Università degli Studi di Napoli Federico II

Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione

Corso di Laurea Triennale in Informatica

Classe n. L-31

Progetto di sistemi operativi

Traccia A

Professore:
Finzi Alberto

Candidati:
Turco Mario
Matr. N8600/2503
Longobardi Francesco
Matr. N8600/2468

Anno Accademico
2019/2020

Indice

1 Istruzioni preliminari

1.1 Modalità di compilazione

Il progetto è provvisto di un file makefile il quale è in grado di compilare autonomamente l'intero progetto. Per utilizzare il makefile aprire la cartella del progetto tramite la console di sistema e digitare "make".
In alternativa è possibile compilare manualemente il client ed il server con i seguenti comandi:

```
gcc -o server server.c boardUtility.c parser.c list.c -lpthread
gcc -o client client.c boardUtility.c parser.c list.c -lpthread
```

2 Guida all'uso

2.1 Server

Una volta compilato il progetto è possibile avviare il server digitando da console il seguente comando

```
./server users
```

L'identificativo *users* ri riferisce al nome del file sul quale sarà salvata la lista degli utenti e delle loro credenziali.
È possibile scegliere un nome a piacimento per il file purchè esso sia diverso da *Log*.

2.2 Client

Una volta compilato il progetto è possibile avviare il cient digitando da console il seguente comando:

```
./client ip porta
```

Dove *ip* andrà sostituito con l'ip o l'indirizzo URL del server e *porta* andrà sostituito con la porta del server.
Una volta avviato il client comprarià il menu con le scelte 3 possibili: accedi, registrati ed esci.
Una volta effettuata la registrazione dell'utente è possibile effettuare l'accesso al programma al seguito del quale verranno mostrate sia la mappa sia le istruzioni di gioco.

3 Comunicazione tra client e server

Di seguito verranno illustrate le modalità di comunicazione tra client e server.

3.1 Configurazione del server

Il socket del server viene configurato con famiglia di protoccoli PF_NET, con tipo di trasmissione dati SOCK_STREAM e con protocollo TCP. Mostriamo di seguito il codice sorgente:

Listato 1: Configurazione indirizzo del server

```
struct sockaddr_in configuraIndirizzo()
{
  struct sockaddr_in mio_indirizzo;
  mio_indirizzo.sin_family = AF_INET;
  mio_indirizzo.sin_port = htons(5200);
  mio_indirizzo.sin_addr.s_addr = htonl(INADDR_ANY);
  printf("Indirizzo socket configurato\n");
  return mio_indirizzo;
}
```

Listato 2: Configurazione socket del server

```
void configuraSocket(struct sockaddr_in mio_indirizzo)
{
  if ((socketDesc = socket(PF_INET, SOCK_STREAM, 0)) < 0)
  {
    perror("Impossibile creare socket");
    exit(-1);
  }
  if (setsockopt(socketDesc, SOL_SOCKET, SO_REUSEADDR, &(int){1}, sizeof(int)) <
      0)
    perror("Impossibile impostare il riutilizzo dell'indirizzo ip e della "
           "porta\n");
  if ((bind(socketDesc, (struct sockaddr *)&mio_indirizzo,
            sizeof(mio_indirizzo))) < 0)
  {
    perror("Impossibile effettuare bind");
    exit(-1);
  }
}
```

È importante notare anche come il server riesca a gestire in modo concorrente più client tramite l'uso di un thread dedicato ad ogni client connesso. Una volta aver configurato il socket, il server si mette in ascolto per nuove connessioni in entrata ed, ogni volta che viene stabilita una nuova connessione, il server avvia un thread per gestire tale connessione. Di seguito il relativo codice:

Listato 3: Procedura di ascolto del server

```
void startListening()
{
  pthread_t tid;
  int clientDesc;
  int *puntClientDesc;
  while (1 == 1)
  {
    if (listen(socketDesc, 10) < 0)
      perror("Impossibile mettersi in ascolto"), exit(-1);
    printf("In ascolto..\n");
    if ((clientDesc = accept(socketDesc, NULL, NULL)) < 0)
    {
      perror("Impossibile effettuare connessione\n");
      exit(-1);
    }
    printf("Nuovo client connesso\n");
    struct sockaddr_in address;
    socklen_t size = sizeof(struct sockaddr_in);
    if (getpeername(clientDesc, (struct sockaddr *)&address, &size) < 0)
    {
      perror("Impossibile ottenere l'indirizzo del client");
      exit(-1);
    }
    char clientAddr[20];
    strcpy(clientAddr, inet_ntoa(address.sin_addr));
    Args args = (Args)malloc(sizeof(struct argsToSend));
    args->userName = (char *)calloc(MAX_BUF, 1);
    strcpy(args->userName, clientAddr);
    args->flag = 2;
    pthread_t tid;
    pthread_create(&tid, NULL, fileWriter, (void *)args);

    puntClientDesc = (int *)malloc(sizeof(int));
```

```
34      *puntClientDesc = clientDesc;
35      pthread_create(&tid, NULL, gestisci, (void *)puntClientDesc);
36    }
37    close(clientDesc);
38    quitServer();
39  }
```

In particolare al rigo 35 notiamo la creazione di un nuovo thread per gestire la connessione in entrata a cui passiamo il descrittore del client di cui si deve occupare. Dal rigo 19 al rigo 31, estraiamo invece l'indirizzo ip del client per scriverlo sul file di log.

3.2 Configurazione del client

Il cliente invece viene configurato e si connette al server tramite la seguente funzione:

Listato 4: Configurazione e connessione del client

```
1  int connettiAlServer(char **argv) {
2    char *indirizzoServer;
3    uint16_t porta = strtoul(argv[2], NULL, 10);
4    indirizzoServer = ipResolver(argv);
5    struct sockaddr_in mio_indirizzo;
6    mio_indirizzo.sin_family = AF_INET;
7    mio_indirizzo.sin_port = htons(porta);
8    inet_aton(indirizzoServer, &mio_indirizzo.sin_addr);
9    if ((socketDesc = socket(PF_INET, SOCK_STREAM, 0)) < 0)
10     perror("Impossibile creare socket"), exit(-1);
11   else
12     printf("Socket creato\n");
13   if (connect(socketDesc, (struct sockaddr *)&mio_indirizzo,
14               sizeof(mio_indirizzo)) < 0)
15     perror("Impossibile connettersi"), exit(-1);
16   else
17     printf("Connesso a %s\n", indirizzoServer);
18   return socketDesc;
19 }
```

Si noti come al rigo 9 viene configurato il socket ed al rigo 13 viene invece effettuato il tentativo di connessione al server.
Al rigo 3 invece viene convertita la porta inserita in input (secondo argomento da riga di comando) dal tipo stringa al tipo corretto della porta (uint16_t, unsigned long int).
Al rigo 4 notiamo invece la risoluzione dell'url da parte della funzione ipResolver che è riportata di seguito:

Listato 5: Risoluzione url del client

```
1  char *ipResolver(char **argv) {
2    char *ipAddress;
3    struct hostent *hp;
4    hp = gethostbyname(argv[1]);
5    if (!hp) {
6      perror("Impossibile risolvere l'indirizzo ip\n");
7      sleep(1);
8      exit(-1);
9    }
10   printf("Address:\t%s\n", inet_ntoa(*(struct in_addr *)hp->h_addr_list[0]));
11   return inet_ntoa(*(struct in_addr *)hp->h_addr_list[0]);
12 }
```

Al rigo 4, tramite l'url o l'indirizzo ip viene riempita la struttura hostent da cui poi possiamo estrarre l'indirizzo ip presente nel campo h_addr_list che, in effetti, è un array che contiene i vari indirizzi ip associati a quell'host.
Infine, al rigo 11 ritorniamo il primo indirizzo convertito in Internet dot notation.

3.3 Comunicazione tra client e server

La comunicazione tra client e server avviene tramite write e read sul socket.
Il comportamento del server e del client è determinato da particolari messaggi inviati e/o ricevuti che codificano, tramite interi o caratteri, la richiesta da parte del client di usufruire di un determinato servizio e la relativa risposta del server.

3.3.1 Esempio: la prima comunicazione

In particolare, una volta effettuata la connessione, il server attenderà un messaggio dal client per poter avviare una delle tre possibili procedure, ovvero login, registrazione ed uscita (rispettivamente codici: 1,2,3).
Di seguito sono riportate le realtive funzioni di gestione che entrano in esecuzione subito dopo aver stabilito la connessione tra client e server.

