

```
1 #include "Server.h"
2 #include <time.h>
3
4 /*
5  * VARIABILI GLOBALI:
6  * LogFile serverLog, una struttura dati contenente il file
7  * di Log della specifica sessione server e il mutex che ne
8  * regola l'accesso;
9  * Game *g, un puntatore alla partita in corso, inizialmente
10 * non allocata (definizione di Game in costanti.h);
11 * int gameId=-1, l'id della partita che verrà assegnato al
12 * momento dell'allocazione della stessa;
13 * LoggedUser loggati, una struttura dati contenente i
14 * giocatori correntemente loggati e il mutex che ne regola l'
15 * accesso;
16 * pthread_t clientsId[MAX_PLAYER_N], un array contenente il
17 * tid dei thread giocanti utilizzato per gestire il fine
18 * partita;
19 * int gameEnded = 0, un valore utilizzato per indicare al
20 * Server quando si trova all'interno di una partita;
21 */
22
23
24 LogFile serverLog;
25 Game *g;
26 int gameId=-1;
27 loggedUser loggati;
28 pthread_t clientsId[MAX_PLAYER_N];
29 int gameEnded = 0;
30 struct sockaddr_in client_addr;
31
32 int main(int argc, char* argv[]){
33
34     int i = 0;
35     for(i = 0; i<MAX_PLAYER_N; i++){
36         clientsId[i] = 0;
37     }
38     i = 0;
39
40     clear();
41     system("/sbin/ifconfig");
42     //seed per la generazione di numeri casuali;
43     srand(time(NULL));
44     //signal handler set;
45     signal(SIGINT, handleSignal);
46     signal(SIGHUP, handleSignal);
47     signal(SIGQUIT, handleSignal);
```

```
38     signal(SIGTERM, handleSignal);
39     signal(SIGALRM, handleSignal);
40     signal(SIGSEGV, handleSignal);
41
42     if (argc!=2) {
43         printf("Passa il numero della porta.\n");
44         return -1;
45     }
46
47     int *thread_sd, sock, sockfd, porta =atoi(argv[1]), pid;
48     pthread_t tid;
49
50     socklen_t client_len;
51
52     pthread_mutex_lock(&serverLog.sem);
53     LogServerStart(&serverLog.fd);
54     pthread_mutex_unlock(&serverLog.sem);
55     initializeLoggedUser(&loggati);
56     if((sock = creaSocket(porta))<0){
57         if(sock == ERR_SOCKET_CREATION){
58             printf("%s", SOCKET_CREATION_ERR_MESSAGE);
59             pthread_mutex_lock(&serverLog.sem);
60             LogErrorMessage(&serverLog.fd,
61                             SOCKET_CREATION_ERR_MESSAGE);
62             pthread_mutex_unlock(&serverLog.sem);
63         }else if(sock == ERR_SOCKET_BINDING){
64             printf("%s", SOCKET_BINDING_ERR_MESSAGE);
65             pthread_mutex_lock(&serverLog.sem);
66             LogErrorMessage(&serverLog.fd,
67                             SOCKET_BINDING_ERR_MESSAGE);
68             pthread_mutex_unlock(&serverLog.sem);
69         }else{
70             printf("%s", NOT_SURE_ERR_MESSAGE);
71             pthread_mutex_lock(&serverLog.sem);
72             LogErrorMessage(&serverLog.fd, NOT_SURE_ERR_MESSAGE
73 );
74             pthread_mutex_unlock(&serverLog.sem);
75         }
76     }else{
77         if((listen(sock,MAXIMUM_SOCKET_BACKLOG))<0){
78             printf("%s", SOCKET_LISTEN_ERR_MESSAGE);
79             pthread_mutex_lock(&serverLog.sem);
80             LogErrorMessage(&serverLog.fd,
81                             SOCKET_LISTEN_ERR_MESSAGE);
82             pthread_mutex_unlock(&serverLog.sem);
83         }
84     }
85 }
```

```

80         while(1){
81             while(!gameEnded){
82                 client_len = sizeof(client_addr);
83                 if((sockfd = accept(sock,(struct sockaddr *) &
84                     client_addr, &client_len))<0){
85                     printf("%s", ACCEPT_SOCKET_ERR_MESSAGE);
86                     pthread_mutex_lock(&serverLog.sem);
87                     LogErrorMessage(&serverLog.fd,
88                         ACCEPT_SOCKET_ERR_MESSAGE);
89                     pthread_mutex_unlock(&serverLog.sem);
90                     exit(-1);
91                 }
92                 int* thread_sd;
93                 thread_sd = (int*) malloc(sizeof(int));
94                 thread_sd = &sockfd;
95                 if((pthread_create(&tid, NULL, run, (void *)
96                     thread_sd))<0){
97                     printf(" %s", THREAD_CREATION_ERR_MESSAGE);
98                     pthread_mutex_lock(&serverLog.sem);
99                     LogErrorMessage(&serverLog.fd,
100                         THREAD_CREATION_ERR_MESSAGE);
101                     pthread_mutex_unlock(&serverLog.sem);
102                     }
103                     clientsId[i]=tid;
104                     i++;
105                 }
106             }
107             close(sock);
108             close(sockfd);
109             pthread_mutex_lock(&serverLog.sem);
110             LogServerClose(&serverLog.fd);
111             pthread_mutex_unlock(&serverLog.sem);
112             return 1;
113         }
114
115 /*
116     La funzione che esegue ogni nuovo thread;}
117
118 */
119 void * run(void *arg){
120     char msg[1000];

```

```

121  int n_b_r;
122  int sockfd=*((int *)arg);
123  char user[50];
124  int done=0, result;
125
126  n_b_r=sendMsg(sockfd,WELCOME_MESSAGE,msg);
127  if(n_b_r<0){
128      pthread_exit((int*)-1);
129  }
130  while(!done){
131      if(n_b_r==1){
132          switch (msg[0]) {
133              case 'l': case 'L':
134                  result = logInUserMenu(sockfd,user,&serverLog,&
loggati,inet_ntoa(client_addr.sin_addr));
135                  if(result == ERR_NO_USER_FILE || result ==
ERR_INPUT_OUTPUT){
136                      clear();
137                      write(sockfd,"-1", strlen("-1"));
138                      n_b_r = read(sockfd, msg, 50);
139                      if(strcmp(msg, USER_LOG_OUT)){
140                          pthread_mutex_lock(&serverLog.sem);
141                          LogUnknownClientDisconnection(&serverLog.fd,
inet_ntoa(client_addr.sin_addr));
142                          pthread_mutex_unlock(&serverLog.sem);
143                      }
144                      pthread_exit((int*)1);
145                      break;
146                  }
147                  done = 1;
148                  break;
149              case 'r': case 'R':
150                  result = signInUserMenu(sockfd,user,&serverLog,&
loggati,inet_ntoa(client_addr.sin_addr));
151                  if(result == ERR_NO_USER_FILE || result ==
ERR_INPUT_OUTPUT){
152                      clear();
153                      write(sockfd,"-1", strlen("-1"));
154                      n_b_r = read(sockfd, msg, 50);
155                      if(strcmp(msg, USER_LOG_OUT)){
156                          pthread_mutex_lock(&serverLog.sem);
157                          LogUnknownClientDisconnection(&serverLog.fd,
inet_ntoa(client_addr.sin_addr));
158                          pthread_mutex_unlock(&serverLog.sem);
159                      }
160                      pthread_exit((int*)1);

```

```

161         break;
162     }
163     done = 1;
164     break;
165 case 'e': case 'E':
166     clear();
167     write(sockfd, "-1", strlen("-1"));
168     n_b_r = read(sockfd, msg, 50);
169     if(strcmp(msg, USER_LOG_OUT)){
170         pthread_mutex_lock(&serverLog.sem);
171         LogUnknownClientDisconnection(&serverLog.fd,
172             inet_ntoa(client_addr.sin_addr));
173         pthread_mutex_unlock(&serverLog.sem);
174     }
175     pthread_exit((int*)1);
176     break;
177 default:
178     n_b_r=sendMsg(sockfd,WELCOME_MESSAGE,msg);
179     if(n_b_r<0){
180         pthread_exit((int*)-1);
181     }
182     break;
183 }else{
184     n_b_r=sendMsg(sockfd,WELCOME_MESSAGE,msg);
185     if(n_b_r<0){
186         pthread_exit((int*)-1);
187     }
188 }
189 }
190 if(g==NULL){
191     initializeNewGame(&g,sockfd,user,&serverLog,&loggati);
192 }else{
193     spawnNewPlayer(&g,user, sockfd, &serverLog,&loggati);
194 }
195 }
196
197 void handleSignal(int Sig){
198     switch (Sig) {
199     case SIGINT:
200         pthread_mutex_lock(&serverLog.sem);
201         LogServerClose(&serverLog.fd);
202         pthread_mutex_unlock(&serverLog.sem);
203         exit(1);
204     break;
205     case SIGALRM:

```

```
206     alarm(0);
207     pthread_mutex_lock(&g->sem);
208     g->timeOver = 1;
209     gameEnded = 1;
210     pthread_mutex_unlock(&g->sem);
211     break;
212 }
213 }
214
215 void * endGameManagement(void * arg){
216
217     for(int i = 0; i<0; i++){
218         if(clientsId[i]!=0){
219             pthread_join(clientsId[i], NULL);
220         }
221     }
222     if(g!= NULL){
223         pthread_mutex_lock(&g->sem);
224         deleteGrid(g->grid);
225         Game * tmp = g;
226         pthread_mutex_lock(&serverLog.sem);
227         LogEndGame(&serverLog.fd,g->gameId);
228         pthread_mutex_unlock(&serverLog.sem);
229         pthread_mutex_unlock(&g->sem);
230         g = NULL;
231         free(tmp);
232     }
233     gameEnded = 0;
234     pthread_exit((int *) 1);
235 }
236 }
```

```

1 #include <netinet/in.h>
2 #include <sys/socket.h>
3 #include <arpa/inet.h>
4 #include <sys/un.h>
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <unistd.h>
8 #include <sys/types.h>
9 #include <sys/stat.h>
10 #include <fcntl.h>
11 #include <pthread.h>
12 #include <string.h>
13 #include <errno.h>
14 #include <signal.h>
15 #include <sys/mman.h>
16 #include "../costanti.h"
17 #include "../messaggi.h"
18 #include "../codici_errori.h"
19 #include "Librerie/Log.c"
20 #include "Librerie/Gameplay.c"
21 #include "Librerie/Users.c"
22 #include "Librerie/Communication.c"
23
24
25
26 /*
27   La funzione handleSignal viene utilizzata per la gestione
   dei segnali. In particolare il Server gestisce in maniera
   particolare
28   i segnali di SIGINT e SIGALRM.
29   Il segnale di SIGINT viene utilizzato solo per loggare l'
   evento di chiusura sessione server, mentre il segnale di
   SIGALRM viene usato
30   per gestire il fine partita. Si imposta SIGALRM per
   essere lanciato quando si è raggiunto il tempo massimo di
   gioco oppure il segnale viene lanciato,
31   in vari punti del codice, quando si raggiunge un'altra
   condizione di terminazione. Raccogliere SIGALRM si traduce
   poi nel cambio del bit timeOver della struttura
32   Game da 0 ad 1 e, di conseguenza, nell'invio di un
   segnale di terminazione partita ai thread giocanti.
33   Parametri:
34     Sig, un intero che corrisponde al segnale da gestire;
35   */
36 void handleSignal(int Sig);
37

