EEE442-FINAL PROJECT REPORT

Utku Acar – 250206062
Egemen Uluçam – 240206034

We have updated our old project which belongs to **CENG421 Computer Network Programming** Course. The "OLD" refers to the function has been used in old project, "UPDATED" refers to the function has been used in old project and we are still using it with some updates, and "NEW" refers to the function has not been used before and we are using it for now on.

Function Descriptions:

Client Structure:

Our client structure had some basic features like storing its own user ID, name, IP address, connection file descriptor, socket address. We added group IDs and some variables for our mini games on our server (Lines [34-40]) as you can see in **Figure 1**.

```
29  typedef struct {//Client structure  UPDATED
30      struct sockaddr_in addr;     //Client remote address
31      int connfd;                 //Connection file descriptor
32      int user_id;               //User identifier
33      int group_id;             //Group 0 is lobby, updates when group change
34      char name[50];           //Client name-nick
35      int msgtogrp;           //Flag of message range
36      float rep;             //Reputation Point
37      float health;         //Health of the User
38      float attack_power;  //Power of the User
39      float luck;         //Luck of the user
40  } client_t;
```

Figure 1: Client Struct

q_adr(.):

This function adds new clients to the client array which is a global variable such that it stores all possible clients, as you can see in **Figure 2**.

```
159  void q_adr(client_t *client){//Add client to queue when client connected OLD
160      int i;
161      pthread_mutex_lock(&clients_mutex);
162      for (i = 0; i < C_NUM; i++) {
163          if (!client_array[i]) {
164              client_array[i] = client;
165              break;
166          }
167      }
168      pthread_mutex_unlock(&clients_mutex);
169  }
```

Figure 2: Adding to the Queue Function

q_dlt(.):

This function removes old clients from the client array (**Figure 3**).

```
171  void q_dlt(int user_id){//Removing client from the list if client quits OLD
172      int i;
173      pthread_mutex_lock(&clients_mutex);
174      for (i = 0; i < C_NUM; i++) {
175          if (client_array[i]) {
176              if (client_array[i]->user_id == user_id) {
177                  client_array[i] = NULL;
178                  break;
179              }
180          }
181      }
182      pthread_mutex_unlock(&clients_mutex);
183  }
```

Figure 3: Removing from the Queue Function

send_msg_all(.):

The server sends messages to all clients with this function (**Figure 4).**

```
225  void send_msg_all(char *str){//Send message to all clients  OLD
226      int i;
227      pthread_mutex_lock(&clients_mutex);
228      for (i = 0; i <C_NUM; i++){
229          if (client_array[i]) {
230              if (write(client_array[i]->connfd, str, strlen(str)) < 0) {
231                  perror("Sending message to all clients error");
232                  break;
233              }
234          }
235      }
236      pthread_mutex_unlock(&clients_mutex);
237  }
```

Figure 4: Messaging from Server to all Clients Function

msg_from_sv_to_clt(.):

A server sends messages to a specific client with this function (**Figure 5**).

```
246  void msg_from_sv_to_clt(char *str, int user_id){//Sending message to client OLD
247      int i;
248      pthread_mutex_lock(&clients_mutex);
249      for (i = 0; i < C_NUM; i++){
250          if (client_array[i]) {
251              if (client_array[i]->user_id == user_id) {
252                  if (write(client_array[i]->connfd, str, strlen(str))<0) {
253                      perror("Message from server to client error ");
254                      break;
255                  }
256              }
257          }
258      }
259      pthread_mutex_unlock(&clients_mutex);
260  }
```

Figure 5: Messaging from Server to a Specific Client Function

forward_message(.)

This function sent one client message to the other clients before **(Figure 6)**. Since we added encryption and private rooms features to the project, we are sending the message to the other clients in an encrypted way. Encryption will be explained later.

```
185  void forward_message(char *str, int user_id){//Forwarding messages to other clients UPDATED
186      int key, i;
187      char buff_out[B_SIZE];
188      pthread_mutex_lock(&clients_mutex);
189      for (i = 0; i < C_NUM; i++) {
190          buff_out[0] = '\0';//initializing
191          if (client_array[i]) {
192              if (client_array[i]->user_id != user_id) {//until equals self
193                  key=enc_key_array[client_array[i]->group_id];//reaching session key which is special to group
194                  snprintf(buff_out, sizeof(buff_out), "[%s] %s\r\n", client_array[user_id-1]->name, ceaser_dec(str, key));
195                  if (write(client_array[i]->connfd, buff_out, strlen(buff_out)) < 0) {
196                      perror("Forwarding error");
197                      break;
198                  }
199              }
200          }
201      }
202      pthread_mutex_unlock(&clients_mutex);
203  }
```

Figure 6: Message Forwarding Function

forward_message_self(.)

This function sends messages to the client himself/herself (**Figure 7**).

```
239  void forward_message_self(const char *str, int connfd){//Send message to self   OLD
240      if (write(connfd, str, strlen(str)) < 0) {
241          perror("Forwarding message to self error");
242          exit(-1);
243      }
244  }
```

Figure 7: Message Forwarding Function

group_message(.)

