Shikhar Chaurasia
Gunin Wasan

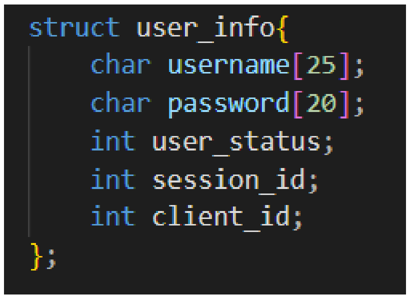
Text Conferencing Application

We implemented a Text Conferencing Application in Lab 4 and 5 as part of ECE361. We divided our workload equally between the server and client files. Our application fulfills all the essential functionalities mentioned in the handout, and as part of the additional functionalities, we implemented **private messaging** and **user registration.** Below is a breakdown of our implementation. First, we elaborate on the required functionalities (Lab 4), and then the additional ones (Lab 5).

Server

Our server implementation made use of synchronous IO Multiplexing, with help from the Beej's Programming Guide.

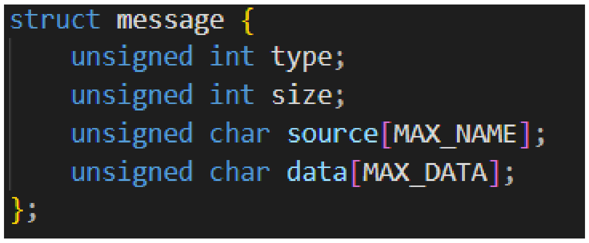
For keeping track of the users, it accesses a file called userInfoDetails.txt, which stored the user's credentials. Whenever a user tries to login, this file is looked up and login only proceeds if the credentials match. All the user's information is stored in an array of user_info, a struct defined as follows:



```
struct user_info{
    char username[25];
    char password[20];
    int user_status;
    int session_id;
    int client_id;
};
```

- user_status indicates if user is online or not (1 or 0)
- session_id shows which session a user is part of.
- client_id is the file descriptor that is used by a given user.

For any given command, most validation checks are done on the client side of things. (e.g., right number of commands, correct command spellings, etc.) Once they are done, a packet is sent in the format **commandType:size:source:data**, which is deserialized and fed into a message struct as below:



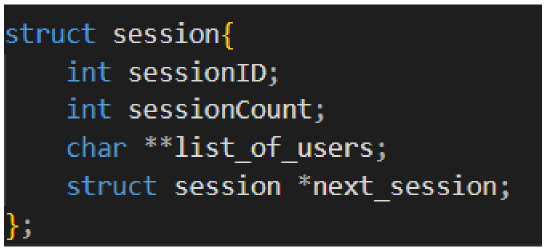
```
struct message {
    unsigned int type;
    unsigned int size;
    unsigned char source[MAX_NAME];
    unsigned char data[MAX_DATA];
};
```

- commandType: type of command (login, leavesession, etc)
- size: size of the message sent.
- source: username of the sender client
- data: the arguments along with the command sent.

Once this is decomposed, we check for the type of command against a list of acceptable commands, and upon match, we execute the necessary functionality.

A user sends messages in two forms: 1) in a session and 2) in private messages.

Our application's infrastructure for the sessions consists of a linked list of sessions, each of which contains:



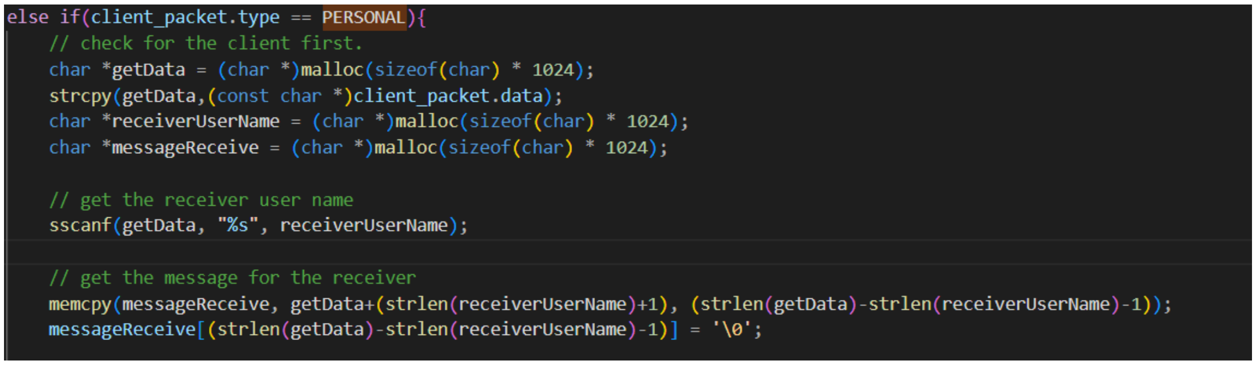
```
struct session{
    int sessionID;
    int sessionCount;
    char **list_of_users;
    struct session *next_session;
};
```

- sessionID is the unique session ID
- sessionCount is the number of people in a session (max = 10)
- list_of_users is an array of users in the session.

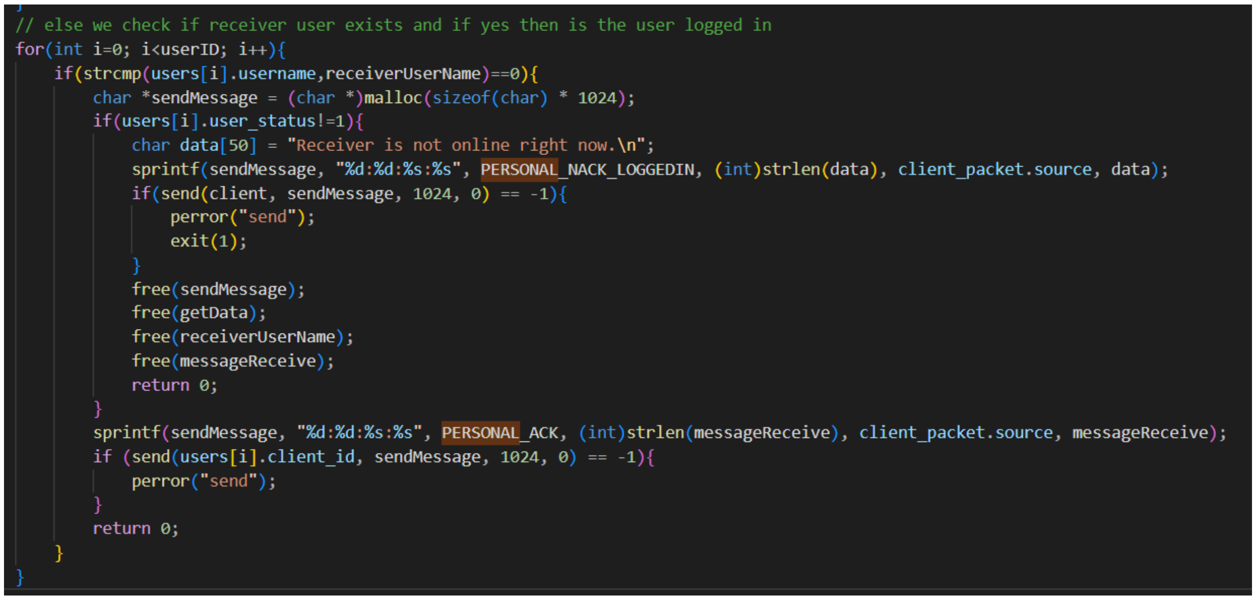
Our additional feature, **personal messages**, involves a client entering the following format of input:

/personal destination_user message

Once client side of things is handled, on the server side, necessary checks are made so that the receiver is valid, that they're both different users, and if they're logged in. Once all these checks are made, a packet is formed, and sent in the form of a serialized string, to the client_id of the target user intended to.



```c
else if(client_packet.type == PERSONAL){
    // check for the client first.
    char *getData = (char *)malloc(sizeof(char) * 1024);
    strcpy(getData,(const char *)client_packet.data);
    char *receiverUserName = (char *)malloc(sizeof(char) * 1024);
    char *messageReceive = (char *)malloc(sizeof(char) * 1024);

    // get the receiver user name
    sscanf(getData, "%s", receiverUserName);

    // get the message for the receiver
    memcpy(messageReceive, getData+(strlen(receiverUserName)+1), (strlen(getData)-strlen(receiverUserName)-1));
    messageReceive[(strlen(getData)-strlen(receiverUserName)-1)] = '\0';
```



```c
// else we check if receiver user exists and if yes then is the user logged in
for(int i=0; i<userID; i++){
    if(strcmp(users[i].username,receiverUserName)==0){
        char *sendMessage = (char *)malloc(sizeof(char) * 1024);
        if(users[i].user_status!=1){
            char data[50] = "Receiver is not online right now.\n";
            sprintf(sendMessage, "%d:%d:%s:%s", PERSONAL_NACK_LOGGEDIN, (int)strlen(data), client_packet.source, data);
            if(send(client, sendMessage, 1024, 0) == -1){
                perror("send");
                exit(1);
            }
            free(sendMessage);
            free(getData);
            free(receiverUserName);
            free(messageReceive);
            return 0;
        }
        sprintf(sendMessage, "%d:%d:%s:%s", PERSONAL_ACK, (int)strlen(messageReceive), client_packet.source, messageReceive);
        if (send(users[i].client_id, sendMessage, 1024, 0) == -1){
            perror("send");
        }
        return 0;
    }
}
```

Other commands such as /leavesession, /joinsession, etc. involve moving around the user from one node in the sessions linked list to another, creation, and removal of nodes, etc.

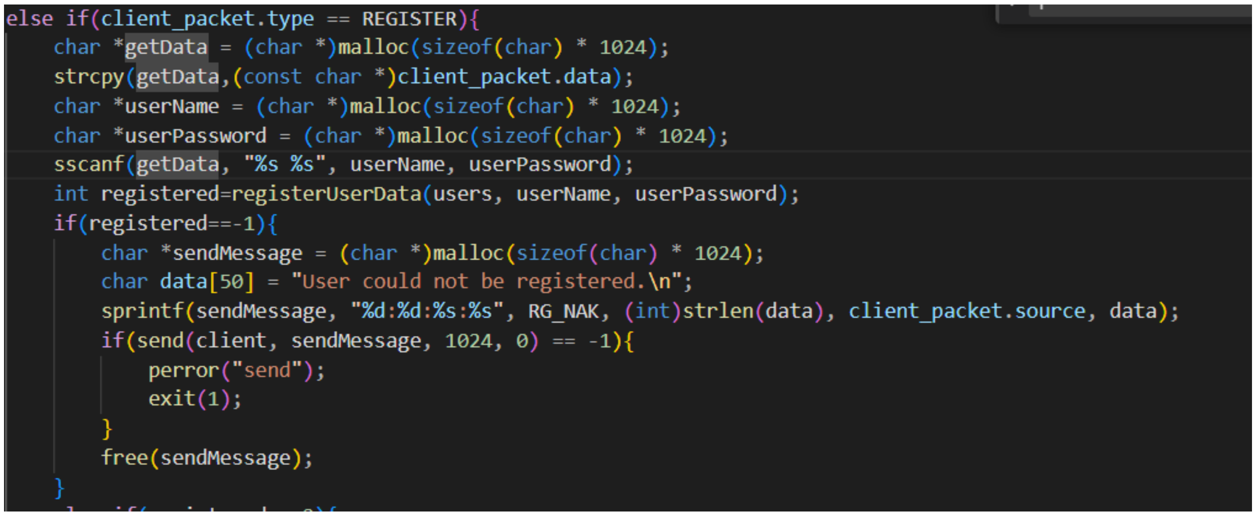
The second additional feature is **registering users**, using /register. Client enters the input in the following format:

/register user_name password

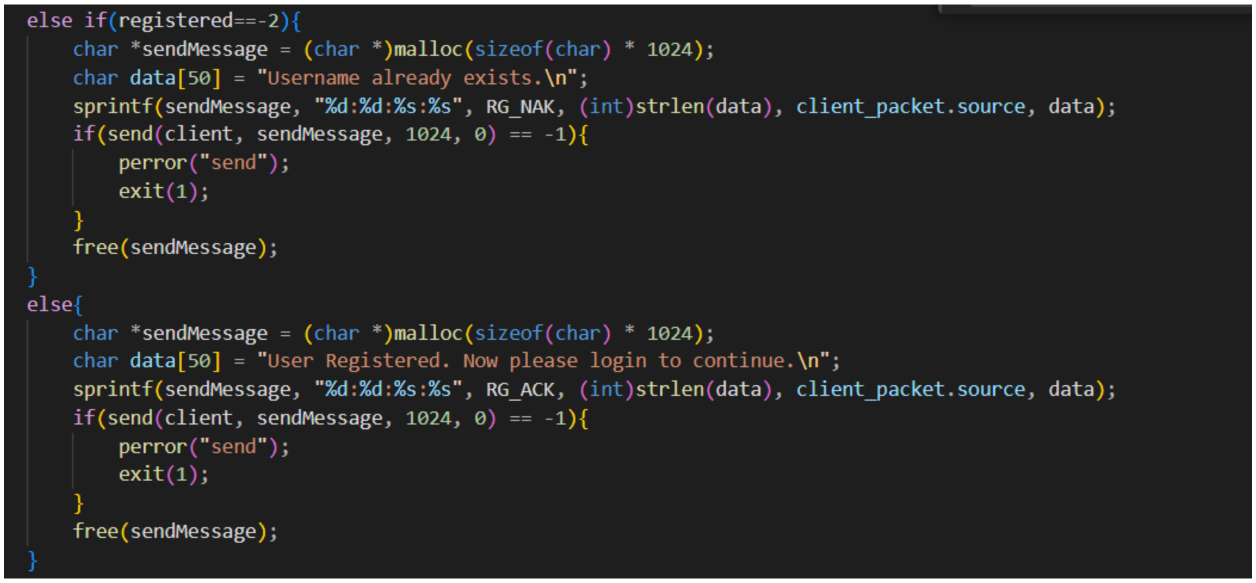
We first gather the details of the user to be registered following deserialization, after which we make some checks to ensure if the user has been registered or not. The user information (usernames and passwords of all users) is stored in a file userInfoDetails.txt. This is done so that the login information is persistent, so even when the server stops, the login information is still available.

There are also checks placed to see if a user has been registered already. If so, then we don't allow them to register again. This is done in correspondence to the userInfoDetails.txt, which stores the username and password for each user.

Once all these checks pass, we proceed to register the user successfully, and write into the userInfoDetails.txt. This allows the registered user to use the application as any other user.



