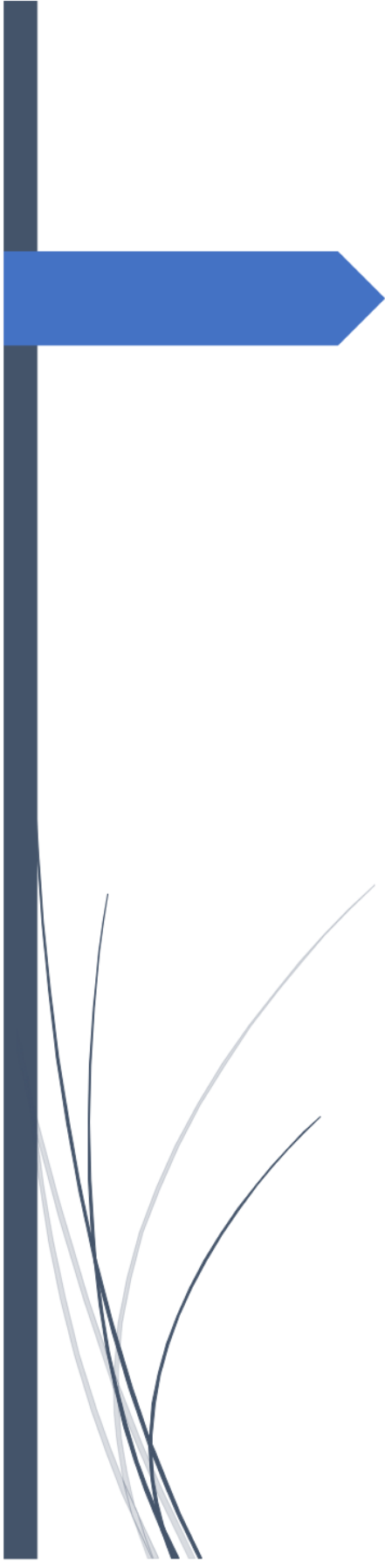


Networked Application

by Asgar Ali

FILE	4101_ASGAR_ALI_NETWORKED_APPLICATION_252504_1783923229.DO CX (801.35K)		
TIME SUBMITTED	01-MAY-2020 01:06AM (UTC+0100)	WORD COUNT	2211
SUBMISSION ID	125760891	CHARACTER COUNT	12722



NETWORKED APPLICATION

ACCA6015

Asgar Ali

1706594@student.uwtsd.ac.uk

School of Applied Computing

University of Wales Trinity Saint David

Supervisor: Dr Tim Bashford

1st May 2020

1000 Words

TABLE OF CONTENTS

1. INTRODUCTION	3
2. IMPLEMENTATION	3
2.1 Server Design	3
2.2 Client Design	7
2.3 Client/ Server message Design	8
3. CONCLUSION	10
4. REFERENCES	10
5. APPENDIX A - CHATSERVER.CPP	11
6. APPENDIX B - CHATCLIENT.CPP	13

TABLE OF FIGURES

Figure 1- Concurrency example [1].....	3
Figure 2 – Server start message.	4
Figure 3 – New connection message.	4
Figure 4 – Input Server thread flow.....	5
Figure 5 – Input handler thread flow.	5
Figure 6 – Output Server thread flow.	6
Figure 7 – Removing a connection.	7
Figure 8 – Input Client thread flow.	8
Figure 9 - Starting the Client.	8
Figure 10 – Here are the methods to send and receive messages.	9
Figure 11 - Writing a chat message in Client no.1.	9
Figure 12 - The chat message is transmitted to Client no.2.	10
Figure 13 - ChatServer.cpp.....	13
Figure 14 - chatClient.cpp	16

NETWORKED APPLICATION

1. INTRODUCTION

This report will discuss the implementation of a Client-Server network application which supports multiple simultaneous users and avoids I/O blocking through multithreading [1]. Low-level data transport was incorporated through sockets through a C++ console.