This function sends messages to a specific group from the Server (**Figure 8**).

```
262  void group_message(char *str,int group_id){//Sending message to group   NEW
263      int i;
264      pthread_mutex_lock(&clients_mutex);
265      for (i = 0; i < C_NUM; i++){
266          if (client_array[i]) {
267              if (client_array[i]->group_id == group_id) {
268                  if (write(client_array[i]->connfd, str, strlen(str))<0) {
269                      perror("Group message error");
270                      break;
271                  }
272              }
273          }
274      }
275      pthread_mutex_unlock(&clients_mutex);
276  }
```

Figure 8: Messaging from Server to a Specific Group Function

group_forward(.)

This function sends one client's message to other clients in the same group in an encrypted way (**Figure 9**).

```
205  void group_forward(char *str, int user_id, int group_id){//     NEW
206      int key, i;
207      char buff_out[B_SIZE];
208      pthread_mutex_lock(&clients_mutex);
209      for (i = 0; i < C_NUM; i++) {
210          buff_out[0] = '\0';//initializing
211          if (client_array[i]) {
212              if ((client_array[i]->user_id != user_id) && (client_array[i]->group_id== group_id)) {//until equals self
213                  key=enc_key_array[client_array[i]->group_id];
214                  snprintf(buff_out, sizeof(buff_out), "[%s] %s\r\n", client_array[user_id-1]->name, ceaser_dec(str, key));
215                  if (write(client_array[i]->connfd, buff_out, strlen(buff_out)) < 0) {
216                      perror("Forwarding error");
217                      break;
218                  }
219              }
220          }
221      }
222      pthread_mutex_unlock(&clients_mutex);
223  }
```

Figure 9: Messaging from Server to a Specific Group Function

create(.)

This function creates a group with a given password (**Figure 10**)**.** Passwords are stored in a global array. When a new group is created, the group number has been determined by looking for the lowest available group number by checking first indices of whether the group is zero or not. If its zero, that means the group is available; if not the group is already created. The client who created the group becomes the admin of that group.

```
834  int create(int password,int user_id){//Creating a Group with password       NEW
835      int i,j;
836      if(group[GROUP_NUM-1][0] != 0){
837          return -1;
838      }
839      else{
840          for(i=1;i<GROUP_NUM;i++){
841              if (group[i][0] == 0){
842                  group[i][0] = user_id;
843                  password_arr[i] = password;
844                  printf("The Group %d has been created with %d\n",i,password);
845                  return i;
846              }
847          }
848      }
849  }
```

Figure 10: Creating a New Group Function

join(.)

This function is used to join a group with a given group ID and password **(Figure 11).** If there is no group with the given group ID function returns (-1) in client_handler function and client gets a message from the server as “There is no group with that number”. If password is wrong function returns (0) and server sends "Your password is wrong” to the client in client_handler function. If the password is correct but the group is full function returns (2) and client gets "This group is full” message. Lastly, if the password is correct and the group is not full, the client joins to the group.

```
851  int join(int group_id,int password,int user_id){//Joining a Group with a password   NEW
852      int i;
853
854      if(group[group_id][0] == 0){
855          return -1;
856      }
857      else{
858          if(password == password_arr[group_id]){
859              if(group[group_id][GROUP_MEM_NUM] != 0){
860                  return 2;
861              }
862              else{
863                  for(i=1;i<GROUP_MEM_NUM;i++){
864                      if(group[group_id][i] == 0){
865                          group[group_id][i]=user_id;
866                          return 1;
867                      }
868                  }
869              }
870          }
871          else if(password != password_arr[group_id]){
872              return 0;
873          }
874      }
875
876  }
```

Figure 11: Joining a Group Function

disconnect(.)

This function used to disconnecting a group (**Figure 12**). Clients in the group array shifts one index to fill the empty index in the group array. If the client who disconnects the group is the admin of that group, the oldest client who joined the group becomes the admin.

```
878  void disconnect(int group_id, int user_id ){//Disconnecting from  a Group   NEW
879      int i,j;
880      int admin_check = 0;
881      for (i = 0; i < GROUP_MEM_NUM, group[group_id][i] != 0; i++) {
882          if(group[group_id][i] == user_id){
883              if(i == 0)  admin_check = 1;
884              for (; i < GROUP_MEM_NUM, group[group_id][i] != 0; i++){
885                  if (i == GROUP_MEM_NUM - 1) {
886                      group[group_id][i] = 0;
887                  }
888                  else {
889                      group[group_id][i] = group[group_id][i+1];
890                  }
891              }
892          }
893      }
894      if (admin_check == 1)
895          msg_from_sv_to_clt("You are the Admin now. Have Fun.\r\n", group[group_id][0]);
896  }
```

Figure 12: Disconnecting from Group Function

active_group_list(.)

This function shows active groups to the client who wants to learn with forward_message_self function (Figure 13). To understand if a group is active or not, we check first the index of the groups. Active groups always have a client as admin in their first index.

```
314  void active_group_list(int connfd){//Send list of active groups NEW
315      char s[64];
316      int i, counter = 0;
317      pthread_mutex_lock(&clients_mutex);
318      for (i = 0; i < GROUP_NUM; i++){
319          if (group[i][0] != 0) {
320              counter++;
321              sprintf(s, "Group[%d] is Active\r\n", i);
322              forward_message_self(s, connfd);
323          }
324      }
325      if (counter == 0){
326          forward_message_self("There is no Active group. To create one </create>\r\n", connfd);
327      }
328      pthread_mutex_unlock(&clients_mutex);
329  }
```

Figure 13: Showing a List of Active Groups Function

active_client_list(.)