Listato 6: Prima comunicazione del server

```
1  void *gestisci(void *descriptor)
2  {
3    int bufferReceive[2] = {1};
4    int client_sd = *(int *)descriptor;
5    int continua = 1;
6    char name[MAX_BUF];
7    while (continua)
8    {
9      read(client_sd, bufferReceive, sizeof(bufferReceive));
10     if (bufferReceive[0] == 2)
11       registraClient(client_sd);
12     else if (bufferReceive[0] == 1)
13       if (tryLogin(client_sd, name))
14       {
15         play(client_sd, name);
16         continua = 0;
17       }
18       else if (bufferReceive[0] == 3)
19         disconnettiClient(client_sd);
20       else
21       {
22         printf("Input invalido, uscita...\n");
23         disconnettiClient(client_sd);
24       }
25   }
26   pthread_exit(0);
27 }
```

Si noti come il server riceva, al rigo 9, il messaggio codificato da parte del client e metta in esecuzione la funzione corrispondente.

Listato 7: Prima comunicazione del client

```
1  int gestisci() {
2    char choice;
3    while (1) {
4      printMenu();
5      scanf("%c", &choice);
6      fflush(stdin);
7      system("clear");
8      if (choice == '3') {
9        esciDalServer();
10       return (0);
11     } else if (choice == '2') {
12       registrati();
13     } else if (choice == '1') {
14       if (tryLogin())
15         play();
16     } else
17       printf("Input errato, inserire 1,2 o 3\n");
18   }
19 }
```

4 Comunicazione durante la partita

Una volta effettuato il login, il client potrà iniziare a giocare tramite la funzione play (Vedi List. 8 e List. 9) che rappresentano il cuore della comunicazione tra client e server.

4.1 Funzione core del server

La funzione play del server consiste di un ciclo nel quale il server invia al client tre informazioni importanti:

- La griglia di gioco (Rigo 32)
- Il player con le relative informazioni (Righi 34 a 37)
- Un messaggio che notifica al client se è iniziato un nuovo turno oppure no (Righi 52,57,62,71)

Dopodichè il thread del server rimane in attesa di ricevere l'input del client per spostare il giocaore sulla mappa tramite la relativa funzione. (Rigo 33) (Vedi List. 17 Rigo 430 e List. 19 Rigo 296, 331,367, 405) Oltre questo, la funzione play del server si occupa anche di generare la posizione del player appena entra in partita, generare la nuova posizione (Righi 56 a 65) quando viene effettuato il cambio di mappa ed inviare il tempo rimanente o la lista degli utente loggati su richiesta del client.

È anche importante notare il seguente dettaglio implementativo: la griglia di gioco è una matrice globale definita nel file del server che contiene tutti i player, i punti di raccolta ed i pacchi, mentre gli ostacoli sono contenuti in una seconda matrice globale del server. Ogni client però deve vedere soltanto gli ostacoli che ha già scoperto, per questo motivo ad ogni client non viene mandata direttamente la matrice di gioco, bensì, dai righi 22 a 24, inizializziamo una nuova matrice temporanea a cui aggiungiamo gli ostacoli già scoperti dal client (rigo 24) prima di mandarla al client stesso.
In questo modo ci assicuriamo che ogni client visualizzi soltanto gli ostacoli che ha già scoperto.

Listato 8: Funzione play del server

```
1  void play(int clientDesc, char name[])
2  {
3    int true = 1;
4    int turnoFinito = 0;
5    int turnoGiocatore = turno;
6    int posizione[2];
7    int destinazione[2] = {-1, -1};
8    PlayerStats giocatore = initStats(destinazione, 0, posizione, 0);
9    Obstacles listaOstacoli = NULL;
10   char inputFromClient;
11   if (timer != 0)
12   {
13     inserisciPlayerNellaGrigliaInPosizioneCasuale(
14         grigliaDiGiocoConPacchiSenzaOstacoli, grigliaOstacoliSenzaPacchi,
15         giocatore->position);
16     pthread_mutex_lock(&PlayerGeneratiMutex);
17     playerGenerati++;
18     pthread_mutex_unlock(&PlayerGeneratiMutex);
19   }
20   while (true)
21   {
22     if (clientDisconnesso(clientDesc))
23     {
24       freeObstacles(listaOstacoli);
25       disconnettiClient(clientDesc);
26       return;
27     }
28     char grigliaTmp[ROWS][COLUMNS];
29     clonaGriglia(grigliaTmp, grigliaDiGiocoConPacchiSenzaOstacoli);
30     mergeGridAndList(grigliaTmp, listaOstacoli);
31     // invia la griglia
32     write(clientDesc, grigliaTmp, sizeof(grigliaTmp));
33     // invia la struttura del player
34     write(clientDesc, giocatore->deploy, sizeof(giocatore->deploy));
35     write(clientDesc, giocatore->position, sizeof(giocatore->position));
36     write(clientDesc, &giocatore->score, sizeof(giocatore->score));
37     write(clientDesc, &giocatore->hasApack, sizeof(giocatore->hasApack));
38     // legge l'input
39     if (read(clientDesc, &inputFromClient, sizeof(char)) > 0){
40       pthread_mutex_lock(&numMosseMutex);
41       numMosse++;
42       pthread_mutex_unlock(&numMosseMutex);
43     }
44     if (inputFromClient == 'e' || inputFromClient == 'E')
45     {
46       freeObstacles(listaOstacoli);
```

```
47        listaOstacoli = NULL;
48        disconnettiClient(clientDesc);
49      }
50      else if (inputFromClient == 't' || inputFromClient == 'T')
51      {
52        write(clientDesc, &turnoFinito, sizeof(int));
53        sendTimerValue(clientDesc);
54      }
55      else if (inputFromClient == 'l' || inputFromClient == 'L')
56      {
57        write(clientDesc, &turnoFinito, sizeof(int));
58        sendPlayerList(clientDesc);
59      }
60      else if (turnoGiocatore == turno)
61      {
62        write(clientDesc, &turnoFinito, sizeof(int));
63        giocatore =
64            gestisciInput(grigliaDiGiocoConPacchiSenzaOstacoli,
65                          grigliaOstacoliSenzaPacchi, inputFromClient, giocatore,
66                          &listaOstacoli, deployCoords, packsCoords, name);
67      }
68      else
69      {
70        turnoFinito = 1;
71        write(clientDesc, &turnoFinito, sizeof(int));
72        freeObstacles(listaOstacoli);
73        listaOstacoli = NULL;
74        inserisciPlayerNellaGrigliaInPosizioneCasuale(
75            grigliaDiGiocoConPacchiSenzaOstacoli, grigliaOstacoliSenzaPacchi,
76            giocatore->position);
77        giocatore->score = 0;
78        giocatore->hasApack = 0;
79        giocatore->deploy[0] = -1;
80        giocatore->deploy[1] = -1;
81        turnoGiocatore = turno;
82        turnoFinito = 0;
83        pthread_mutex_lock(&PlayerGeneratiMutex);
84        playerGenerati++;
85        pthread_mutex_unlock(&PlayerGeneratiMutex);
86      }
87    }
88  }
```

4.2 Funzione core del client

Dall'altro lato, la funzione play del client, è stata mantenuta il puù semplice possibile. Lo scopo del client è unicamente quello di ricevere i dati forniti dal server, stampare la mappa di gioco e ed inviare un input al server che rappresenta la volontà del giocatore di muoversi, vedere la lista degli utenti, uscire o stampare il timer. Unica eccezione è il rigo 30 del client che non richiede la ricezione di ulteriori dati dal server: al rigo 23, infatti si avvia la procedura di disconnessione del client (Vedi List. 16 rigo 59).

