**Programming Assignment 2**

Chenjie Luo (UIN: 324007289)

Yu-Wen Chen(UIN: 227009499)

1. **Description**

This project is mainly divided into two parts: server and client. In this assignment Chenjie Luo is responsible for the server side while Yu-Wen Chen is responsible for the client side and test cases.

For the server side, when target file is executed, users will have to enter three inputs which are: server’s IP address, port number and allowed capacity of clients. After that a socket\_fd will be created and bind to the input port number. Then I designed a struct and a vector to store all current clients info including fd and username. The server will listen and accept connection as long as number of clients doesn’t surpass the capacity. An infinite while loop is running and select function will check if any socket has new incoming messages. The timeout is set to NULL and it means select() will not return unless there is a change in any socket or an error happens. When a change happens, select returns and we identified whether it is from server(socket\_fd) or clients. If it is from server, we checked if new clients had joined by creating a new\_socket using accept(). If number of clients right now is less than capacity requirement, we added it into fd\_list, added its fd and username into vector<struct SBCP\_CLIENT\_INFO>clients. This is used to check whether a username has existed. When a client has successfully joined the chat, an ACK message including current number of clients and their usernames will be sent to the new client and a broadcast message will be sent every other client that a new client has joined in. Otherwise, a NAK message will send to the client and tell the reason that number of clients has reached maximum capacity. Similarly, when a client leaves the chat room we will close this socket and broadcast leaving message to every other client. Eventually, when a socket is found having incoming SEND(4) messages, a forward message is created and the header of it is set to FWD(3). The attribute payload of forward message is set to equal to received message’s attribute payload. Still, the message will be broadcast to every other client using write() function. When a incoming message with header equals IDLE(9), it means the client is turning to IDLE mode. A forward message is sent to other clients with the attribute payload that client is in IDLE mode now.

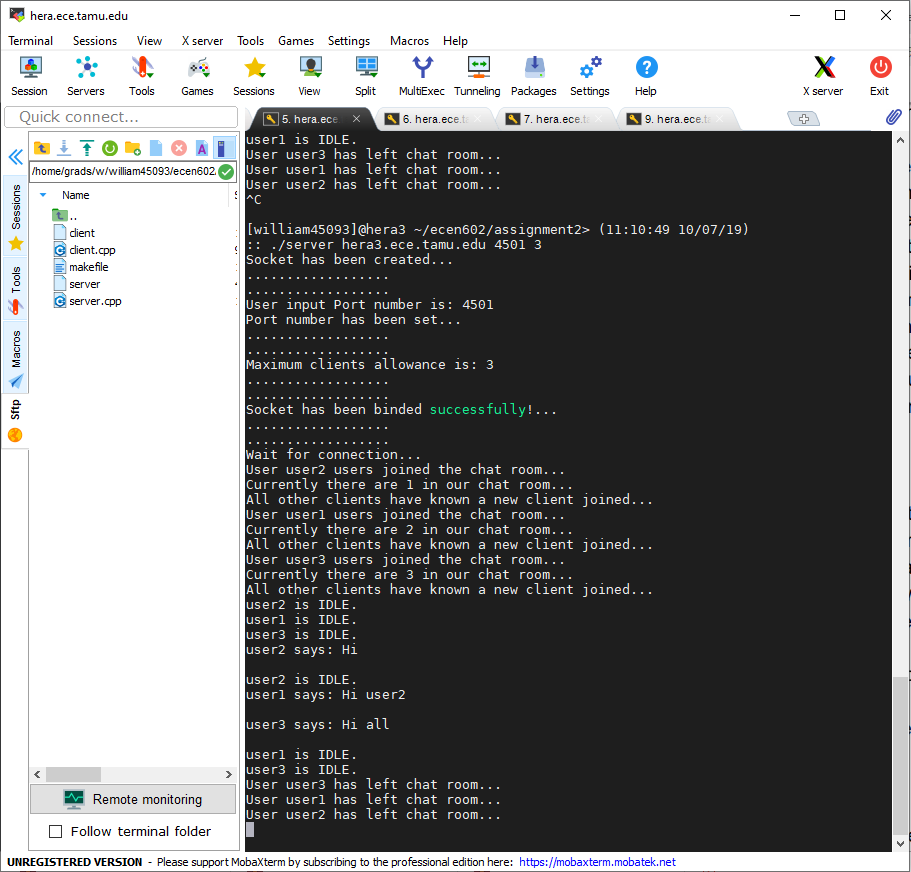
For the client part, users need to type “./client USERNAME SERVERNAME PORT\_NUMBER” to open the client and connect it to the server. After connecting to the server, the client will attempt to join the chat room on server by sending a JOIN(2) message to the server. Then, I initialize the timeout variable to 10 seconds for the bonus feature 2 and start a infinite loop in the client program. Inside the infinite loop, the setup for select function, including FD\_ZERO function to clear the file descriptor set “master” and two FD\_SET functions to add the file descriptors “stdin” and “socket” to the set, is initialized at the beginning of the loop to prevent the set from losing the setup after previous select function is activated. After the setup, the select function is activated to detect if there is any data ready to be read in one of the file descriptors (stdin and socket), and three situations will occur:

1. The first situation is when there is a string to be read in the stdin, and the client will put the string from the stdin into a SEND(4) message, send the SEND message to the server, and then reset the idle state of the client and the timeout back to 10 seconds.
2. The second one is when there is a message from the server arriving at the socket. Then the client will identify which type of message is received and do the corresponding operation to it as follow:
3. If an FWD message is received, client will print the payload string directly.
4. If an ACK message is received, client will print how may clients in the chat room and who is in the chat room.
5. If an NAK message is received, client will print the reason why the client could not join the chat room.
6. If an ONLINE message is received, client will print who is join the chat room.
7. If an OFFLINE message is received, client will print who is leaving the chat room.
8. If an IDLE message is received, client will print who has not sent any message for 10 seconds and thus is timeout.
9. The third situation is that the client has not detected any message from stdin and thus is timeout. In that, the client will send an IDLE message to the chat room and then set the IDLE indicator “idle” to high.