This function shows active clients to the client who wants to learn with *forward_message_self* function (**Figure 14**). While using this function the client can list all the clients on the server or only the group he/she involved.

```
278  void active_client_list(int connfd, int group_id){//Send list of active clients UPDATED
279      char s[128];
280      int i;
281      pthread_mutex_lock(&clients_mutex);//for sync between clients
282      if (group_id == -1){//for everyone
283          for (i = 0; i < C_NUM; i++){
284              if (client_array[i]) {
285                  sprintf(s, " [%d] %s\r\n", client_array[i]->user_id, client_array[i]->name);
286                  forward_message_self(s, connfd);
287              }
288          }
289      }
290      else if (group_id == 0) {//for request of group member list from lobby
291          forward_message_self("You are not in a group.\r\nUse </list> command to see all users.\r\n", connfd);
292      }
293      else { // for seeing group members in same group
294      int counter = 0;
295          for (i = 0; i < GROUP_MEM_NUM; i++) {
296              if (group[group_id][i]){
297                  counter++;//active group member
298                  if(i == 0){//If member is first member of a group(ADMIN)
299                      sprintf(s, "[ADMIN]  [%d] %s\r\n", group[group_id][i], client_array[group[group_id][i]-1]->name);
300                      forward_message_self(s, connfd);
301                  }
302                  else {// Other users(MEMBERS)
303                      sprintf(s, "[MEMBER] [%d] %s\r\n", group[group_id][i], client_array[group[group_id][i]-1]->name);
304                      forward_message_self(s, connfd);
305                  }
306              }
307          }
308          sprintf(s, "Total number of member this group is %d\r\n", counter);
309          forward_message_self(s, connfd);
310      }
311      pthread_mutex_unlock(&clients_mutex);//unlock the thread
312  }
```

Figure 14: Showing a List of Clients Function

ceaser_enc(.)

This function encrypts the message with a given key (**Figure 15**). Key is generated randomly everytime when a client sends a message and it's stored into enc_key_array with client's group ID index.

```
141  void ceaser_enc(char arr[],int key){//Ceaser cryption   NEW
142      int i;
143      for(i = 0; i < strlen(arr); i++)
144      {
145          arr[i]= arr[i] + key;
146      }
147  }
```

Figure 15: Encrypting a Meassage Function

ceaser_dec(.)

This function decrypts the message with a given key (**Figure 16**). This function used in *forward_message* and *group_forward* functions. Keys are found from enc_key_array[client's group ID].

```
149  char *ceaser_dec(char arr[],int key){//Ceaser decryption    NEW
150      int i;
151      char *ptr = str_duplicate(arr);
152      for(i = 0; i < strlen(arr); i++)
153      {
154          ptr[i] = ptr[i] - key;
155      }
156      return ptr;
157  }
```

Figure 16: Decrypting a Meassage Function

fight(.)

This function **(Figure 17)** is something we did for fun. Clients can play a mini game between themselves. Each client has their own health points, attack points, and reputation points. Clients can challenge each other to a duel and fight according to these points. Winner takes defeated client's reputation points. Then the defeated player is automatically kicked from the group. Reputation points can be traded for health and attack points. Attack turns in the fight are determined randomly to make the fight fair. Someone with lower attack and health points can win against his/her rival with more attack and health points.

```
908  int fight(int duelist_1,int duelist_2){//   NEW
909      duelist_1= duelist_1 - 1;
910      duelist_2= duelist_2 - 1;
911      int turn;
912      srand(time(NULL));
913      while(client_array[duelist_1]->health > 0.0 && client_array[duelist_2]->health > 0.0){
914          turn = (rand() % 2) + 1;
915          if(turn == 2){
916              client_array[duelist_1]->health=client_array[duelist_1]->health - client_array[duelist_2]->attack_power;
917              client_array[duelist_2]->health=client_array[duelist_2]->health - client_array[duelist_1]->attack_power;
918          }
919          else if(turn == 1){
920              client_array[duelist_2]->health=client_array[duelist_2]->health - client_array[duelist_1]->attack_power;
921              client_array[duelist_1]->health=client_array[duelist_1]->health - client_array[duelist_2]->attack_power;
922          }
923      }
924      if(client_array[duelist_1]->health <= 0.0){
925          return 2;
926      }
927      else if(client_array[duelist_2]->health <= 0.0){
928          return 1;
929      }
930      else {
931          return 0;
932      }
933  }
```

Figure 17: Fight Function

**str_duplicate(.)*

This function copies a string (**Figure 18**).

```
132 char *str_duplicate(const char *str) {//Storing a copy of the input OLD
133     size_t size = strlen(str) + 1;
134     char *ptr = malloc(size);
135     if (ptr) {
136         memcpy(ptr, str, size);
137     }
138     return ptr;
139 }
```

Figure 18: Copying String Function

string_ender(.)