```

```
38 /*
39   La funzione Lanciata da ogni nuovo thread che si occupa
   di gestire la comunicazione con il Client. Ad ogni Client
   viene associato un thread.
40   Parametri:
41     arg[0], un intero che rappresenta il file descriptor
       della socket TCP adibita alla comunicazione;
42   */
43 void * run(void *arg);
44
45 /*
46   La funzione che si occupa di gestire il fine partita,
   viene Lanciata da handleSignal ogniqualvolta viene raccolto
   il segnale di SIGALRM. Aspetta la terminazione
47   dei vari thread giocanti e dealloca la partita. Al
   termine di endGameManagement il valore intero gameEnded
   viene reimpostato ad 1.
48   */
49 void * endGameManagement(void *arg);
50
```

```

1 #include "Log.h"
2
3 void substituteChar(char st[],char a, char b){
4     int i=0;
5     while(st[i]!='\0'){
6         if(st[i]==a)
7             st[i]=b;
8         i++;
9     }
10    return;
11 }
12
13 void oraEsatta(char s[]){
14     char buffer[26];
15     time_t ora;
16     time(&ora);
17     strcpy(s,ctime_r(&ora, buffer));
18     substituteChar(s,'\n','\0');
19 }
20
21 void LogServerStart(int *fdLog){
22
23     char ora[80];
24     char fileName[45] = "File/Log/LOG_";
25     int n_b_w;
26
27     oraEsatta(ora);
28     strcat(fileName,ora);
29     substituteChar(fileName,' ', '_');
30     substituteChar(fileName,':', '_');
31     strcat(fileName,".txt");
32
33     if((*fdLog=open(fileName,O_CREAT | O_APPEND | O_WRONLY,
S_IRWXU))<0){
34         /*Gestire cosa succede in caso di errore*/
35     } else {
36         if((n_b_w = write(*fdLog,ora,strlen(ora))< strlen(ora
))) {
37             /*Gesire mancata scrittura su LOG*/
38         } else {
39             if((n_b_w = write(*fdLog,LOG_START_SERVER,sizeof(
LOG_START_SERVER)-1)) < sizeof(LOG_START_SERVER)-1){
40                 /*Gesire mancata scrittura su LOG*/
41             }
42         }
43     }

```

```
44     return ;
45 }
46
47 void LogServerClose(int*fdLog){
48
49     char ora[80];
50     int n_b_w;
51
52     oraEsatta(ora);
53
54     if((n_b_w = write(*fdLog,ora,strlen(ora))< strlen(ora))){
55         /*Gesire mancata scrittura su LOG*/
56     } else {
57         if((n_b_w = write(*fdLog,LOG_CLOSE_SERVER,sizeof(
58             LOG_CLOSE_SERVER)-1)) < sizeof(LOG_CLOSE_SERVER)-1){
59             /*Gesire mancata scrittura su LOG*/
60         }
61     }
62 }
63
64 void LogNewGame(int* fdLog, int gameId){
65
66     char ora[80];
67     int n_b_w;
68     char buf[100];
69
70     oraEsatta(ora);
71     sprintf(buf, LOG_NEW_GAME, gameId);
72
73     if((n_b_w = write(*fdLog,ora,strlen(ora))< strlen(ora))){
74         /*Gesire mancata scrittura su LOG*/
75     } else {
76         if((n_b_w = write(*fdLog,buf,strlen(buf))) < strlen(buf
77 ))){
78             /*Già sai*/
79         }
80     }
81 }
82
83 void LogEndGame(int* fdLog, int gameId){
84     char ora[80];
85     int n_b_w;
86     char buf[100];
87 }
```

```

88     oraEsatta(ora);
89     sprintf(buf, LOG_END_GAME, gameId);
90
91     if((n_b_w = write(*fdLog,ora,strlen(ora)) < strlen(ora
92 ))){
93         /*Gesire mancata scrittura su LOG*/
94     } else {
95         if((n_b_w = write(*fdLog,buf,strlen(buf))) < strlen(
96         buf)){
97             /*Già sai*/
98         }
99     }
100
101 void LogPlayerJoin(int* fdLog, int gameId, char* player){
102     char ora[80];
103     int n_b_w;
104     char buf[100];
105
106     oraEsatta(ora);
107     sprintf(buf, LOG_NEW_PLAYER_JOIN, player,gameId);
108
109     if( (n_b_w = write(*fdLog,ora,strlen(ora) ) < strlen(ora
110 )) ) {
111         /*Gesire mancata scrittura su LOG*/
112     }else{
113         if((n_b_w = write(*fdLog,buf,strlen(buf))) < strlen(
114         buf)){
115             /*Già sai*/
116         }
117     }
118
119 void LogPlayerMoves(int* fdLog, int gameId, char* player,
120     char* src, char *dst ){
121     char ora[80];
122     int n_b_w;
123     char buf[100];
124
125     oraEsatta(ora);
126     sprintf(buf, LOG_PLAYER_MOVES, gameId, player, src, dst
127 );
128
129     if( (n_b_w = write(*fdLog,ora,strlen(ora)) < strlen(ora

```

```

127 )) ) {
128     //Gesire mancata scrittura su LOG
129 } else {
130     if( (n_b_w= write(*fdLog, buf ,strlen(buf)) ) < strlen
131 (buf) ){
132         //Già sai
133     }
134     return;
135 }
136
137 void LogPlayerTakePackage(int* fdLog,int gameId,char
138 player[], int pacco, char loc[]){
139     char ora[80];
140     int n_b_w;
141     char buf[100];
142
143     oraEsatta(ora);
144     sprintf(buf, LOG_PLAYER_TAKE, gameId,player,pacco,loc);
145
146     if( (n_b_w = write(*fdLog,ora,strlen(ora)) < strlen(ora
147 )) ) {
148         //Gesire mancata scrittura su LOG
149     }else{
150         if((n_b_w = write(*fdLog, buf ,strlen(buf))) < strlen(
151 buf)){
152             /*Già sai*/
153         }
154     }
155
156 void LogPlayerLeavePackage(int *fdLog, int gameId, char
157 player[], int pacco, char loc[]){
158     char ora[80];
159     int n_b_w;
160     char buf[100];
161
162     oraEsatta(ora);
163     sprintf(buf, LOG_PLAYER_DELIVER, gameId,player,pacco,loc
164 );
165     if( (n_b_w = write(*fdLog,ora,strlen(ora)) < strlen(ora
166 )) ) {
167         //Gesire mancata scrittura su LOG

```

```

166    }else{
167        if((n_b_w = write(*fdLog, buf ,strlen(buf))) < strlen(
168            buf)) {
169            /*Già sai*/
170        }
171    return;
172 }
173
174 void LogPlayerMakeAPoint(int *fdLog, int gameId, char
175 player[]){
176     char ora[80];
177     int n_b_w;
178     char buf[100];
179
180     oraEsatta(ora);
181     sprintf(buf, LOG_PLAYER_MAKE_POINT, gameId,player);
182     if( (n_b_w = write(*fdLog,ora,strlen(ora)) < strlen(ora
183 )) ) {
184         //Gesire mancata scrittura su LOG
185     }else{
186         if((n_b_w = write(*fdLog, buf ,strlen(buf))) < strlen(
187             buf)) {
188             /*Già sai*/
189         }
190     return;
191 }
192
193 void LogPlayerWin(int* fdLog, int gameId, char * player){
194     char ora[80];
195     int n_b_w;
196     char buf[100];
197
198     oraEsatta(ora);
199     sprintf(buf, LOG_PLAYER_WINS, player,gameId);
200
201     if((n_b_w = write(*fdLog,ora,strlen(ora)) < strlen(ora
202 ))) {
203         /*Gesire mancata scrittura su LOG*/
204     } else {
205         if((n_b_w= write(*fdLog,buf,strlen(buf))) < strlen(buf
206 )) {
207             /*Già sai*/

```

```
206     }
207 }
208
209 return;
210 }
211
212 void LogNewUser(int* fdLog, char* player) {
213     char ora[80];
214     int n_b_w;
215     char buf[64];
216
217     oraEsatta(ora);
218     sprintf(buf,LOG_NEW_USER,player);
219
220     if((n_b_w = write(*fdLog,ora,strlen(ora))) < strlen(ora
221 )) {
222         /*Gesire mancata scrittura su LOG*/
223     } else {
224         if((n_b_w= write(*fdLog,buf ,strlen(buf))) < strlen(
225             buf)){
226             /*Già sai*/
227         }
228     }
229
230     void LogUserSignIn(int* fdLog, char* player){
231     char ora[80];
232     int n_b_w;
233     char buf[64];
234
235     oraEsatta(ora);
236     sprintf(buf,LOG_SIGN_IN,player);
237
238     if((n_b_w = write(*fdLog,ora,strlen(ora))) < strlen(ora
239 )) {
240         /*Gesire mancata scrittura su LOG*/
241     } else {
242         if((n_b_w= write(*fdLog,buf ,strlen(buf))) < strlen(
243             buf)){
244             /*Già sai*/
245         }
246     }
247 }
```

```

248 void LogUserSignOut(int* fdLog, char* player){
249     char ora[80];
250     int n_b_w;
251     char buf[64];
252
253     oraEsatta(ora);
254     sprintf(buf,LOG_SIGN_OUT,player);
255
256     if((n_b_w = write(*fdLog,ora,strlen(ora)) < strlen(ora
    ))){
257         /*Gesire mancata scrittura su LOG*/
258     } else {
259         if((n_b_w= write(*fdLog,buf ,strlen(buf))) < strlen(
    buf)){
260             /*Già sai*/
261         }
262     }
263     return;
264 }
265
266 void LogErrorMessage(int* fdLog, char* err){
267     char ora[80];
268     int n_b_w;
269     char buf[1000];
270
271     oraEsatta(ora);
272     sprintf(buf,LOG_ERROR,err);
273     if((n_b_w = write(*fdLog,ora,strlen(ora)) < strlen(ora
    ))){
274         /*Gesire mancata scrittura su LOG*/
275     }else{
276         if((n_b_w= write(*fdLog, buf ,strlen(buf))) < strlen(
    buf)){
277             /*che fare che fare*/
278         }
279     }
280     return;
281 }
282
283 void LogUnknownClientDisconnection (int*fdLog,char addr
    []){
284
285     char ora[80],buf[1000];
286     int n_b_w;
287
288     oraEsatta(ora);