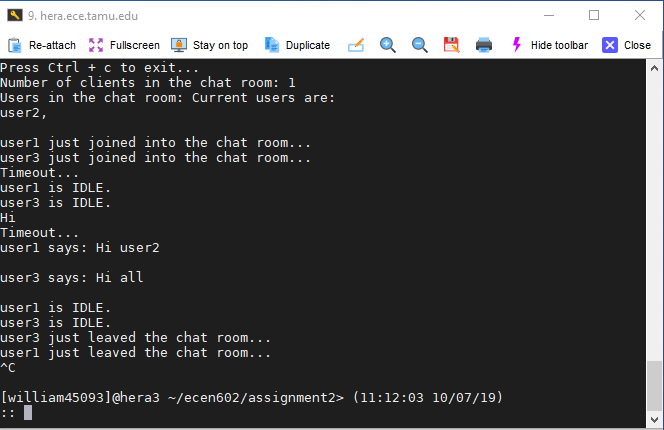
Finally, I would like to explain my implementation to handle timeout of the select function. In that, I first setup the timeout time of the select function to 10 seconds, which means that the select function will return a zero if a timeout occurs. Then, I use an if statement to detect whether the return from select function is zero and whether the indicator ”idle” is zero, which indicates if the current state of client is idle and is used to prevent the client from keeping sending the IDLE message to the server. If the select function is timeout, the client will send an IDLE message to the server and set the indicator “idle” to 1. Besides, since that only a new string from stdin is detected can reset the timeout, I implement the reset of timeout in the end of the stdin operation so that the timeout of the select function will keep counting down to zero (here, the timeout will not be reset to initial value automatically when a read from the file descriptor set is detected) unless a new string is detected from stdin.

1. **Instruction to run our code**
2. After downloading the file from github, type “make all” in the command line to generate the execution files: server and client.
3. Second, type “./server SERVER\_IP\_ADDRESS PORT\_NUMBER MAX\_CLIENT” to execute the server, and type “./client USERNAME SERVERNAME PORT\_NUMBER” to execute the client.
4. Now, users can enter messages to the client to send them to the server and test our code, and if users want to terminate the client and server, press Ctrl + C on the keyboard. Here, notice that users need to terminate all the clients before terminating the server because the server of a real chat room application should always be executed, or the clients will corrupt if the server shutdowns randomly.
5. **Test result**
6. Three clients connected to one server

