Programming Assignment 2

Chenjie Luo (UIN: 324007289)
Yu-Wen Chen(UIN: 227009499)

a. 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:
   a. If an FWD message is received, client will print the payload string directly.
   b. If an ACK message is received, client will print how may clients in the chat room and who is in the chat room.
   c. If an NAK message is received, client will print the reason why the client could not join the chat room.
   d. If an ONLINE message is received, client will print who is join the chat room.
   e. If an OFFLINE message is received, client will print who is leaving the chat room.
   f. If an IDLE message is received, client will print who has not sent any message for 10 seconds and thus is timeout.
3. 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.

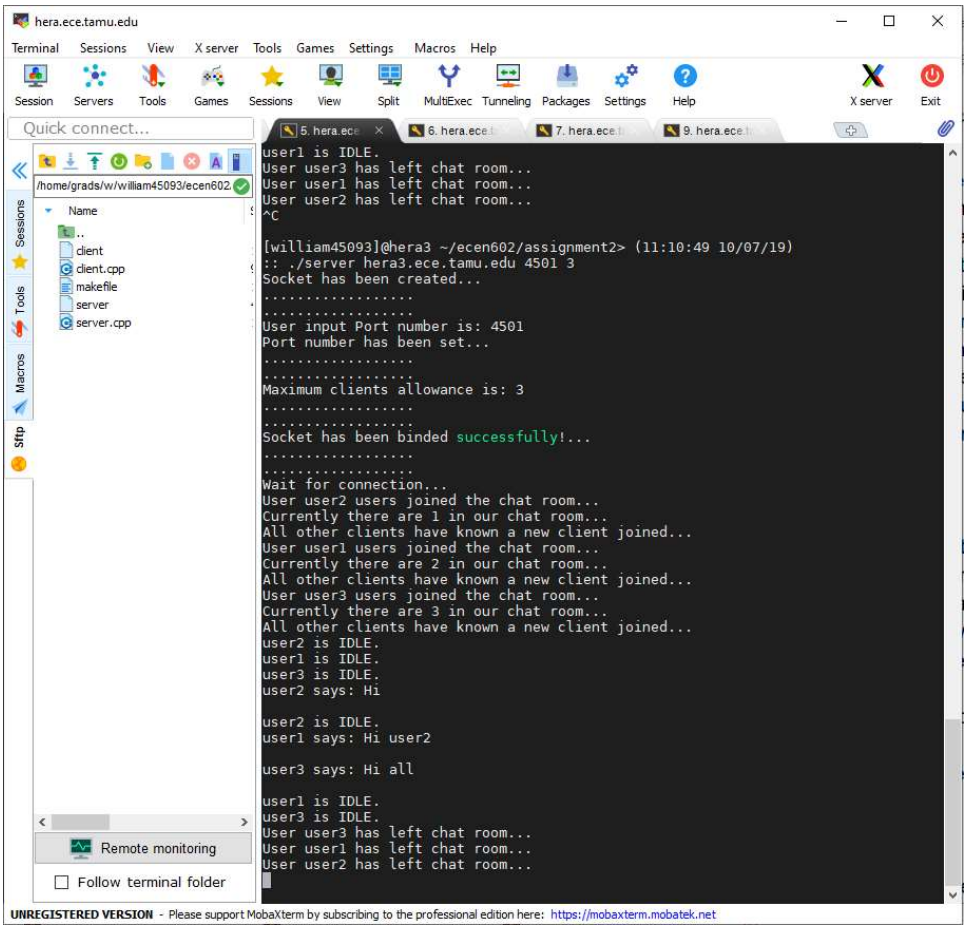
b. Instruction to run our code

1. After downloading the file from github, type "make all" in the command line to generate the execution files: server and client.
2. 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.
3. 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.