```

```
289     sprintf(buf,LOG_UNKNOWN_CLIENT_DISCONNECTION,addr);
290     if((n_b_w = write(*fdLog,ora,strlen(ora)))< strlen(ora
291 )){           /*Gesire mancata scrittura su LOG*/
292     } else {
293     if((n_b_w = write(*fdLog,buf,strlen(buf))) < strlen(
294 buf)){           /*Gesire mancata scrittura su LOG*/
295     }
296   }
297   return ;
298 }
299
```

```
1 #include "../codici_errori.h"
2 #include "../messaggi.h"
3 #include "../costanti.h"
4
5 /*
6   La libreria si occupa di gestire il logging degli eventi
7   . Come tale ogni funzione in essa contenuta
8   riceve come parametro il fd del file di Log relativo alla
9   specifica sessione Server e le informazioni
10  necessarie al logging dell'evento specifico. I file di
11  Log si trovano nella directory File/Log/<nome_file_log>
12 */
13
14 /*
15   La funzione LogServerStart logga l'evento dell'inizio
16  della sessione Server, si occupa anche di creare il file
17  di Log stesso.
18 */
19
20 void LogServerStart(int* fdLog);
21
22 /*
23   La funzione LogServerClose logga l'evento di fine
24  sessione Server.
25 */
26
27 void LogServerClose(int* fdLog);
28
29 /*
30   La funzione LogNewGame logga l'evento di inizio nuova
31  partita.
32  Parametri:
33    int gameId, l'id della partita specifica.
34 */
35
36 void LogNewGame(int* fdLog, int gameId);
37
38 /*
39   La funzione LogEndGame logga l'evento di fine partita.
40  Parametri:
41    int gameId, l'id della partita.
42 */
43
44 void LogEndGame(int* fdLog, int gameId);
45
46 /*
47   La funzione LogPlayerJoin logga l'evento giocatore si
48  unisce a partita.
49  Parametri:
50    int gameId, l'id della partita;
```

```
40     char * player, il nickname del giocatore;
41 */
42 void LogPlayerJoin(int* fdLog, int gameId, char* player);
43
44 /*
45 La funzione LogPlayerMoves Logga L'evento giocatore va da
46 casella (x,y) a casella (x',y').
47 Parametri:
48     int gameId, l'id della partita;
49     char * player, il nickname del giocatore;
50     char * src, rappresentazione testuale della casella di
51     origine;
52     char * dst, rappresentazione testuale della casella di
53     destinazione.
54 */
55 void LogPlayerMoves(int* fdLog, int gameId, char* player,
56                     char* src , char* dst);
57
58 /*
59 La funzione LogPlayerWin Logga L'evento giocatore x vince
60 partita.
61 Parametri:
62     int gameId, l'id della partita;
63     char * player, il nickname del giocatore;
64 */
65 void LogPlayerWin(int* fdLog, int gameId, char* player);
66
67 /*
68 La funzione LogNewUser Logga L'evento giocatore x si è
69 registrato sul server:
70 Parametri:
71     int gameId, l'id della partita;
72     char * player, il nickname del giocatore;
73 */
74 void LogNewUser(int* fdLog, char* player);
75
76 /*
77 La funzione LogPlayerTakePackage Logga L'evento giocatore
78 raccoglie pacco.
79 Parametri:
80     int gameId, l'id della partita;
81     char * player, il nickname del giocatore;
82     int pacco, l'id dell pacco specifico;
83     char * loc, la casella da cui si è raccolto il pacco.
84 */
85 void LogPlayerTakePackage(int* fdLog,int gameId, char*
```

```

78 player, int pacco, char* loc);
79
80 /*
81   La funzione LogPlayerLeavePackage Logga L'evento
82   giocatore lascia pacco.
83   Parametri:
84     int gameId, l'id della partita;
85     char * player, il nickname del giocatore;
86     int pacco, l'id dell pacco specifico;
87     char * Loc, la casella su cui si è lasciato il pacco.
88 */
89 void LogPlayerLeavePackage(int *fdLog, int gameId, char*
90   player, int pacco, char* loc);
91
92 /*
93   La funzione LogPlayerMakeAPoint Logga L'evento giocatore
94   fa un punto:
95   Parametri:
96     int gameId, l'id della partita;
97     char * player, il nickname del giocatore.
98 */
99 void LogPlayerMakeAPoint(int *fdLog, int gameId, char*
100   player);
101
102 /*
103   La funzione LogUserSignIn Logga L'evento utente x si è
104   connesso al Server:
105   Parametri:
106     int gameId, l'id della partita;
107     char * player, il nickname del giocatore.
108 */
109 void LogUserSignIn(int* fdLog, char* player);
110
111 /*
112   La funzione LogUserSignIn Logga L'evento utente x si è
113   disconnesso Server:
114   Parametri:
115     int gameId, l'id della partita;
116     char * player, il nickname del giocatore.
117 */
118 void LogUserSignOut(int* fdLog, char* player);
119
120 /*
121   La funzione LogErrorMessage Logga un eventuale messaggio
122   d'errore:
123   Parametri:

```

```
117     int gameId, l'id della partita;
118     char * err, il messaggio d'errore da Loggare.
119 */
120 void LogErrorMessage(int* fdLog, char* err);
121
122 /*
123 La funzione LogUnknownClientDisconnection logga l'evento
124 un Client non identificato da un nickname
125 si è disconnesso dal Server.
126 Parametri:
127     char addr*, l'indirizzo IP del client che si è appena
128     disconnesso.
129 */
128 void LogUnknownClientDisconnection (int*fdLog, char* addr
    );
129
```

```

1 #include "Users.h"
2
3 int signInUserMenu(int sockfd, char usrn[], LogFile *log,
4 loggedUser *loggati,char addrClient[]){
5     int n_b_r;
6     char pssw[50],msg[100];
7     int err;
8     n_b_r=sendMsg(sockfd,INSERT_USERNAME_SIM,usrn);
9     if(n_b_r<0){
10         pthread_mutex_lock(&log->sem);
11         LogUnknownClientDisconnection(&log->fd,addrClient);
12         pthread_mutex_unlock(&log->sem);
13         pthread_exit((int*)1);
14     }
15     n_b_r=sendMsg(sockfd,INSERT_PASSWORD_SIM,pssw);
16     if(n_b_r<0){
17         pthread_mutex_lock(&log->sem);
18         LogUnknownClientDisconnection(&log->fd,addrClient);
19         pthread_mutex_unlock(&log->sem);
20         pthread_exit((int*)1);
21     }
22     while((err=registerUser(usrn, pssw, loggati)) != 0){
23         //L'utente non è stato trovato tra quelli registrati
24         switch(err){
25             case ERR_NO_USER_FILE:
26                 pthread_mutex_lock(&log->sem);
27                 LogErrorMessage(&log->fd,
28 USER_FILE_OPEN_ERR_MESSAGE);
29                 pthread_mutex_unlock(&log->sem);
30                 return ERR_NO_USER_FILE;
31                 break;
32             case ERR_INPUT_OUTPUT:
33                 pthread_mutex_lock(&log->sem);
34                 LogErrorMessage(&log->fd, I_O_ERR_MESSAGE);
35                 pthread_mutex_unlock(&log->sem);
36                 return ERR_INPUT_OUTPUT;
37                 break;
38             case ERR_INVALID_USERNAME:
39                 n_b_r=sendMsg(sockfd,USER_ALREADY_PRESENT_SIM,
40 usrn);
41                 if(n_b_r<0){
42                     pthread_mutex_lock(&log->sem);
43                     LogUnknownClientDisconnection(&log->fd,
44 addrClient);
45                     pthread_mutex_unlock(&log->sem);
46                     pthread_exit((int*)1);

```

```

43         }
44         n_b_r=sendMsg(sockfd,INSERT_PASSWORD_SIM,pssw);
45         if(n_b_r<0){
46             pthread_mutex_lock(&log->sem);
47             LogUnknownClientDisconnection(&log->fd,
48                 addrClient);
49             pthread_mutex_unlock(&log->sem);
50             pthread_exit((int*)1);
51         }
52     default:
53         break;
54     }
55 }
56 pthread_mutex_lock(&log->sem);
57 LogNewUser(&log->fd, usrn);
58 pthread_mutex_unlock(&log->sem);
59 n_b_r=sendMsgNoReply(sockfd,SUCCESS_MESSAGE_SIM);
60 return 1;
61 }
62
63 int checkUsername(char* username, loggedUser *loggati){
64     int fdUserFile, i=0, res,r=1;
65     char c, str[MAX_SIZE_USERNAME];
66     if((fdUserFile=open(USER_FILE,O_RDONLY))<0){
67         return ERR_NO_USER_FILE;
68     }else{
69         while(r>0){
70             while(((r=read(fdUserFile,&c,1))>0)&&c != '%')
71                 str[i++]=c;
72             str[i++]='\0';
73             if(strcmp(str,username)==0){
74                 if(isLogged(username,loggati)){
75                     close(fdUserFile);
76                     return ERR_USER_ALREADY_LOGGED;
77                 }
78                 res=lseek(fdUserFile,0,SEEK_CUR);
79                 close(fdUserFile);
80                 return res;
81             }
82             while(((r=read(fdUserFile,&c,1))>0)&&c != '\n');
83             i=0;
84         }
85         close(fdUserFile);
86     }
87     return ERR_USERNAME_NOT_FOUND;