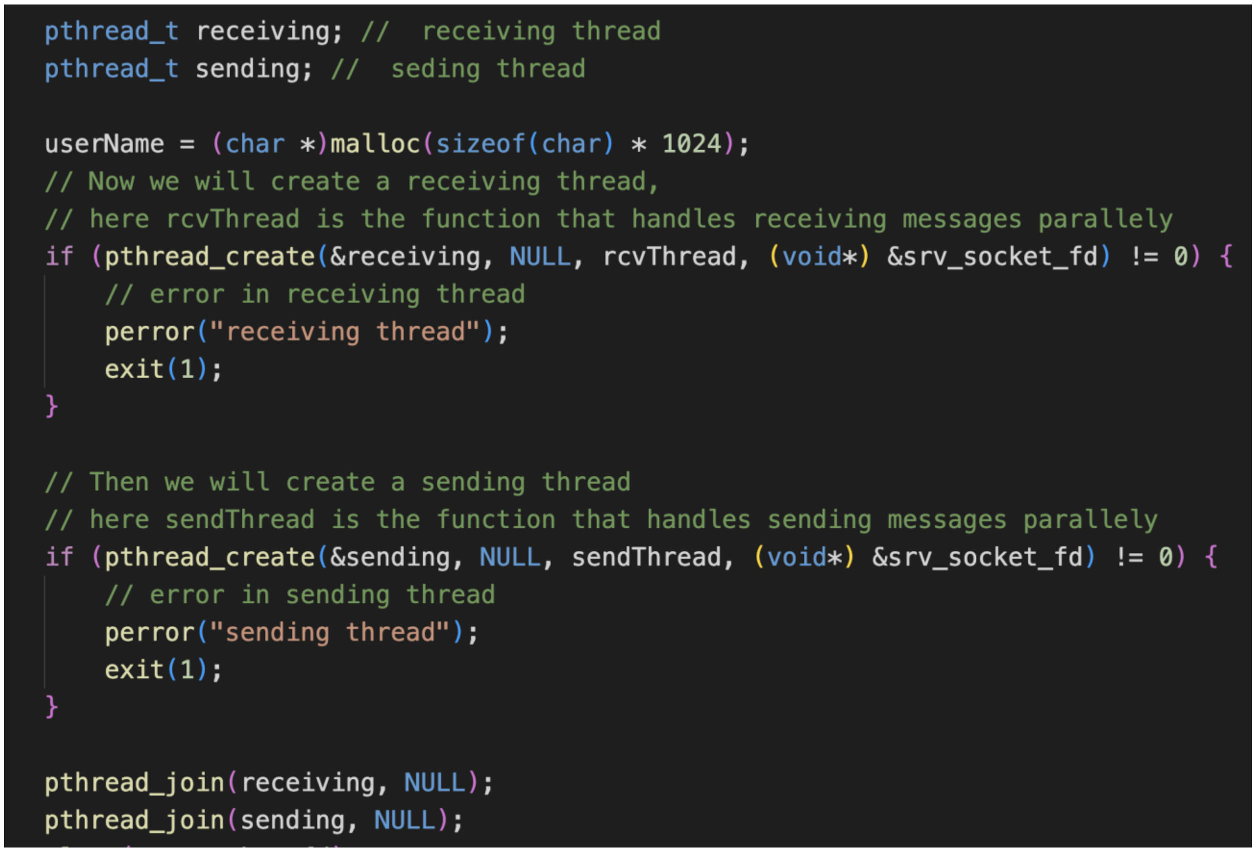
```
else if(client_packet.type == REGISTER){
    char *getData = (char *)malloc(sizeof(char) * 1024);
    strcpy(getData,(const char *)client_packet.data);
    char *userName = (char *)malloc(sizeof(char) * 1024);
    char *userPassword = (char *)malloc(sizeof(char) * 1024);
    sscanf(getData, "%s %s", userName, userPassword);
    int registered=registerUserData(users, userName, userPassword);
    if(registered==-1){
        char *sendMessage = (char *)malloc(sizeof(char) * 1024);
        char data[50] = "User could not be registered.\n";
        sprintf(sendMessage, "%d:%d:%s:%s", RG_NAK, (int)strlen(data), client_packet.source, data);
        if(send(client, sendMessage, 1024, 0) == -1){
            perror("send");
            exit(1);
        }
        free(sendMessage);
    }
```



```
else if(registered==-2){
    char *sendMessage = (char *)malloc(sizeof(char) * 1024);
    char data[50] = "Username already exists.\n";
    sprintf(sendMessage, "%d:%d:%s:%s", RG_NAK, (int)strlen(data), client_packet.source, data);
    if(send(client, sendMessage, 1024, 0) == -1){
        perror("send");
        exit(1);
    }
    free(sendMessage);
}
else{
    char *sendMessage = (char *)malloc(sizeof(char) * 1024);
    char data[50] = "User Registered. Now please login to continue.\n";
    sprintf(sendMessage, "%d:%d:%s:%s", RG_ACK, (int)strlen(data), client_packet.source, data);
    if(send(client, sendMessage, 1024, 0) == -1){
        perror("send");
        exit(1);
    }
    free(sendMessage);
}
```