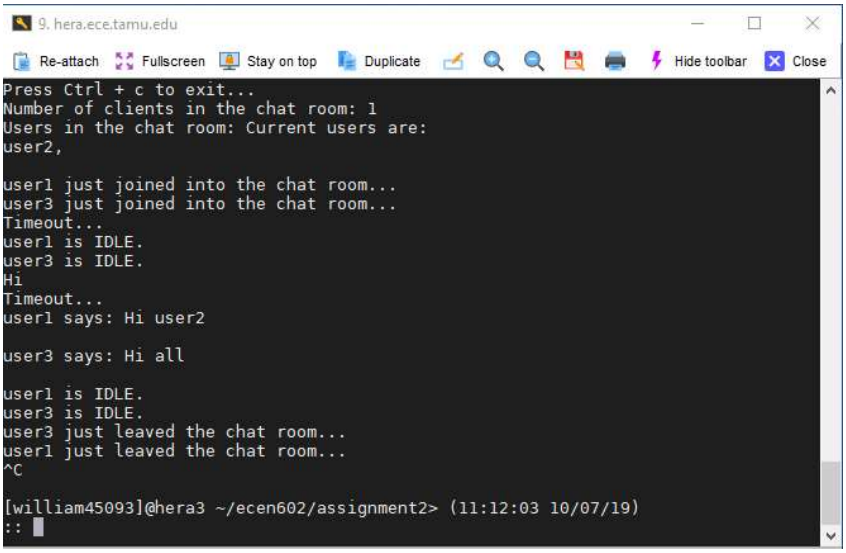
c. Test result

1. Three clients connected to one server

server



user2



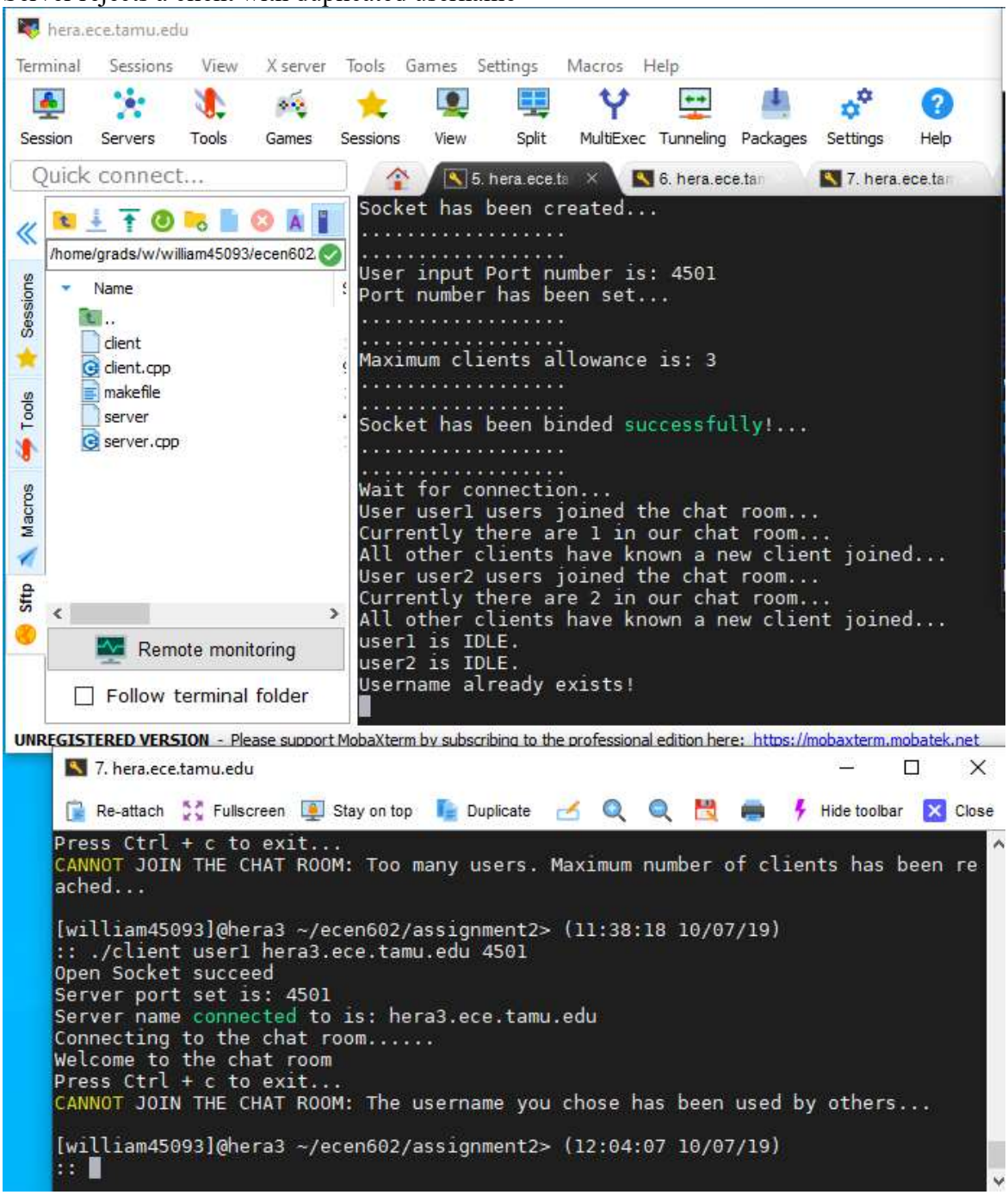
user1

```
6. hera.ece.tamu.edu
Re-attach  Fullscreen  Stay on top  Duplicate  Hide toolbar  Close
Welcome to the chat room
Press Ctrl + c to exit...
Number of clients in the chat room: 2
Users in the chat room: Current users are:
user2,
user1,

user3 just joined into the chat room...
user2 is IDLE.
Timeout...
user3 is IDLE.
user2 says: Hi

user2 is IDLE.
Hi user2
user3 says: Hi all

Timeout...
user3 is IDLE.
user3 just leaved the chat room...
^C

[william45093]@hera3 ~/ecen602/assignment2> (11:12:01 10/07/19)
::
```