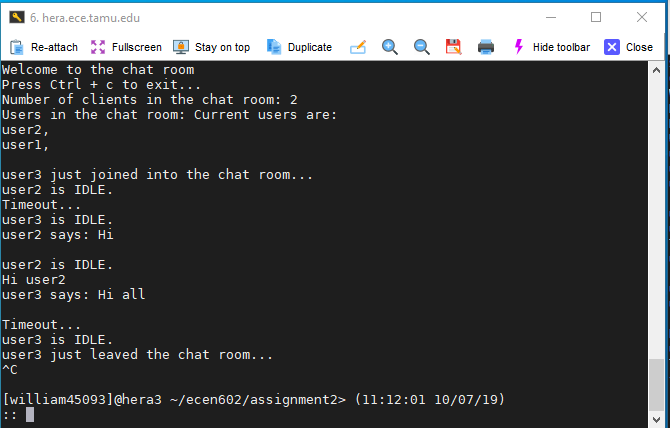
server



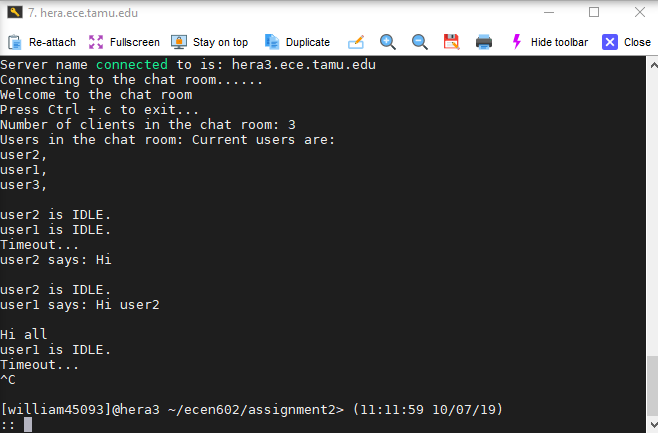
user2



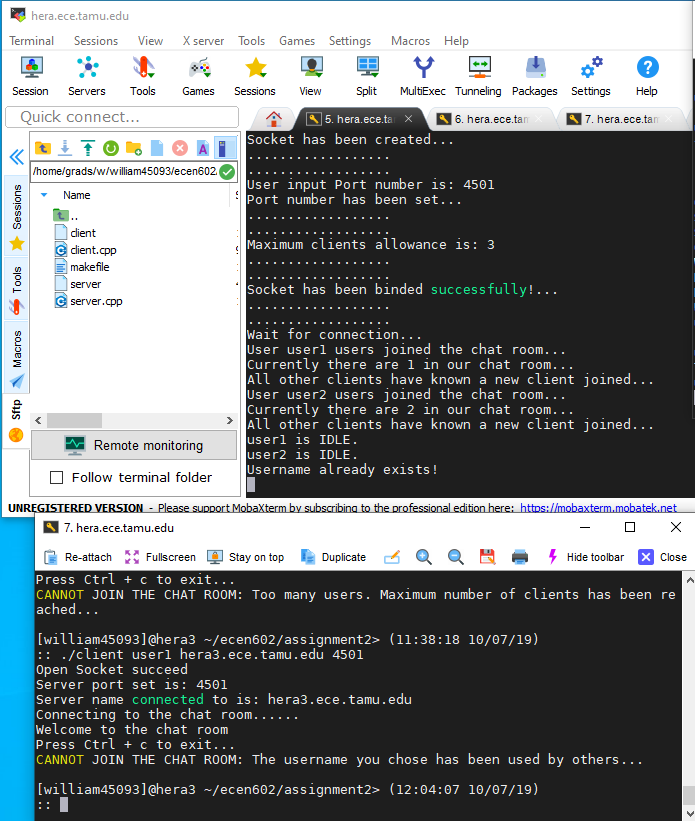
user1



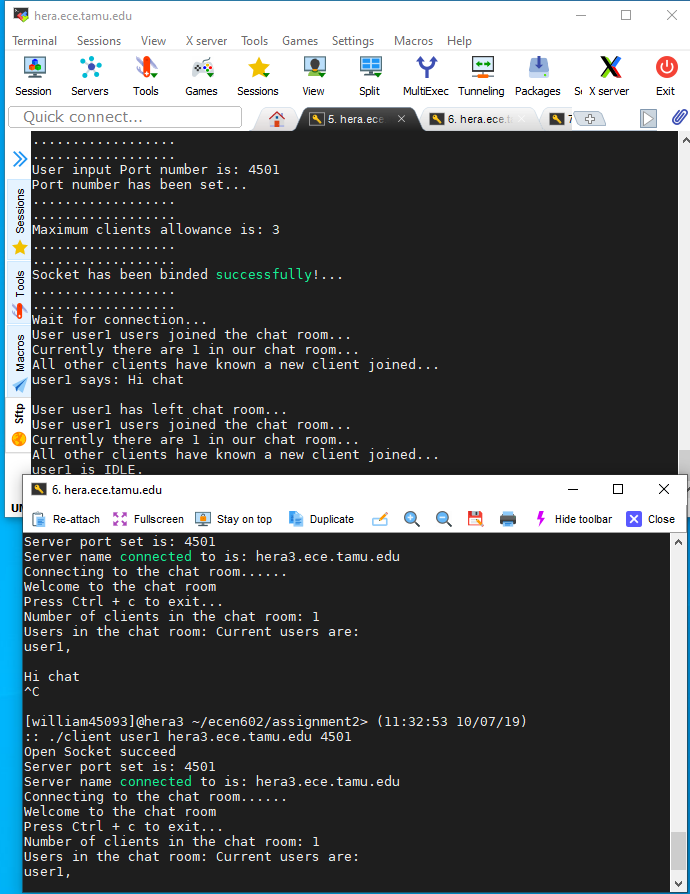
user3



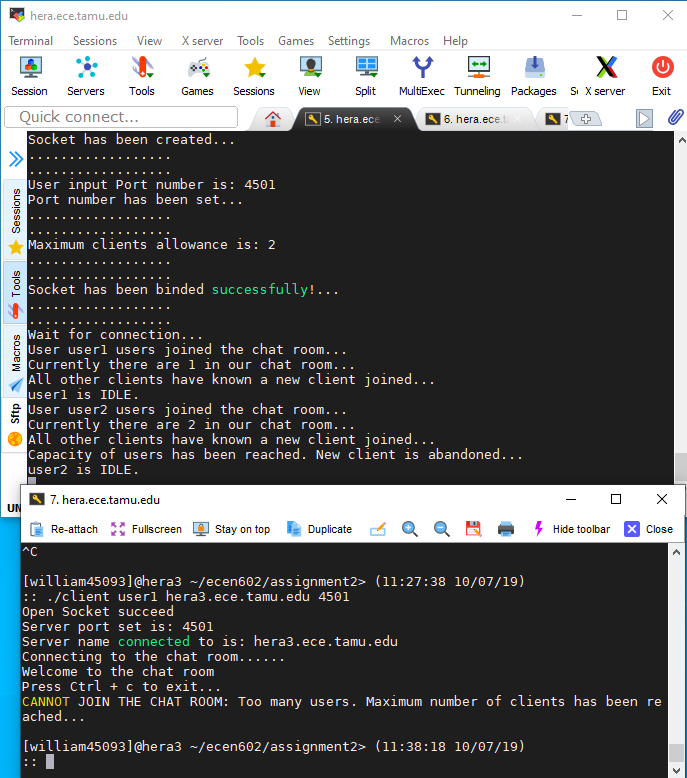
1. Server rejects a client with duplicated username



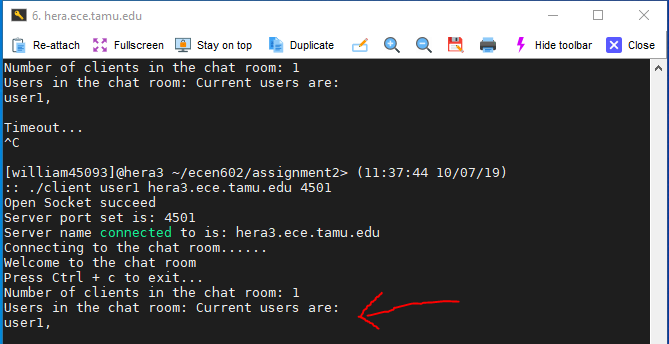
1. Server allow previously used username to be reused



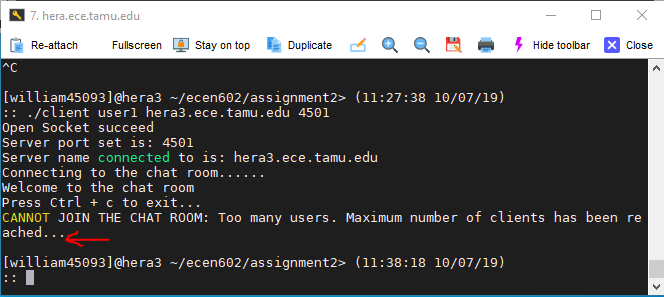
1. Server rejects the client because it exceeds the maximum number of clients allowed