The Server will be run first, then any number of Clients. In a Client, characters will be typed and sent by pressing Enter. Then the message will appear in all Clients.

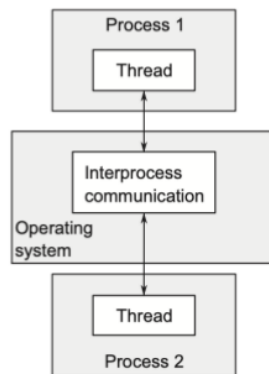


Figure 1- Concurrency example [1].

2. IMPLEMENTATION

2.1 Server Design

Immediately after its starts, the Server application creates two threads to accept TCP connections. The connections on port 50018 are used to transmit data from the Server to the Client. These connections do not require multithread processing, so this thread only adds every connection to the pool of connections for data transmitting `outputConnects (std::vector <ConnectionInstance*>)` and transmits the first message containing Client id [2].

The connections on port 50019 are used to asynchronously receive data from the Clients, that is why a separate thread is created for every such connection. Every thread waits to receive a string through its connection. In case there is an exception, the connection is closed, and the thread terminates. In case a string is received, the thread transmits it through all **outputConnects** connections. If transmitting through a connection has failed, it is regarded as a disconnection, and is deleted from **outputConnects**. To avoid conflicts as a result of simultaneous access to **outputConnects**, all operations with it are protected by a mutex or “mutual exclusion object” [1] [3].

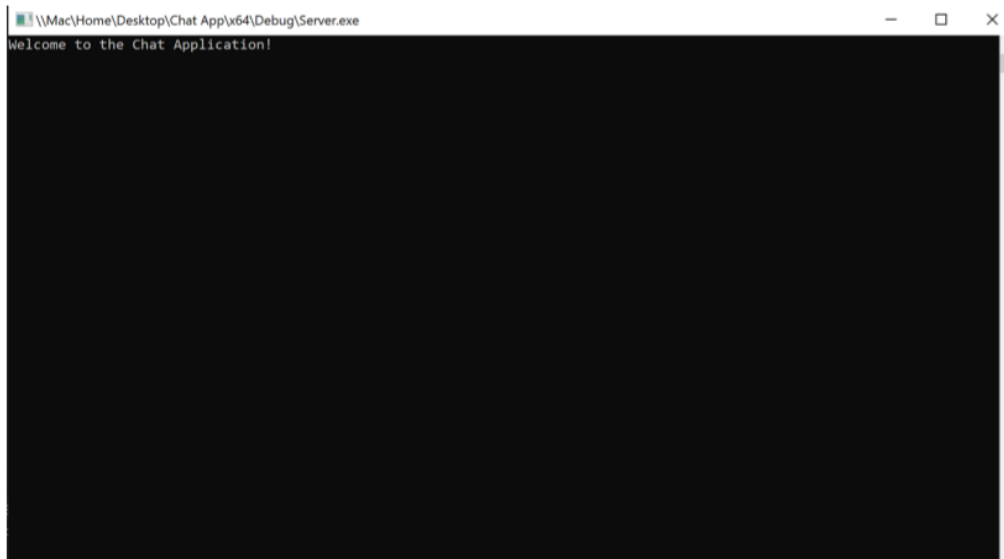


Figure 2 – Server start message.

