints to the console a list of all connected clients (ip address and port number) like the following:

```
Numer of Clients: 2

IP Address      Port

localhost      51717

localhost      51718
```

Please enter command: q - closes all sockets and terminates the program

2) The data being sent back and forth will use the following packet structure:

```
struct tcpMessage
{
  unsigned char nVersion;
  unsigned char nType;
  unsigned short nMsgLen;
  char chMsg[1000];
};
```

The TCP server needs to have the following functionality for received messages:

- If **nVersion** is not equal to 1 then the message is ignored.

- If **nType == 0** then the message should be sent to all other connected clients exactly as received (but <u>not</u> the client that sent the message).

- If **nType == 1** then the message is reversed and sent back to **only** the client that sent the message.

Client-30 points

In order to test your server, write a console program that takes as a **command line argument** the **IP Address** and **port number** of the server as shown below:

./a.out localhost 51717

The program should prompt the user for inputs and display any messages received.

Here are example user inputs:

Please enter command: v #	the user enters a "v", a space, and then a version number. This version number is now used in all new messages.
Please enter command: t # message string	the user enters a "t", a space, and then a type number, followed by the message. Be sure you are able to handle the spaces in the message string.
Please enter command: q	the user enters a "q" causes the socket to be closed and the program to terminate.

Any messages received from the server should be displayed as followed:

Received Msg Type: #**; Msg:** *message received*

Problem #2 UDP Sockets

UDP Server (60 points)

Write a console program that takes as a **command line argument** the **port number** on which the UDP Server will receive messages. The UDP server collects parts of a larger **composite message** from the clients. The UDP server collects these message parts from different clients and assembles them in order, based on the **`lSeqNum`**. The UDP server keeps a running list of all clients that have sent a message to it and will broadcast the composite message to all clients when commanded. The UDP server needs to be able to receive data from clients without blocking the main application thread. The program needs to respond to user input while handling socket communications at the same time.

The program should continuously prompt the user for commands to execute like so:

Please enter command: 0

Please enter command: 1

Please enter command: 2

Composite Msg: *current composite message*

The data being sent back and forth will use the following packet structure:

```
struct udpMessage
{
  unsigned char  nVersion;
  unsigned char  nType;
  unsigned short nMsgLen;
  unsigned long  lSeqNum;
  char chMsg[1000];
};
```

The UDP server needs to have the following functionality from the **command prompt**:

- If **command** == 0 the server immediately sends to all clients the current composite message and clears out the composite message.
- If **command** == 1 the server immediately clears out the composite message.
- If **command** == 2 the server immediately displays to the console the composite message but takes no other actions.

The UDP server needs to have the following functionality when **receiving messages**:

- If **nVersion** is not equal to 1 then the message is ignored.

- If **nType == 0** the composite message is immediately cleared and anything in the chMsg buffer is ignored.
- If **nType == 1** the composite message is immediately cleared and the message in chMsg is used as the start of a new composite message
- If **nType == 2** the message in chMsg is added to the composite message based on its **lSeqNum**
- If **nType == 3** the server immediately sends to all clients the current composite message and clears out the composite message.

If the composite message becomes larger than 1000 then the composite message (up to 1000) is immediately set out to all clients and any remaining characters are used to start a new composite message with a `lSeqNum` = 0.

UDP Client (40 points)

In order to test your server, write a console program that takes as a **command line argument** the **IP Address** and **port number** of the server as shown below:

./a.out localhost 51717

The program should prompt the user for inputs and display any messages received.

Here are example user inputs:

Please enter command: v #	the user enters a "v", a space, and then a version number. This version number is now used in all new messages.
Please enter command: t # # **message string**	the user enters a "t", a space, and then a **type** number, and then a **sequence** number, followed by the **message** (if any). Be sure you are able to handle the spaces in the message.
Please enter command: q	the user enters a "q" causes the socket to be closed and the program to terminate.

Any messages received should be displayed as followed:

Received Msg Type: 1, Seq: 33, Msg: *message received*

Appendix A: Coding Standards

Indentation:

When using *if/for/while* statements, make sure you indent 4 spaces for the content inside those. Also make sure that you use spaces to make the code more readable.
For example:

```
for (int i; i < 10; i++)
{
    j = j + i;
}
```

If you have nested statements, you should use multiple indentions. Each { should be on its own line (like the *for* loop) If you have *else* or *else if* statements after your *if* statement, they should be on their own line.

```
for (int i; i < 10; i++)
{
   if (i < 5)
   {
       counter++;
       k -= i;
   }
   else
   {
       k +=1;
   }
   j += i;
}
```

Camel Case:

This naming convention has the first letter of the variable be lower case, and the first letter in each new word be capitalized (e.g. firstSecondThird). This applies for functions and member functions as well! The main exception to this is class names, where the first letter should also be capitalized.

Variable and Function Names:

Your variable and function names should be clear about what that variable or function is. Do not use one letter variables, but use abbreviations when it is appropriate (for example: “imag" instead of “imaginary”). The more descriptive your variable and function names are, the more readable your code will be. This is the idea behind self-documenting code.

__File Headers__:
Every file should have the following header at the top

```
/*
Author: <your name>
Class: ECE4122 or ECE6122
Last Date Modified: <date>

Description:

What is the purpose of this file?

*/
```

__Code Comments:__

1. Every function must have a comment section describing the purpose of the function, the input and output parameters, the return value (if any).
2. Every class must have a comment section to describe the purpose of the class.
3. Comments need to be placed inside of functions/loops to assist in the understanding of the flow of the code.

Appendix B: Accessing PACE-ICE Instructions

ACCESSING LINUX PACE-ICE CLUSTER (SERVER)

To access the PACE-ICE cluster you need certain software on your laptop or desktop system, as described below.