```

```

88 }
89
90 int registerUser(char* newuser, char* newpassw, loggedUser
91     *loggati){
92     int res, err, fdUserFile, n_b_w, lenght_user, lenght_passw;
93     lenght_user = strlen(newuser);
94     lenght_passw = strlen(newpassw);
95
96     if( lenght_user > MAX_SIZE_USERNAME || lenght_passw >
97         MAX_SIZE_PASSW ){
98         return ERR_INVALID_USERNAME;
99     }else{
100         if((res=checkUsername(newuser, loggati)) ==
101             ERR_USERNAME_NOT_FOUND){// Se checkUsername ritorna un
102             valore minore di 0 vuol dire che quel username non è
103             presente nel file ed è quindi disponibile
104         if((fdUserFile = open(USER_FILE,O_WRONLY|O_APPEND,
105             S_IRWXU))<0){
106             return ERR_NO_USER_FILE;
107         }else{
108             if((n_b_w = write(fdUserFile,newuser,lenght_user
109             ))<lenght_user){
110                 close(fdUserFile);
111                 return ERR_INPUT_OUTPUT;
112             }else{
113                 write(fdUserFile,"%",1);
114                 if((n_b_w = write(fdUserFile,newpassw,
115                     lenght_passw))<lenght_passw){
116                     close(fdUserFile);
117                     return ERR_INPUT_OUTPUT;
118                 }else{
119                     if(err=insertLoggedUser(newuser,loggati)<0){
120                         return ERR_INSERT_LOGGED_USER;
121                     }else{
122                         close(fdUserFile);
123                         return 0;
124                     }
125                 }
126             }else{
127                 return res;
128             }
129         }
130     }
131 }
132 }
```

```

126 }
127
128 int logInUser(char* user, char* passw, loggedUser *loggati
129 )
130     int fdUserFile, pos, err,n_b_r,i=0;
131     char c,str[MAX_SIZE_PASSW];
132     str[0]='\0';
133     if((pos=checkUsername(user,loggati))<0){
134         return pos;
135     }else{
136         if((fdUserFile=open(USER_FILE,O_RDONLY))<0){
137             return ERR_NO_USER_FILE;
138         }else{
139             if(lseek(fdUserFile,pos,SEEK_SET)!=pos){
140                 return ERR_INPUT_OUTPUT;
141             }else{
142                 while((n_b_r=read(fdUserFile,&c,1))>0 && (c!=13
143 && c!='\n'))
144                     str[i++]=c;
145                     str[i]='\0';
146                     if(strcmp(passw,str)==0){
147                         if(err=insertLoggedUser(user,loggati)<0){
148                             return ERR_INSERT_LOGGED_USER;
149                         }else{
150                             return 0;
151                         }
152                     }
153                 }
154             }
155         }
156     }
157     return -1;
158 }
159
160 int logInUserMenu(int sockfd, char usrn[],LogFile *log,
161 loggedUser *loggati,char addrClient[]){
162     int n_b_r;
163     char pssw[50],msg[100];
164     int err;
165
166     n_b_r=sendMsg(sockfd,INSERT_USERNAME_LIM,usrn);
167     if(n_b_r<0){
168         pthread_mutex_lock(&log->sem);

```

```

169     LogUnknownClientDisconnection(&log->fd,addrClient);
170     pthread_mutex_unlock(&log->sem);
171     pthread_exit((int*)1);
172 }
173 while((err = checkUsername(usrn, loggati))<0){
174     if(err == ERR_NO_USER_FILE){
175         pthread_mutex_lock(&log->sem);
176         LogErrorMessage(&log->fd,
177             USER_FILE_OPEN_ERR_MESSAGE);
178         pthread_mutex_unlock(&log->sem);
179         return ERR_NO_USER_FILE;
180     }
181     if(err == ERR_USER_ALREADY_LOGGED){
182         pthread_mutex_lock(&log->sem);
183         LogErrorMessage(&log->fd,
184             USER_FILE_OPEN_ERR_MESSAGE);
185         pthread_mutex_unlock(&log->sem);
186         n_b_r=sendMsg(sockfd,USER_ALREADY_LOGGED,usrn);
187         if(n_b_r<0){
188             pthread_mutex_lock(&log->sem);
189             LogUnknownClientDisconnection(&log->fd,
190                 addrClient);
191             pthread_mutex_unlock(&log->sem);
192             pthread_exit((int*)1);
193         }
194     }
195     else{
196         n_b_r=sendMsg(sockfd,WRONG_USERNAME_LIM,usrn);
197         if(n_b_r<0){
198             pthread_mutex_lock(&log->sem);
199             LogUnknownClientDisconnection(&log->fd,
200                 addrClient);
201             pthread_mutex_unlock(&log->sem);
202             pthread_exit((int*)1);
203         }
204     }
205     n_b_r=sendMsg(sockfd,INSERT_PASSWORD_LIM,pssw);
206     if(n_b_r<0){
207         pthread_mutex_lock(&log->sem);
208         LogUnknownClientDisconnection(&log->fd,addrClient);
209         pthread_mutex_unlock(&log->sem);
210         pthread_exit((int*)1);
211     }
212     while((err=logInUser(usrn, pssw, loggati)) != 0){
213         switch(err){
214             case ERR_NO_USER_FILE:

```

```

211     pthread_mutex_lock(&log->sem);
212     LogErrorMessage(&log->fd,
213         USER_FILE_OPEN_ERR_MESSAGE);
214     pthread_mutex_unlock(&log->sem);
215     return ERR_NO_USER_FILE;
216     case ERR_INPUT_OUTPUT:
217         pthread_mutex_lock(&log->sem);
218         LogErrorMessage(&log->fd, I_O_ERR_MESSAGE);
219         pthread_mutex_unlock(&log->sem);
220         return ERR_INPUT_OUTPUT;
221         case ERR_WRONG_PASSWORD:
222             n_b_r=sendMsg(sockfd,WRONG_PASSWORD_LIM,pssw);
223             if(n_b_r<0){
224                 pthread_mutex_lock(&log->sem);
225                 LogUnknownClientDisconnection(&log->fd,
226                     addrClient);
227                 pthread_mutex_unlock(&log->sem);
228                 pthread_exit((int*)1);
229             }
230             break;
231             case ERR_USERNAME_NOT_FOUND:
232                 n_b_r=sendMsg(sockfd,WRONG_USERNAME_LIM,usrn);
233                 if(n_b_r<0){
234                     pthread_mutex_lock(&log->sem);
235                     LogUnknownClientDisconnection(&log->fd,
236                         addrClient);
237                     pthread_mutex_unlock(&log->sem);
238                     if(n_b_r<0){
239                         pthread_mutex_lock(&log->sem);
240                         LogUnknownClientDisconnection(&log->fd,
241                             addrClient);
242                         pthread_mutex_unlock(&log->sem);
243                         pthread_exit((int*)1);
244                     }
245                     break;
246                     default:
247                         break;
248                     }
249                     pthread_mutex_lock(&log->sem);
250                     LogUserSignIn(&log->fd, usrn);
251                     pthread_mutex_unlock(&log->sem);
252                     n_b_r=sendMsgNoReply(sockfd,SUCCESS_MESSAGE_LIM);

```

```
253     return 1;
254 }
255
256 int deleteLoggedUser(char user[], loggedUser* loggati){
257
258     int i, res=0;
259
260     pthread_mutex_lock(&loggati->sem);
261     for(i=0;i<MAX_PLAYER_N;i++){
262         if(strcmp(user,loggati->user[i])==0){
263             strcpy(loggati->user[i],"");
264             res=1;
265             break;
266         }
267     }
268     pthread_mutex_unlock(&loggati->sem);
269     return res;
270 }
271
272 int insertLoggedUser(char user[],loggedUser *loggati){
273
274     int i, res=0;
275     pthread_mutex_lock(&loggati->sem);
276     for(i=0;i<MAX_PLAYER_N;i++){
277         if(strcmp(loggati->user[i],"")==0){
278             strcpy(loggati->user[i],user);
279             res=1;
280             break;
281         }
282     }
283     pthread_mutex_unlock(&loggati->sem);
284     return res;
285 }
286
287 int isLogged(char user[], loggedUser *loggati){
288
289     int i,res=0;
290     pthread_mutex_lock(&loggati->sem);
291     for(i=0;i<MAX_PLAYER_N; i++){
292         if(strcmp(user,loggati->user[i])==0){
293             res=1;
294             break;
295         }
296     }
297     pthread_mutex_unlock(&loggati->sem);
298     return res;
```

```
299 }
300
301 void initializeLoggedUser(loggedUser *loggati){
302     int i;
303     pthread_mutex_lock(&loggati->sem);
304     for(i=0;i<MAX_PLAYER_N;i++){
305         strcpy(loggati->user[i], "");
306     }
307     pthread_mutex_unlock(&loggati->sem);
308 }
309
```

```
1 #include "../codici_errori.h"
2 #include "../messaggi.h"
3 #include "../costanti.h"
4 #include "Log.h"
5 #include "Communication.h"
6
7 /*
8  La libereria contiene le funzioni utili a gestire
9  registrazione/Login degli utenti. Per tal motivo tutte le
10 funzioni ricevono un puntatore ad
11 una struttura di tipo loggedUser, che contiene le
12 informazione sugli utenti correntemente collegati al Server
13 .
14 */
15 /*
16  La funzione registerUser aggiunge un nuovo utente al file
17 RegisteredUser.txt.
18 Al termine della registrazione l'utente viene considerato
19 loggato e aggiunto, tramite la funzione insertLoggedUser
20 tra
21 gli utenti correntemente connessi al Server,
22 Parametri:
23     char* newuser, il nome utente da aggiungere;
24     char* newpasswd, la password del nuovo utente;
25 Valori di Ritorno:
26     ERR_INVALID_USERNAME, nel caso il nome utente non sia
27     valido (Lunghezza > MAX_SIZE_USERNAME);
28     ERR_NO_USER_FILE, nel caso sia fallita l'apertura del
29     file RegisteredUser.txt;
30     ERR_INPUT_OUTPUT, nel caso sia fallita la scrittura su
31     RegisteredUser.txt;
32     ERR_INSERT_LOGGED_USER, nel caso sia fallita la
33     funzione insertLoggedUser.
34 */
35 int registerUser(char* newuser, char* newpasswd, loggedUser
36 * );
37 /*
38  La funzione checkUsername controlla se lo username
39  passato come parametro sia presente all'interno del file.
40  Al suo interno richiama
41  La funzione isLogged per controllare che l'utente non sia
42  già loggato sul Server. Se la ricerca è andata a buon fine
43  ritorna l'offset
44  del nome utente cercato.
```