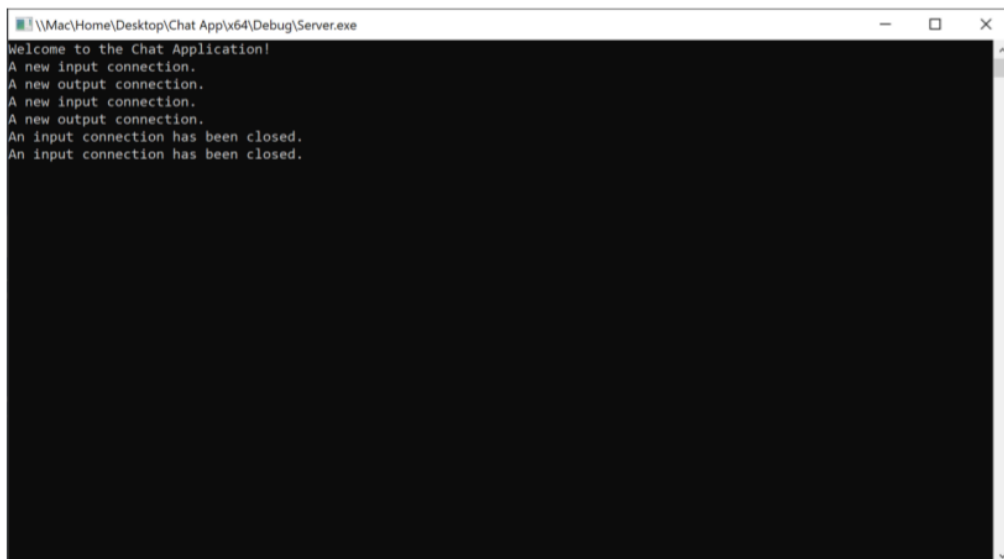


Figure 3 – New connection message.

Figures 4-6 present the workflow of the Server.

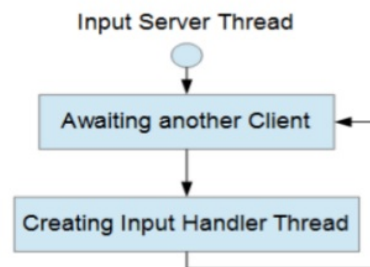


Figure 4 – Input Server thread flow.

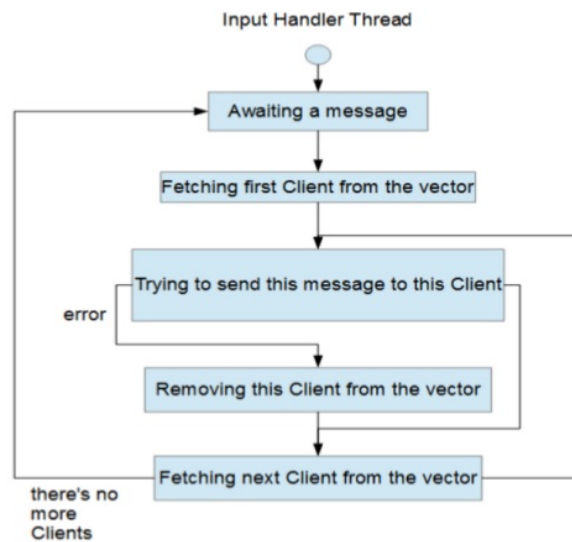


Figure 5 – Input handler thread flow.

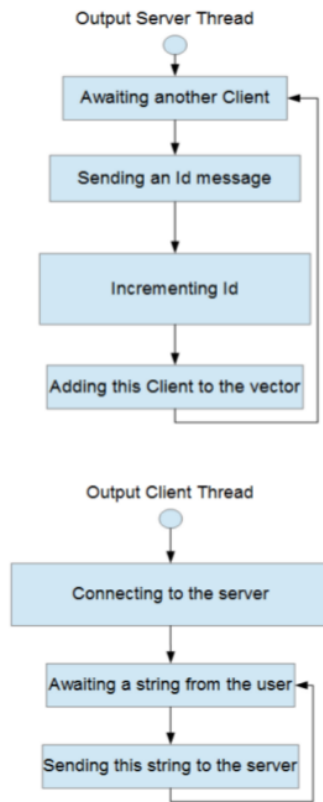


Figure 6 – Output Server thread flow.