Windows Users:

Option 0 (Using SecureCRT)- THIS IS THE EASIEST OPTION!

The Georgia Tech Office of Information Technology (*OIT)* maintains a web page of software that can be downloaded and installed by students and faculty. That web page is:

http://software.oit.gatech.edu

From that page you will need to install SecureCRT.

To access this software, you will first have to log in with your Georgia Tech user name and password, then answer a series of questions regarding export controls.

Connecting using SecureCRT should be easy.

- Open SecureCRT, you'll be presented with the "Quick Connect" screen.
- Choose protocol "ssh2".
- Enter the name of the PACE machine you wish to connect to in the "HostName" box (i.e. *coc-ice.pace.gatech.edu*)
- Type your username in the "Username" box.
- Click "Connect".
- A new window will open, and you'll be prompted for your password.

Option 1 (Using Ubuntu for Windows 10):

Option 1 uses the Ubuntu on Windows program. This can only be downloaded if you are running Windows 10 or above. If using Windows 8 or below, use Options 2 or 3. It also requires the use of simple bash commands.

1. Install Ubuntu for Windows 10 by following the guide from the following link:

   https://msdn.microsoft.com/en-us/commandline/wsl/install-win10

2. Once Ubuntu for Windows 10 is installed, open it and type the following into the command line:

   *ssh **YourGTUsername**@coc-ice.pace.gatech.edu* where **YourGTUsername** is

   replaced with your alphanumeric GT login. Ex: bkim334

3. When it asks if you’re sure you want to connect, type in:
   yes

and type in your password when prompted (Note: When typing in your password, it will not show any characters typing)

4. You're now connected to PACE-ICE. You can edit files using vim by typing in:

   vi filename.cc OR *nano vilename.cpp*

   For a list of vim commands, use the following link:

   https://coderwall.com/p/adv71w/basic-vim-commands-for-getting-started

5. You're able to edit, compile, run, and submit your code from this command line.

Option 2 (Using PuTTY):

Option 2 uses a program called PuTTY to ssh into the PACE-ICE cluster. It is easier to set up, but doesn't allow you to access any local files from the command line. It also requires the use of simple bash commands.

1. Download and install PuTTY from the following link: www.putty.org

2. Once installed, open PuTTY and for the Host Name, type in:
   coc-ice.pace.gatech.edu and

   for the port, leave it as 22.

3. Click Open and a window will pop up asking if you trust the host. Click Yes and it will then ask you for your username and password. (Note: When typing in your password, it will not show any characters typing)

4. You're now connected to PACE-ICE. You can edit files using vim by typing in: *vim filename.cc* OR *nano vilename.cpp*

   For a list of vim commands, use the following link:

   https://coderwall.com/p/adv71w/basic-vim-commands-for-getting-started

5. You're able to edit, compile, run, and submit your code from this command line.

MacOS Users:.

Option 0 (Using the Terminal to SSH into PACE-ICE):

This option uses the built-in terminal in MacOS to ssh into PACE-ICE and use a command line text editor to edit your code.

1. Open Terminal from the Launchpad or Spotlight Search.

2. Once you're in the terminal, ssh into PACE-ICE by typing:

   *ssh **YourGTUsername**@ coc-ice.pace.gatech.edu* where **YourGTUsername** is

   replaced with your alphanumeric GT login. Ex: bkim334

3. When it asks if you're sure you want to connect, type in:
   yes

   and type in your password when prompted (Note: When typing in your password, it will not show any characters typing)

4. You're now connected to PACE-ICE. You can edit files using vim by typing in:

   vi filename.cc OR *nano filename.cpp*

   For a list of vim commands, use the following link: https://coderwall.com/p/adv71w/basic-vim-commands-for-getting-started

5. You're able to edit, compile, run, and submit your code from this command line.

<u>Linux Users:</u>

If you're using Linux, follow Option 0 for MacOS users.