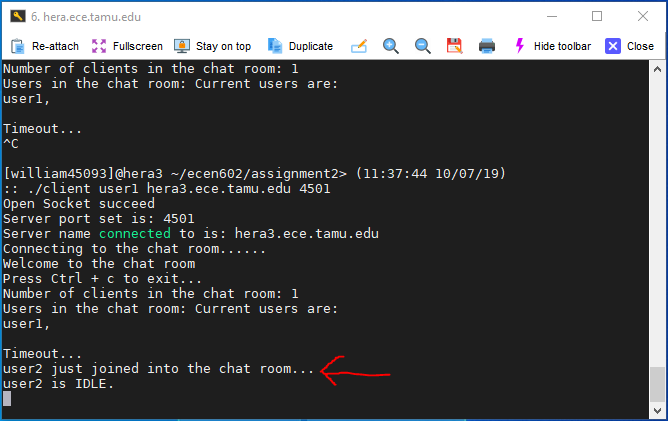
1. Bonus feature 1:
2. ACK message from server to show who is in the chat room



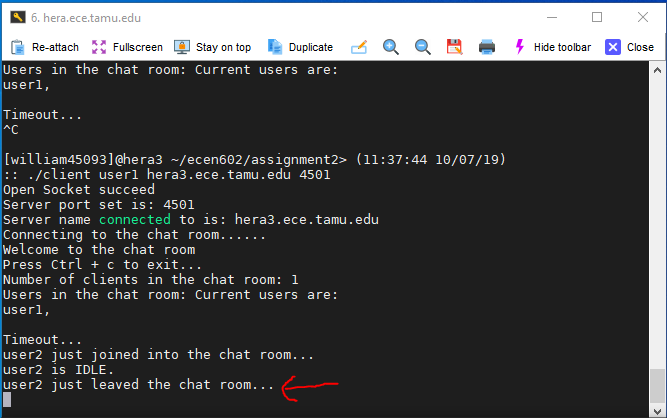
1. NAK message from server to show the reason why the client could not join the chat room



1. ONLINE message from server to show who joins the chat room

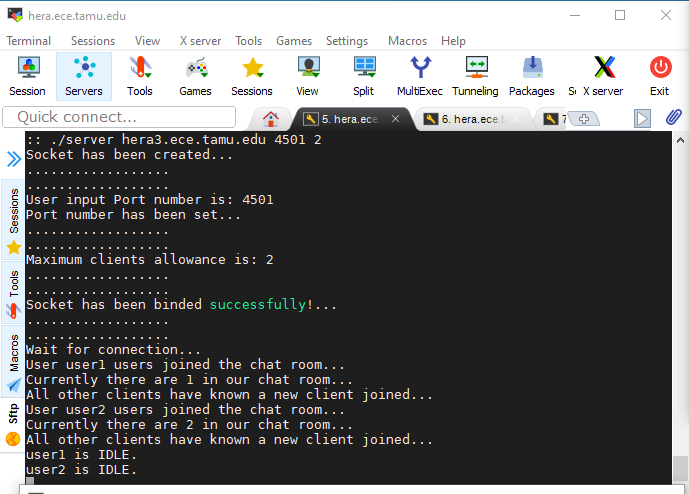


1. OFFLINE message from server to show who leaves the chat room.

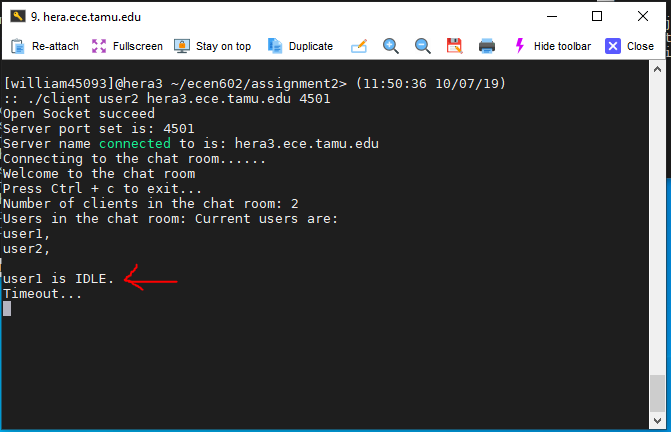


1. Bonus feature 2: Client sends IDLE message to the server indicating that no input from user over 10 seconds, and then server broadcasts IDLE message to all other clients to show that one client is in idle state.

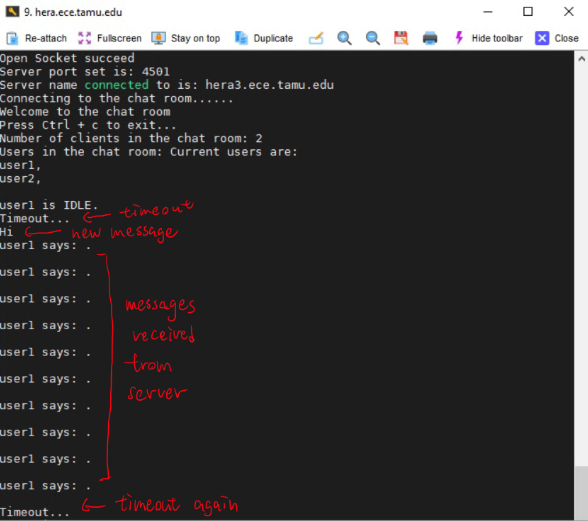
Server shows that who is idle



Broadcasted message received by client



After receiving a new message from user, timeout will be reset. However, receiving new messages from server could not reset the timeout. Finally, client is timeout again.



1. **Test result**
2. server

#include <stdio.h>

#include <string.h>

#include <vector>

#include <stdlib.h>

#include <unistd.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <sys/errno.h>

#include <iostream>

#include <sys/types.h>

#include <arpa/inet.h>

#include <sys/time.h>

#include <vector>

#include <queue>

#include <netdb.h>

#include <unordered\_map>

// PAYLOAD LENGTH

#define MAXLINE 512

// HEADER TYPE

#define JOIN 2

#define SEND 4

#define FWD 3

// ATTRIBUTE TYPE

#define USERNAME 2

#define MESSAGE 4

#define REASON 1

#define CLIENT\_COUNT 3

// JOIN AND LEAVE MSG

#define LEAVE\_MSG " leaves"

#define JOIN\_MSG " joins"

#define REASON\_TOO\_MANY\_USER "Too many users. Maximum number of clients has been reached..."