31 *Parametri:*
32 *char* username, il nome utente da cercare;.*
33 *Valori di Ritorno:*
34 *int res, l'offset del nome utente cercato all'interno del file;*
35 *ERR_NO_USER_FILE, nel caso sia fallita l'apertura del file RegisteredUser.txt;*
36 *ERR_USER_ALREADY_LOGGED, nel caso l'utente specificato sia già loggato;*
37 *ERR_USERNAME_NOT_FOUND, nel caso l'utente specificato non sia stato trovato.*
38 */
39 **int** checkUsername(**char*** username, **loggedUser***);
40
41 /*
42 *La funzione LogInUser autentica un utente passato come parametro tramite la password passata come parametro. Se l'autenticazione è andata a*
43 *a buon fine inserisce l'utente all'interno della struttura loggedUser. Richiama al suo interno la funzione checkUsername per controllare se il nome utente è effettivamente presente all'interno*
44 *del file RegisteredUser.txt. L'effettivo inserimento dell'utente tra quelli loggati è affidato alla funzione insertLoggedUser.*
45 *Parametri:*
46 *char* user, il nome utente da autenticare;*
47 *char* passw, la password da usare per l'autenticazione;*
48 *Valori di Ritorno:*
49 *ERR_NO_USER_FILE, nel caso sia fallita l'apertura del file RegisteredUser.txt;*
50 *ERR_INPUT_OUTPUT, nel caso sia fallita la funzione di lseek;*
51 *ERR_INSERT_LOGGED_USER, nel caso sia fallito l'inserimento nella struttura di tipo LoggedUser;*
52 *ERR_WRONG_PASSWORD, nel caso la password registrata non corrisponda a quella passata come parametro;*
53 *0, nel caso il login sia andato a buon fine;*
54 *-1, negli altri casi d'errore.*
55 */
56 **int** logInUser(**char*** user, **char*** passw, **loggedUser***);
57
58 /*
59 *La funzione gestisce il flusso d'interazione con l'utente durante il login. Invia un prompt per l'inserimento dell'username e, dopo aver effettuato controlli sullo stesso*

60 chiede all'utente la propria password. Se la coppia username, password è valida invia un messaggio di successo di login al client per portare avanti l'interazione.

61 Parametri:

62 int sockfd, la socket su cui scrivere;

63 char usrn[], il buffer per il nome utente;

64LogFile* log, la struttura di log;

65 char addrClient[], l'indirizzo ip del Client specifico

66 Valori di Ritorno:

67 ERR_NO_USER_FILE, nel caso sia fallita l'apertura del file RegisteredUser.txt;

68 ERR_USER_ALREADY_LOGGED, nel caso l'utente specificato sia già loggato;

69 ERR_INPUT_OUTPUT, nel caso sia fallita la funzione di lseek;

70 ERR_WRONG_PASSWORD, nel caso la password registrata non corrisponda a quella passata come parametro;

71 1, se l'utente è stato correttamente autenticato.

72 */

73 **int** logInUserMenu(**int** sockfd, **char** usrn[], **LogFile*** log, **loggedUser*** loggati, **char** addrClient[]);

74

75 /*

76 La funzione controlla il flusso dell'interazione con l'utente durante la registrazione. Invia un prompt per l'inserimento dello username e, se questo è valido, chiede di inserire una password. Dopo di che chiama la funzione registerUser sui dati inseriti.

77 Parametri:

78 int sockfd, la socket su cui scrivere;

79 char usrn[], il buffer per il nome utente;

80LogFile* log, la struttura di log;

81 char addrClient[], l'indirizzo ip del Client specifico

82 Valori di Ritorno:

83 1, se la funzione registerUser è andata a buon fine;

84 ERR_NO_USER_FILE, nel caso sia fallita l'apertura del file RegisteredUser.txt;

85 ERR_INPUT_OUTPUT, nel caso sia fallita la funzione di lseek;

86 ERR_INVALID_USERNAME, nel caso il nome utente non sia valido.

87 */

88 **int** signInUserMenu(**int** sockfd, **char** usrn[], **LogFile*** log, **loggedUser*** loggati, **char** addrClient[]);

```
90
91 /*
92  La funzione controlla se l'utente passato come parametro
93  è tra quelli loggati al Server correntemente.
94  Parametri:
95      char user[], il nome utente da controllare;
96  Valori di Ritorno:
97      1, se l'utente è Loggato;
98      0, altrimenti.
99 */
100
101 /*
102  La funzione inserisce l'utente passato come parametro
103  tra quelli correntemente loggati al Server.
104 Parametri:
105     char user[], il nome utente da inserire;
106 Valori di Ritorno:
107     1, se l'inserimento è andato a buon fine;
108     0, altrimenti.
109 */
110
111 /*
112  La funzione cancella l'utente passato come parametro tra
113  quelli correntemente loggati al Server.
114 Parametri:
115     char user[], il nome utente da cancellare;
116 Valori di Ritorno:
117     1, se la cancellazione è andata a buon fine;
118     0, altrimenti.
119 */
120
121 /*
122  La funzione alloca la memoria necessaria alla struttura
123  globale Loggati di tipo LoggedUser*.
124 Parametri:
125     LoggedUser* Loggati, il puntatore alla struttura
126     globale.
127 */
128 void inicializaLoggedUser(loggedUser* loggati);
```

```
1 #include "Gameplay.h"
2
3 Game *createGame(){
4     Game* temp;
5     temp=(Game*)calloc(1,sizeof(Game));
6     temp->grid=NULL;
7     int i;
8     for(i=0;i<MAX_PLAYER_N;i++)
9         temp->giocatori[i].posx=-1;
10    pthread_mutex_init(&(temp->sem),NULL);
11    return temp;
12 }
13
14 int playGame(Game * game, int idGiocatore, int gameId,int
15 sockfd,LogFile *serverLog){
16     int n_b_r, result=0, out=0;
17     char msg[100], matrix[5000];
18
19     GameGridToText(game->grid,matrix,idGiocatore,&game->
20 giocatori[idGiocatore]);
21     if(sendMsg(sockfd,matrix,msg)<0){
22         azioneGiocatore(game,idGiocatore,'0',game->gameId,&
23 serverLog->fd);
24         return PLAYER_EXITS;
25     }
26     while(!(game->timeOver)){
27         pthread_mutex_lock(&serverLog->sem);
28         result=azioneGiocatore(game,idGiocatore,msg[0],game->
29 gameId,&serverLog->fd);
30         pthread_mutex_unlock(&serverLog->sem);
31         pthread_mutex_lock(&game->sem);
32         GameGridToText(game->grid,matrix,idGiocatore,&game->
33 giocatori[idGiocatore]);
34         if(result<0){
35             strcat(matrix,"\\a");
36         }
37         pthread_mutex_unlock(&game->sem);
38         if(result == PLAYER_EXITS){
39             sendMsgNoReply(sockfd,"GETOUT");
40             return PLAYER_EXITS;
41         }
42         if( result==PLAYER_MAKE_A_POINT ){
43             pthread_mutex_lock(&game->sem);
44             if(gameHasToEnd(game,idGiocatore)){
45                 out=1;
46             }
47         }
48     }
49 }
```

```

42         pthread_mutex_unlock(&game->sem);
43         raise(SIGALRM);
44         break;
45     }
46     pthread_mutex_unlock(&game->sem);
47 }
48 if(out==0){
49     if(sendMsg(sockfd,matrix,msg)<0){
50         azioneGiocatore(game,idGiocatore,'0',game->gameId,&
serverLog->fd);
51         return PLAYER_EXITS;
52     }
53 }
54 }
55 if(didIWin(game, idGiocatore)){
56     sendMsgNoReply(sockfd, VICTORY_MESSAGE);
57     pthread_mutex_lock(&serverLog->sem);
58     LogPlayerWin(&serverLog->fd, gameId,&game->giocatori[
idGiocatore].nome);
59     pthread_mutex_unlock(&serverLog->sem);
60     return GAME_END_FOR_TIME;
61 }else{
62     sendMsgNoReply(sockfd, LOSS_MESSAGE);
63     return GAME_END_FOR_TIME;
64 }
65 return 0;
66 }
67
68 int azioneGiocatore(Game *game, int giocatore, char action
, int gameId, int * fdLog){
69
70     srand(time(NULL));
71     pthread_mutex_lock(&game->sem);
72     char src[9], dest[9];
73     player *player = &game->giocatori[giocatore];
74     GameGrid ** grid = game->grid;
75     int x=player->posx,y=player->posy;
76     int destx;
77     int desty;
78     switch (action) {
79         case 'w': case 'W':
80         if(y>0){
81             setPermessi(x,y-1,giocatore,grid);
82             if(grid[y-1][x].ostacolo || grid[y-1][x].giocatore){
83                 pthread_mutex_unlock(&game->sem);
84                 return -1;

```

```

85      }else{
86          sprintf(src,"%d,%d]",player->posx,player->posy);
87          grid[y][x].giocatore=0;
88          grid[y-1][x].giocatore=1;
89          grid[y-1][x].codiceGiocatore=giocatore;
90          player->posy=y-1;
91          sprintf(dest,"%d,%d]",player->posx,player->posy);
92          LogPlayerMoves(fdLog,gameId,player->nome,src,dest
93      );
94  }
95  }else{
96      pthread_mutex_unlock(&game->sem);
97      return -1;
98  }
99  break;
100 case 'a': case 'A':
101 if(x>0){
102     setPermessi(x-1,y,giocatore,grid);
103     if(grid[y][x-1].ostacolo || grid[y][x-1].giocatore
104 ){
105         pthread_mutex_unlock(&game->sem);
106         return -1;
107     }else{
108         sprintf(src,"%d,%d]",player->posx,player->posy
109     );
110         grid[y][x].giocatore=0;
111         grid[y][x-1].giocatore=1;
112         grid[y][x-1].codiceGiocatore=giocatore;
113         player->posx=x-1;
114         sprintf(dest,"%d,%d]",player->posx,player->posy
115     );
116         LogPlayerMoves(fdLog,gameId,player->nome,src,
117 dest);
118     }
119  }else{
120      pthread_mutex_unlock(&game->sem);
121      return -1;
122  }
123  break;
124 case 's': case 'S':
125 if(y<MAX_GRID_SIZE_H-1){
126     setPermessi(x,y+1,giocatore,grid);
127     if(grid[y+1][x].ostacolo || grid[y+1][x].giocatore){
128         pthread_mutex_unlock(&game->sem);
129         return -1;
130     }else{