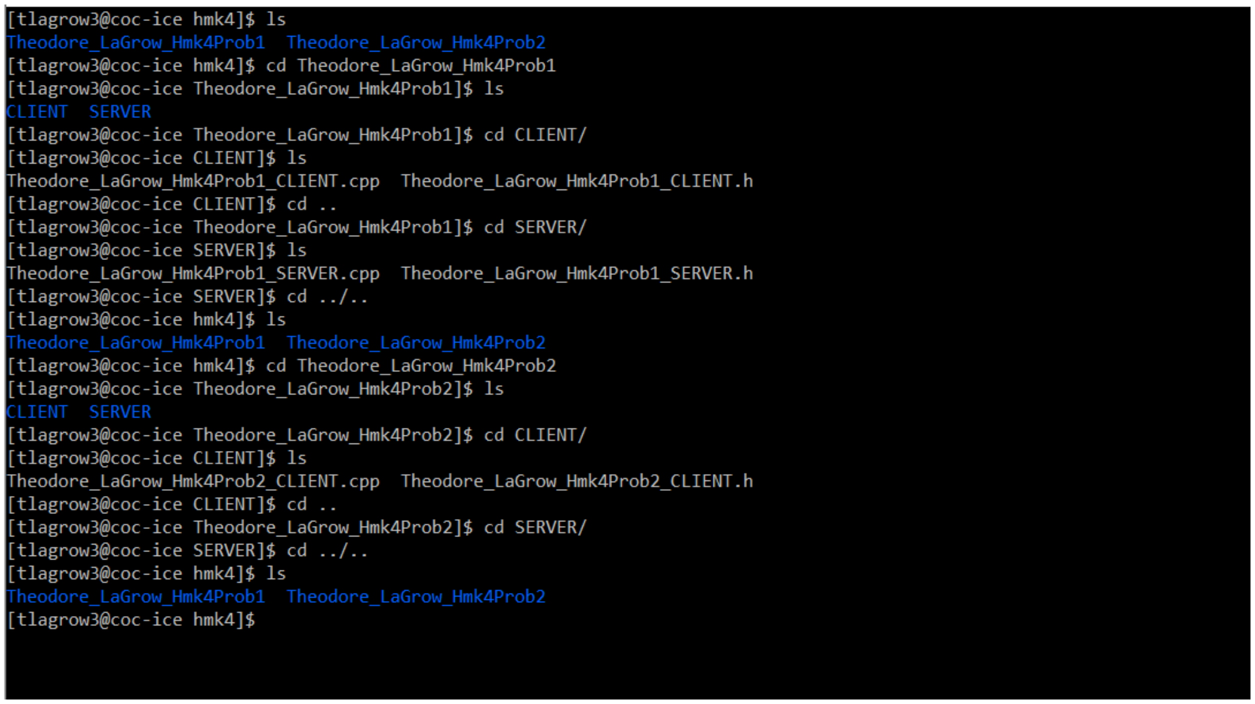
Appendix B: Submitting Assignment

Step-By-Step Submitting a Tar.gz

PLEASE NOTE: All the following instructions are run on PACE. If you are struggling with any of the steps, please come see the TAs during office hours. Better to start and test early than wait until the last minute.

Below is a step-by-step tutorial on how to create the .tgz file for Homework 4 for ECE 4122/6122. Please know, if you are in 4122 you do not need to submit anything for Problem 2. Additionally, if you do not use a header file for a solution, you do not need to submit a header file. Please adjust these instructions accordingly. The tarball should contain a folder with your buzzid containing subfolders for each problem you are assigned with the corresponding .cpp and .h files. Each problem will have a SERVER and CLIENT folder with corresponding code.

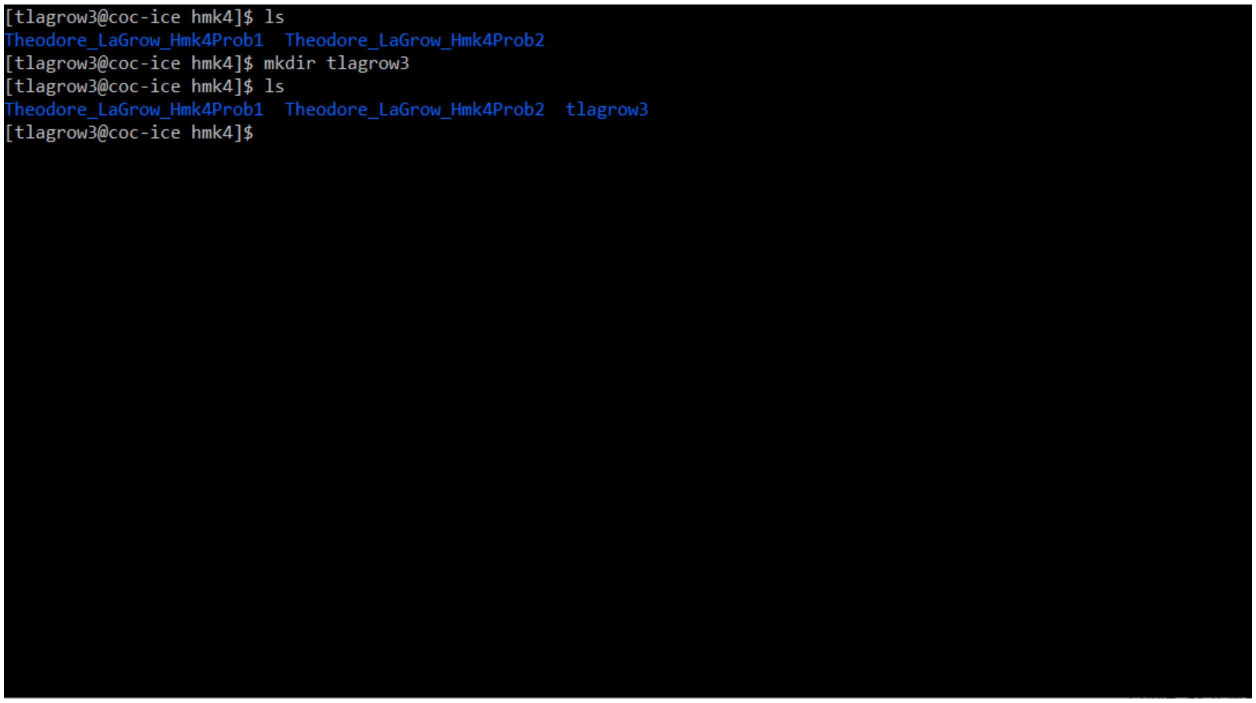
Please make sure that each problem is in its own subfolder with the naming conventions:
<FirstName_LastName>_Hmk4Prob# (where # is either 1 or 2) in addition to the SERVER and CLIENT folders:



```
[tlagrow3@coc-ice hmk4]$ ls
Theodore_LaGrow_Hmk4Prob1  Theodore_LaGrow_Hmk4Prob2
[tlagrow3@coc-ice hmk4]$ cd Theodore_LaGrow_Hmk4Prob1
[tlagrow3@coc-ice Theodore_LaGrow_Hmk4Prob1]$ ls
CLIENT  SERVER
[tlagrow3@coc-ice Theodore_LaGrow_Hmk4Prob1]$ cd CLIENT/
[tlagrow3@coc-ice CLIENT]$ ls
Theodore_LaGrow_Hmk4Prob1_CLIENT.cpp  Theodore_LaGrow_Hmk4Prob1_CLIENT.h
[tlagrow3@coc-ice CLIENT]$ cd ..
[tlagrow3@coc-ice Theodore_LaGrow_Hmk4Prob1]$ cd SERVER/
[tlagrow3@coc-ice SERVER]$ ls
Theodore_LaGrow_Hmk4Prob1_SERVER.cpp  Theodore_LaGrow_Hmk4Prob1_SERVER.h
[tlagrow3@coc-ice SERVER]$ cd ../..
[tlagrow3@coc-ice hmk4]$ ls
Theodore_LaGrow_Hmk4Prob1  Theodore_LaGrow_Hmk4Prob2
[tlagrow3@coc-ice hmk4]$ cd Theodore_LaGrow_Hmk4Prob2
[tlagrow3@coc-ice Theodore_LaGrow_Hmk4Prob2]$ ls
CLIENT  SERVER
[tlagrow3@coc-ice Theodore_LaGrow_Hmk4Prob2]$ cd CLIENT/
[tlagrow3@coc-ice CLIENT]$ ls
Theodore_LaGrow_Hmk4Prob2_CLIENT.cpp  Theodore_LaGrow_Hmk4Prob2_CLIENT.h
[tlagrow3@coc-ice CLIENT]$ cd ..
[tlagrow3@coc-ice Theodore_LaGrow_Hmk4Prob2]$ cd SERVER/
[tlagrow3@coc-ice SERVER]$ cd ../..
[tlagrow3@coc-ice hmk4]$ ls
Theodore_LaGrow_Hmk4Prob1  Theodore_LaGrow_Hmk4Prob2
[tlagrow3@coc-ice hmk4]$
```