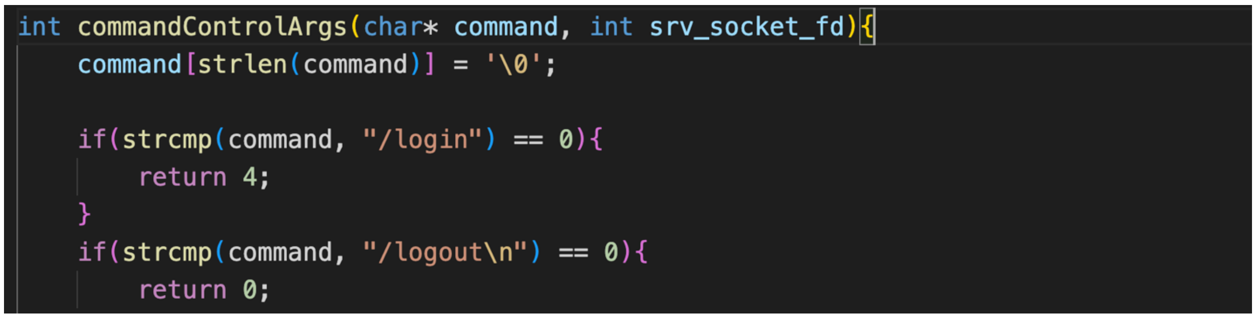
Client

Our client implementation involves the use of multithreading, where we created 2 threads – one for receiving messages from the server and one for sending messages to the server.



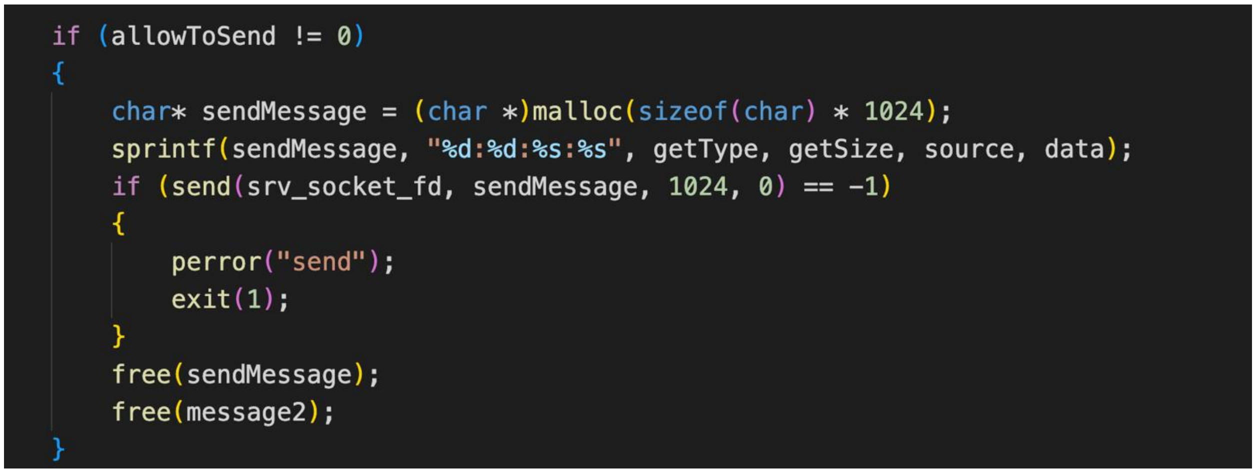
```
pthread_t receiving; //  receiving thread
pthread_t sending; //  seding thread

userName = (char *)malloc(sizeof(char) * 1024);
// Now we will create a receiving thread,
// here rcvThread is the function that handles receiving messages parallely
if (pthread_create(&receiving, NULL, rcvThread, (void*) &srv_socket_fd) != 0) {
    // error in receiving thread
    perror("receiving thread");
    exit(1);
}

// Then we will create a sending thread
// here sendThread is the function that handles sending messages parallely
if (pthread_create(&sending, NULL, sendThread, (void*) &srv_socket_fd) != 0) {
    // error in sending thread
    perror("sending thread");
    exit(1);
}

pthread_join(receiving, NULL);
pthread_join(sending, NULL);
```