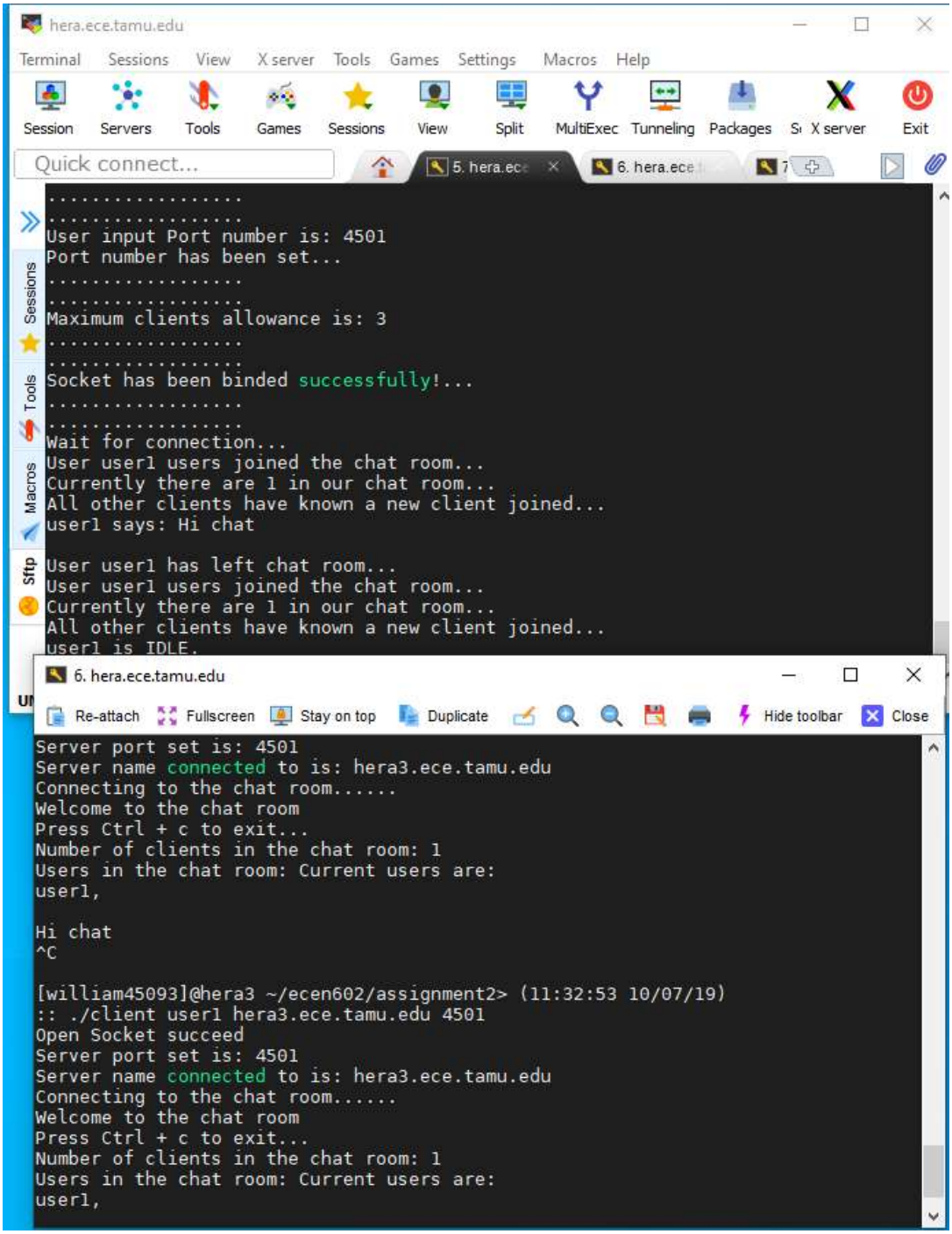
user3

```
7. hera.ece.tamu.edu
Re-attach  Fullscreen  Stay on top  Duplicate  Hide toolbar  Close
Server name connected to is: hera3.ece.tamu.edu
Connecting to the chat room......
Welcome to the chat room
Press Ctrl + c to exit...
Number of clients in the chat room: 3
Users in the chat room: Current users are:
user2,
user1,
user3,

user2 is IDLE.
user1 is IDLE.
Timeout...
user2 says: Hi

user2 is IDLE.
user1 says: Hi user2

Hi all
user1 is IDLE.
Timeout...
^C

[william45093]@hera3 ~/ecen602/assignment2> (11:11:59 10/07/19)
::
```