Listato 9: Funzione play del client

```
1  void play() {
2    PlayerStats giocatore = NULL;
3    int score, deploy[2], position[2], timer;
4    int turnoFinito = 0;
5    int exitFlag = 0, hasApack = 0;
6    while (!exitFlag) {
7      if (serverCaduto())
8        serverCrashHandler();
9      if (read(socketDesc, grigliaDiGioco, sizeof(grigliaDiGioco)) < 1)
10       printf("Impossibile comunicare con il server\n"), exit(-1);
11     if (read(socketDesc, deploy, sizeof(deploy)) < 1)
12       printf("Impossibile comunicare con il server\n"), exit(-1);
13     if (read(socketDesc, position, sizeof(position)) < 1)
14       printf("Impossibile comunicare con il server\n"), exit(-1);
15     if (read(socketDesc, &score, sizeof(score)) < 1)
16       printf("Impossibile comunicare con il server\n"), exit(-1);
17     if (read(socketDesc, &hasApack, sizeof(hasApack)) < 1)
18       printf("Impossibile comunicare con il server\n"), exit(-1);
19     giocatore = initStats(deploy, score, position, hasApack);
20     printGrid(grigliaDiGioco, giocatore);
21     char send = getUserInput();
22     if (send == 'e' || send == 'E') {
23       esciDalServer();
24       exit(0);
25     }
26     write(socketDesc, &send, sizeof(char));
27     read(socketDesc, &turnoFinito, sizeof(turnoFinito));
28     if (turnoFinito) {
29       system("clear");
30       printf("Turno finito\n");
31       sleep(1);
```

```
32      } else {
33        if (send == 't' || send == 'T')
34          printTimer();
35        else if (send == 'l' || send == 'L')
36          printPlayerList();
37      }
38    }
39  }
```

5 Dettagli implementativi degni di nota

In questa sezione verranno trattati alcuni dettagli implementativi da noi giudicati interessanti in realzione a ciò che è stato studiato durante il corso di sistemi operativi.

5.1 Timer

Lo svolgimento della partite è legato al timer: ogni round durerà un numero finito di secondi od oppure terminarà quando un client raccoglierà il numero massimo di pacchi.

Subito dopo aver configurato il socket, il server inizia la procedura di avvio del timer (Vedi List. 17 rigo 126 e 187) che farà partire un thread il quale si occuperà di decrementare e resettare correttamente il timer (definito come variabile globale del server).

Listato 10: Funzione di gestione del timer

```
1  void *timer(void *args)
2  {
3    int cambiato = 1;
4    while (1)
5    {
6      if (almenoUnClientConnesso() && valoreTimerValido() &&
7          almenoUnPlayerGenerato() && almenoUnaMossaFatta())
8      {
9        cambiato = 1;
10       sleep(1);
11       timerCount--;
12       fprintf(stdout, "Time left: %d\n", timerCount);
13     }
14     else if (numeroClientLoggati == 0)
15     {
16       timerCount = TIME_LIMIT_IN_SECONDS;
17       if (cambiato)
18       {
19         fprintf(stdout, "Time left: %d\n", timerCount);
20         cambiato = 0;
21       }
22     }
23     if (timerCount == 0 || scoreMassimo == packageLimitNumber)
24     {
25       pthread_mutex_lock(&PlayerGeneratiMutex);
26       playerGenerati = 0;
27       pthread_mutex_unlock(&PlayerGeneratiMutex);
28       pthread_mutex_lock(&numMosseMutex);
29       numMosse = 0;
30       pthread_mutex_unlock(&numMosseMutex);
31       printf("Reset timer e generazione nuova mappa..\n");
32       startProceduraGenrazioneMappa();
33       pthread_join(tidGeneratoreMappa, NULL);
34       turno++;
35       timerCount = TIME_LIMIT_IN_SECONDS;
36     }
37   }
38 }
```

Analizzando il codice della funzione di modifica del timer si può notare un dettaglio abbastanza interessante: il thread che esegue la funzione del timer è legato ad un altro thread, ovvero quello della generazione di una nuova mappa. Oltre ad un thread per gestire ogni client abbiamo quindi anche un altro thread che va a gestire il tempo , il quale attraverso un altro thread riesce a controllare la generazione della mappa e degli utenti allo scadere del tempo. Si noti anche come, tramite il pthread_join, il timer attenda la terminazione del secondo thread prima di resettare il timer e ricominciare il contro alla rovescia. [1]

Per completezza si riporta anche la funzionione iniziale del thread di generazione mappa

Listato 11: Generazione nuova mappa e posizione players

```
1  void *threadGenerazioneMappa(void *args)
2  {
3    fprintf(stdout, "Rigenerazione mappa\n");
4    inizializzaGrigliaVuota(grigliaDiGiocoConPacchiSenzaOstacoli);
5    generaPosizioniRaccolta(grigliaDiGiocoConPacchiSenzaOstacoli,
```

[1] Altro dettaglio meno importante, ma comunque degno di nota è il fatto che il timer non inizia il conto alla rovescia se non c'è almeno un giocatore loggato, se questo non è stato posizionato sulla mappa e se questo non ha effettuato la prima mossa. Al rigo 3 c'è anche da giustificare la variabile "cambiato" che non è nient'altro che un flag, il quale impedisce al server di stampare in stdout il valore del timer nel caso in cui esso sia stato appena resettato e non sia ancora iniziato il conto alla rovescia. Ciò evita che, prima che inizi il conto alla rovescia, il server continui a stampare il valore massimo del timer

```
                          grigliaOstacoliSenzaPacchi, deployCoords);
  riempiGrigliaConPacchiInPosizioniGenerateCasualmente(
      grigliaDiGiocoConPacchiSenzaOstacoli, packsCoords);
  generaPosizioneOstacoli(grigliaDiGiocoConPacchiSenzaOstacoli,
                          grigliaOstacoliSenzaPacchi);
  printf("Mappa generata\n");
  pthread_exit(NULL);
}
```

5.2 Gestione del file di Log

Una delle funzionalità del server è quella di creare un file di log con varie informazioni durante la sua esecuzione. Riteniamo l'implmentazione di questa funzione piuttosto interessante poichè, oltre ad essere una funzione gestita tramite un thread, fa uso sia di molte chiamate di sistema studiate durante il corso ed utilizza anche il mutex per risolvere eventuali race condition. Riportiamo di seguito il codice:

Listato 12: Funzione di log

```
void *fileWriter(void *args)
{
  int fDes = open("Log", O_RDWR | O_CREAT | O_APPEND, S_IWUSR | S_IRUSR);
  if (fDes < 0)
  {
    perror("Error while opening log file");
    exit(-1);
  }
  Args info = (Args)args;
  char dateAndTime[64];
  putCurrentDateAndTimeInString(dateAndTime);
  if (logDelPacco(info->flag))
  {
    char message[MAX_BUF] = "";
    prepareMessageForPackDelivery(message, info->userName, dateAndTime);
    pthread_mutex_lock(&LogMutex);
    write(fDes, message, strlen(message));
    pthread_mutex_unlock(&LogMutex);
  }
  else if (logDelLogin(info->flag))
  {
    char message[MAX_BUF] = "\"";
    prepareMessageForLogin(message, info->userName, dateAndTime);
    pthread_mutex_lock(&LogMutex);
    write(fDes, message, strlen(message));
    pthread_mutex_unlock(&LogMutex);
  }
  else if (logDellaConnessione(info->flag))
  {
    char message[MAX_BUF] = "\"";
    prepareMessageForConnection(message, info->userName, dateAndTime);
    pthread_mutex_lock(&LogMutex);
    write(fDes, message, strlen(message));
    pthread_mutex_unlock(&LogMutex);
  }
  close(fDes);
  free(info);
  pthread_exit(NULL);
}
```

Analizzando il codice si può notare l'uso open per aprire in append o, in caso di assenza del file, di creare il file di log ed i vari write per scrivere sul suddetto file; possiamo anche notare come la sezione critica, ovvero la scrittura su uno stesso file da parte di più thread, è gestita tramite un mutex.

5.3 Modifica della mappa di gioco da parte di più thread

La mappa di gioco è la stessa per tutti i player e c'è il rischio che lo spostamento dei player e/o la raccolta ed il deposito di pacchi possano provocare problemi a causa della race condition che si viene a creare tra i thread. Tutto ciò è stato risolto con una serie di semplici accorgimenti implementativi. Il primo accorgimento, e forse anche il più importante, è la funzione spostaPlayer mostrata qui di seguito.

Listato 13: Funzione spostaPlayer

```
void spostaPlayer(char griglia[ROWS][COLUMNS], int vecchiaPosizione[2],
                  int nuovaPosizione[2], Point deployCoords[],
                  Point packsCoords[])
{

  pthread_mutex_lock(&MatrixMutex);
```

```
  griglia[nuovaPosizione[0]][nuovaPosizione[1]] = 'P';
  if (eraUnPuntoDepo(vecchiaPosizione, deployCoords))
    griglia[vecchiaPosizione[0]][vecchiaPosizione[1]] = '_';
  else if (eraUnPacco(vecchiaPosizione, packsCoords))
    griglia[vecchiaPosizione[0]][vecchiaPosizione[1]] = '$';
  else
    griglia[vecchiaPosizione[0]][vecchiaPosizione[1]] = '-';
  pthread_mutex_unlock(&MatrixMutex);
}
```

Questa funzione rappresenta l'unico punto del programma che effettivamente modifica la matrice di gioco in seguito ad una richiesta di un client. È possibile notare come l'intera funzione sia racchiusa in un mutex in modo da evitare che contemporaneamente più thread modifichino la mappa di gioco e quindi evita che due player si trovino nella stessa poszione.