#define REASON\_EXISTED\_USER\_NAME "The username you chose has been used by others..."

using namespace std;

struct SBCP\_ATTRIBUTE{

unsigned int type : 16;

unsigned int length : 16;

char payload[512]; // maximum size of MESSAGE equals to 512 bytes

};

struct SBCP\_MSG{

unsigned int vrsn : 9;

unsigned int type : 7;

unsigned int length : 16;

struct SBCP\_ATTRIBUTE attribute[2];

};

struct SBCP\_CLIENT\_INFO{

char username[16];

int fd;

};

//PRINT OUT SUCCESS MESSAGE FOR CREATING SOCKETS

void socket\_created(){

std::cout << "Socket has been created..." << std::endl;

std::cout << ".................." << std::endl;

std::cout << ".................." << std::endl;

}

//USER INPUT TO GET PORT NUMBER

void PORT\_obtained(int &PORT, std::string &INPUT){

PORT = stoi(INPUT);

std::cout << "User input Port number is: " << PORT << std::endl;

std::cout << "Port number has been set..." << std::endl;

std::cout << ".................." << std::endl;

std::cout << ".................." << std::endl;

}

void print\_status(std::string s){

std::cout << s << "..." << std::endl;

std::cout << ".................." << std::endl;

std::cout << ".................." << std::endl;

}

//WRITEN FUNCTION WRITE len BYTES TO THE socket\_fd.

//IF FAILED, IT SHOULD RETURN -1. OTHERWISE, len SHOULD BE RETURNED.

int writen(int &socket\_fd, char\* buffer, int len){

int currptr = 0;

int has\_written = 0;

while (currptr < len){

has\_written = write(socket\_fd, buffer, len - currptr);

if (has\_written <= 0)

return -1;

buffer += has\_written;

currptr += has\_written;

}

return currptr;

}

void address\_set(struct sockaddr\_in &address, int &PORT){

address.sin\_family = AF\_INET;

address.sin\_addr.s\_addr = INADDR\_ANY;

address.sin\_port = htons(PORT);

}

// SET max\_clients AND CHECK IF THE INPUT IS REASONABLE

void max\_clients\_obtained(int &max\_clients, string &max\_clients\_str){

max\_clients = stoi(max\_clients\_str);

if (max\_clients > 20 or max\_clients < 0){

errno = EPERM;

perror("Illegal Input! The maximum clients number can only be positive integer and cannot be greater than 20");

exit(EXIT\_FAILURE);

}

std::cout << "Maximum clients allowance is: " << max\_clients << std::endl;

std::cout << ".................." << std::endl;

std::cout << ".................." << std::endl;

}

bool isempty(char\*buffer){

if (buffer[0] == '\0')

return true;

else

return false;

}

// CHECK INSIDE clients TO SEE IF THE USERNAME HAS BEEN USED

bool user\_exist(char user\_name[], vector<struct SBCP\_CLIENT\_INFO> &clients, int &num\_clients){

for(int i = 0; i < num\_clients; i++){

if(strcmp(user\_name,clients[i].username) == 0){

return false;

}

}

return true;

}

// CHECK IF CERTAIN USER HAS JOINED BEFORE

bool is\_joined(int client\_fd, vector<struct SBCP\_CLIENT\_INFO> &clients, int &num\_clients){

struct SBCP\_MSG join\_msg;

struct SBCP\_ATTRIBUTE join\_msg\_attribute;

char username[16];

read(client\_fd,(struct SBCP\_MSG \*) &join\_msg,sizeof(join\_msg));

join\_msg\_attribute = join\_msg.attribute[0];

strcpy(username, join\_msg\_attribute.payload);

if (user\_exist(username, clients, num\_clients) == false){

cout << "Username already exists!" << endl;

return true;

}

strcpy(clients[num\_clients].username, username);

clients[num\_clients].fd = client\_fd;

num\_clients += 1;

return false;

}

// IF A NEW USER JOINED THE CHAT ROOM, AN ACK MESSAGE WILL BE SENT TO HIM TO SHOW NUMBER OF CLIENTS RIGHT NOW AND THEIR NAMES

void send\_ACK(int &num\_clients, vector<struct SBCP\_CLIENT\_INFO> &clients, int &index){

struct SBCP\_MSG new\_msg;

new\_msg.vrsn = 3;

new\_msg.type = 7;

new\_msg.attribute[0].type = 3;

char cnt\_in\_array[10];

sprintf(cnt\_in\_array, "%d", num\_clients);

strcpy(new\_msg.attribute[0].payload, cnt\_in\_array);

new\_msg.attribute[1].type = 4;

strcpy(new\_msg.attribute[1].payload, "Current users are: \n");

for (int i = 0; i < num\_clients; i++){

strncat(new\_msg.attribute[1].payload, clients[i].username, sizeof(clients[i].username));

strncat(new\_msg.attribute[1].payload, ", \n", sizeof(", \n"));

}

if (write(clients[index].fd, (void \*)&new\_msg, sizeof(new\_msg)) < 0){

perror("Fialed to ACK...");

}

return;

}

// IF A NEW USER FAILED TO JOIN THE CHAT ROOM, A NAK MESSAGE WILL BE SENT TO HIM TO TELL HIM FALURE TO JOIN AND REASON

void send\_NAK(int &fd, int reasons){

struct SBCP\_MSG new\_msg;

new\_msg.vrsn = 3;

new\_msg.type = 5;

new\_msg.attribute[0].type = 1;

if (reasons == 0)

strcpy(new\_msg.attribute[0].payload, REASON\_TOO\_MANY\_USER);

else if (reasons == 1)

strcpy(new\_msg.attribute[0].payload, REASON\_EXISTED\_USER\_NAME);

if (write(fd, (void \*)&new\_msg, sizeof(new\_msg)) < 0){

perror("Fialed to NAK...");

}

return;

}

int main(int argc, char \*\*argv){

if (argc != 4){

errno = EPERM;

perror("Illegal Input! Please only enter your IP addr, server port and max clients in order");

exit(EXIT\_FAILURE);