```

```

126         sprintf(src, "[%d,%d]", player->posx, player->posy);
127         grid[y][x].giocatore=0;
128         grid[y+1][x].giocatore=1;
129         grid[y+1][x].codiceGiocatore=giocatore;
130         player->posy=y+1;
131         sprintf(dest, "[%d,%d]", player->posx, player->posy);
132         LogPlayerMoves(fdLog, gameId, player->nome, src, dest
133     );
134 }
135 }else{
136     pthread_mutex_unlock(&game->sem);
137     return -1;
138 }
139 break;
140 case 'd': case 'D':
141     if(x<MAX_GRID_SIZE_L-1){
142         setPermessi(x+1,y,giocatore,grid);
143         if(grid[y][x+1].ostacolo || grid[y][x+1].giocatore
144     ){
145         pthread_mutex_unlock(&game->sem);
146         return -1;
147     }else{
148         sprintf(src, "[%d,%d]", player->posx, player->posy
149     );
150         grid[y][x].giocatore=0;
151         grid[y][x+1].giocatore=1;
152         grid[y][x+1].codiceGiocatore=giocatore;
153         player->posx=x+1;
154         sprintf(dest, "[%d,%d]", player->posx, player->posy
155     );
156         LogPlayerMoves(fdLog, gameId, player->nome, src,
157         dest);
158     }
159 }else{
160     pthread_mutex_unlock(&game->sem);
161     return -1;
162 }
163 break;
164 case 'q': case 'Q':
165     if(player->pacco==0){
166         if(grid[y][x].pacco){
167             player->pacco = 1;
168             player->codicePacco = grid[y][x].codicePacco;
169             grid[y][x].codicePacco = 0;
170             grid[y][x].pacco = 0;
171             sprintf(dest, "[%d,%d]", player->posx, player->posy);

```

```

167      LogPlayerTakePackage(fdLog, gameId, player->nome,
168      player->codicePacco, dest);
169      pthread_mutex_unlock(&game->sem);
170      return -1;
171  }
172 }else{
173     pthread_mutex_unlock(&game->sem);
174     return -1;
175 }
176 break;
177 case 'e': case 'E':
178 if(player->pacco){
179     if(grid[y][x].locazione){
180         //controllo se la locazione corrisponde al pacco
181         if(grid[y][x].codiceLocazione == player->
182         codicePacco){
183             game->punteggio[giocatore]++;
184             grid[y][x].locazione = 0;
185             grid[y][x].codiceLocazione = 0;
186             player->pacco = 0;
187             player->codicePacco = 0;
188             LogPlayerMakeAPoint(fdLog, gameId, player->nome
189 );
190             pthread_mutex_unlock(&game->sem);
191             return PLAYER_MAKE_A_POINT;
192 }else{
193             pthread_mutex_unlock(&game->sem);
194             return -1;
195 }else{
196     if(grid[y][x].pacco==0){
197         grid[y][x].pacco=1;
198         grid[y][x].codicePacco = player->codicePacco;
199         player->pacco = 0;
200         player->codicePacco = 0;
201         for(int i = 0; i<MAX_PLAYER_N; i++){
202             setPermessi(x, y, i, grid);
203         }
204         sprintf(dest, "[%d,%d]", player->posx, player->
205         posy);
206         LogPlayerLeavePackage(fdLog, gameId, player->
207         nome, grid[y][x].codicePacco, dest);
208     }else{
209         pthread_mutex_unlock(&game->sem);
210         return -1;

```

```

208         }
209     }
210 }else{
211     pthread_mutex_unlock(&game->sem);
212     return -1;
213 }
214 break;
215 case 'r': case 'R'://refresh
216 break;
217 case '0'://esci
218     grid[y][x].giocatore = 0;
219     if(player->pacco){
220         destx = x;
221         desty = y;
222         while(grid[desty][destx].pacco || grid[desty][
223             destx].locazione || grid[desty][destx].giocatore || grid[
224             desty][destx].ostacolo ){
225             destx = rand()%MAX_GRID_SIZE_L;
226             desty = rand()%MAX_GRID_SIZE_L;
227         }
228         grid[desty][destx].pacco=1;
229         grid[desty][destx].codicePacco = player->
230         codicePacco;
231         for(int i = 0; i<MAX_PLAYER_N; i++){
232             setPermessi(destx, desty, i, grid);
233         }
234         player->pacco = 0;
235         player->codicePacco = 0;
236     }
237     game->giocatori[giocatore].posx = -1;
238     game->punteggio[giocatore] = 0;
239     if(game->piena){
240         game->piena = 0;
241     }
242     pthread_mutex_unlock(&game->sem);
243     return PLAYER_EXITS;
244 }
245 pthread_mutex_unlock(&game->sem);
246 return 0;
247 }
248
249 int isGameEmpty(Game* game){
250

```

```
251 int empty=1;
252 for(int i = 0; i<MAX_PLAYER_N; i++ ){
253     if(game->giocatori[i].posx>=0){
254         empty = 0;
255     }
256 }
257 return empty;
258 }
259
260 void setPermessi(int x, int y, int giocatore, GameGrid ** grid){
261     grid[y][x].permessi[giocatore] = 1;
262     return;
263 }
264
265 void setPermessiToAll(int x, int y, GameGrid ** grid){
266     int i;
267     for(i=0; i<MAX_PLAYER_N; i++){
268         grid[y][x].permessi[i]=1;
269     }
270     return;
271 }
272
273 int createGameGrid(Game *g){
274     srand(time(NULL));
275     int i, j, x, y, numOstacoli=0;
276     GameGrid **p=(GameGrid**)malloc(MAX_GRID_SIZE_H * sizeof
277     (GameGrid*));
278     if(p!=NULL){
279         for(i=0;i<MAX_GRID_SIZE_H;i++)
280             p[i]=(GameGrid*)calloc(MAX_GRID_SIZE_L, sizeof(
281             GameGrid));
282         y=rand()%MAX_GRID_SIZE_H;
283         x=rand()%MAX_GRID_SIZE_L;
284         p[y][x].giocatore=1;
285         p[y][x].codiceGiocatore=0;
286         setPermessi(x, y, 0, p);
287         g->giocatori[0].posx=x;
288         g->giocatori[0].posy=y;
289         for (i=0; i < MAX_PACCHI; i++) {
290             do {
291                 y=rand()%MAX_GRID_SIZE_H;
292                 x=rand()%MAX_GRID_SIZE_L;
293             } while(p[y][x].giocatore || p[y][x].pacco);
294             p[y][x].pacco=1;
```

```

294     setPermessiToAll(x, y, p);
295     p[y][x].codicePacco=i;
296 }
297 for (i=0; i < MAX_PACCHI; i++) {
298     do {
299         y=rand()%MAX_GRID_SIZE_H;
300         x=rand()%MAX_GRID_SIZE_L;
301     } while(p[y][x].giocatore || p[y][x].pacco || p[y][x].
302             .locazione);
303     p[y][x].locazione=1;
304     setPermessiToAll(x, y, p);
305     p[y][x].codiceLocazione=i;
306     g->locazioneXPacchi[i]=x;
307     g->locazioneYPacchi[i]=y;
308 }
309 for(i=0;i<MAX_OBSTACLES_N;i++){
310     y=rand()%MAX_GRID_SIZE_H;
311     x=rand()%MAX_GRID_SIZE_L;
312     if(!(p[y][x].giocatore || p[y][x].pacco || p[y][x].
313         locazione)){
314         p[y][x].ostacolo=1;
315         numOstacoli++;
316     }
317     while(numOstacoli<MIN_OBSTACLES){
318         y=rand()%MAX_GRID_SIZE_H;
319         x=rand()%MAX_GRID_SIZE_L;
320         if(!(p[y][x].giocatore || p[y][x].pacco || p[y][x].
321             locazione)){
322             p[y][x].ostacolo=1;
323             numOstacoli++;
324         }
325     }
326 }
327 g->grid=p;
328 return 0;
329
330 }
331
332 void spawnNewPlayer(Game** game, char* username,int sockfd
333 ,LogFile* serverLog, loggedUser *loggati){
334     srand(time(NULL));
335     int x, y, i, time;

```

```

336     player playerToAdd;
337     int result;
338     Game *g=*game;
339     strcpy(playerToAdd.nome, username);
340     playerToAdd.codicePacco = 0;
341     playerToAdd.pacco = 0;
342
343     pthread_mutex_lock(&g->sem);
344     for(i = 0; i < MAX_PLAYER_N; i++){
345         if(g->giocatori[i].posx == -1){
346             g->giocatori[i] = playerToAdd;
347             break;
348         }
349     }
350     if (i < MAX_PLAYER_N) {
351         GameGrid** p = g->grid;
352         do{
353             y=rand()%MAX_GRID_SIZE_H;
354             x=rand()%MAX_GRID_SIZE_L;
355             }while(p[y][x].giocatore || p[y][x].pacco || p[y][x].
356             locazione || p[y][x].ostacolo);
357             p[y][x].giocatore = 1;
358             p[y][x].codiceGiocatore = i;
359             g->giocatori[i].posx = x;
360             g->giocatori[i].posy = y;
361             if(i == MAX_PLAYER_N-1){
362                 g->piena = 1;
363             }
364             setPermessi(x, y, i, p);
365             pthread_mutex_lock(&serverLog->sem);
366             LogPlayerJoin(&serverLog->fd, g->gameId, username);
367             pthread_mutex_unlock(&serverLog->sem);
368             pthread_mutex_unlock(&g->sem);
369
370             if(result =(playGame(g,i,g->gameId,sockfd,serverLog
371             ))){
372                 switch(result){
373                     case PLAYER_EXITS:
374                         pthread_mutex_lock(&serverLog->sem);
375                         LogUserSignOut(&serverLog->fd,username);
376                         pthread_mutex_unlock(&serverLog->sem);
377                         deleteLoggedUser(username,loggati);
378                         close(sockfd);
379                         if(isGameEmpty(g))
380                             raise(SIGALRM);