Every new output connection is added to the vector **outputConnects**. When we receive a new message through some input connection, it must be transmitted through all output connections in the vector. In the case of an exception, the connection is presumed closed, and removed from the vector (Figure 6).

```

while (i < outputConnects.size())
{
    try
    {
        outputConnects[i]->SendString(message);
        i++;
    }
    catch (...) // if an
exception is raised, the connection is deleted
    {
        delete outputConnects[i];
        outputConnects.erase(outputConnects.begin() + i);
        cout << "An output connection has been closed.\n";
    }
}

```

Figure 7 – Removing a connection.

2.2 Client Design

Immediately upon starting, the Client application asks a user for the Server IP address. Then it detects the screen buffer size and on its base, creates a data structure **ScreenBuffer** (`std::vector<std::string>`), for displaying all chat messages in all but the last screen lines. The last line is used for messages input by the user.

The application then starts two threads, and both of them create their own connection with the Server. Since both threads work with the console, the mutex [1] **CoutMutex** is used to avoid simultaneous access. The connection on port 50018 serves to receive data from the Server. The first message is always an Id of the Client. The application shows it in its title. After the message is received, the thread adds it to **screenBuffer** (and, if necessary, removes from it the oldest message), takes **CoutMutex**, remembers the current position of the cursor, displays **ScreenBuffer** on the screen, restores the remembered position of the cursor and gives **CoutMutex**.

In a chat application we do not want to miss any user messages, therefore, TCP should be use. The Clients will not be able to connect with each other directly and all interactions will be made through the Server. We will permit any number of users (i.e. any number of Clients) to connect to the Server simultaneously. Since users can send messages asynchronously, a two-Client-Server connection per Client needs to be established. The first connection is for receiving other user messages from the Server and the second one for transmitting these user messages to the Server.

Initially, there will be two threads in the Server application. The first thread accepts all output connections (i.e. connections to transmit data to Clients). The second thread accepts all input connections (i.e. connections to receive data from Clients) and creates a new dedicated thread for each of them. Since every Client needs only two connections, there will be two appropriate threads in it.

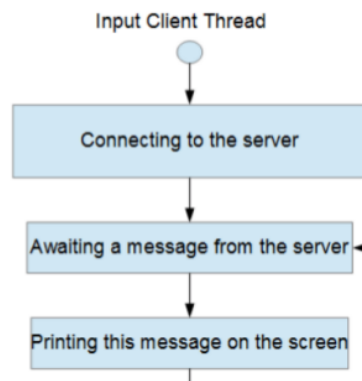


Figure 8 – Input Client thread flow.

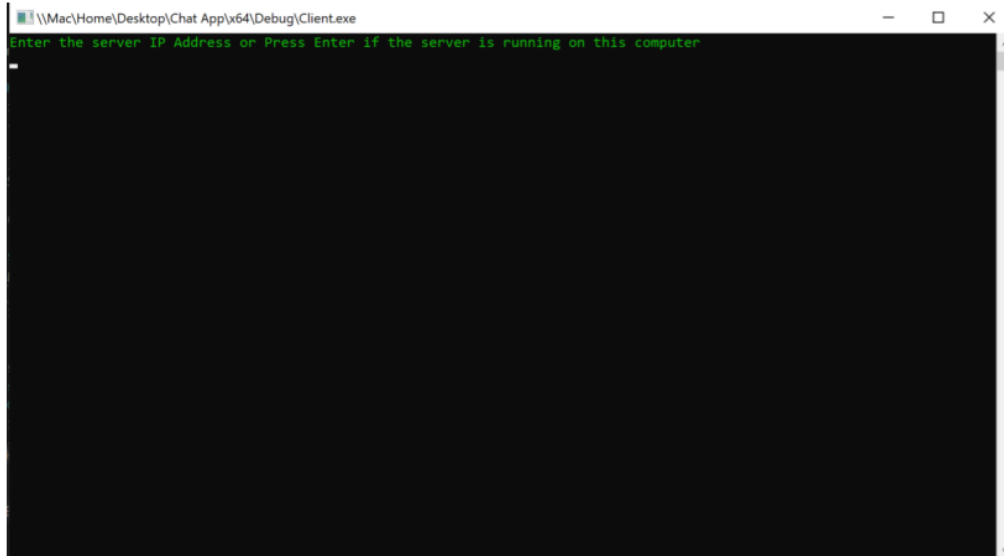


Figure 9 - Starting the Client.