}

std::string IP\_addr = argv[1];

std::string port\_str = argv[2];

std::string max\_clients\_str = argv[3];

int socket\_fd;

int new\_socket;

int PORT;

int max\_clients;

std::string str\_read;

struct sockaddr\_in address;

int addrlen = sizeof(address);

fd\_set fd\_list;

int current\_fd;

int max\_fd;

char \*buffer;

char \*to\_send;

std::queue<char\*> to\_send\_queue;

int val\_read;

int writenout;

char welcome\_message[45] = "Welcome! You have connected to the server! ";

int num\_clients;

fd\_set temp\_fd\_list;

struct SBCP\_MSG received\_msg;

struct SBCP\_MSG forward\_msg;

struct SBCP\_MSG broadcast\_join\_msg;

struct SBCP\_MSG broadcast\_leave\_msg;

struct SBCP\_ATTRIBUTE client\_attribute;

FD\_ZERO(&fd\_list);

FD\_ZERO(&temp\_fd\_list);

int no = 0;

// CREATE A SOCKET WITH A DESCRIPTER socket\_fd WHICH BOTH SUPPORT IPv6 and IPv4

if ((socket\_fd = socket(AF\_INET, SOCK\_STREAM, 0)) < 0){

errno = ETIMEDOUT;

perror("Failed to create socket...");

exit(EXIT\_FAILURE);

}

socket\_created();

PORT\_obtained(PORT, port\_str);

max\_clients\_obtained(max\_clients, max\_clients\_str);

int client\_fd[max\_clients];

for (int i = 0; i < max\_clients; i++)

client\_fd[i] = 0;

address\_set(address, PORT);

memset(&(address.sin\_zero), '\0', 8);

//BIND THE SOCKET TO THE IP ADDRESS AND PORT

if (::bind(socket\_fd, (struct sockaddr \*)&address, sizeof(address)) < 0){

errno = EADDRINUSE;

perror("Failed to bind...");

exit(EXIT\_FAILURE);

}

print\_status("Socket has been binded successfully!");

std::cout << "Wait for connection..." << std::endl;

// RESERVE MEMORY SPACE FOR STROING JOINED CLIENTS INFO INCLUDING FILE DESCRIPTER AND USERNAME

struct SBCP\_CLIENT\_INFO new\_client;

struct sockaddr\_in new\_addr;

vector<struct SBCP\_CLIENT\_INFO> clients(max\_clients, new\_client);

vector<struct sockaddr\_in> clients\_addr(max\_clients, new\_addr);

//SET server\_fd TO PASSIVE SOCKET AND COULD ACCEPT CONNECTION, SET MAXIMUM CONNECTION AT A TIME TO 10

if (listen(socket\_fd, 10) < 0){

errno = ETIMEDOUT;

perror("Failed to listen...");

exit(EXIT\_FAILURE);

}

FD\_SET(socket\_fd, &fd\_list);

max\_fd = socket\_fd;

while (true){

temp\_fd\_list = fd\_list;

// USE select() TO CHECK ALL THE SOCKETS IF NEW MESSAGES ARRIVE. THE TIMEOUT IS SET TO INFINITE UNTIL ANY OF SOCKET HAS AN UPDATE

if (select(max\_fd + 1, &temp\_fd\_list, NULL, NULL, NULL) < 0){

perror("Failed to select...");

exit(EXIT\_FAILURE);

}

// CHECK ALL THE FILE DESCRIPTER TO SEE IF NEW MESSAGES ARRIVE

for (int i = 0; i <= max\_fd; i++){

if (FD\_ISSET(i, &temp\_fd\_list)){

if (i == socket\_fd){

socklen\_t client\_addr\_size = sizeof(clients\_addr[num\_clients]);

new\_socket = accept(socket\_fd, (struct sockaddr \*)&clients\_addr[num\_clients], &client\_addr\_size);

if (new\_socket < 0){

perror("Failed to accept...");

exit(EXIT\_FAILURE);

}

if (num\_clients < max\_clients){

if (is\_joined(new\_socket, clients, num\_clients) == false){

FD\_SET(new\_socket, &fd\_list);

max\_fd = max(max\_fd, new\_socket);

int index = 0;

for (; index < num\_clients; index++){

if (clients[index].fd == new\_socket)

break;

}

std::cout << "User " << clients[index].username << " users joined the chat room..." << std::endl;

std::cout << "Currently there are "<< num\_clients << " in our chat room..." << std::endl;

// WHEN A NEW CLIENT JOIN THE CHAT, SEND AN ACK TO HIM TO INDICATE HE HAS JOIN SUCCESSFULLY

send\_ACK(num\_clients, clients, index);

broadcast\_join\_msg.vrsn = 3;

broadcast\_join\_msg.type = 8;

broadcast\_join\_msg.attribute[0].type = 2;

strcpy(broadcast\_join\_msg.attribute[0].payload, clients[index].username);

for (int j = 0; j <= max\_fd; j++){

if (FD\_ISSET(j, &fd\_list)){

if (j != socket\_fd and j != new\_socket){

// BROADCAST TO EVERY OTHER CLIENT THAT A NEW CLIENT HAS JOINED IN

if (write(j, (void \*)&broadcast\_join\_msg, sizeof(broadcast\_join\_msg)) < 0){

perror("Failed to broadcast...");

exit(EXIT\_FAILURE);

}

}

}

}

cout << "All other clients have known a new client joined..." << endl;

}

else{

send\_NAK(new\_socket, 1);

}

}

else{

// IF THE NUMBER OF CAPACITY HAS BEEN REACHED NO MORE USERS CAN CONNECT

send\_NAK(new\_socket, 0);

std::cout << "Capacity of users has been reached. New client is abandoned..."<< std::endl;

}

}

else{

val\_read = read(i, (struct SBCP\_MSG \*) &received\_msg, sizeof(received\_msg));

if (val\_read < 0){

perror("Failed to read message...");

}

if (val\_read == 0){

int k = 0;

for (; k < num\_clients; k++){

if (clients[k].fd == i)

break;