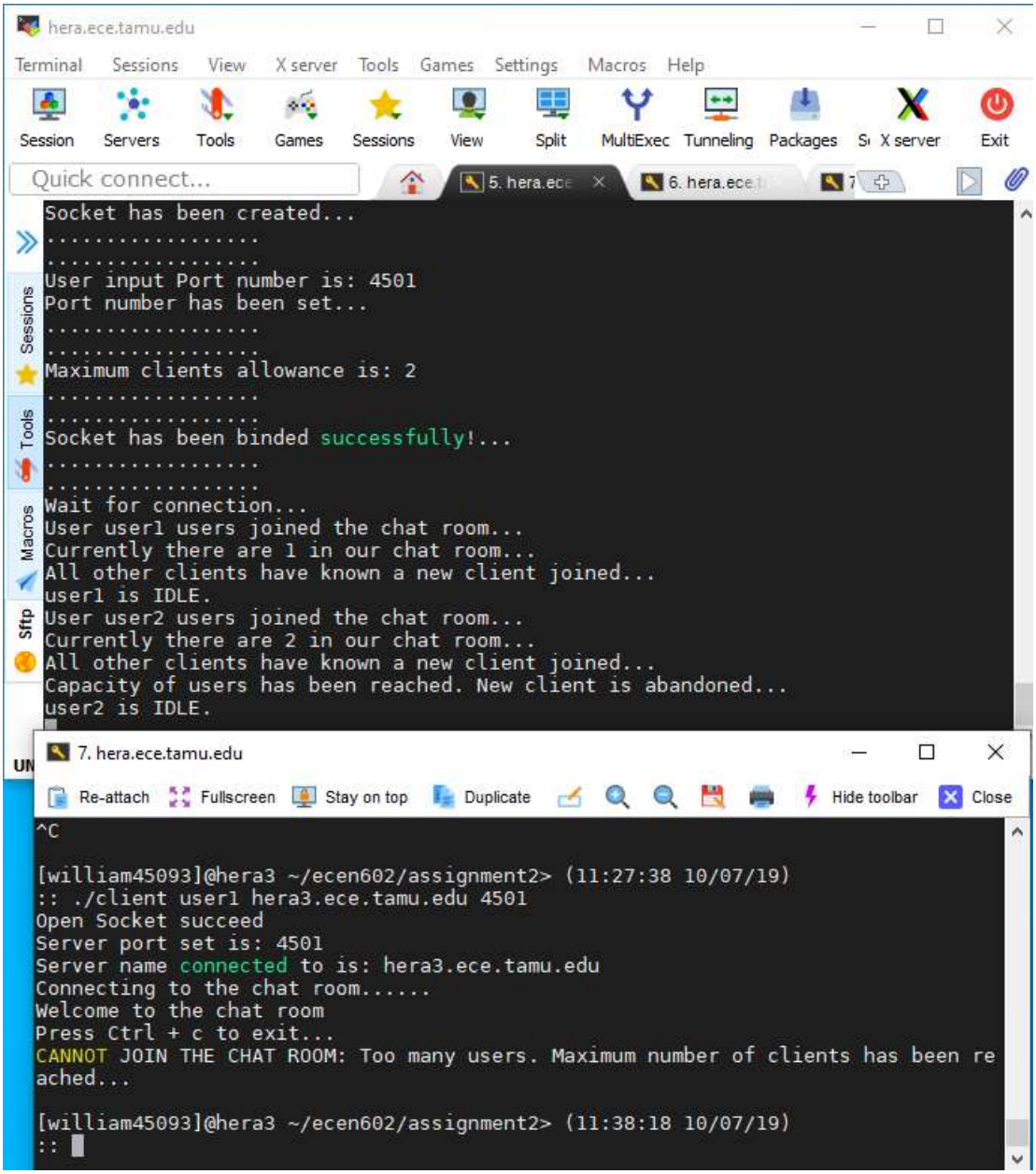
2. Server rejects a client with duplicated username