Create a subdirectory named *buzzid* in the current working directory by executing the following command in the shell:
$ mkdir *buzzed*

(In this case my *buzzid* is tlagrow3)

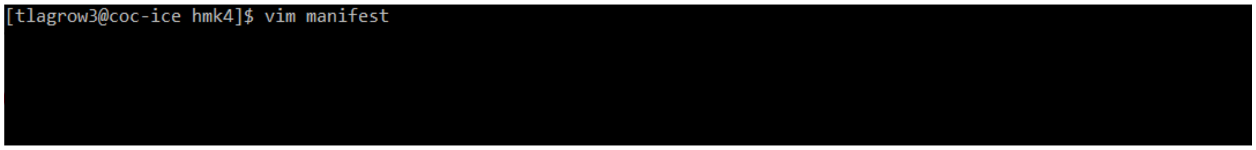


```
[tlagrow3@coc-ice hmk4]$ ls
Theodore_LaGrow_Hmk4Prob1  Theodore_LaGrow_Hmk4Prob2
[tlagrow3@coc-ice hmk4]$ mkdir tlagrow3
[tlagrow3@coc-ice hmk4]$ ls
Theodore_LaGrow_Hmk4Prob1  Theodore_LaGrow_Hmk4Prob2  tlagrow3
[tlagrow3@coc-ice hmk4]$
```

Create a text file named **manifest.**

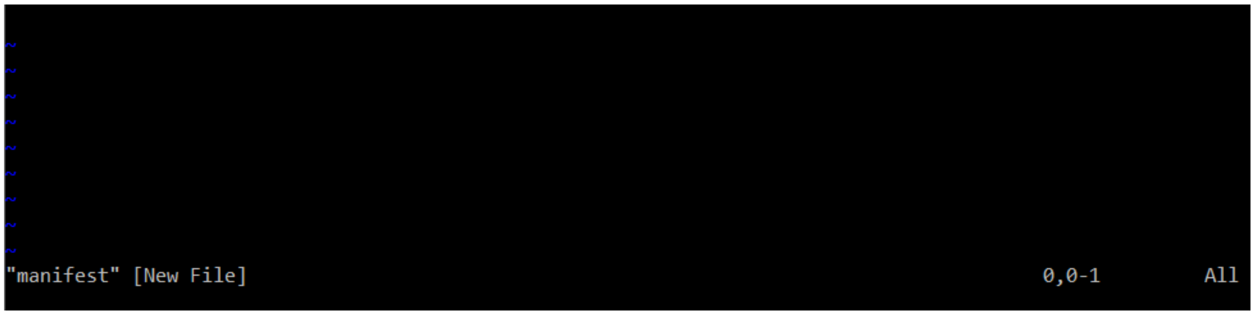
Please make sure that the **manifest** file is a plain text file and not a rich text file. Microsoft word will generate a rich text file. The easiest method to create a plain text file is to use a command line editor such as vi, vim, or nano.

(If you want a tutorial, these can be usefule: vi: http://heather.cs.ucdavis.edu/~matloff/UnixAndC/Editors/ViIntro.html, vim: https://openvim.com/, nano: https://www.howtogeek.com/howto/42980/the-beginners-guide-to-nano-the-linux-command-line-text-editor/)



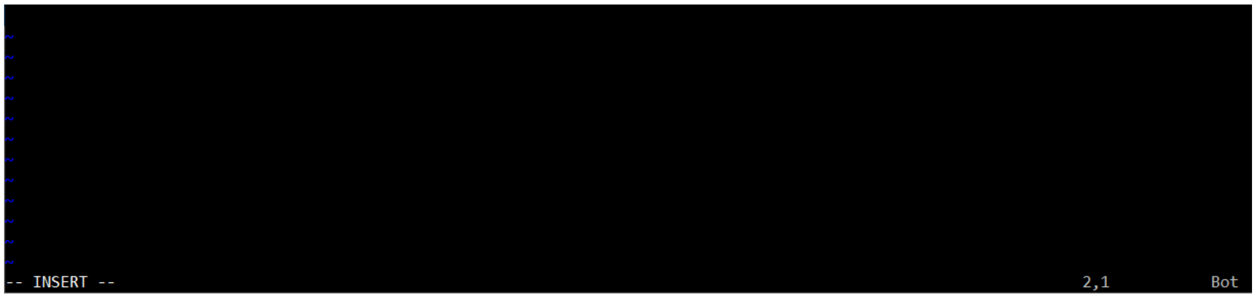
```
[tlagrow3@coc-ice hmk4]$ vim manifest
```