Il secondo accorgimento è stato quello di far in modo che un player possa raccogliere un pacco solo quando si trova nella poszione del pacco (”sia sovrapposto al pacco”) e possa depositare un pacco solo nella posizione in cui il player stesso si trova (”deposita il pacco su se stesso”).

Questi due accorgimenti, assieme, evitano qualsiasi tipo di conflitto tra i player: due player non potranno mai trovarsi nella stessa posizione e, di conseguenza non potranno mai raccogliere lo stesso pacco o depositare due pacchi nella stessa posizione contemporaneamente.

5.4 Gestione del login

La gestione del login è il quarto ed ultimo dettagli implementativo giusdicato abbastanza interessante poichè fa uso della system call system() per utilizzare le chiamate di sistema unix studiate durante la prima parte del corso. Di seguito riportiamo il codice e la spiegazione

Listato 14: "Gestion del login 1"

```
int isRegistered(char *name, char *file) {
  char command[MAX_BUF] = "cat  ";
  strcat(command, file);
  char toApp[] = " |cut -d\" \" -f1|grep \"^";
  strcat(command, toApp);
  strcat(command, name);
  char toApp2[] = "$\">tmp";
  strcat(command, toApp2);
  int ret = 0;
  system(command);
  int fileDes = openFileRDON("tmp");
  struct stat info;
  fstat(fileDes, &info);
  if ((int)info.st_size > 0)
    ret = 1;
  close(fileDes);
  system("rm tmp");
  return ret;
}
```

La funzione isRegistered tramite varie concatenazioni produce ed esegue il seguente comando

```
cat file | cut -d" " -f1|grep "^name$">tmp
```

Ovvero andiamo a leggere la prima colonna (dove sono conservati tutti i nomi utente) dal file degli utenti registrati, cerchiamo la stringa che combacia esattamente con name e la scriviamo sul file temporaneo ”tmp”.
Dopodichè andiamo a verificare la dimensione del file tmp tramite la struttura stat: se la dimensione è maggiore di 0 allora significa che è il nome esisteva nella lista dei client registrati ed è stato quindi trascritto in tmp altrimenti significa che il nome non era presente nella lista dei player registrati. A questo punto eliminiamo il file temporaneo e restituiamo il valore appropriato.

Listato 15: "Gestion del login 2"

```
int validateLogin(char *name, char *pwd, char *file) {
  if (!isRegistered(name, file))
    return 0;
  char command[MAX_BUF] = "cat  ";
  strcat(command, file);
  char toApp[] = " |grep \"^";
  strcat(command, toApp);
```

```
8     strcat(command, name);
9     strcat(command, " ");
10    strcat(command, pwd);
11    char toApp2[] = "$\">tmp";
12    strcat(command, toApp2);
13    int ret = 0;
14    system(command);
15    int fileDes = openFileRDON("tmp");
16    struct stat info;
17    fstat(fileDes, &info);
18    if ((int)info.st_size > 0)
19      ret = 1;
20    close(fileDes);
21    system("rm tmp");
22    return ret;
23  }
```

La funziona validateLogin invece, tramite concatenazioni successive crea ed esegue il seguente comando:

```
cat file | grep "^nome password$">tmp
```

Verificando se la coppia nome password sia presente nel file degli utenti registrati, trascrivendola sul file tmp Ancora una volta si va a verificare tramite la struttura stat se è stato trascritto qualcosa oppure no, ritornando il valore appropriato.

A Codici sorgente

Di seguito sono riportati tutti i codici sorgenti integrali del progetto.