The client-side checks for a command and/or a message within the file itself. If the input is a command, they are validated with correct number of arguments and is then sent to the server.



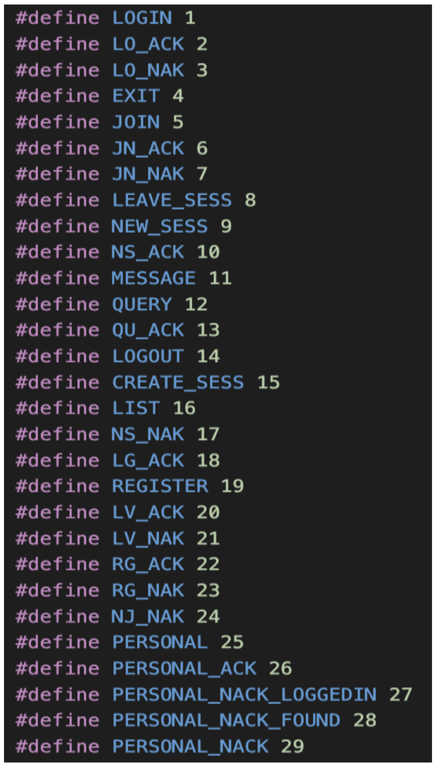
```
int commandControlArgs(char* command, int srv_socket_fd){
    command[strlen(command)] = '\0';

    if(strcmp(command, "/login") == 0){
        return 4;
    }
    if(strcmp(command, "/logout\n") == 0){
        return 0;
```

When the command or message meets all the checks, then a packet is sent to the server in a in the format **commandType:size:source:data**, which as described above in the server, is deserialized.



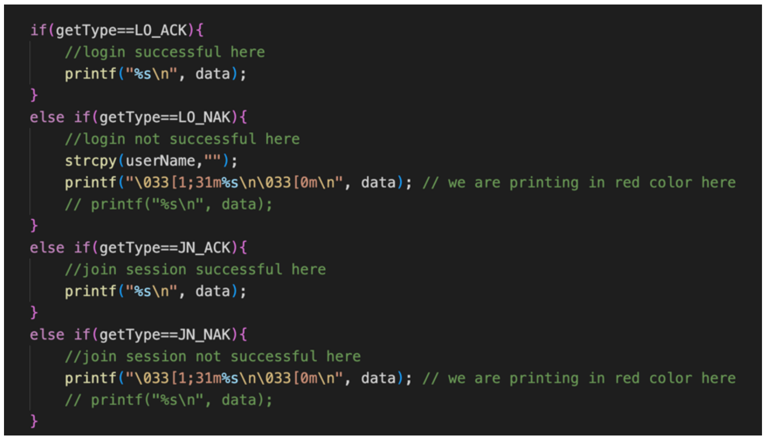
```c
if (allowToSend != 0)
{
    char* sendMessage = (char *)malloc(sizeof(char) * 1024);
    sprintf(sendMessage, "%d:%d:%s:%s", getType, getSize, source, data);
    if (send(srv_socket_fd, sendMessage, 1024, 0) == -1)
    {
        perror("send");
        exit(1);
    }
    free(sendMessage);
    free(message2);
}
```

Now for the receiving thread, the messages received from the server are of the format **ACK:size:source:data or NACK:size:source:data.**
Here, ACK and NACK defined are one of the defined numbers below. As the client