3. Server allow previously used username to be reused

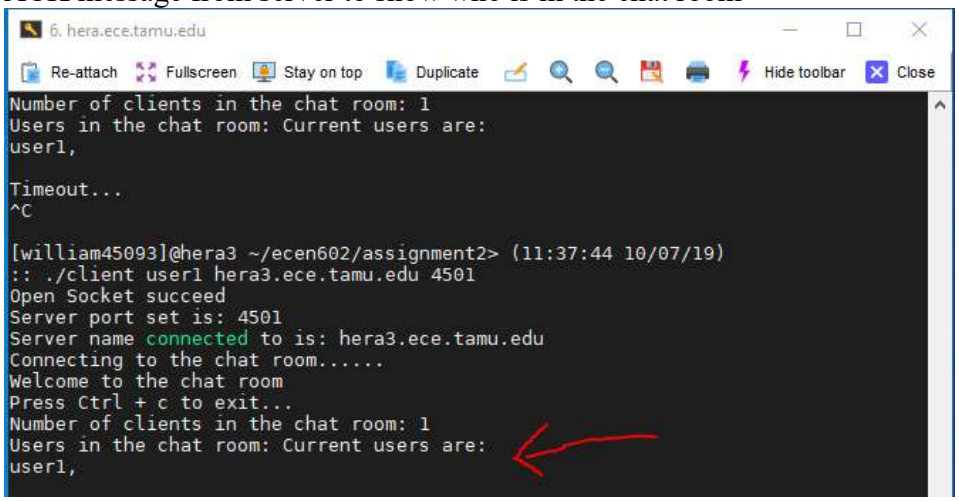


4. Server rejects the client because it exceeds the maximum number of clients allowed

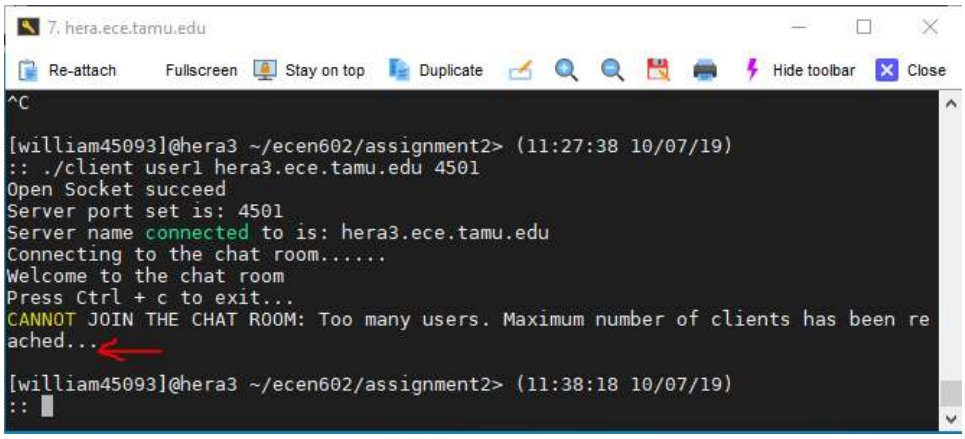


5. Bonus feature 1:

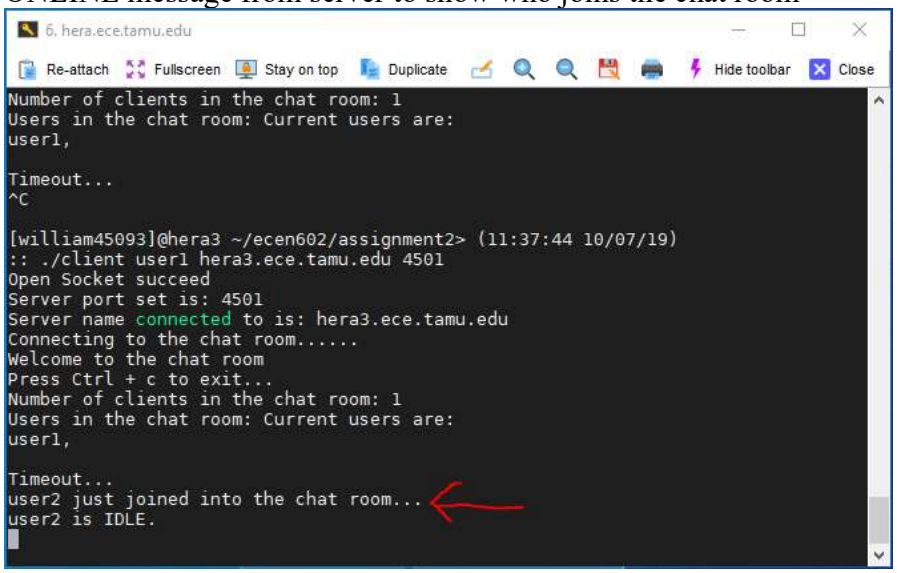
(1) ACK message from server to show who is in the chat room



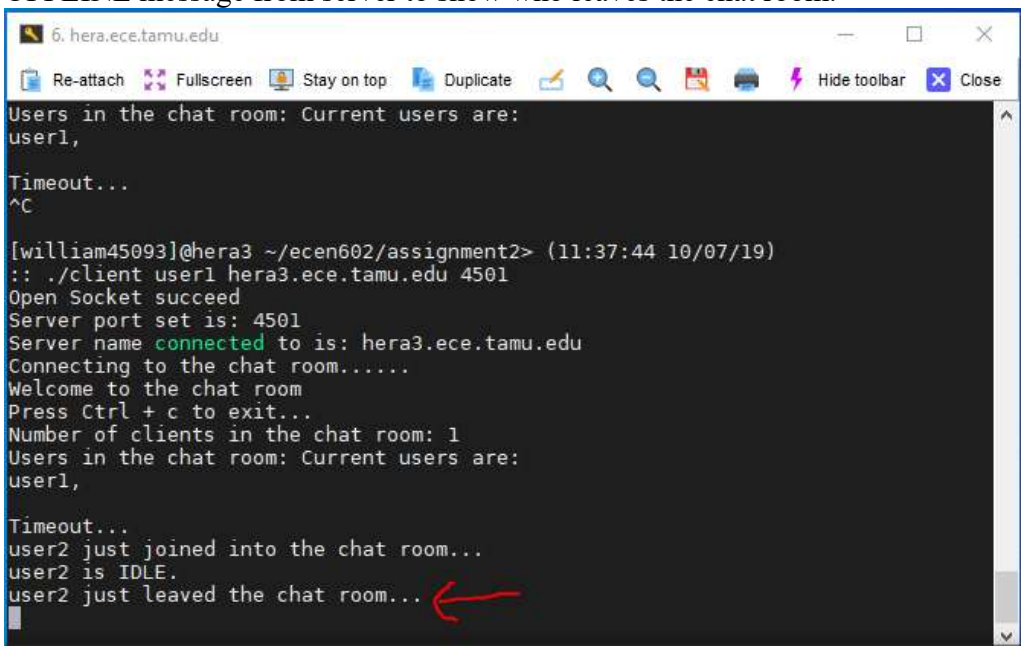
(2) NAK message from server to show the reason why the client could not join the chat room