A.1 Codice sorgente del client

Listato 16: Codice sorgente del client

```
#include "boardUtility.h"
#include "list.h"
#include "parser.h"
#include <arpa/inet.h>
#include <fcntl.h>
#include <netdb.h>
#include <netinet/in.h> //conversioni
#include <netinet/in.h>
#include <netinet/ip.h> //struttura
#include <pthread.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>

void printPlayerList();
int getTimer();
void printTimer();
void play();
int tryLogin();
void printMenu();
int connettiAlServer(char **argv);
char *ipResolver(char **argv);
int registrati();
int gestisci();
char getUserInput();
void clientCrashHandler();
void serverCrashHandler();
int serverCaduto();
void esciDalServer();
int isCorrect(char);

int socketDesc;
char grigliaDiGioco[ROWS][COLUMNS];

int main(int argc, char **argv) {
  signal(SIGINT, clientCrashHandler); /* CTRL-C */
  signal(SIGHUP, clientCrashHandler); /* Chiusura della console */
  signal(SIGQUIT, clientCrashHandler);
  signal(SIGTSTP, clientCrashHandler); /* CTRL-Z*/
  signal(SIGTERM, clientCrashHandler); /* generato da 'kill' */
  signal(SIGPIPE, serverCrashHandler);
  char bufferReceive[2];
  if (argc != 3) {
    perror("Inserire indirizzo ip/url e porta (./client 127.0.0.1 5200)");
    exit(-1);
  }
  if ((socketDesc = connettiAlServer(argv)) < 0)
    exit(-1);
  gestisci(socketDesc);
  close(socketDesc);
  exit(0);
}
void esciDalServer() {
  int msg = 3;
  printf("Uscita in corso\n");
  write(socketDesc, &msg, sizeof(int));
  close(socketDesc);
}
int connettiAlServer(char **argv) {
  char *indirizzoServer;
  uint16_t porta = strtoul(argv[2], NULL, 10);
  indirizzoServer = ipResolver(argv);
  struct sockaddr_in mio_indirizzo;
  mio_indirizzo.sin_family = AF_INET;
  mio_indirizzo.sin_port = htons(porta);
  inet_aton(indirizzoServer, &mio_indirizzo.sin_addr);
  if ((socketDesc = socket(PF_INET, SOCK_STREAM, 0)) < 0)
    perror("Impossibile creare socket"), exit(-1);
  else
    printf("Socket creato\n");
  if (connect(socketDesc, (struct sockaddr *)&mio_indirizzo,
```

```
78                sizeof(mio_indirizzo)) < 0)
79      perror("Impossibile connettersi"), exit(-1);
80    else
81      printf("Connesso a %s\n", indirizzoServer);
82    return socketDesc;
83  }
84  int gestisci() {
85    char choice;
86    while (1) {
87      printMenu();
88      scanf("%c", &choice);
89      fflush(stdin);
90      system("clear");
91      if (choice == '3') {
92        esciDalServer();
93        return (0);
94      } else if (choice == '2') {
95        registrati();
96      } else if (choice == '1') {
97        if (tryLogin())
98          play();
99      } else
100       printf("Input errato, inserire 1,2 o 3\n");
101   }
102 }
103 int serverCaduto() {
104   char msg = 'y';
105   if (read(socketDesc, &msg, sizeof(char)) == 0)
106     return 1;
107   else
108     write(socketDesc, &msg, sizeof(msg));
109   return 0;
110 }
111 void play() {
112   PlayerStats giocatore = NULL;
113   int score, deploy[2], position[2], timer;
114   int turnoFinito = 0;
115   int exitFlag = 0, hasApack = 0;
116   while (!exitFlag) {
117     if (serverCaduto())
118       serverCrashHandler();
119     if (read(socketDesc, grigliaDiGioco, sizeof(grigliaDiGioco)) < 1)
120       printf("Impossibile comunicare con il server\n"), exit(-1);
121     if (read(socketDesc, deploy, sizeof(deploy)) < 1)
122       printf("Impossibile comunicare con il server\n"), exit(-1);
123     if (read(socketDesc, position, sizeof(position)) < 1)
124       printf("Impossibile comunicare con il server\n"), exit(-1);
125     if (read(socketDesc, &score, sizeof(score)) < 1)
126       printf("Impossibile comunicare con il server\n"), exit(-1);
127     if (read(socketDesc, &hasApack, sizeof(hasApack)) < 1)
128       printf("Impossibile comunicare con il server\n"), exit(-1);
129     giocatore = initStats(deploy, score, position, hasApack);
130     printGrid(grigliaDiGioco, giocatore);
131     char send = getUserInput();
132     if (send == 'e' || send == 'E') {
133       esciDalServer();
134       exit(0);
135     }
136     write(socketDesc, &send, sizeof(char));
137     read(socketDesc, &turnoFinito, sizeof(turnoFinito));
138     if (turnoFinito) {
139       system("clear");
140       printf("Turno finito\n");
141       sleep(1);
142     } else {
143       if (send == 't' || send == 'T')
144         printTimer();
145       else if (send == 'l' || send == 'L')
146         printPlayerList();
147     }
148   }
149 }
150 void printPlayerList() {
151   system("clear");
152   int lunghezza = 0;
153   char buffer[100];
154   int continua = 1;
155   int number = 1;
156   fprintf(stdout, "Lista dei player: \n");
157   if (!serverCaduto(socketDesc)) {
158     read(socketDesc, &continua, sizeof(continua));
159     while (continua) {
160       read(socketDesc, &lunghezza, sizeof(lunghezza));
161       read(socketDesc, buffer, lunghezza);
162       buffer[lunghezza] = '\0';
163       fprintf(stdout, "%d) %s\n", number, buffer);
164       continua--;
165       number++;
166     }
```

```
167     sleep(1);
168   }
169 }
170 void printTimer() {
171   int timer;
172   if (!serverCaduto(socketDesc)) {
173     read(socketDesc, &timer, sizeof(timer));
174     printf("\t\tTempo restante: %d...\n", timer);
175     sleep(1);
176   }
177 }
178 int getTimer() {
179   int timer;
180   if (!serverCaduto(socketDesc))
181     read(socketDesc, &timer, sizeof(timer));
182   return timer;
183 }
184 int tryLogin() {
185   int msg = 1;
186   write(socketDesc, &msg, sizeof(int));
187   system("clear");
188   printf("Inserisci i dati per il Login\n");
189   char username[20];
190   char password[20];
191   printf("Inserisci nome utente(MAX 20 caratteri): ");
192   scanf("%s", username);
193   printf("\nInserisci password(MAX 20 caratteri):");
194   scanf("%s", password);
195   int dimUname = strlen(username), dimPwd = strlen(password);
196   if (write(socketDesc, &dimUname, sizeof(dimUname)) < 0)
197     return 0;
198   if (write(socketDesc, &dimPwd, sizeof(dimPwd)) < 0)
199     return 0;
200   if (write(socketDesc, username, dimUname) < 0)
201     return 0;
202   if (write(socketDesc, password, dimPwd) < 0)
203     return 0;
204   char validate;
205   int ret;
206   read(socketDesc, &validate, 1);
207   if (validate == 'y') {
208     ret = 1;
209     printf("Accesso effettuato\n");
210   } else if (validate == 'n') {
211     printf("Credenziali Errate o Login giá effettuato\n");
212     ret = 0;
213   }
214   sleep(1);
215   return ret;
216 }
217 int registrati() {
218   int msg = 2;
219   write(socketDesc, &msg, sizeof(int));
220   char username[20];
221   char password[20];
222   system("clear");
223   printf("Inserisci nome utente(MAX 20 caratteri): ");
224   scanf("%s", username);
225   printf("\nInserisci password(MAX 20 caratteri):");
226   scanf("%s", password);
227   int dimUname = strlen(username), dimPwd = strlen(password);
228   if (write(socketDesc, &dimUname, sizeof(dimUname)) < 0)
229     return 0;
230   if (write(socketDesc, &dimPwd, sizeof(dimPwd)) < 0)
231     return 0;
232   if (write(socketDesc, username, dimUname) < 0)
233     return 0;
234   if (write(socketDesc, password, dimPwd) < 0)
235     return 0;
236   char validate;
237   int ret;
238   read(socketDesc, &validate, sizeof(char));
239   if (validate == 'y') {
240     ret = 1;
241     printf("Registrato con successo\n");
242   }
243   if (validate == 'n') {
244     ret = 0;
245     printf("Registrazione fallita\n");
246   }
247   sleep(1);
248   return ret;
249 }
250 char *ipResolver(char **argv) {
251   char *ipAddress;
252   struct hostent *hp;
253   hp = gethostbyname(argv[1]);
254   if (!hp) {
255     perror("Impossibile risolvere l'indirizzo ip\n");
```

```
256     sleep(1);
257     exit(-1);
258   }
259   printf("Address:\t%s\n", inet_ntoa(*(struct in_addr *)hp->h_addr_list[0]));
260   return inet_ntoa(*(struct in_addr *)hp->h_addr_list[0]);
261 }
262 void clientCrashHandler() {
263   int msg = 3;
264   int rec = 0;
265   printf("\nChiusura client...\n");
266   do {
267     write(socketDesc, &msg, sizeof(int));
268     read(socketDesc, &rec, sizeof(int));
269   } while (rec == 0);
270   close(socketDesc);
271   signal(SIGINT, SIG_IGN);
272   signal(SIGQUIT, SIG_IGN);
273   signal(SIGTERM, SIG_IGN);
274   signal(SIGTSTP, SIG_IGN);
275   exit(0);
276 }
277 void serverCrashHandler() {
278   system("clear");
279   printf("Il server á stato spento o á irraggiungibile\n");
280   close(socketDesc);
281   signal(SIGPIPE, SIG_IGN);
282   premiEnterPerContinuare();
283   exit(0);
284 }
285 char getUserInput() {
286   char c;
287   c = getchar();
288   int daIgnorare;
289   while ((daIgnorare = getchar()) != '\n' && daIgnorare != EOF) {
290   }
291   return c;
292 }
```