2.3 Client/ Server message Design

Two connections are established with the Server, and messages typed by the user in the console is transmit through to one of them, and output in the console messages received through the second of these connections.

The application will have two message types. The first type is a message from the Server to the Client, containing a string with the Client Id (i.e. 1, 2, 3 etc). This message is sending strictly one for every Client, and

it is always the first message. The second, and the main type, are user chat messages. It is bidirectional, i.e. messages from Server to Clients and messages from Clients to Server have the same format:

- Integer number (with network byte order), containing number of message characters.
- Then bytes containing those characters.

Figure 7 shows the **Readn** method and **Writen** method that receive and transmit the given number of bytes.

```
void ConnectionInstance::SendString(string value)
{
    SendInt((int)value.length());
    Writen(value.c_str(), (int)value.length());}

string ConnectionInstance::ReceiveString()
{
    int incMessageSize = ReceiveInt();
    char* incMessage = new char[incMessageSize + 1];
    Readn(incMessage, incMessageSize);
    incMessage[incMessageSize] = '\0';
    string s(incMessage);
    delete incMessage;
    return s;}
```

Figure 10 – Here are the methods to send and receive messages.

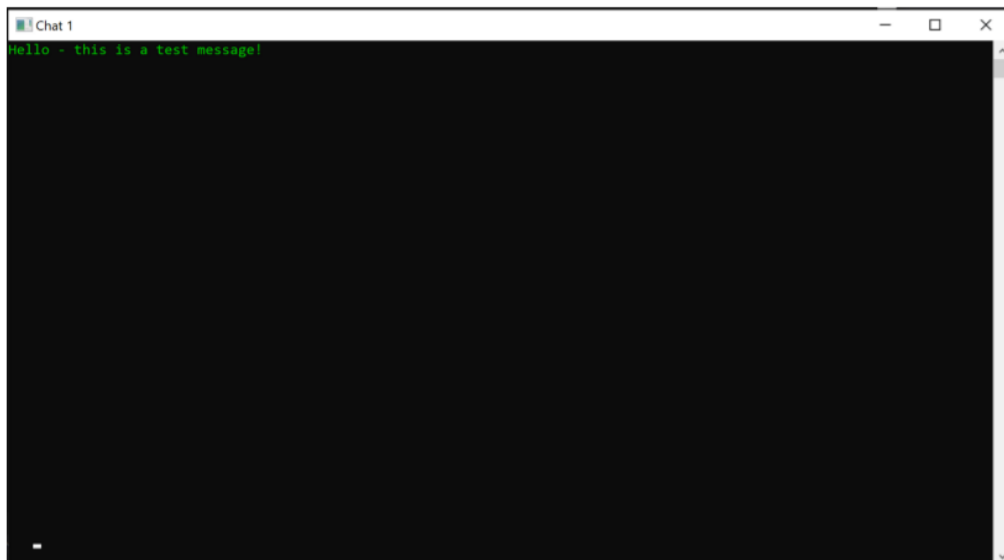


Figure 11 - Writing a chat message in Client no.1.