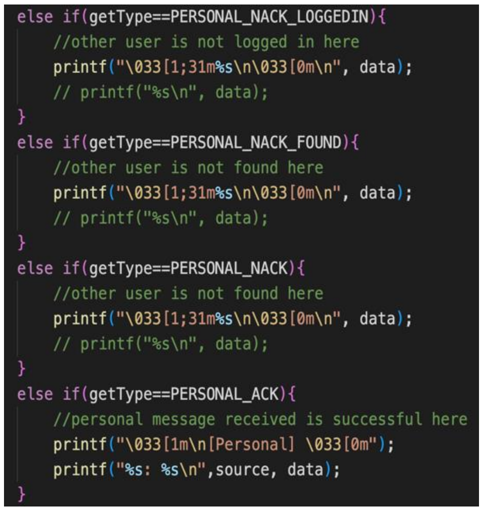
```c
#define LOGIN 1
#define LO_ACK 2
#define LO_NAK 3
#define EXIT 4
#define JOIN 5
#define JN_ACK 6
#define JN_NAK 7
#define LEAVE_SESS 8
#define NEW_SESS 9
#define NS_ACK 10
#define MESSAGE 11
#define QUERY 12
#define QU_ACK 13
#define LOGOUT 14
#define CREATE_SESS 15
#define LIST 16
#define NS_NAK 17
#define LG_ACK 18
#define REGISTER 19
#define LV_ACK 20
#define LV_NAK 21
#define RG_ACK 22
#define RG_NAK 23
#define NJ_NAK 24
#define PERSONAL 25
#define PERSONAL_ACK 26
#define PERSONAL_NACK_LOGGEDIN 27
#define PERSONAL_NACK_FOUND 28
#define PERSONAL_NACK 29
```

receives the message from the server, it is passed through a function to identify the command that the ACK/NACK is sent in reference to. The message is printed on the client side accordingly, then.

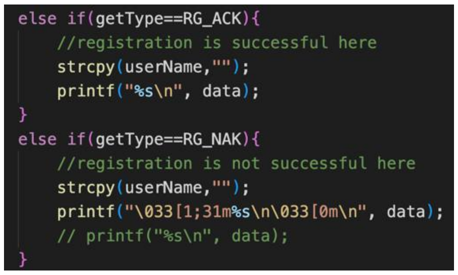


```
if(getType==LO_ACK){
    //login successful here
    printf("%s\n", data);
}
else if(getType==LO_NAK){
    //login not successful here
    strcpy(userName,"");
    printf("\033[1;31m%s\n\033[0m\n", data); // we are printing in red color here
    // printf("%s\n", data);
}
else if(getType==JN_ACK){
    //join session successful here
    printf("%s\n", data);
}
else if(getType==JN_NAK){
    //join session not successful here
    printf("\033[1;31m%s\n\033[0m\n", data); // we are printing in red color here
    // printf("%s\n", data);
}
```

For the additional features, like register and personal, there are some additional ACK/NACK defined.



```
else if(getType==PERSONAL_NACK_LOGGEDIN){
    //other user is not logged in here
    printf("\033[1;31m%s\n\033[0m\n", data);
    // printf("%s\n", data);
}
else if(getType==PERSONAL_NACK_FOUND){
    //other user is not found here
    printf("\033[1;31m%s\n\033[0m\n", data);
    // printf("%s\n", data);
}
else if(getType==PERSONAL_NACK){
    //other user is not found here
    printf("\033[1;31m%s\n\033[0m\n", data);
    // printf("%s\n", data);
}
else if(getType==PERSONAL_ACK){
    //personal message received is successful here
    printf("\033[1m\n[Personal] \033[0m");
    printf("%s: %s\n",source, data);
}
```



```
else if(getType==RG_ACK){
    //registration is successful here
    strcpy(userName,"");
    printf("%s\n", data);
}
else if(getType==RG_NAK){
    //registration is not successful here
    strcpy(userName,"");
    printf("\033[1;31m%s\n\033[0m\n", data);
    // printf("%s\n", data);
}
```