A.2 Codice sorgente del server

Listato 17: Codice sorgente del server

```
1  #include "boardUtility.h"
2  #include "list.h"
3  #include "parser.h"
4  #include <arpa/inet.h>
5  #include <errno.h>
6  #include <fcntl.h>
7  #include <netinet/in.h> //conversioni
8  #include <netinet/ip.h> //struttura
9  #include <pthread.h>
10 #include <signal.h>
11 #include <stdio.h>
12 #include <stdlib.h>
13 #include <string.h>
14 #include <sys/socket.h>
15 #include <sys/stat.h>
16 #include <sys/types.h>
17 #include <time.h>
18 #include <unistd.h>
19
20 //struttura di argomenti da mandare al thread che scrive sul file di log
21 struct argsToSend
22 {
23   char *userName;
24   int flag;
25 };
26
27 typedef struct argsToSend *Args;
28 void prepareMessageForLogin(char message[], char username[], char date[]);
29 void sendPlayerList(int clientDesc);
30 PlayerStats gestisciC(char grigliaDiGioco[ROWS][COLUMNS], PlayerStats giocatore,
31                       Point deployCoords[], Point packsCoords[], char name[]);
32 PlayerStats gestisciInput(char grigliaDiGioco[ROWS][COLUMNS],
33                           char grigliaOstacoli[ROWS][COLUMNS], char input,
34                           PlayerStats giocatore, Obstacles *listaOstacoli,
35                           Point deployCoords[], Point packsCoords[],
36                           char name[]);
37 void clonaGriglia(char destinazione[ROWS][COLUMNS], char source[ROWS][COLUMNS]);
38 int almenoUnClientConnesso();
39 void prepareMessageForConnection(char message[], char ipAddress[], char date[]);
40 void spostaPlayer(char griglia[ROWS][COLUMNS], int vecchiaPosizione[2],
41                   int nuovaPosizione[2], Point deployCoords[],
42                   Point packsCoords[]);
43 int valoreTimerValido();
```

```
44  PlayerStats gestisciW(char grigliaDiGioco[ROWS][COLUMNS],
45                        char grigliaOstacoli[ROWS][COLUMNS],
46                        PlayerStats giocatore, Obstacles *listaOstacoli,
47                        Point deployCoords[], Point packsCoords[]);
48  PlayerStats gestisciA(char grigliaDiGioco[ROWS][COLUMNS],
49                        char grigliaOstacoli[ROWS][COLUMNS],
50                        PlayerStats giocatore, Obstacles *listaOstacoli,
51                        Point deployCoords[], Point packsCoords[]);
52  PlayerStats gestisciD(char grigliaDiGioco[ROWS][COLUMNS],
53                        char grigliaOstacoli[ROWS][COLUMNS],
54                        PlayerStats giocatore, Obstacles *listaOstacoli,
55                        Point deployCoords[], Point packsCoords[]);
56  PlayerStats gestisciS(char grigliaDiGioco[ROWS][COLUMNS],
57                        char grigliaOstacoli[ROWS][COLUMNS],
58                        PlayerStats giocatore, Obstacles *listaOstacoli,
59                        Point deployCoords[], Point packsCoords[]);
60  int almenoUnPlayerGenerato();
61  int almenoUnaMossaFatta();
62  void sendTimerValue(int clientDesc);
63  void putCurrentDateAndTimeInString(char dateAndTime[]);
64  void startProceduraGenrazioneMappa();
65  void *threadGenerazioneMappa(void *args);
66  void *fileWriter(void *);
67  int tryLogin(int clientDesc, char name[]);
68  void disconnettiClient(int);
69  int registraClient(int);
70  void *timer(void *args);
71  void *gestisci(void *descriptor);
72  void quitServer();
73  void clientCrashHandler(int signalNum);
74  void startTimer();
75  void configuraSocket(struct sockaddr_in mio_indirizzo);
76  struct sockaddr_in configuraIndirizzo();
77  void startListening();
78  int clientDisconnesso(int clientSocket);
79  void play(int clientDesc, char name[]);
80  void prepareMessageForPackDelivery(char message[], char username[], char date[]);
81  int logDelPacco(int flag);
82  int logDelLogin(int flag);
83  int logDellaConnessione(int flag);
84
85  char grigliaDiGiocoConPacchiSenzaOstacoli[ROWS][COLUMNS]; //protetta
86  char grigliaOstacoliSenzaPacchi[ROWS][COLUMNS];           //protetta
87  int numeroClientLoggati = 0;                              //protetto
88  int playerGenerati = 0;                                   //mutex
89  int timerCount = TIME_LIMIT_IN_SECONDS;
90  int turno = 0; //lo cambia solo timer
91  pthread_t tidTimer;
92  pthread_t tidGeneratoreMappa;
93  int socketDesc;
94  Players onLineUsers = NULL; //protetto
95  char *users;
96  int scoreMassimo = 0; //mutex
97  int numMosse = 0;     //mutex
98  Point deployCoords[numberOfPackages];
99  Point packsCoords[numberOfPackages];
100 pthread_mutex_t LogMutex = PTHREAD_MUTEX_INITIALIZER;
101 pthread_mutex_t RegMutex = PTHREAD_MUTEX_INITIALIZER;
102 pthread_mutex_t PlayerMutex = PTHREAD_MUTEX_INITIALIZER;
103 pthread_mutex_t MatrixMutex = PTHREAD_MUTEX_INITIALIZER;
104 pthread_mutex_t PlayerGeneratiMutex = PTHREAD_MUTEX_INITIALIZER;
105 pthread_mutex_t ScoreMassimoMutex = PTHREAD_MUTEX_INITIALIZER;
106 pthread_mutex_t numMosseMutex = PTHREAD_MUTEX_INITIALIZER;
107
108 int main(int argc, char **argv)
109 {
110   if (argc != 2)
111   {
112     printf("Wrong parameters number(Usage: ./server usersFile)\n");
113     exit(-1);
114   }
115   else if (strcmp(argv[1], "Log") == 0)
116   {
117     printf("Cannot use the Log file as a UserList \n");
118     exit(-1);
119   }
120   users = argv[1];
121   struct sockaddr_in mio_indirizzo = configuraIndirizzo();
122   configuraSocket(mio_indirizzo);
123   signal(SIGPIPE, clientCrashHandler);
124   signal(SIGINT, quitServer);
125   signal(SIGHUP, quitServer);
126   startTimer();
127   inizializzaGiocoSenzaPlayer(grigliaDiGiocoConPacchiSenzaOstacoli,
128                               grigliaOstacoliSenzaPacchi, packsCoords);
129   generaPosizioniRaccolta(grigliaDiGiocoConPacchiSenzaOstacoli,
130                           grigliaOstacoliSenzaPacchi, deployCoords);
131   startListening();
132   return 0;
```

```
133 }
134 void startListening()
135 {
136   pthread_t tid;
137   int clientDesc;
138   int *puntClientDesc;
139   while (1 == 1)
140   {
141     if (listen(socketDesc, 10) < 0)
142       perror("Impossibile mettersi in ascolto"), exit(-1);
143     printf("In ascolto..\n");
144     if ((clientDesc = accept(socketDesc, NULL, NULL)) < 0)
145     {
146       perror("Impossibile effettuare connessione\n");
147       exit(-1);
148     }
149     printf("Nuovo client connesso\n");
150     struct sockaddr_in address;
151     socklen_t size = sizeof(struct sockaddr_in);
152     if (getpeername(clientDesc, (struct sockaddr *)&address, &size) < 0)
153     {
154       perror("Impossibile ottenere l'indirizzo del client");
155       exit(-1);
156     }
157     char clientAddr[20];
158     strcpy(clientAddr, inet_ntoa(address.sin_addr));
159     Args args = (Args)malloc(sizeof(struct argsToSend));
160     args->userName = (char *)calloc(MAX_BUF, 1);
161     strcpy(args->userName, clientAddr);
162     args->flag = 2;
163     pthread_t tid;
164     pthread_create(&tid, NULL, fileWriter, (void *)args);
165
166     puntClientDesc = (int *)malloc(sizeof(int));
167     *puntClientDesc = clientDesc;
168     pthread_create(&tid, NULL, gestisci, (void *)puntClientDesc);
169   }
170   close(clientDesc);
171   quitServer();
172 }
173 struct sockaddr_in configuraIndirizzo()
174 {
175   struct sockaddr_in mio_indirizzo;
176   mio_indirizzo.sin_family = AF_INET;
177   mio_indirizzo.sin_port = htons(5200);
178   mio_indirizzo.sin_addr.s_addr = htonl(INADDR_ANY);
179   printf("Indirizzo socket configurato\n");
180   return mio_indirizzo;
181 }
182 void startProceduraGenrazioneMappa()
183 {
184   printf("Inizio procedura generazione mappa\n");
185   pthread_create(&tidGeneratoreMappa, NULL, threadGenerazioneMappa, NULL);
186 }
187 void startTimer()
188 {
189   printf("Thread timer avviato\n");
190   pthread_create(&tidTimer, NULL, timer, NULL);
191 }
192 int tryLogin(int clientDesc, char name[])
193 {
194   char *userName = (char *)calloc(MAX_BUF, 1);
195   char *password = (char *)calloc(MAX_BUF, 1);
196   int dimName, dimPwd;
197   read(clientDesc, &dimName, sizeof(int));
198   read(clientDesc, &dimPwd, sizeof(int));
199   read(clientDesc, userName, dimName);
200   read(clientDesc, password, dimPwd);
201   int ret = 0;
202   pthread_mutex_lock(&PlayerMutex);
203   if (validateLogin(userName, password, users) &&
204       !isAlreadyLogged(onLineUsers, userName))
205   {
206     ret = 1;
207     write(clientDesc, "y", 1);