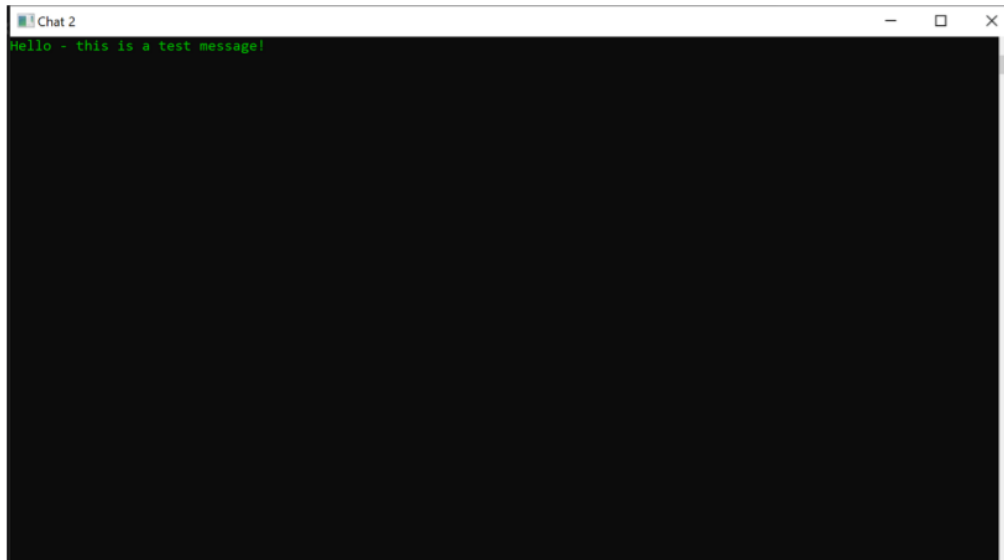


Figure 12 - The chat message is transmitted to Client no.2.

3. CONCLUSION

The output to the console in the Client was difficult to implement. For example, redrawing messages when a new one is received, concurrent access for output from two different threads. Although it was found that C++ multithreading is a difficult topic to understand with a lot of concepts, concurrency is an essential software engineering skill, as modern computers have more CPU power due to Moore's Law [4].

4. REFERENCES

- [1] A. Williams, *C++ Concurrency in Action, 2nd Edition*. 2019.
- [2] W. Goralski, *The Illustrated Network*. Elsevier Inc., 2009.
- [3] B. Stroustrup, *The C++ programming language*. .
- [4] D. Wu, L. Chen, Y. Zhou, and B. Xu, "An Empirical Study on C++ Concurrency Constructs," in *International Symposium on Empirical Software Engineering and Measurement*, 2015, vol. 2015-November, pp. 257–266, doi: 10.1109/ESEM.2015.7321187.

5. APPENDIX A - CHATSERVER.CPP

```
#include "..\Common\ServerConnection.h"
#include <thread>
#include "..\Common\INet4Address.h"
#include <vector>
#include <mutex>

#pragma comment(lib, "Ws2_32.lib")
#pragma comment(lib, "Mswsock.lib")
#pragma comment(lib, "AdvApi32.lib")

using namespace std;

vector<ConnectionInstance*> outputConnects; // output connections with Clients, i.e. connections to transmit
data from the server
mutex OutputMutex; // to avoid simultaneous access to outputConnects

void OutputServerThread() // to accept from Clients output connections
{
    INet4Address clientAddress(50018 /*port number*/);

    ServerConnection conn(&clientAddress);

    conn.StartServer(100);

    int clientNumber = 1; // to number Clients

    while (true)
    {
        ConnectionInstance* ci = conn.AcceptClient(); // accepting a new connection
        ci->SendString(to_string(clientNumber++)); // first message is Client Id
        cout << "A new output connection.\n";
        {
            lock_guard<mutex> lock(OutputMutex);
            outputConnects.push_back(ci); // adding it to outputConnects
        }
    }
}

void InputHandler(ConnectionInstance* ci) // a thread to receive and process data from a Client through the
specific input connection
{
    while (true)
    {
        string message;
        try
        {
            message = ci->ReceiveString(); // trying to receive a chat message
        }
    }
}
```

```

    }
    catch (...) // if an exception is raised, the connection is deleted and the
thread terminates
    {
        delete ci;
        cout << "An input connection has been closed.\n";
        return;
    }
    unsigned int i = 0; // else transmitting the message through all output
connections
    {
        lock_guard<mutex> lock(OutputMutex);
        while (i < outputConnects.size())
        {
            try
            {
                outputConnects[i]->SendString(message);
                i++;
            }
            catch (...) // if an exception is raised, the connection is deleted
            {
                delete outputConnects[i];
                outputConnects.erase(outputConnects.begin() + i);
                cout << "An output connection has been closed.\n";
            }
        }
    }
}

void InputServerThread() // to accept from Clients input connections
{
    INet4Address clientAddress(50019 /*port number*/);

    ServerConnection conn(&clientAddress);

    conn.StartServer(100);

    while (true)
    {
        ConnectionInstance* ci = conn.AcceptClient(); // accepting a new connection
        cout << "A new input connection.\n";
        thread(InputHandler, ci).detach(); // running a separate thread for it
    }
}

int main(int argc, char** argv)
{
    thread(OutputServerThread).detach();
    thread(InputServerThread).detach();
}

