ECE 4122/6122 Lab 5: TCP Sockets

(100 pts)

Category: sockets
Due: Tuesday November 21st, 2023 by 11:59 PM

Objective:

Implement a TCP server and TCP client application that can be used to receive and send messages from multiple clients.

Description:

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. You are free to use the SFML socket sample as a starting point.

Server-60 points

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

> **Please enter command: msg** - prints to the console the last message received (if any)
> **Last Message:** *last message received*
>
> **Please enter command: clients** - prints to the console a list of all connected clients (ip address and port number) like the following:
>
> Number of Clients: 2
> IP Address: localhost | Port: 51717
> IP Address: localhost | Port: 51718
>
> **Please enter command: exit** - 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 102 then the message is ignored.

- If **nType == 77** then the message should be sent to all other connected clients exactly as received (but **not** the client that sent the message).
- If **nType == 201** then the message is reversed and sent back to **only** the client that sent the message.

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, 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*

Turn-In Instructions

Zip up your server file(s) into **Lab5Server.zip** and your client files into **Lab5Client.zip** upload these zip files on the assignment section of Canvas.

Grading Rubric:

If a student's program runs correctly and produces the desired output, the student has the potential to get a 100 on his or her homework; however, TA's will look through your code for other elements needed to meet the lab requirements. The table below shows typical deductions that could occur.

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	Up to 90%	The code compiles but does not produce correct outputs.
Clear Self-Documenting Coding Styles	Up to 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 – 0.5 * H	H = number of hours (ceiling function) passed deadline

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 represents. 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 (section)
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.