208     strcpy(name, userName);
209     Args args = (Args)malloc(sizeof(struct argsToSend));
210     args->userName = (char *)calloc(MAX_BUF, 1);
211     strcpy(args->userName, name);
212     args->flag = 0;
213     pthread_t tid;
214     pthread_create(&tid, NULL, fileWriter, (void *)args);
215     printf("Nuovo client loggato, client loggati : %d\n", numeroClientLoggati);
216     onLineUsers = addPlayer(onLineUsers, userName, clientDesc);
217     numeroClientLoggati++;
218     pthread_mutex_unlock(&PlayerMutex);
219     printPlayers(onLineUsers);
220     printf("\n");
221   }
```

```
222   else
223   {
224     write(clientDesc, "n", 1);
225   }
226   return ret;
227 }
228 void *gestisci(void *descriptor)
229 {
230   int bufferReceive[2] = {1};
231   int client_sd = *(int *)descriptor;
232   int continua = 1;
233   char name[MAX_BUF];
234   while (continua)
235   {
236     read(client_sd, bufferReceive, sizeof(bufferReceive));
237     if (bufferReceive[0] == 2)
238       registraClient(client_sd);
239     else if (bufferReceive[0] == 1)
240       if (tryLogin(client_sd, name))
241       {
242         play(client_sd, name);
243         continua = 0;
244       }
245       else if (bufferReceive[0] == 3)
246         disconnettiClient(client_sd);
247       else
248       {
249         printf("Input invalido, uscita...\n");
250         disconnettiClient(client_sd);
251       }
252   }
253   pthread_exit(0);
254 }
255 void play(int clientDesc, char name[])
256 {
257   int true = 1;
258   int turnoFinito = 0;
259   int turnoGiocatore = turno;
260   int posizione[2];
261   int destinazione[2] = {-1, -1};
262   PlayerStats giocatore = initStats(destinazione, 0, posizione, 0);
263   Obstacles listaOstacoli = NULL;
264   char inputFromClient;
265   if (timer != 0)
266   {
267     inserisciPlayerNellaGrigliaInPosizioneCasuale(
268         grigliaDiGiocoConPacchiSenzaOstacoli, grigliaOstacoliSenzaPacchi,
269         giocatore->position);
270     pthread_mutex_lock(&PlayerGeneratiMutex);
271     playerGenerati++;
272     pthread_mutex_unlock(&PlayerGeneratiMutex);
273   }
274   while (true)
275   {
276     if (clientDisconnesso(clientDesc))
277     {
278       freeObstacles(listaOstacoli);
279       disconnettiClient(clientDesc);
280       return;
281     }
282     char grigliaTmp[ROWS][COLUMNS];
283     clonaGriglia(grigliaTmp, grigliaDiGiocoConPacchiSenzaOstacoli);
284     mergeGridAndList(grigliaTmp, listaOstacoli);
285     // invia la griglia
286     write(clientDesc, grigliaTmp, sizeof(grigliaTmp));
287     // invia la struttura del player
288     write(clientDesc, giocatore->deploy, sizeof(giocatore->deploy));
289     write(clientDesc, giocatore->position, sizeof(giocatore->position));
290     write(clientDesc, &giocatore->score, sizeof(giocatore->score));
291     write(clientDesc, &giocatore->hasApack, sizeof(giocatore->hasApack));
292     // legge l'input
293     if (read(clientDesc, &inputFromClient, sizeof(char)) > 0){
294       pthread_mutex_lock(&numMosseMutex);
295       numMosse++;
296       pthread_mutex_unlock(&numMosseMutex);
297     }
298     if (inputFromClient == 'e' || inputFromClient == 'E')
299     {
300       freeObstacles(listaOstacoli);
301       listaOstacoli = NULL;
302       disconnettiClient(clientDesc);
303     }
304     else if (inputFromClient == 't' || inputFromClient == 'T')
305     {
306       write(clientDesc, &turnoFinito, sizeof(int));
307       sendTimerValue(clientDesc);
308     }
309     else if (inputFromClient == 'l' || inputFromClient == 'L')
310     {
```

```
311       write(clientDesc, &turnoFinito, sizeof(int));
312       sendPlayerList(clientDesc);
313     }
314     else if (turnoGiocatore == turno)
315     {
316       write(clientDesc, &turnoFinito, sizeof(int));
317       giocatore =
318           gestisciInput(grigliaDiGiocoConPacchiSenzaOstacoli,
319                         grigliaOstacoliSenzaPacchi, inputFromClient, giocatore,
320                         &listaOstacoli, deployCoords, packsCoords, name);
321     }
322     else
323     {
324       turnoFinito = 1;
325       write(clientDesc, &turnoFinito, sizeof(int));
326       freeObstacles(listaOstacoli);
327       listaOstacoli = NULL;
328       inserisciPlayerNellaGrigliaInPosizioneCasuale(
329           grigliaDiGiocoConPacchiSenzaOstacoli, grigliaOstacoliSenzaPacchi,
330           giocatore->position);
331       giocatore->score = 0;
332       giocatore->hasApack = 0;
333       giocatore->deploy[0] = -1;
334       giocatore->deploy[1] = -1;
335       turnoGiocatore = turno;
336       turnoFinito = 0;
337       pthread_mutex_lock(&PlayerGeneratiMutex);
338       playerGenerati++;
339       pthread_mutex_unlock(&PlayerGeneratiMutex);
340     }
341   }
342 }
343 void sendTimerValue(int clientDesc)
344 {
345   if (!clientDisconnesso(clientDesc))
346     write(clientDesc, &timerCount, sizeof(timerCount));
347 }
348 void clonaGriglia(char destinazione[ROWS][COLUMNS],
349                   char source[ROWS][COLUMNS])
350 {
351   int i = 0, j = 0;
352   for (i = 0; i < ROWS; i++)
353   {
354     for (j = 0; j < COLUMNS; j++)
355     {
356       destinazione[i][j] = source[i][j];
357     }
358   }
359 }
360 void clientCrashHandler(int signalNum)
361 {
362   char msg[0];
363   int socketClientCrashato;
364   int flag = 1;
365   // TODO eliminare la lista degli ostacoli dell'utente
366   if (onLineUsers != NULL)
367   {
368     Players prec = onLineUsers;
369     Players top = prec->next;
370     while (top != NULL && flag)
371     {
372       if (write(top->sockDes, msg, sizeof(msg)) < 0)
373       {
374         socketClientCrashato = top->sockDes;
375         printPlayers(onLineUsers);
376         disconnettiClient(socketClientCrashato);
377         flag = 0;
378       }
379       top = top->next;
380     }
381   }
382   signal(SIGPIPE, SIG_IGN);
383 }
384 void disconnettiClient(int clientDescriptor)
385 {
386   pthread_mutex_lock(&PlayerMutex);
387   if (numeroClientLoggati > 0)
388     numeroClientLoggati--;
389   onLineUsers = removePlayer(onLineUsers, clientDescriptor);
390   pthread_mutex_unlock(&PlayerMutex);
391   printPlayers(onLineUsers);
392   int msg = 1;
393   printf("Client disconnesso (client attualmente loggati: %d)\n",
394          numeroClientLoggati);
395   write(clientDescriptor, &msg, sizeof(msg));
396   close(clientDescriptor);
397 }
398 int clientDisconnesso(int clientSocket)
399 {
```

```
400   char msg[1] = {'u'}; // UP?
401   if (write(clientSocket, msg, sizeof(msg)) < 0)
402     return 1;
403   if (read(clientSocket, msg, sizeof(char)) < 0)
404     return 1;
405   else
406     return 0;
407 }
408 int registraClient(int clientDesc)
409 {
410   char *userName = (char *)calloc(MAX_BUF, 1);
411   char *password = (char *)calloc(MAX_BUF, 1);
412   int dimName, dimPwd;
413   read(clientDesc, &dimName, sizeof(int));
414   read(clientDesc, &dimPwd, sizeof(int));
415   read(clientDesc, userName, dimName);
416   read(clientDesc, password, dimPwd);
417   pthread_mutex_lock(&RegMutex);
418   int ret = appendPlayer(userName, password, users);
419   pthread_mutex_unlock(&RegMutex);
420   char risposta;
421   if (!ret)
422   {
423     risposta = 'n';
424     write(clientDesc, &risposta, sizeof(char));
425     printf("Impossibile registrare utente, riprovare\n");
426   }
427   else
428   {
429     risposta = 'y';
430     write(clientDesc, &risposta, sizeof(char));
431     printf("Utente registrato con successo\n");
432   }
433   return ret;
434 }
435 void quitServer()
436 {
437   printf("Chiusura server in corso..\n");
438   close(socketDesc);
439   exit(-1);
440 }
441 void *threadGenerazioneMappa(void *args)
442 {
443   fprintf(stdout, "Rigenerazione mappa\n");
444   inizializzaGrigliaVuota(grigliaDiGiocoConPacchiSenzaOstacoli);
445   generaPosizioniRaccolta(grigliaDiGiocoConPacchiSenzaOstacoli,
446                           grigliaOstacoliSenzaPacchi, deployCoords);
447   riempiGrigliaConPacchiInPosizioniGenerateCasualmente(
448       grigliaDiGiocoConPacchiSenzaOstacoli, packsCoords);
449   generaPosizioneOstacoli(grigliaDiGiocoConPacchiSenzaOstacoli,
450                           grigliaOstacoliSenzaPacchi);
451   printf("Mappa generata\n");
452   pthread_exit(NULL);
453 }
454 int almenoUnaMossaFatta()
455 {
456   if (numMosse > 0)
457     return 1;
458   return 0;
459 }
460 int almenoUnClientConnesso()