The editor will open:



```
~
~
~
~
~
~
~
~
~
"manifest" [New File]                                       0,0-1         All
```

You will need to insert test into this file. Press the 'i' key on the keyboard to insert text:

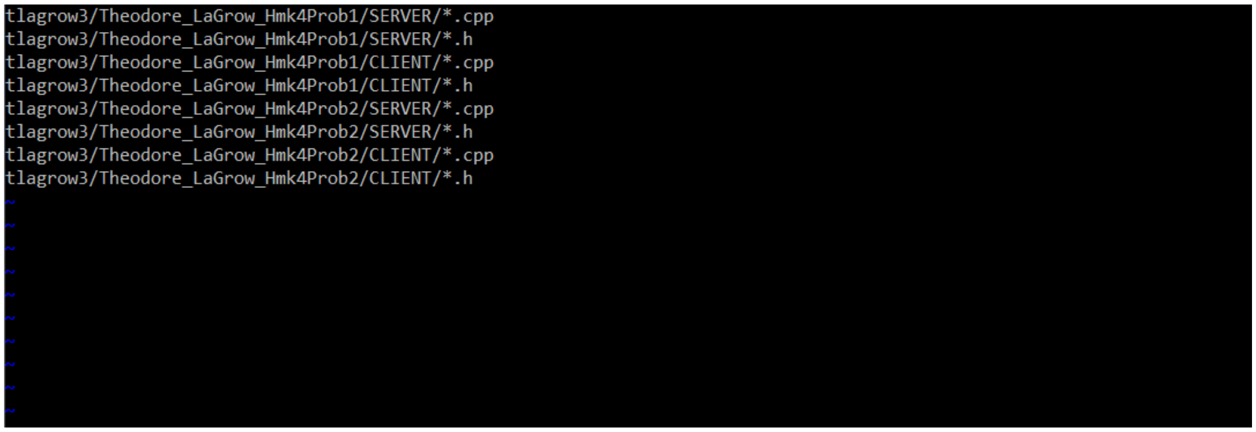


```
~
~
~
~
~
~
~
~
~
~
~
~
-- INSERT --                                                                          2,1           Bot
```

Populate the **manifest** file with the following:

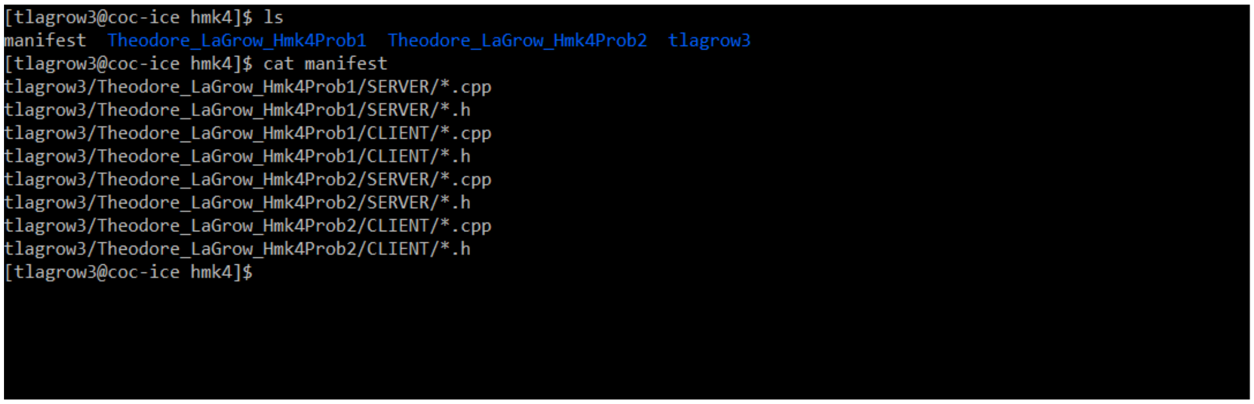
buzzid/<FirstName_LastName>_Hmk4Prob1/SERVER/*.cpp
buzzid/<FirstName_LastName>_Hmk4Prob1/SERVER/*.h
buzzid/<FirstName_LastName>_Hmk4Prob1/CLIENT/*.cpp
buzzid/<FirstName_LastName>_Hmk4Prob1/CLIENT/*.h
buzzid/<FirstName_LastName>_Hmk4Prob2/SERVER/*.cpp
buzzid/<FirstName_LastName>_Hmk4Prob2/SERVER/*.h
buzzid/<FirstName_LastName>_Hmk4Prob2/CLIENT/*.cpp
buzzid/<FirstName_LastName>_Hmk4Prob2/CLIENT/*.h

(***PLEASE NOTE***: this is subject to change with depending on your class section. The wildcard (*) character will grab all the .cpp and .h files you generated for each problem.)



```
tlagrow3/Theodore_LaGrow_Hmk4Prob1/SERVER/*.cpp
tlagrow3/Theodore_LaGrow_Hmk4Prob1/SERVER/*.h
tlagrow3/Theodore_LaGrow_Hmk4Prob1/CLIENT/*.cpp
tlagrow3/Theodore_LaGrow_Hmk4Prob1/CLIENT/*.h
tlagrow3/Theodore_LaGrow_Hmk4Prob2/SERVER/*.cpp
tlagrow3/Theodore_LaGrow_Hmk4Prob2/SERVER/*.h
tlagrow3/Theodore_LaGrow_Hmk4Prob2/CLIENT/*.cpp
tlagrow3/Theodore_LaGrow_Hmk4Prob2/CLIENT/*.h
~
~
~
~
~
~
~
~
~
~
```