This function prevents empty messages from being sent to other clients (**Figure 19**).

```
331 void string_ender(char *str){//String ender(adding \0 while necessary ) OLD
332     while (*str != '\0') {// scan until last element of string
333         if (*str == '\n' || *str == '\r') { //if newline then close the string (/0)(replace)
334             *str = '\0';
335         }
336         str++;
337     }
338 }
```

Figure 19: String Ender Function

print_client_addr(.)

This function shows IP adress of client (**Figure 20**).

```
340 void print_client_addr(struct sockaddr_in addr){//Printing the ip address by shifting 8 bit recursively OLD
341     printf("%d.%d.%d.%d",
342     addr.sin_addr.s_addr & 0xff,
343     (addr.sin_addr.s_addr & 0xff00) >> 8,
344     (addr.sin_addr.s_addr & 0xff0000) >> 16,
345     (addr.sin_addr.s_addr & 0xff000000) >> 24);
346 }
```

Figure 20: Printing IP Address of Client Function

toggler(.)

This function toggles the input and returns the result (**Figure 21**).

```
902    int toggler(int input){//   NEW
903        if(input == 1){
904            input = 0;
905        }
906        else if(input == 0){
907            input = 1;
908        }
909        return input;
910    }
```

Figure 21: Toggling Function

Client_handler (.)

This function provides communication between clients to clients and clients to servers. Most of the other functions we mentioned above are used in this function, such as sending messages to groups or everyone, listing active groups or clients, encrypting and decrypting the message. There are commands to use these features and to see these commands the client can type "/help" (**Figure 22**). Also, when a client first connects to the server, he/she gets the information about "/help" command.

```
779             } else if (strcmp(command, "/help") == 0) {
780                 strcat(buff_out, "/quit ==========> Exit from server\r\n");
781                 strcat(buff_out, "/info ==========> Ask your info to server\r\n");
782                 strcat(buff_out, "/topic ==========> <topic_msg> Create a topic\r\n");
783                 strcat(buff_out, "/nick ==========> <name> Create a nickname\r\n");
784                 strcat(buff_out, "/list ==========> Show connected clients\r\n");
785                 strcat(buff_out, "/msg ==========> <user_id>(number) <msg> Send private message\r\n");
786                 strcat(buff_out, "/create =========> <password> Create a group with password(number only)\r\n");
787                 strcat(buff_out, "/join ==========> <group_id> <password> Join a group with password(number only)\r\n");
788                 strcat(buff_out, "/dc ==========> disconnected from group\r\n");
789                 strcat(buff_out, "/grp_mem ========> Show connected clients in the group members\r\n");
790                 strcat(buff_out, "/grp_list =======> Show active groups\r\n");
791                 strcat(buff_out, "/grp_m ==========> Send one message to the group\r\n");
792                 strcat(buff_out, "/grp_a ==========> [default=everyone]Toggles the destination of the messages Group or everyone\r\n");
793                 strcat(buff_out, "/kick ==========> <user_id>(number) Kicks a member of group (Admin command) \r\n");
794                 strcat(buff_out, "/dice ==========> <face_number> Rolls a die\r\n");
795                 strcat(buff_out, "/rps ==========> <choose>(r or p or s) Rock paper scissors\r\n");
796                 strcat(buff_out, "/duel ==========> <user_id> Invite to someone to Duel\r\n");
797                 strcat(buff_out, "/accept ==========> <user_id> Accept Duel request from user\r\n");
798                 strcat(buff_out, "/hp ==========>  <Requested health> Trading health points with rep points \r\n");
799                 strcat(buff_out, "/attack ==========>  <Requested attack points> Trading attack points with rep points \r\n");
800                 strcat(buff_out, "/help ==========> Show help\r\n");
801                 forward_message_self(buff_out, client_pointer->connfd);
802             }
```

Figure 22: "/help" Command

main(.)

In this TCP server we create variables to establish a connection from lines 67 to 71. At line 72 we check if the input in the command line to execute the project has exactly two arguments; if not, we print the usage. To open the server, a port number is needed to enter while executing. Then, from lines 77 to 93, we adjust the socket settings as it can be seen in **Figure 23**. After that server is become ready to listen clients.

```
66  int main(int argc, char *argv[]){
67      int listenfd = 0, connfd = 0;
68      int i;
69      struct sockaddr_in serv_addr;
70      struct sockaddr_in client_address;
71      pthread_t thread_id;
72     if (argc != 2) {
73      fprintf(stderr, "Usage: %s <port>\n", argv[0]);
74      exit(1);
75     }
76      /* Socket settings */
77      listenfd = socket(AF_INET, SOCK_STREAM, 0);//storing socket
78      serv_addr.sin_family = AF_INET;
79      serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);//default IP
80      serv_addr.sin_port = htons(atoi(argv[1]));// for giving port number manually
81
82
83      signal(SIGPIPE, SIG_IGN);// for early disconnections(Ignore pipe signals)
84
85
86      if (bind(listenfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) < 0) {//Binding sockets
87          perror("Socket binding error");
88          return FAIL;
89      }
90      if (listen(listenfd, 10) < 0) {//Listening sockets
91          perror("Socket listen error");
92          return FAIL;
93      }
94
95      printf("Server has been started\n");
```