```

```

    cout << "Welcome to the Chat Application!\n";
    while (true);
}

```

Figure 13 - ChatServer.cpp

6. APPENDIX B - CHATCLIENT.CPP

```

#include "../Common/ClientConnection.h"
#include <thread>
#include <windows.h>
#include <stdio.h>
#include <mutex>
#include <vector>
#include <utilapiset.h>

#pragma comment (lib, "Ws2_32.lib")
#pragma comment (lib, "Mswsock.lib")
#pragma comment (lib, "AdvApi32.lib")

using namespace std;

vector<string> ScreenBuffer; // contains last chat messages to display them in the console
unsigned int ScreenLine = 0; // ScreenBuffer line number for the next received chat message
unsigned int ColsCount = 80; // console width in characters with default value
mutex CoutMutex; // to avoid simultaneous modifying access to the console
string ServerIp // IP-address of the server
= "127.0.0.1"; // default value - "this computer"

void SetPosition(COORD position) // set the cursor position in the console
{
    SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE), position);
}

COORD GetPosition() // get the cursor position in the console
{
    CONSOLE_SCREEN_BUFFER_INFO cbsi;
    COORD defPos = { 0, 0 };
    return GetConsoleScreenBufferInfo(GetStdHandle(STD_OUTPUT_HANDLE), &cbsi) ?
        cbsi.dwCursorPosition :
        defPos;
}

void InputThread() // to create and process the input connection to the server to receive all chat messages
{
    InetAddress serverAddress(ServerIp.c_str(), 50018 /*port number*/);
    ClientConnection conn(&serverAddress);
}

```

```

ConnectionInstance* ci = NULL;
try
{
    ci = conn.ConnectToServer();
}
catch (...)
{
    cout << "Can't establish the input connection.\n";
    getchar();
    exit(-1);
}
try
{
    string title = "Chat " + ci->ReceiveString(); // first message from server is our Id
    SetConsoleTitle(title.c_str()); // setting it as window title
    while (true)
    {
        string message = ci->ReceiveString(); // receiving a message
        if (message.length() > ColsCount)
            message.erase(ColsCount); // truncating the message to the console width (Client
doesn't support multiline messages)
        else
            message += string(ColsCount - message.length(), ' '); // complementing the message to the console
width with spaces (to fully overwrite the previous text in the console)
        if (ScreenLine < ScreenBuffer.size()) // if next line number doesn't exceed ScreenBuffer size
            ScreenBuffer[ScreenLine++] = message; // simply putting the message in this
ScreenBuffer line
        else // but if it does exceed
        {
            ScreenBuffer.erase(ScreenBuffer.begin()); // removing the oldest message from
ScreenBuffer
            ScreenBuffer.push_back(message); // and adding the new one to the end of
ScreenBuffer
        }
        for (unsigned int n = 0; n < ScreenBuffer.size(); n++) // through all ScreenBuffer lines
        {
            COORD pos = { 0, short(n) };
            lock_guard<mutex> lock(CoutMutex);
            COORD prevPos = GetPosition(); // remembering a current position of the cursor
            SetPosition(pos); // setting the cursor in the beginning of console line with the
same number
            cout << ScreenBuffer[n]; // displaying a corresponding message
            SetPosition(prevPos); // restoring the remembered position of the cursor (for
another thread)
        }
        Beep(400, 300); // playing sound for every message
    }
}
catch (...)
{
    delete ci;
}