}

broadcast\_leave\_msg.type = 6;

broadcast\_leave\_msg.attribute[0].type = 2;

broadcast\_leave\_msg.vrsn = 3;

broadcast\_leave\_msg.length = 520;

broadcast\_leave\_msg.attribute[0].length = 516;

strcpy(broadcast\_leave\_msg.attribute[0].payload, clients[k].username);

cout << "User " << clients[k].username << " has left chat room..." << endl;

// BROADCAST clients[k].username HAS LEFT THE CHAT ROOM

for (int j = 0; j <= max\_fd; j++){

if (FD\_ISSET(j, &fd\_list)){

if (j != socket\_fd){

if (write(j, (void\*)&broadcast\_leave\_msg, sizeof(broadcast\_leave\_msg)) < 0)

perror("Failed to broadcast...");

}

}

}

}

if (val\_read <= 0){

close(i);

FD\_CLR(i, &fd\_list);

for (int a = i; a < num\_clients; a++)

clients[a] = clients[a + 1];

num\_clients -= 1;

}

else{

// IT IS A SEND MESSAGE AND JUST FORWARD TO OTHERS

forward\_msg.vrsn = received\_msg.vrsn;

forward\_msg.type = 3;

forward\_msg.attribute[0].length = received\_msg.attribute[0].length;

forward\_msg.attribute[0].type = 4;

// CHECK THE INFO OF SEND BY USING FILE DESCRIPTER

int k = 0;

for (; k < num\_clients; k++){

if (clients[k].fd == i)

break;

}

if (received\_msg.type == 9){

strcpy(forward\_msg.attribute[0].payload, clients[k].username);

strncat(forward\_msg.attribute[0].payload, " is IDLE.", sizeof(" is IDLE."));

cout << forward\_msg.attribute[0].payload << endl;

}

else{

strcpy(forward\_msg.attribute[0].payload, clients[k].username);

strncat(forward\_msg.attribute[0].payload, " says: ", sizeof(" says: "));

strncat(forward\_msg.attribute[0].payload, received\_msg.attribute[0].payload, sizeof(received\_msg.attribute[0].payload));

cout << forward\_msg.attribute[0].payload << endl;

}

for (int j = 0; j <= max\_fd; j++){

if (FD\_ISSET(j, &fd\_list)){

if (j != socket\_fd and j != i){

// FORWARD THE MESSAGE TO EVERY OTHER JOINED CLIENTS

if (write(j, (void\*) &forward\_msg, val\_read) < 0)

perror("Failed to write...");

}

}

}

}

}

}

}

}

return 0;

}

1. client

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <sys/errno.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <sys/select.h>

#include <sys/time.h>

#include <arpa/inet.h>

#include <stdlib.h>

#include <iostream>

using namespace std;

// PAYLOAD LENGTH

#define MAXLINE 512

// HEADER TYPE

#define JOIN 2

#define SEND 4

#define FWD 3

#define ACK 7

#define NAK 5

#define ONLINE 8

#define OFFLINE 6

#define IDLE 9

// ATTRIBUTE TYPE

#define USERNAME 2

#define MESSAGE 4

#define REASON 1

#define CLIENT\_COUNT 3

struct SBCP\_ATTRIBUTE{

unsigned int type : 16;

unsigned int length : 16;

char payload[512]; // maximum size of MESSAGE equals to 512 bytes

};

struct SBCP\_MSG{

unsigned int vrsn : 9;

unsigned int type : 7;

unsigned int length : 16;

struct SBCP\_ATTRIBUTE attribute[2];

};

int main(int argc, char \*\*argv){

if (argc != 4) {

errno = EPERM;

printf("CORRECT FORMAT: ./client USERNAME SERVER\_NAME SERVER\_PORT \n");

printf("INPUT\_ERROR: ERRNO: \t%s\n", strerror(errno));

return -1;

}

char\* username = argv[1];

char\* server\_name = argv[2];

string \_server\_port = argv[3];

int server\_port = -1;

int socketfd;

char input[MAXLINE];

struct sockaddr\_in server\_addr;

struct SBCP\_MSG \*msg\_to\_server;

struct SBCP\_MSG \*msg\_from\_server;

struct timeval tv;

fd\_set master;

fd\_set readfd;

int act;

int idle = 0; // variable indicates whether the client is idle or not

// Reset

memset(&tv, 0, sizeof(struct timeval));

if ((socketfd = socket(AF\_INET, SOCK\_STREAM, 0)) < 0) {

printf("SOCKET\_OPEN: ERRNO: \t%s\n", strerror(errno));

return -1;

}

printf("Open Socket succeed\n");

// Input the server port to be connected

server\_port = stoi(\_server\_port); // Transform format of server port from string to integer

printf("Server port set is: %d\n", server\_port);

memset(&server\_addr, 0, sizeof(server\_addr)); // Set all bits of server address to zero

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_port = htons(server\_port);

if (inet\_pton(AF\_INET, server\_name, &server\_addr.sin\_addr) < 0) {

printf("ADDR\_TRANS: ERRNO: \t%s\n", strerror(errno));

return -1;

}

printf("Server name connected to is: %s\n", server\_name);

// Connect the socket to the server

if ((connect(socketfd, (struct sockaddr \*)&server\_addr, sizeof(server\_addr))) < 0) {

printf("CONNECT: RRNO: \t%s\n", strerror(errno));

return -1;

}

// Send JOIN message to server

msg\_to\_server = (struct SBCP\_MSG \*) malloc(sizeof(struct SBCP\_MSG));

msg\_to\_server->vrsn = 3; // protocol version is 3

msg\_to\_server->type = JOIN; // SBCP message type is JOIN to join server

msg\_to\_server->length = 24; // 4 bytes for vsrn and type, and 20 bytes for attribute

msg\_to\_server->attribute[0].type = USERNAME; // indicates that payload stores username

msg\_to\_server->attribute[0].length = 20; // 4 bytes for type and length, and 16 bytes for username

strcpy(msg\_to\_server->attribute[0].payload, username); // copy string username to payload

printf("Connecting to the chat room......\n");

if ((write(socketfd, msg\_to\_server, sizeof (struct SBCP\_MSG))) < 0) {

printf("JOIN: ERRNO: \t%s\n", strerror(errno));

free(msg\_to\_server); // release malloc of msg\_to\_server

return -1;