Figure 23: main Function (Part I)

In while loop server waits and listens clients (**Figure 24**). When a new client tries to connect line 97 and 98, assign a connfd (Connection file descriptor) but if maximum client is achieved, server closes the connection. After checking the maximum client number, a new client struct variable is created and filled with unique personal information. Then add the client to the queue (client array). After that creates a thread for the client at line 126 to handle all clients in parallel.

```
96      while (1) {//Accepting clients
97          socklen_t client_address_len = sizeof(client_address);
98          connfd = accept(listenfd, (struct sockaddr*)&client_address, &client_address_len);
99          if ((client_count + 1) == C_NUM) {//Checking the client capacity of the server
100             printf(" Maximum number of clients has been connected\n");
101             printf(" Rejection due to out of capacity!! ");
102             print_client_addr(client_address);
103             printf("\n");
104             close(connfd);
105             continue;
106         }
107         client_t *client_pointer = (client_t *)malloc(sizeof(client_t));//For store settings of clients (dynamic allocation)
108         client_pointer->addr = client_address;
109         client_pointer->connfd = connfd;
110         for (i = 0; i < C_NUM; i++) {//determining user id numbers whenever a new client has connected
111             if (client_array[i] == 0) {//if corresponding space is empty
112                 user_id = i+1; // then make user id as i +1 which is 1 more of client array number 0->1 or 1->2 (there is no 0th client)
113                 break;
114             }
115         }
116         srand(time(NULL));
117         client_pointer->user_id = user_id;// giving userid to the client struct for gathering all the info at one place
118         client_pointer->group_id = 0;// at first everyone at lobby group 0
119         client_pointer->msgtogrp = 0;// at first users send their message to everyone encrypted or not
120         client_pointer->rep = 100.0;
121         client_pointer->luck = rand()%10;
122         client_pointer->health = 100.0 + client_pointer->luck;
123         client_pointer->attack_power =10.0 + client_pointer->luck;
124         sprintf(client_pointer->name, "%d", client_pointer->user_id); //Default client name is user id (until nickname)
125         q_adr(client_pointer);// queing
126         pthread_create(&thread_id, NULL, &client_handler, (void*)client_pointer); //forking
127         sleep(1);// wait for connection do not close the server
128     }
129     return SUCCESS;
```

Figure 24: main Function (Part II)

Project Description with Examples:

Our project was designing a telnet-based client server which provides communication between users via server and adding security to this communication with Caesar-based encryption and decryption algorithm.

The first step was understanding how we can make connections between users and servers and between users themselves. We figure it out by doing some research on books and the net. Then our project appeared in our head; we made a To-Do list and started to do those things one by one. We realized we should design a set of functions and manager functions to use those functions and get input from users and process these inputs for every user separately. This manager function is called “Client handler” which can handle all things which we stated previously. A more detailed explanation is stated further in the project report.

Server requires port number to work. This feature allows us to by-pass the port problems of fixed ports in Linux and allows us to create multiple servers on different ports. When a user connects to the server, server gives him/her a unique user id and removes it when client disconnected from the server. Each connected

user has their information in a struct-based variable. It stores specific features such as connection file descriptors, socket addresses, user id, group id, and some other extra variables for using the server's different features, as you can see in **Figure 25**.

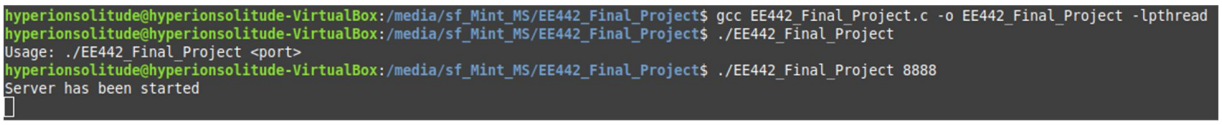

Figure 25: Server input requirements

When a telnet-based user connects to a server, they receive a message like in **Figure 26** which gives information to the client on what he/she can do in server.

Figure 26: Client terminal connection to the server

Meanwhile, in the server terminal a message appears which states that the connection between client and server has succeeded. Like below in **Figure 27.**

Figure 27: Server response when a client connected

Whenever a client connects to the server, server automatically sets her/him group id as 0 which is lobby and client can give any message to server and server forward these messages to other clients respect to group ids and the encryption process starts right here. The condition for getting messages as decrypted is being in the same group. If a user 1 in lobby wants to communicate with user 2 in group 1 the solution is basically joining group one; otherwise, user 1 cannot understand what user 2 is saying at that time because its group-based Caesar encryption. The messages which have been sent to the server are decrypted with respect to random keys which have been generated by rand function with NULL-seed and special to the group which changes at every sending process. This random key process is also valid for the encryption part. The Caesar cipher is weak by itself, but we have added random keys for every session which provides enough security for sending the same message multiple times. The groups can be created with /create command with providing number-based password. Let us assume user 1 has created a group and wrote a message. User 2 should not be able to understand that message. You can check this out from **Figure 28** below.