```

```

380         pthread_exit((int *) 1);
381     break;
382     case GAME_END_FOR_TIME:
383         pthread_mutex_lock(&serverLog->sem);
384         LogUserSignOut(&serverLog->fd,username);
385         pthread_mutex_unlock(&serverLog->sem);
386         deleteLoggedUser(username,loggati);
387         close(sockfd);
388         pthread_exit((int *) 1);
389     break;
390 }
391 }
392 }
393 return;
394 }
395
396 void initializeNewGame(Game ** game, int sockfd, char user
[],LogFile *toLog, loggedUser *loggati){
397
398     LogFile serverLog= *toLog;
399     char matrix[2000];
400     char msg[50];
401     int result;
402     int n_b_r;
403     Game *g;
404     *game=createGame();
405     g=*game;
406     strcpy(g->giocatori[0].nome,user);
407     g->giocatori[0].nome[strlen(user)]='\0';
408
409     if(createGameGrid(g) == 0){
410         pthread_mutex_lock(&g->sem);
411         g->gameId = rand()%10000;
412
413         pthread_mutex_lock(&serverLog.sem);
414         LogNewGame(&serverLog.fd,g->gameId);
415         pthread_mutex_unlock(&serverLog.sem);
416         pthread_mutex_lock(&serverLog.sem);
417         LogPlayerJoin(&serverLog.fd, g->gameId, user);
418         pthread_mutex_unlock(&serverLog.sem);
419         pthread_mutex_unlock(&g->sem);
420         //Set timer
421         if(alarm(MAX_TIME)>0){
422             alarm(0);
423             alarm(MAX_TIME);
424         }

```

```
425
426     if(result ==(playGame(g,0,g->gameId,sockfd,toLog))){
427         switch(result){
428             case PLAYER_EXITS:
429                 pthread_mutex_lock(&serverLog.sem);
430                 LogUserSignOut(&serverLog.fd,user);
431                 pthread_mutex_unlock(&serverLog.sem);
432                 deleteLoggedUser(user,loggati);
433                 close(sockfd);
434                 if(isGameEmpty(g))
435                     raise(SIGALRM);
436                 pthread_exit((int *) 1);
437                 break;
438             case GAME_END_FOR_TIME:
439                 pthread_mutex_lock(&serverLog.sem);
440                 LogUserSignOut(&serverLog.fd,user);
441                 pthread_mutex_unlock(&serverLog.sem);
442                 deleteLoggedUser(user,loggati);
443                 close(sockfd);
444                 pthread_exit((int *) 1);
445                 break;
446         }
447     }
448 }else{
449     /*Gestione errore*/
450 }
451
452 return;
453 }
454
455 void deleteGrid(GameGrid **g){
456     int i;
457     for(i=0;i<MAX_GRID_SIZE_H;i++)
458         free(g[i]);
459     free(g);
460     return;
461 }
462
463 int didIWin(Game * g, int idGiocatore){
464
465     int i = 0, max;
466     max = g->punteggio[0];
467
468     for(i = 1; i< MAX_PLAYER_N; i++){
469         if(g->punteggio[i]>max){
470             max = g->punteggio[i];
```

```
471     }
472 }
473
474     return ((max == g->punteggio[idGiocatore]) && max>0);
475 }
476
477 int gameHasToEnd(Game *game, int idGiocatore){
478
479     int i,sum=0;
480
481     if( game->punteggio[idGiocatore] > (MAX_PACCHI/2) ){
482         return 1;
483     }
484     for(i=0;i<MAX_PLAYER_N;i++)
485         sum+=game->punteggio[i];
486     if(sum==MAX_PACCHI)
487         return 1;
488     return 0;
489 }
490
```

```
1 #include "../codici_errori.h"
2 #include "../messaggi.h"
3 #include "../costanti.h"
4 #include "Communication.h"
5 #include "Users.h"
6
7 /*La Libreria contiene tutte le funzioni utili a gestire
una partita.*/
8
9 /*
10 La funzione modifica la griglia di gioco in accordo con
Le mosse del giocatore.
11 Parametri:
12     Game game, è un puntatore alla partita specifica;
13     int giocatore, contiene l'identificativo del giocatore
specifico;
14     char action, contiene l'identificativo testuale della
mossa:
15         action può assumere i seguenti valori: {'W', 'A', 'S', '
D', 'Q', 'E', '0', 'R'};
16         int gameId, contiene l'id della partita specifica.
17         int* fdLog, è il file descriptor del file di log
specifico della sessione server in corso.
18     Valori di Ritorno:
19         -1, se la mossa descritta da action non era possibile (
es. spostamento al di fuori della matrice di gioco);
20         PLAYER_EXITS, un valore costante definito in costanti.h
se la mossa specificata corrispondeva ad uscita giocatore;
21 */
22 int azioneGiocatore(Game *game, int giocatore, char action
, int gameId, int * fdLog);
23
24 /*
25 Setta i permessi per la visibilità di una casella in
accordo con le mosse di un giocatore.
26 Parametri:
27     int x, posizione x della cella all'interno della
matrice;
28     int y, posizione y della cella all'interno della
matrice;
29     int giocatore, identificativo del giocatore per cui
settare i permessi;
30     GameGrid ** grid, La matrice di gioco;
31 */
32 void setPermessi(int x, int y, int giocatore, GameGrid **
grid);
```

```
33 /*
34  * Setta i permessi per la visibilità di una casella per
35  * ogni giocatore.
36  * Parametri:
37  *     int x, posizione x della cella all'interno della
38  *     matrice;
39  *     int y, posizione y della cella all'interno della
38  *     matrice;
39  *     GameGrid ** grid, la matrice di gioco;
40 */
41 void setPermessiToAll(int x, int y, GameGrid ** grid);
42
43 /*
44  * Controlla se la partita corrente deve terminare. Ritorna
45  * 0 se la partita deve continuare, 1 altrimenti.
46  * Parametri:
47  *     Game * game, il puntatore alla partita corrente;
48  *     int idGiocatore, l'id del giocatore corrente.
49  * Valori di Ritorno:
50  *     1, se il giocatore specificato da idGiocatore ha
51  *     consegnato la metà dei pacchi più uno 0
52  *     se il numero totali di pacchi consegnati corrisponde
53  *     al numero totale di pacchi disponibili;
54  *     0, se le condizioni sopracitate non avvengono;
55 */
56 int gameHasToEnd(Game *game, int idGiocatore);
57
58 /*
59  * Crea la matrice di gioco associata ad una partita.
60  * Parametri:
61  *     Game * g, il puntatore alla partita corrente;
62  * Valori di Ritorno:
63  *     1, se la creazione ha avuto successo;
64  *     0, altrimenti;
65 */
66 int createGameGrid(Game *g);
67
68 /*
69  * Al momento della disconessione di un Client controlla che
70  * la partita corrente contiene almeno
71  * un giocatore.
72  * Parametri:
73  *     Game* g, il puntatore alla partita corrente;
74  * Valori di Ritorno:
75  *     1, se la partita corrente è vuota;
```

```
72     0, altrimenti;
73 */
74 int isGameEmpty(Game* game);
75
76 /*
77   La funzione controlla se il giocatore specificato abbia
    vinto la partita, controllando se il
78   suo id corrisponde al valore massimo presente all'
    interno dell'array Punteggio contenuto in game.
79   Parametri:
80       Game * game, il puntatore alla partita corrente;
81       int idGiocatore, l'id del giocatore corrente.
82   Valori di ritorno:
83       1, se il giocatore corrente ha vinto;
84       0, altrimenti.
85 */
86 int didIWin(Game* g, int idGiocatore);
87
88 /*
89   La funzione alloca una partita e inizializza il semaforo
    contenuto in Game.
90   Valori di Ritorno:
91       un puntatore ad una struttura game se l'allocazione è
        andata a buon fine;
92       NULL altrimenti.
93 */
94 Game * createGame();
95
96 /*
97   La funzione playGame consente al giocatore di giocare
    alla partita corrente. Preso un giocatore ed una partita
98   si mette in ascolto dei messaggi del Client e richiama
    la funzione azioneGiocatore su quel giocatore per
    calcolare
99   il risultato della mossa indicata dal client. Dopo di
    che invia la matrice aggiornata al Client. Esegue inoltre
    i controlli
100  necessari a verificare la fine della partita. Se avviene
    il fine partita PlayGame si occupa anche di inviare ai
    vari client
101  giocanti i messaggi di vittoria e sconfitta.
102  Parametri:
103      Game* game, il puntatore alla partita corrente;
104      int idGiocatore, l'id del giocatore corrente;
105      int gameId, l'id del giocatore corrente;
106      int sockfd, il fd della socket di comunicazione;
```