```

```

        cout << "The input connection is closed.";
        getchar();
        exit(-1);
    }
}

void OutputThread() // to create the output connection to the server and transmit the messages typed by user
{
    InetAddress serverAddress(ServerIp.c_str(), 50019 /*port number*/);
    ClientConnection conn(&serverAddress);
    ConnectionInstance* ci = NULL;
    try
    {
        ci = conn.ConnectToServer();
    }
    catch (...)
    {
        cout << "Can't establish the output connection.\n";
        getchar();
        exit(-1);
    }
    try
    {
        while (true)
        {
            string message;
            COORD pos = { 0, short(ScreenBuffer.size()) };
            {
                lock_guard<mutex> lock(CoutMutex);
                SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE), pos); // setting the cursor position
                // to the last console line
                cout << "=>" + string(ColsCount - 5, ' '); // displaying the prompt and the full line of spaces
                // to fully overwrite the previous text
                pos.X = 3;
                SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE), pos); // setting the cursor position
                // after the prompt
            }
            getline(cin, message); // getting a message typed by user
            if (message == "exit") // "exit" is the command to close the application
                exit(0);
            ci->SendString(message); // sending it to the server
        }
    }
    catch (...)
    {
        delete ci;
        cout << "The output connection is closed.";
        getchar();
        exit(-1);
    }
}

```

```

int main(int argc, char** argv)
{
    HANDLE hOut = GetStdHandle(STD_OUTPUT_HANDLE);
    if (hOut == INVALID_HANDLE_VALUE)
    {
        return GetLastError();
    }
    SetConsoleTextAttribute(hOut, FOREGROUND_GREEN | FOREGROUND_INTENSITY); // set green colour
    for text

    cout << "Enter the server IP Address or Press Enter if the server is running on this computer" << endl;

    string addr;
    getline(cin, addr);

    if (!addr.empty()) // if the user doesn't enter an address, the default value is used
        ServerIp = addr;

    system("cls");
    size_t linesCount = 25; // number of console lines, with the default value
    CONSOLE_SCREEN_BUFFER_INFO cbsi;
    if (GetConsoleScreenBufferInfo(hOut, &cbsi)) // if we can't get buffer info, linesCount and
    ColsCount will keep their default values
    {
        // else calculating their values from the screen buffer dimensions
        linesCount = cbsi.srWindow.Bottom - cbsi.srWindow.Top + 0;
        ColsCount = cbsi.srWindow.Right - cbsi.srWindow.Left + 1;
    }
    for (size_t n = 1; n < linesCount; n++) // ScreenBuffer gets number of lines as in screen buffer -
    1 (the last screen line is reserved for the user input)
        ScreenBuffer.push_back("");

    thread(InputThread).detach();
    thread(OutputThread).detach();

    while (true);
}

```

Figure 14 - chatClient.cpp

Networked Application

GRADEMARK REPORT

FINAL GRADE

72/100

GENERAL COMMENTS

Instructor

A very good effort, Asgar. What you have is working nicely - a prototype IRC server and clients with some nicely-developed, extensible code. I'm sure, given more time, you would've added a lot more features like aliases, but what you have does the job nicely for the brief and the code underneath is of a high quality. Well done!

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