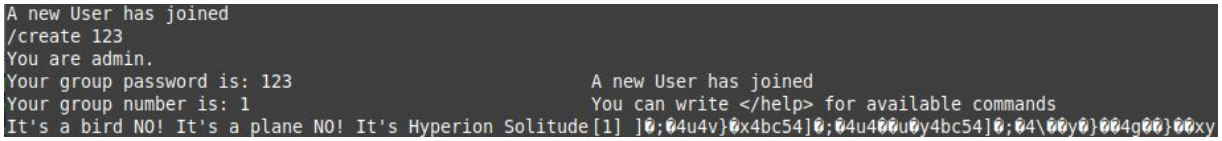

Figure 28: The proof of encryption works

As we expected user 2 cannot understand what user 1 is saying because user 1 is in group 1 while user 2 is still in group 0, which is the lobby (**Figure 28**).

Now user 2 joins to group 1 with password 124 which is wrong, then joins with correct password which is 123. Then user 1 sends the same message (**Figure 29**).

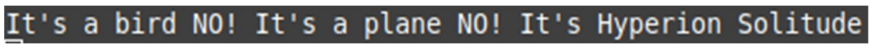

Figure 29: Sample message to test encryption

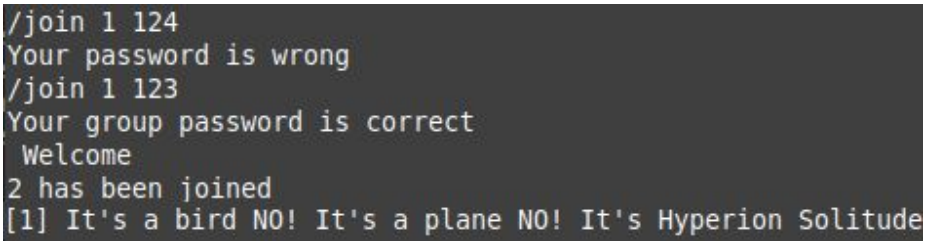

Figure 30: Same group can see messages as it is,

As we can see above in Figure 30 second user understood the message as it is. Now user 2 wants to leave the server. At this point the user 1 sends the same message **(Figure 31)** which should have a different ciphertext from the first case, and user 2 should not be able to understand it again **(Figure 32)**.

Figure 31: Test message for test

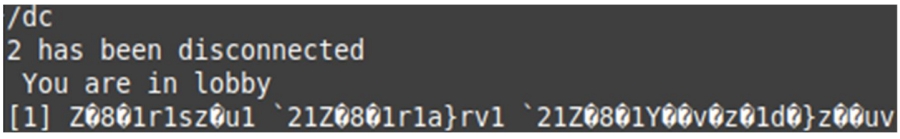

Figure 32: The proof of the encryption keys based on randomness

We can clearly see that user 2 could not understand the message now because of group differences. Now user 1 wants to know what he/she can do in this server. He writes "/help" and he saw available commands. Then he wanted to change his nickname to Hyperion-Solitude with /nick command and wanted to know his info. Like **Figure 33** below:

```
/help
/quit ===========> Exit from server
/info ===========> Ask your info to server
/topic ==========> <topic_msg> Create a topic
/nick ===========> <name> Create a nickname
/list ===========> Show connected clients
/msg ============> <user_id>(number) <msg> Send private message
/create =========> <password> Create a group with password(number only)
/join ===========> <group_id> <password> Join a group with password(number only)
/dc =============> disconnected from group
/grp_mem ========> Show connected clients in the group members
/grp_list =======> Show active groups
/grp_m ==========> Send one message to the group
/grp_a ==========> [default=everyone]Toggles the destination of the messages Group or everyone
/kick ===========> <user_id>(number) Kicks a member of group (Admin command)
/dice ===========> <face_number> Rolls a die
/rps ============> <choose>(r or p or s) Rock paper scissors
/duel ===========> <user_id> Invite to someone to Duel
/accept ============> <user_id> Accept Duel request from user
/hp ============>  <Requested health> Trading health points with rep points
/attack ============>  <Requested attack points> Trading attack points with rep points
/help ===========> Show help
/nick Hyperion-Solitude
 1 is now known as Hyperion-Solitude
/info
User id = 1
Name = Hyperion-Solitude
Group = 1
Message to group only = 0
Reputation points = 100.00
Attack power = 19.00
Health= 109.00
```

Figure 33: Available features of the server and usage of some features

Then he wanted to play rock paper scissors with his reputation points. Then wanted to roll a die again with his reputation points. (**Figure 34**).

```
/rps
Please give your choice ! Usage: /dice <r or p or s>.
/rps r
 [You lose!] You choose r. Server choose p. You lost 5 rep points
/dice
Please give face number of a die! Usage: /dice <face_number>.
/dice 20
 You rolled 3.
 You lose 7.00 rep points.
```

Figure 34: Server's rock paper scissors feature

Then user 2 (AgemennoN) rejoined group 1, but Hyperion Solitude does not want him in his group. So, he kicked AgemennoN (**Figure 35**).

