ECE 4122/6122 Lab #5
UDP Socket Class

(100 pts)

Section	Due Date
All Sections	Nov 13th, 2020 by 11:59 PM

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 to act as the server and one for the clients.

Submitting the Assignment:

Submit two zip files. One zip file contains the code to build your server application and the other zip file contains the code for your client application.

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.
Server Application	5% for each command 5% for each receive msg action	There are 4 user prompt commands and 5 receive message actions
Client Application	15% for each user prompt command	There are 3 user prompt commands
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*$H_{weekend}$

Develop an **ECE_UDPSocket** class that will be used in both the server and client applications, with the following features

1. Only has one constructor that takes an unsigned short as an input parameter. This input parameter is the port number for the socket to receive the custom messages defined below.
    a. Delete the default constructor.
    b. The constructor should start a std::thread to receive messages.
    c. The received messages should be placed into a **std::list** using the **lSeqNum** to order them.
    d. The list will be accessed by multiple threads and needs to be protected.
2. Destructor
    a. Closes the socket
    b. Terminates the thread
3. Functions
    a. bool getNextMessage(udpMessage &msg)
    b. void sendMessage(const std::string &strTo, unsigned short usPortNum, const udpMessage &msg)
4. List of all message sources so that the composite messages can be sent back to all sources

UDP Server (50 points)

Write a console program that takes as a **command line argument** the **port number** on which the UDP Server will receive messages using the **ECE_UDPSocket** class . The UDP server collects parts of a larger **composite message** from 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.

Below is how the port number should be set as a command line argument:

./a.out 51717

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

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

The UDP server application 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.
- If **command** == 3 the server application terminates.

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 **nType == 4** indicates this is a 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 (50 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.