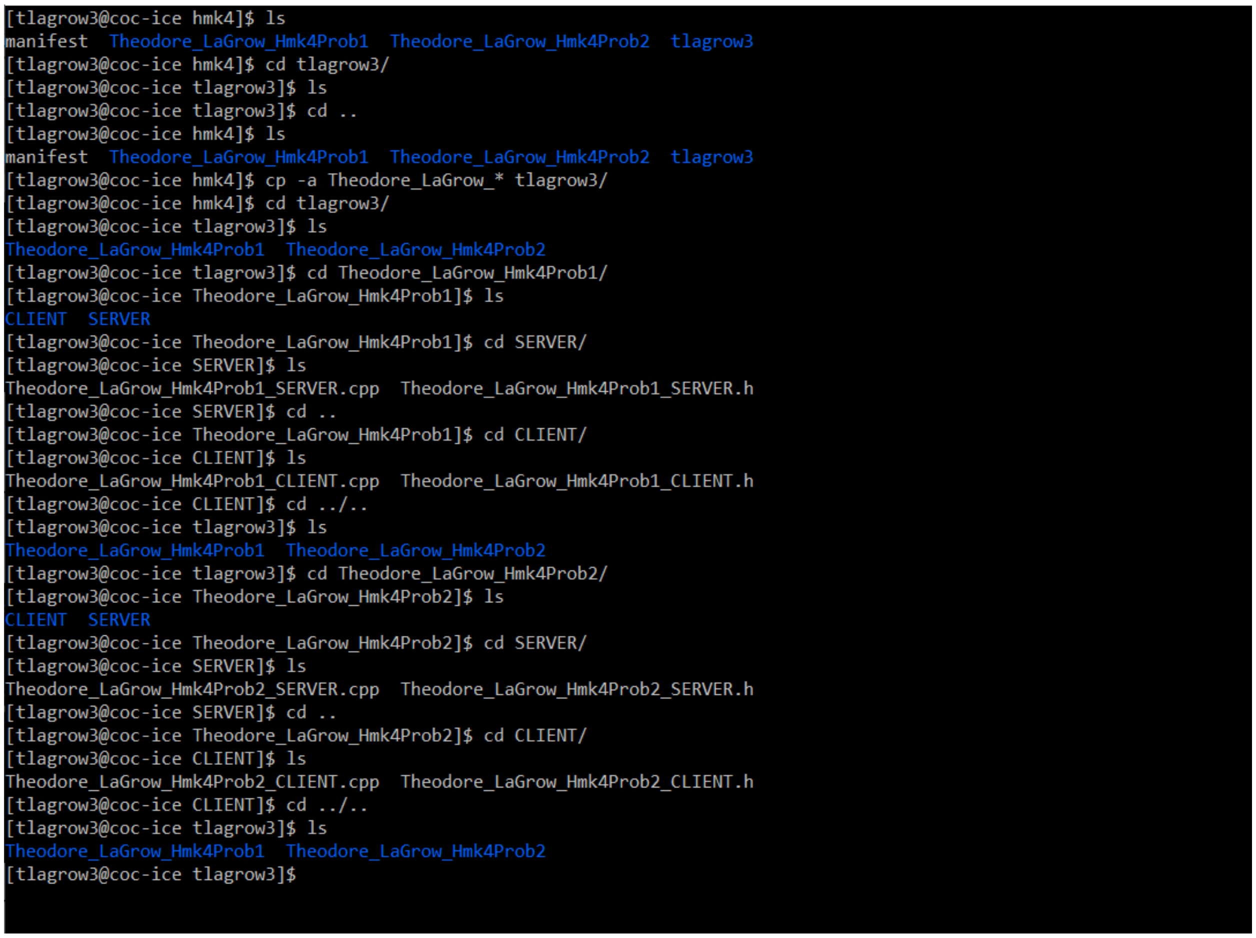
To save the file, hold the Shift key and press ‘ZZ’. This command will save the text and exit in the vim text editor. To check that the file saved, you can use the command ‘$cat <filename>’ to print out all the lines:



```
[tlagrow3@coc-ice hmk4]$ ls
manifest  Theodore_LaGrow_Hmk4Prob1  Theodore_LaGrow_Hmk4Prob2  tlagrow3
[tlagrow3@coc-ice hmk4]$ cat manifest
tlagrow3/Theodore_LaGrow_Hmk4Prob1/SERVER/*.cpp
tlagrow3/Theodore_LaGrow_Hmk4Prob1/SERVER/*.h
tlagrow3/Theodore_LaGrow_Hmk4Prob1/CLIENT/*.cpp
tlagrow3/Theodore_LaGrow_Hmk4Prob1/CLIENT/*.h
tlagrow3/Theodore_LaGrow_Hmk4Prob2/SERVER/*.cpp
tlagrow3/Theodore_LaGrow_Hmk4Prob2/SERVER/*.h
tlagrow3/Theodore_LaGrow_Hmk4Prob2/CLIENT/*.cpp
tlagrow3/Theodore_LaGrow_Hmk4Prob2/CLIENT/*.h
[tlagrow3@coc-ice hmk4]$
```

Now you need to construct the correct file structure for the submission. This means that you need to put all the homework files into your *buzzid* folder. Execute the following lines in the shell:

$ cp -a <FirstName_LastName>_* *buzzid*



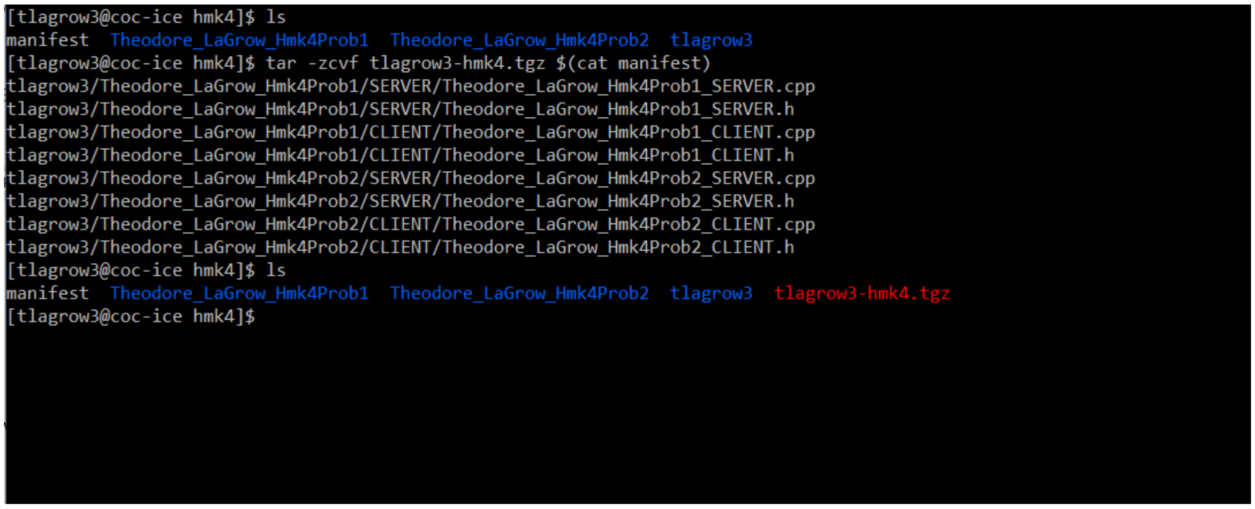
```
[tlagrow3@coc-ice hmk4]$ ls
manifest  Theodore_LaGrow_Hmk4Prob1  Theodore_LaGrow_Hmk4Prob2  tlagrow3
[tlagrow3@coc-ice hmk4]$ cd tlagrow3/
[tlagrow3@coc-ice tlagrow3]$ ls
[tlagrow3@coc-ice tlagrow3]$ cd ..
[tlagrow3@coc-ice hmk4]$ ls
manifest  Theodore_LaGrow_Hmk4Prob1  Theodore_LaGrow_Hmk4Prob2  tlagrow3
[tlagrow3@coc-ice hmk4]$ cp -a Theodore_LaGrow_* tlagrow3/
[tlagrow3@coc-ice hmk4]$ cd tlagrow3/
[tlagrow3@coc-ice tlagrow3]$ ls
Theodore_LaGrow_Hmk4Prob1  Theodore_LaGrow_Hmk4Prob2
[tlagrow3@coc-ice tlagrow3]$ cd Theodore_LaGrow_Hmk4Prob1/
[tlagrow3@coc-ice Theodore_LaGrow_Hmk4Prob1]$ ls
CLIENT  SERVER
[tlagrow3@coc-ice Theodore_LaGrow_Hmk4Prob1]$ cd SERVER/
[tlagrow3@coc-ice SERVER]$ ls
Theodore_LaGrow_Hmk4Prob1_SERVER.cpp  Theodore_LaGrow_Hmk4Prob1_SERVER.h
[tlagrow3@coc-ice SERVER]$ cd ..
[tlagrow3@coc-ice Theodore_LaGrow_Hmk4Prob1]$ cd CLIENT/
[tlagrow3@coc-ice CLIENT]$ ls
Theodore_LaGrow_Hmk4Prob1_CLIENT.cpp  Theodore_LaGrow_Hmk4Prob1_CLIENT.h
[tlagrow3@coc-ice CLIENT]$ cd ../..
[tlagrow3@coc-ice tlagrow3]$ ls
Theodore_LaGrow_Hmk4Prob1  Theodore_LaGrow_Hmk4Prob2
[tlagrow3@coc-ice tlagrow3]$ cd Theodore_LaGrow_Hmk4Prob2/
[tlagrow3@coc-ice Theodore_LaGrow_Hmk4Prob2]$ ls
CLIENT  SERVER
[tlagrow3@coc-ice Theodore_LaGrow_Hmk4Prob2]$ cd SERVER/
[tlagrow3@coc-ice SERVER]$ ls
Theodore_LaGrow_Hmk4Prob2_SERVER.cpp  Theodore_LaGrow_Hmk4Prob2_SERVER.h
[tlagrow3@coc-ice SERVER]$ cd ..
[tlagrow3@coc-ice Theodore_LaGrow_Hmk4Prob2]$ cd CLIENT/
[tlagrow3@coc-ice CLIENT]$ ls
Theodore_LaGrow_Hmk4Prob2_CLIENT.cpp  Theodore_LaGrow_Hmk4Prob2_CLIENT.h
[tlagrow3@coc-ice CLIENT]$ cd ../..
[tlagrow3@coc-ice tlagrow3]$ ls
Theodore_LaGrow_Hmk4Prob1  Theodore_LaGrow_Hmk4Prob2
[tlagrow3@coc-ice tlagrow3]$
```

Many people have had issue with this step. You need to make sure the naming conventions are consistent to avoid potential problems.