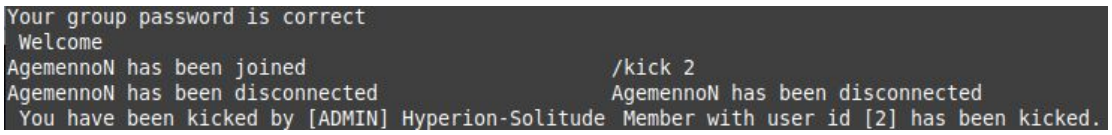

Figure 35: Joining group and kicking somebody

Then AgemennoN becomes upset and wants to disconnect Hyperion from his group. So, he invited Hyperion to duel with himself. Hyperion accepts his duel request, which is based on random turns of attacks. The duel has been started. (**Figure 36**)

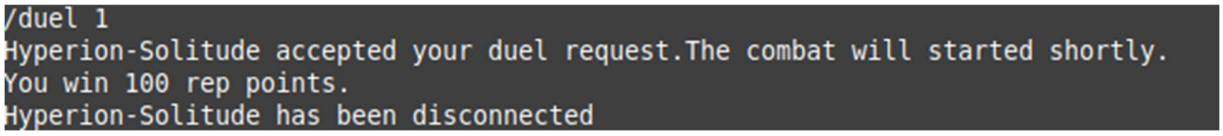

Figure 36: Duel request usage

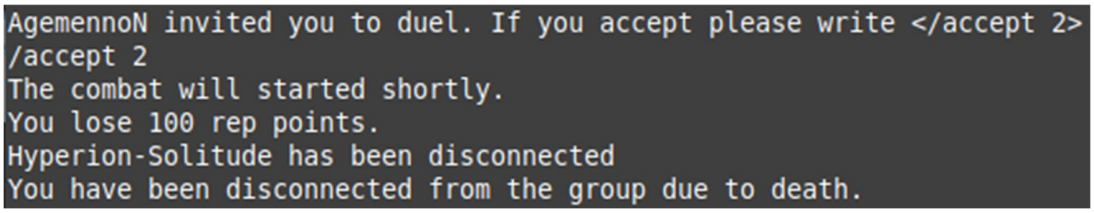

Figure 37: Accepting Duel request

AgemennoN has accomplished his goal which was to destroy Hyperion's group, and he looted all his rep-points. Then AgemennoN wants to know how many rep-points he has, then spend these rep points to gain Attack power and Health in case of further battles (**Figure 37** and **Figure 38**).

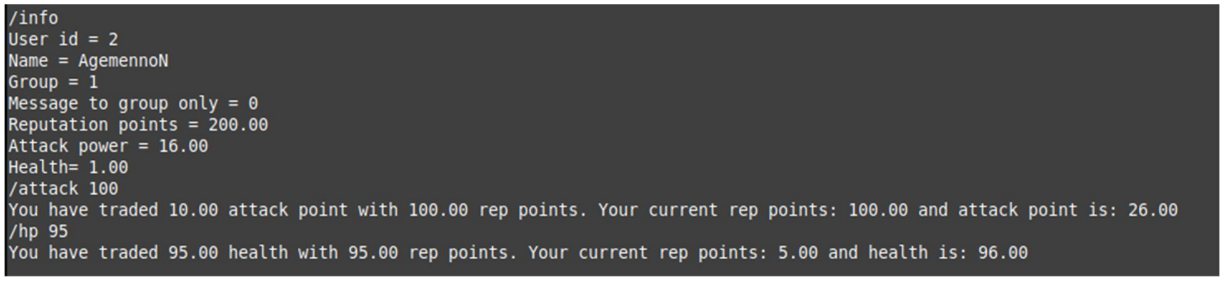

Figure 38: Displaying user info and usage of adding attack and health point

Then AgemennoN wants to create a group to secure messaging with his new friend Garry. First, he messaged Garry privately. Then he said group password and special command for toggling message range.

```
A new User has joined
You can write </help> for available commands
/nick Garry
 3 is now known as Garry
Hey AgemennoN whats up
[PM][From User AgemennoN] Garry wait, I will create a group
/msg 2 OK I am waiting
[PM][From User AgemennoN] Garry you can come to my group with /join 1 123 command
/msg 2 I am coming.
/join 1 123
Your group password is correct
 Welcome
Garry has been joined
[AgemennoN] wow Welcome man whats up?
Fine man what about u?
[AgemennoN] I'm fine
[AgemennoN] Garry Use /grp_a command to block our message to send Hyperion
/grp_a
Ok i did that
[AgemennoN] Niceeeee!
YEY!
```

Figure 39: Usage of private message and always send to group toggle

```
A new User has joined
 3 is now known as Garry
[Garry] Hey AgemennoN whats up
/msg 3 Garry wait, I will create a group
[PM][From User Garry] OK I am waiting
/create 123
You are admin.
Your group password is: 123
Your group number is: 1
/msg 3 Garry you can come to my group with /join 1 123 command
[PM][From User Garry] I am coming.
Garry has been joined
wow Welcome man whats up?
[Garry] Fine man what about u?
I'm fine
Garry Use /grp_a command to block our message to send Hyperion
[Garry] Ok i did that
/grp_a
Niceeeee!
[Garry] YEY!
```