107 `LogFile*` *serverLog*, il puntatore alla struttura che
 contiene il fd del Log e il relativo mutex.
108 Valori di Ritorno:
109 `PLAYER_EXITS`, un valore costante associato all'evento
 'giocatore esce dalla partita';
110 `GAME_END_FOR_TIME`, un valore costante rappresentante l'
 'evento 'fine tempo massimo di gioco'.
111 */
112 **int** playGame(`Game` * *game*, **int** *idGiocatore*, **int** *gameId*, **int**
 sockfd, `LogFile*` *serverLog*);
113
114 /*
115 La funzione spawnNewPlayer si occupa di gestire l'evento
 giocatore si unisce a partita in corso.
116 Inserisce un giocatore all'interno della matrice di
 gioco e gli assegna un id. Dopodichè popola
117 la corrispettiva cella dell'array di player in game.
 Dopo aver fatto ciò richiama la funzione PlayGame per
118 quella partita e quel giocatore.
119 Se la funzione playGame ritorna `PLAYER_EXITS` il server
 logga l'evento d'uscita
120 giocatore ed elimina il giocatore dalla struttura che
 gestisce i giocatori `LogGame` * *game*, il puntatore alla
 partita corrente;
121 int *idGiocatore*, l'id del giocatore corrente.gati.
 Dopo aver chiuso la socket
122 controlla se la partita contiene ancora almeno un
 giocatore, se ciò non avviene lancia un segnale di `SIGALRM`
 .
123 In entrambi casi a quel punto l'esecuzione del thread
 viene terminata.
124 Se playGame ritorna `GAME_END_FOR_TIME` allora si logga l'
 evento d'uscita, si cancella l'utente dall'elenco di
 giocatori Loggati,
125 si chiude la socket e si termina l'esecuzione del thread
 .
126 Parametri:
127 `Game **` *game*, un puntatore a puntatore alla partita
 corrente;
128 `char*` *username*, una stringa contenente il nickname
 dell'utente;
129 `int` *sockfd*, il fd della socket di comunicazione;
130 `LogFile*` *serverLog*, il puntatore alla struttura che
 contiene il fd del Log e il relativo mutex,
131 `LoggedUser*` *loggati*, il puntatore alla struttura che
 contiene i giocatori correntemente loggati.

```
132 */
133 void spawnNewPlayer(Game ** game , char* username, int
134     sockfd, LogFile* serverLog, loggedUser* loggati);
135 /*
136     La funzione dealloca la matrice di gioco associata alla
137     partita.
138     Parametri:
139     GameGrid ** g, un puntatore a puntatore alla matrice
140     associata alla partita.
141 */
142 /*
143     La funzione crea una nuova partita. Sfrutta la funzione
144     createGame.
145     Dopo aver creato la partita e il giocatore che, essendo
146     il primo, riceve id 0 chiama la funzione playGame.
147     Il suo comportamento di gestione dei valori di ritorno
148     di playGame (PLAYER_EXITS e GAME_END_FOR_TIME) è
149     perfettamente
150     uguale a quello della funzione spawnNewPlayer descritto
151     sopra.
152     Parametri:
153     Game ** game, un puntatore a puntatore alla partita
154     corrente;
155     char user[], una stringa contenente il nickname dell'
156     utente;
157     int sockfd, il fd della socket di comunicazione;
158     LogFile* toLog, il puntatore alla struttura che
159     contiene il fd del Log e il relativo mutex,
160     LoggedUser* loggati, il puntatore alla struttura che
161     contiene i giocatori correntemente loggati.
162 */
163 void inicializaNewGame(Game** game,int sockfd, char user
164     [],LogFile* toLog,loggedUser* loggati);
165 
```

```

1 #include "Communication.h"
2
3 /*
4  La funzione creaSocket si occupa di creare un socket per
5  la comunicazione
6  Client Server.
7 */
8 int creaSocket(int porta){
9     struct sockaddr_in server_address;
10    int sock;
11    if((sock=socket(AF_INET, SOCK_STREAM, 0)) < 0){
12        return ERR_SOCKET_CREATION ;
13    }else{
14        server_address.sin_family = AF_INET;
15        server_address.sin_port = htons(porta);
16        memset(&(server_address.sin_zero), '\0', 8);
17        server_address.sin_addr.s_addr = htonl(INADDR_ANY);
18        if((bind(sock,(struct sockaddr *)&server_address,sizeof
19             (server_address)))<0){
20            return ERR_SOCKET_BINDING;
21        }
22    }
23
24 int GameGridToText(GameGrid **p, char msg[], int giocatore
25 , player * gamer){
26
27    int i,j;
28    char mom[150];
29    msg[0]='\0';
30    for(i=0;i<MAX_GRID_SIZE_L+1;i++){
31        if(i!=0){
32            if(i/10==0)
33                sprintf(mom, "\033[30;47m %d \033[0m",i);
34            else
35                sprintf(mom, "\033[30;47m %d \033[0m",i);
36            strcat(msg,mom);
37        }else{
38            sprintf(mom, "    ");
39            strcat(msg, mom);
40        }
41        sprintf(mom, "\n");
42        strcat(msg, mom);
43    for(i=0;i<MAX_GRID_SIZE_H;i++){

```

```

44     if(i+1!=10)
45         sprintf(mom, "\033[30;47m %d \033[0m", i+1);
46     else
47         sprintf(mom, "\033[30;47m %d \033[0m", i+1);
48     strcat(msg,mom);
49     for(j=0;j<MAX_GRID_SIZE_L;j++){
50         printCellForPlayer(p, msg,giocatore, gamer, i, j);
51     }
52     strcat(msg, "\n\n");
53 }
54 strcat(msg, IN_GAME_MENU);
55 return 1;
56 }
57
58 int sendMsgNoReply(int sockfd,char toSend[]){
59
60     int n_b_r, n_b_w;
61     char msg[150];
62
63     clear();
64     sprintf(msg,"%ld",strlen(toSend));
65     n_b_w=write(sockfd,msg,strlen(msg));
66     if(n_b_w<strlen(msg)){
67         return ERR_SENDING_MESSAGE;
68     }
69     n_b_r=read(sockfd,msg,5);
70     if (n_b_r<0) {
71         return ERR RECEIVING_MESSAGE;
72     }
73     n_b_w=write(sockfd,toSend,strlen(toSend));
74     if (n_b_w<strlen(toSend)) {
75         return ERR_SENDING_MESSAGE;
76     }
77     return n_b_w;
78 }
79
80 int sendMsg(int sockfd,char toSend[],char received[]){
81
82     int n_b_r, n_b_w;
83     char msg[250];
84
85     clear();
86     sprintf(msg,"%ld",strlen(toSend));
87     n_b_w=write(sockfd,msg,strlen(msg));
88     if(n_b_w<strlen(msg)){
89         return ERR_SENDING_MESSAGE;

```

```

90     }
91     n_b_r=read(sockfd,msg,5);
92     n_b_w=write(sockfd,toSend,strlen(toSend));
93     if (n_b_w<strlen(toSend)) {
94         return ERR_SENDING_MESSAGE;
95     }
96     n_b_r=read(sockfd,received,50);
97     if( (strcmp(received,USER_LOG_OUT)==0) || (strcmp(
98     received,CLIENT_GONE)==0) ){
99         return -1;
100    }
101    received[n_b_r]='\0';
102    return n_b_r;
103 }
104
105 void printCellForPlayer(GameGrid **p, char msg[], int
giocatore, player * gamer, int i, int j){
106     char mom[150];
107     if (p[i][j].giocatore) {
108         if (p[i][j].codiceGiocatore==giocatore)
109             strcat(msg, "\033[92m ME \033[0m");
110         else
111             strcat(msg, "\033[92m\x1b[35m EN \033[0m");
112     }else{
113         if(p[i][j].permessi[giocatore]==0)
114             strcat(msg, " ? ");
115         else{
116             if (p[i][j].ostacolo)
117                 strcat(msg, "\033[91m X \033[0m");
118             else{
119                 if (p[i][j].pacco)
120                     strcat(msg, "\033[93m P \033[0m");
121                 else {
122                     if(p[i][j].locazione){
123                         if(gamer->pacco && gamer->codicePacco==p[i][j]
124                             .codiceLocazione)
125                             strcat(msg, "\033[102;30;5m L \033[0m ");
126                         else
127                             strcat(msg, "\033[96m L \033[0m");
128                     }else
129                         strcat(msg, " [] ");
130                 }
131             }
132         }
}

```

```
133     return;  
134 }  
135
```

```

1 /*
2  La Libreria conterrà tutte le funzioni utili a gestire la
3   comunicazione tra i Client e il Server.
4 */
5 #include "../codici_errori.h"
6 #include "../messaggi.h"
7 #include "../costanti.h"
8
9 /*
10  La funzione crea una socket tcp sulla porta passata come
11   parametro.
12  Parametri:
13    int porta, un intero rappresentante La porta su cui
14     verrà creata la socket tcp;
15  Valori di Ritorno:
16    int sock, un intero corrispondente al fd della socket
17     appena creata;
18    ERR_SOCKET_CREATION, un codice d'errore per il
19      fallimento della funzione di libreria socket;
20    ERR_SOCKET_BINDING, un codice d'errore per il
21      fallimento della funzione di libreria bind;
22 */
23 int creaSocket(int porta);
24 /*
25  La funzione GameGridToText trasforma La struttura
26  GameGrid associata alla partita in una stringa
27  che potrà poi essere comunicata al Client. Sfrutta la
28  funzione printCellForPlayer per stampare
29  La cella per il giocatore specificato dall'id.
30  Parametri:
31    GameGrid ** p, La matrice da trasformare;
32    char msg[], il buffer su cui sarà scritta la matrice
33     trasformata;
34    int giocatore, l'id del giocatore specifico,
35    player* gamer, il puntatore alla struttura player
36     corrispondente al giocatore.
37 */
38 int GameGridToText(GameGrid **p, char msg[], int giocatore
39 , player* gamer);
40 /*
41  La funzione che si occupa di comunicare con il Client nel
42  caso il Server necessiti di una
43  risposta per continuare l'esecuzione.

```

```
35  Parametri:  
36      int sockfd, il fd della socket di comunicazione;  
37      char toSend[], la stringa da scrivere sulla porta;  
38      char received[], la risposta del Client.  
39  Valori di ritorna:  
40      int n_b_r, corrispondente al numero di bit letti dal  
server;  
41      ERR_SENDING_MESSAGE, codice di errore corrispondente  
alla mancata scrittura sulla socket;  
42      -1, se il messaggio inviato dal Client corrispondeva ad  
uno dei messaggi di log out, cioè  
43          USER_LOG_OUT e CLIENT_GONE.  
44 */  
45 int sendMsg(int , char [], char []);  
46  
47 /*  
48     La funzione che si occupa di comunicare con il Client nel  
caso il Server NON necessiti di  
49     una risposta per continuare l'esecuzione.  
50  Parametri:  
51      int sockfd, il fd della socket di comunicazione;  
52      char toSend[], la stringa da scrivere sulla porta;  
53  Valori di ritorna:  
54      int n_b_w, corrispondente al numero di bit scritti dal  
server;  
55      ERR_SENDING_MESSAGE, codice di errore corrispondente  
alla mancata scrittura sulla socket;  
56      ERR RECEIVING_MESSAGE, codice di errore corrispondente  
alla mancata lettura sulla socket.  
57 */  
58 int sendMsgNoReply(int, char []);  
59  
60 /*  
61     La funzione printCellForPlayer inserisce all'interno del  
messaggio da inviare al Client una traduzione  
62     simbolica del contenuto della cella, in accordo con i  
permessi del giocatore specifico.  
63  Parametri:  
64      GameGrid ** p, la matrice da trasformare;  
65      char msg[], il buffer su cui sarà scritta la matrice  
trasformata;  
66      int giocatore, l'id del giocatore specifico,  
67      player* gamer, il puntatore alla struttura player  
corrispondente al giocatore.  
68      int i, j, specificano la cella della matrice.  
69 */
```

```
70 void printCellForPlayer(GameGrid **p, char msg[], int
    giocatore, player * gamer, int i, int j);
71
```