It is now time to tarball your submission. If all the other steps have gone smoothly, execute the following command:

$ tar -zcvf *buzzid*-hmk4.tgz $(cat manifest)

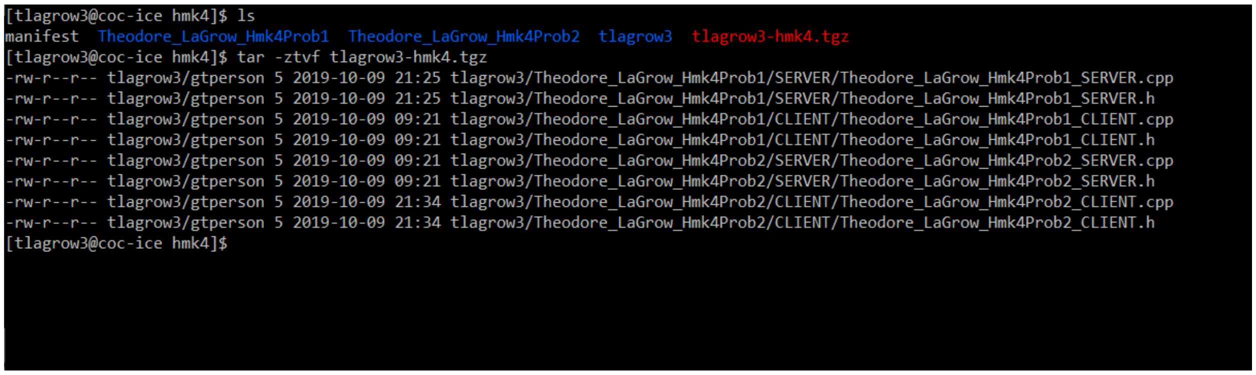
This command will generate a new tgz file with all of the specified files and folders in the manifest text document:



```
[tlagrow3@coc-ice hmk4]$ ls
manifest  Theodore_LaGrow_Hmk4Prob1  Theodore_LaGrow_Hmk4Prob2  tlagrow3
[tlagrow3@coc-ice hmk4]$ tar -zcvf tlagrow3-hmk4.tgz $(cat manifest)
tlagrow3/Theodore_LaGrow_Hmk4Prob1/SERVER/Theodore_LaGrow_Hmk4Prob1_SERVER.cpp
tlagrow3/Theodore_LaGrow_Hmk4Prob1/SERVER/Theodore_LaGrow_Hmk4Prob1_SERVER.h
tlagrow3/Theodore_LaGrow_Hmk4Prob1/CLIENT/Theodore_LaGrow_Hmk4Prob1_CLIENT.cpp
tlagrow3/Theodore_LaGrow_Hmk4Prob1/CLIENT/Theodore_LaGrow_Hmk4Prob1_CLIENT.h
tlagrow3/Theodore_LaGrow_Hmk4Prob2/SERVER/Theodore_LaGrow_Hmk4Prob2_SERVER.cpp
tlagrow3/Theodore_LaGrow_Hmk4Prob2/SERVER/Theodore_LaGrow_Hmk4Prob2_SERVER.h
tlagrow3/Theodore_LaGrow_Hmk4Prob2/CLIENT/Theodore_LaGrow_Hmk4Prob2_CLIENT.cpp
tlagrow3/Theodore_LaGrow_Hmk4Prob2/CLIENT/Theodore_LaGrow_Hmk4Prob2_CLIENT.h
[tlagrow3@coc-ice hmk4]$ ls
manifest  Theodore_LaGrow_Hmk4Prob1  Theodore_LaGrow_Hmk4Prob2  tlagrow3  tlagrow3-hmk4.tgz
[tlagrow3@coc-ice hmk4]$
```

You can now check the new tgz file just generated with the following command:

$ tar -ztvf *buzzid*-hmk4.tgz



```
[tlagrow3@coc-ice hmk4]$ ls
manifest  Theodore_LaGrow_Hmk4Prob1  Theodore_LaGrow_Hmk4Prob2  tlagrow3  tlagrow3-hmk4.tgz
[tlagrow3@coc-ice hmk4]$ tar -ztvf tlagrow3-hmk4.tgz
-rw-r--r-- tlagrow3/gtperson 5 2019-10-09 21:25 tlagrow3/Theodore_LaGrow_Hmk4Prob1/SERVER/Theodore_LaGrow_Hmk4Prob1_SERVER.cpp
-rw-r--r-- tlagrow3/gtperson 5 2019-10-09 21:25 tlagrow3/Theodore_LaGrow_Hmk4Prob1/SERVER/Theodore_LaGrow_Hmk4Prob1_SERVER.h
-rw-r--r-- tlagrow3/gtperson 5 2019-10-09 09:21 tlagrow3/Theodore_LaGrow_Hmk4Prob1/CLIENT/Theodore_LaGrow_Hmk4Prob1_CLIENT.cpp
-rw-r--r-- tlagrow3/gtperson 5 2019-10-09 09:21 tlagrow3/Theodore_LaGrow_Hmk4Prob1/CLIENT/Theodore_LaGrow_Hmk4Prob1_CLIENT.h
-rw-r--r-- tlagrow3/gtperson 5 2019-10-09 09:21 tlagrow3/Theodore_LaGrow_Hmk4Prob2/SERVER/Theodore_LaGrow_Hmk4Prob2_SERVER.cpp
-rw-r--r-- tlagrow3/gtperson 5 2019-10-09 09:21 tlagrow3/Theodore_LaGrow_Hmk4Prob2/SERVER/Theodore_LaGrow_Hmk4Prob2_SERVER.h
-rw-r--r-- tlagrow3/gtperson 5 2019-10-09 21:34 tlagrow3/Theodore_LaGrow_Hmk4Prob2/CLIENT/Theodore_LaGrow_Hmk4Prob2_CLIENT.cpp
-rw-r--r-- tlagrow3/gtperson 5 2019-10-09 21:34 tlagrow3/Theodore_LaGrow_Hmk4Prob2/CLIENT/Theodore_LaGrow_Hmk4Prob2_CLIENT.h
[tlagrow3@coc-ice hmk4]$
```

If your file has the same structure as above, you are all good to submit!