Figure 40: Optimizing the group creation and chatting example

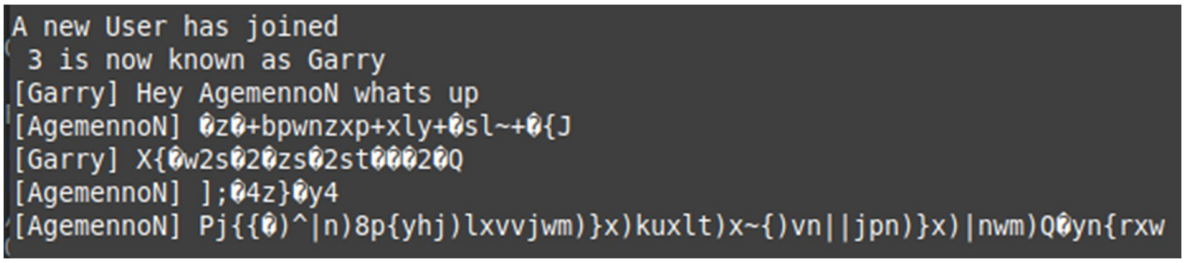

Figure 41: How group messages can be seen from outside of the group

As we can see from above in **Figure 39, Figure 40, Figure 41** Hyperion did not understand messages between AgemennoN and Garry, and after /grp_a command he did not receive even ciphertexts of the messages. However, messages are still encrypted, and the server decrypts it; then group members can see messages clearly while Hyperion cannot even see them.

Being admin of the group gives admin permission for kicking players at the group. Since the requirement of being admin of a group is creating a group or being 0th array element of a specific group disconnection of an admin triggers the system and system automatically sets admin as second user in array of that group. System also controls if there is anyone in the group, if not then the group has been deleted, and can be created again by anyone with any password. This disconnect function makes servers actions very smooth and it can be used in for different features such as duel, quit, dc, kick etc. If someone wants to quit from group instantly (without disconnection first) system checks if the user is in any group or not if the user is not in any group, then system instantly disconnects user from the server and closes his connection to optimize the resource allocation. If user in some group, then system first disconnects the user from the group which increases the optimality of resource usage (group's capacity) then disconnects from the server, then closes his connection. You can see that in **Figure 42**.

Figure 42: Usage of quit function by Hyperion

Since Hyperion has no group, he instantly disconnected.

Figure 43: Usage of quit function by AgemennoN

Since AgemennoN in group 1 and he was ADMIN before leaving, system disconnects him from the group first then give the ADMIN permissions to Garry. As we can see **Figure 44.**

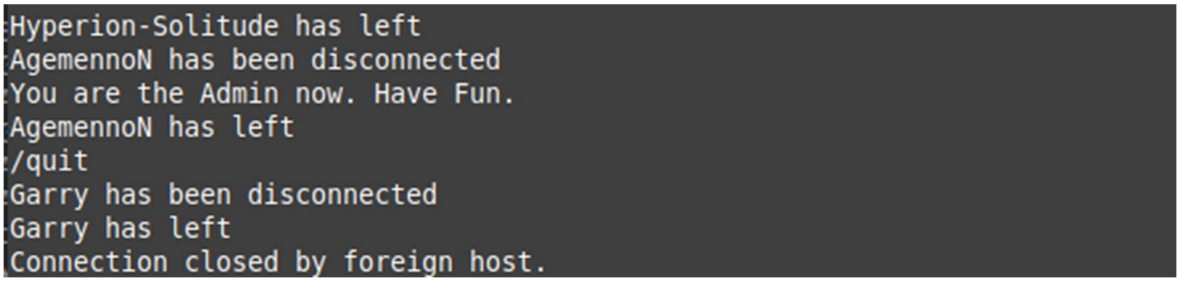

Figure 44: Usage of quit function by Garry

The changes are reported to the server automatically (connections disconnections, duel results etc.). Finally, all the clients have been disconnected and the server can be closed by closing the terminal or pressing "^C."

We have debugged a lot of faulty input errors. Every one of them was causing a "Core dumped "error. We have checked every one of them as we can. We hope we did not miss any bugs in the code. For example, sometimes, even if there is no error in the code, the things that should be printed at that time do not print, we have found that we must add '/n' newline at the end of every buffer, printf, snprintf, etc.

Thus, we have achieved what we wanted in the end. We have even added small games, a duel-based arena, and choices to spend rep points to get more health or attack points for making themselves stronger in the arena. We have increased the security level of the cipher by adding a random number generator to the key creation. We were thinking that we were familiar with the language C, but we realized how much we had forgotten C language as time passed. Meanwhile, socket creation, multithreading, and forking were unknown to us. So, we did so much research to understand these processes. We learned a lot of functions for making this server such as accept(), listen(), sprint(), pthread_mutex_unlock(), pthread_mutex_lock(). We have not used struct in a while, so it was good for us to remember. The hardest point of this project was understanding how a connection has been established from client to server; after understanding this problem, the project gets easier by knowing exactly how we should write required code to match its purpose. Every time we were adding some feature to the server, we have been flattered because of seeing every feature work. We have admitted that debugging was not so easy at first, but as we saw bugs, we were familiarized by them, and after some time we were able to figure out which part is 'bugged' nearly instant. They were generally insufficient conditions for given faulty inputs. We are planning on improving this server in future such that we can make Group War into getting closer RPG-like games in the future.

In conclusion, this project has been fun and instructive to implement small servers and provide secure communication between users and servers.