}

printf("Welcome to the chat room\n");

printf("Press Ctrl + c to exit...\n");

free(msg\_to\_server); // release malloc of msg\_to\_server

// Initialize select timeout to 10 sec

tv.tv\_sec = 10;

tv.tv\_usec = 0;

while (1) {

// Setup select function

FD\_ZERO(&master); // Clear the set before using file descriptor set "master"

FD\_SET(fileno(stdin), &master); // Add stdin file descriptor to the select set

FD\_SET(socketfd, &master); // Add socket file descroptor to the select set

if ((act = select(socketfd + 1, &master, NULL, NULL, &tv)) < 0) {

printf("MASTER SELECT: ERRNO: \t%s\n", strerror(errno));

return -1;

}

else if ((act == 0) && (idle == 0)) { // timeout when return 0 and the client is not in IDLE state currently

// Setup IDLE message to server

msg\_to\_server = (struct SBCP\_MSG \*) malloc(sizeof(struct SBCP\_MSG));

msg\_to\_server->vrsn = 3; // protocol version is 3

msg\_to\_server->type = IDLE; // SBCP message type is SEND to send message to server

msg\_to\_server->length = 520; // 4 bytes for vsrn and type, and 516 bytes for attribute

msg\_to\_server->attribute[0].type = 0; // indicates that payload stores none

msg\_to\_server->attribute[0].length = 516; // 4 bytes for type and length, and 512 bytes for message

memset(msg\_to\_server->attribute[0].payload, 0, 512 \* sizeof(char)); // Nothing to send to the chat room

// Written operation

if (write(socketfd, msg\_to\_server, sizeof(struct SBCP\_MSG)) < 0) {

printf("WRITE: ERRNO: \t%s\n", strerror(errno));

free(msg\_to\_server); // release malloc of msg\_to\_server

return -1;

}

printf("Timeout...\n");

free(msg\_to\_server); // release malloc of msg\_to\_server

idle = 1; // indicate that the client enters into idle state

}

// When user input to the terminal

if (FD\_ISSET(fileno(stdin), &master)) {

// Input string into the client

fgets(input, sizeof(input), stdin);

// Setup SEND message to server

msg\_to\_server = (struct SBCP\_MSG \*) malloc(sizeof(struct SBCP\_MSG));

msg\_to\_server->vrsn = 3; // protocol version is 3

msg\_to\_server->type = SEND; // SBCP message type is SEND to send message to server

msg\_to\_server->length = 520; // 4 bytes for vsrn and type, and 516 bytes for attribute

msg\_to\_server->attribute[0].type = MESSAGE; // indicates that payload stores message

msg\_to\_server->attribute[0].length = 516; // 4 bytes for type and length, and 512 bytes for message

strcpy(msg\_to\_server->attribute[0].payload, input); // copy string input to payload

// Written operation

if (write(socketfd, msg\_to\_server, sizeof(struct SBCP\_MSG)) < 0) {

printf("WRITE: ERRNO: \t%s\n", strerror(errno));

free(msg\_to\_server); // release malloc of msg\_to\_server

return -1;

}

free(msg\_to\_server); // release malloc of msg\_to\_server

idle = 0; // client leave the idle state after receiving message from stdin

// A message is detected from stdin, so the select timeout needs to reset back to 10 sec

tv.tv\_sec = 10;

tv.tv\_usec = 0;

}

// When receiving message from server

else if (FD\_ISSET(socketfd, &master)) {

msg\_from\_server = (struct SBCP\_MSG \*) malloc(sizeof(struct SBCP\_MSG));

// Read messages from the socket

if ((read(socketfd , msg\_from\_server, sizeof(struct SBCP\_MSG))) < 0) {

printf("READ: ERRNO: \t%s\n", strerror(errno));

return -1;

}

// Check type of message from server

// print the attribute directly when receive forward message from server

if (msg\_from\_server->type == FWD) {

if (msg\_from\_server->attribute[0].type == MESSAGE) {

printf("%s\n", msg\_from\_server->attribute[0].payload);

}

}

// when receiving ACK message from server, print the number of clients connected to the server,

// all the client name belonging to the server respectively

else if (msg\_from\_server->type == ACK) {

if (msg\_from\_server->attribute[0].type == CLIENT\_COUNT){

printf("Number of clients in the chat room: %s\n", msg\_from\_server->attribute[0].payload);

}

if (msg\_from\_server->attribute[1].type == MESSAGE) {

printf("Users in the chat room: %s\n", msg\_from\_server->attribute[1].payload);

}

}

// when receiving NAK message from server, print the reason why the client could not join the server

else if (msg\_from\_server->type == NAK) {

if (msg\_from\_server->attribute[0].type == REASON){

printf("CANNOT JOIN THE CHAT ROOM: %s\n", msg\_from\_server->attribute[0].payload);

}

close(socketfd);

return 0;

}

// ONLINE message indicates that a new client joins the server, and the client will print the name of

// the new cleint

else if (msg\_from\_server->type == ONLINE) {

if (msg\_from\_server->attribute[0].type == USERNAME){

printf("%s just joined into the chat room...\n", msg\_from\_server->attribute[0].payload);

}

}

// OFFNLINE message indicates that a client leaves the server, and the client will print the name of

// the left cleint

else if (msg\_from\_server->type == OFFLINE) {

if (msg\_from\_server->attribute[0].type == USERNAME){

printf("%s just leaved the chat room...\n", msg\_from\_server->attribute[0].payload);

}

}

// IDLE message indicates that a client has not sent any message for 10 sec , and the client

// will print the name of the idle client

else if (msg\_from\_server->type == IDLE) {

if (msg\_from\_server->attribute[0].type == USERNAME){

printf("%s is idle...\n", msg\_from\_server->attribute[0].payload);

}

}

free(msg\_from\_server); // release malloc of msg\_from\_server

}

// Reset char array for next turn

memset(input, 0, sizeof(input));

}

}