(3) ONLINE message from server to show who joins the chat room

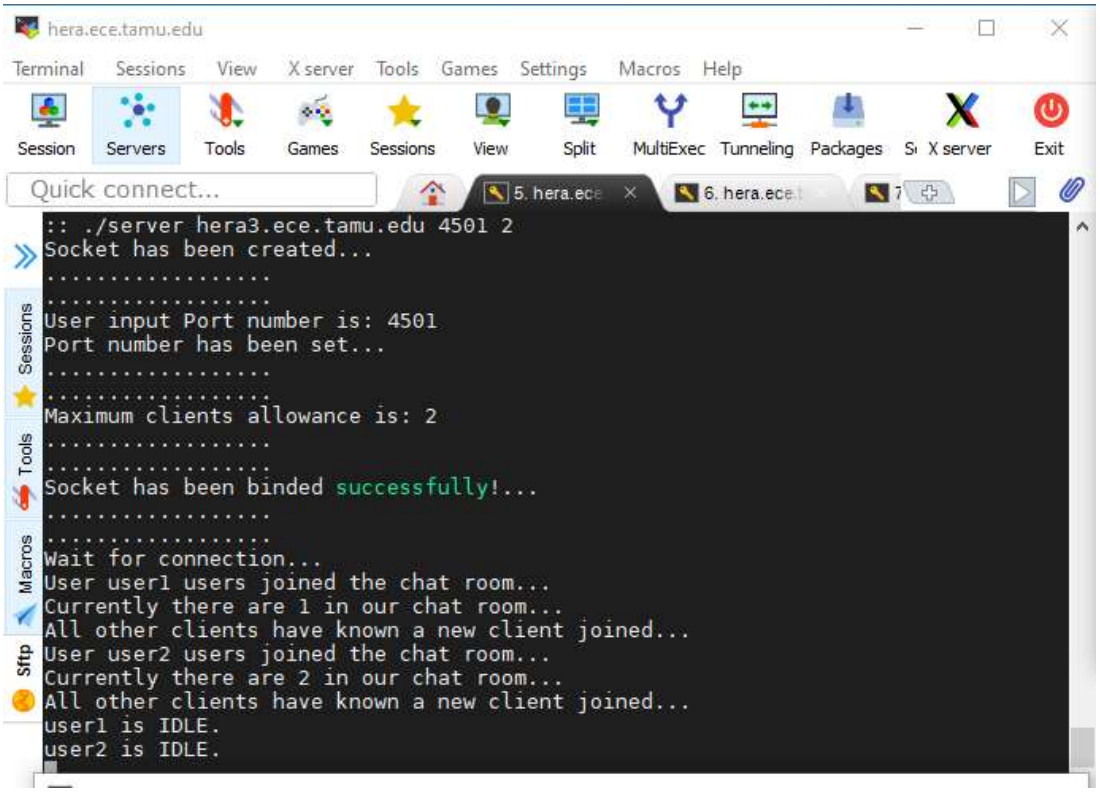


(4) OFFLINE message from server to show who leaves the chat room.



6. 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

```
9. hera.ece.tamu.edu
Re-attach  Fullscreen  Stay on top  Duplicate  Hide toolbar  Close

[william45093]@hera3 ~/ecen602/assignment2> (11:50:36 10/07/19)
:: ./client user2 hera3.ece.tamu.edu 4501
Open Socket succeed
Server port set is: 4501
Server name connected to is: hera3.ece.tamu.edu
Connecting to the chat room......
Welcome to the chat room
Press Ctrl + c to exit...
Number of clients in the chat room: 2
Users in the chat room: Current users are:
user1,
user2,

user1 is IDLE.
Timeout...
```

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.

```
9. hera.ece.tamu.edu
Re-attach  Fullscreen  Stay on top  Duplicate  Hide toolbar  Close

Open Socket succeed
Server port set is: 4501
Server name connected to is: hera3.ece.tamu.edu
Connecting to the chat room......
Welcome to the chat room
Press Ctrl + c to exit...
Number of clients in the chat room: 2
Users in the chat room: Current users are:
user1,
user2,

user1 is IDLE.
Timeout...   ← timeout
Hi           ← new message
user1 says: .
user1 says: .
user1 says: .
user1 says: .
user1 says: .
user1 says: .
user1 says: .
user1 says: .
user1 says: .
user1 says: .
Timeout...   ← timeout again
```

messages received from server

d. Test result

1. 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;
}
```

2. 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));
    }

}
```