461 {
462   if (numeroClientLoggati > 0)
463     return 1;
464   return 0;
465 }
466 int valoreTimerValido()
467 {
468   if (timerCount > 0 && timerCount <= TIME_LIMIT_IN_SECONDS)
469     return 1;
470   return 0;
471 }
472 int almenoUnPlayerGenerato()
473 {
474   if (playerGenerati > 0)
475     return 1;
476   return 0;
477 }
478 void *timer(void *args)
479 {
480   int cambiato = 1;
481   while (1)
482   {
483     if (almenoUnClientConnesso() && valoreTimerValido() &&
484         almenoUnPlayerGenerato() && almenoUnaMossaFatta())
485     {
486       cambiato = 1;
487       sleep(1);
488       timerCount--;
```

```
489       fprintf(stdout, "Time left: %d\n", timerCount);
490     }
491     else if (numeroClientLoggati == 0)
492     {
493       timerCount = TIME_LIMIT_IN_SECONDS;
494       if (cambiato)
495       {
496         fprintf(stdout, "Time left: %d\n", timerCount);
497         cambiato = 0;
498       }
499     }
500     if (timerCount == 0 || scoreMassimo == packageLimitNumber)
501     {
502       pthread_mutex_lock(&PlayerGeneratiMutex);
503       playerGenerati = 0;
504       pthread_mutex_unlock(&PlayerGeneratiMutex);
505       pthread_mutex_lock(&numMosseMutex);
506       numMosse = 0;
507       pthread_mutex_unlock(&numMosseMutex);
508       printf("Reset timer e generazione nuova mappa..\n");
509       startProceduraGenrazioneMappa();
510       pthread_join(tidGeneratoreMappa, NULL);
511       turno++;
512       timerCount = TIME_LIMIT_IN_SECONDS;
513     }
514   }
515 }
516
517 void configuraSocket(struct sockaddr_in mio_indirizzo)
518 {
519   if ((socketDesc = socket(PF_INET, SOCK_STREAM, 0)) < 0)
520   {
521     perror("Impossibile creare socket");
522     exit(-1);
523   }
524   if (setsockopt(socketDesc, SOL_SOCKET, SO_REUSEADDR, &(int){1}, sizeof(int)) <
525       0)
526     perror("Impossibile impostare il riutilizzo dell'indirizzo ip e della "
527            "porta\n");
528   if ((bind(socketDesc, (struct sockaddr *)&mio_indirizzo,
529             sizeof(mio_indirizzo))) < 0)
530   {
531     perror("Impossibile effettuare bind");
532     exit(-1);
533   }
534 }
535
536 PlayerStats gestisciInput(char grigliaDiGioco[ROWS][COLUMNS],
537                           char grigliaOstacoli[ROWS][COLUMNS], char input,
538                           PlayerStats giocatore, Obstacles *listaOstacoli,
539                           Point deployCoords[], Point packsCoords[],
540                           char name[])
541 {
542   if (giocatore == NULL)
543   {
544     return NULL;
545   }
546   if (input == 'w' || input == 'W')
547   {
548     giocatore = gestisciW(grigliaDiGioco, grigliaOstacoli, giocatore,
549                           listaOstacoli, deployCoords, packsCoords);
550   }
551   else if (input == 's' || input == 'S')
552   {
553     giocatore = gestisciS(grigliaDiGioco, grigliaOstacoli, giocatore,
554                           listaOstacoli, deployCoords, packsCoords);
555   }
556   else if (input == 'a' || input == 'A')
557   {
558     giocatore = gestisciA(grigliaDiGioco, grigliaOstacoli, giocatore,
559                           listaOstacoli, deployCoords, packsCoords);
560   }
561   else if (input == 'd' || input == 'D')
562   {
563     giocatore = gestisciD(grigliaDiGioco, grigliaOstacoli, giocatore,
564                           listaOstacoli, deployCoords, packsCoords);
565   }
566   else if (input == 'p' || input == 'P')
567   {
568     giocatore = gestisciP(grigliaDiGioco, giocatore, deployCoords, packsCoords);
569   }
570   else if (input == 'c' || input == 'C')
571   {
572     giocatore =
573         gestisciC(grigliaDiGioco, giocatore, deployCoords, packsCoords, name);
574   }
575
576   // aggiorna la posizione dell'utente
577   return giocatore;
```

```
578 }
579
580 PlayerStats gestisciC(char grigliaDiGioco[ROWS][COLUMNS], PlayerStats giocatore,
581                       Point deployCoords[], Point packsCoords[], char name[])
582 {
583   pthread_t tid;
584   if (giocatore->hasApack == 0)
585   {
586     return giocatore;
587   }
588   else
589   {
590     if (isOnCorrectDeployPoint(giocatore, deployCoords))
591     {
592       Args args = (Args)malloc(sizeof(struct argsToSend));
593       args->userName = (char *)calloc(MAX_BUF, 1);
594       strcpy(args->userName, name);
595       args->flag = 1;
596       pthread_create(&tid, NULL, fileWriter, (void *)args);
597       giocatore->score += 10;
598       if (giocatore->score > scoreMassimo){
599         pthread_mutex_lock(&ScoreMassimoMutex);
600         scoreMassimo = giocatore->score;
601         pthread_mutex_unlock(&ScoreMassimoMutex);
602       }
603       giocatore->deploy[0] = -1;
604       giocatore->deploy[1] = -1;
605       giocatore->hasApack = 0;
606     }
607     else
608     {
609       if (!isOnAPack(giocatore, packsCoords) &&
610           !isOnADeployPoint(giocatore, deployCoords))
611       {
612         int index = getHiddenPack(packsCoords);
613         if (index >= 0)
614         {
615           packsCoords[index]->x = giocatore->position[0];
616           packsCoords[index]->y = giocatore->position[1];
617           giocatore->hasApack = 0;
618           giocatore->deploy[0] = -1;
619           giocatore->deploy[1] = -1;
620         }
621       }
622       else
623         return giocatore;
624     }
625   }
626   return giocatore;
627 }
628
629 void sendPlayerList(int clientDesc)
630 {
631   int lunghezza = 0;
632   char name[100];
633   Players tmp = onLineUsers;
634   int numeroClientLoggati = dimensioneLista(tmp);
635   printf("%d ", numeroClientLoggati);
636   if (!clientDisconnesso(clientDesc))
637   {
638     write(clientDesc, &numeroClientLoggati, sizeof(numeroClientLoggati));
639     while (numeroClientLoggati > 0 && tmp != NULL)
640     {
641       strcpy(name, tmp->name);
642       lunghezza = strlen(tmp->name);
643       write(clientDesc, &lunghezza, sizeof(lunghezza));
644       write(clientDesc, name, lunghezza);
645       tmp = tmp->next;
646       numeroClientLoggati--;
647     }
648   }
649 }
650
651 void prepareMessageForPackDelivery(char message[], char username[], char date[])
652 {
653   strcat(message, "Pack delivered by \"");
654   strcat(message, username);
655   strcat(message, "\" at ");
656   strcat(message, date);
657   strcat(message, "\n");
658 }
659
660 void prepareMessageForLogin(char message[], char username[], char date[])
661 {
662   strcat(message, username);
663   strcat(message, "\" logged in at ");
664   strcat(message, date);
665   strcat(message, "\n");
666 }
```

```
667
668 void prepareMessageForConnection(char message[], char ipAddress[], char date[])
669 {
670   strcat(message, ipAddress);
671   strcat(message, "\" connected at ");
672   strcat(message, date);
673   strcat(message, "\n");
674 }
675
676 void putCurrentDateAndTimeInString(char dateAndTime[])
677 {
678   time_t t = time(NULL);
679   struct tm *infoTime = localtime(&t);
680   strftime(dateAndTime, 64, "%X %x", infoTime);
681 }
682
683 void *fileWriter(void *args)
684 {
685   int fDes = open("Log", O_RDWR | O_CREAT | O_APPEND, S_IWUSR | S_IRUSR);
686   if (fDes < 0)
687   {
688     perror("Error while opening log file");
689     exit(-1);
690   }
691   Args info = (Args)args;
692   char dateAndTime[64];
693   putCurrentDateAndTimeInString(dateAndTime);
694   if (logDelPacco(info->flag))
695   {
696     char message[MAX_BUF] = "";
697     prepareMessageForPackDelivery(message, info->userName, dateAndTime);
698     pthread_mutex_lock(&LogMutex);
699     write(fDes, message, strlen(message));
700     pthread_mutex_unlock(&LogMutex);
701   }
702   else if (logDelLogin(info->flag))
703   {
704     char message[MAX_BUF] = "\"";
705     prepareMessageForLogin(message, info->userName, dateAndTime);
706     pthread_mutex_lock(&LogMutex);
707     write(fDes, message, strlen(message));
708     pthread_mutex_unlock(&LogMutex);
709   }
710   else if (logDellaConnessione(info->flag))
711   {
712     char message[MAX_BUF] = "\"";
713     prepareMessageForConnection(message, info->userName, dateAndTime);
714     pthread_mutex_lock(&LogMutex);
715     write(fDes, message, strlen(message));
716     pthread_mutex_unlock(&LogMutex);
717   }
718   close(fDes);
719   free(info);
720   pthread_exit(NULL);
721 }
722
723 void spostaPlayer(char griglia[ROWS][COLUMNS], int vecchiaPosizione[2],
724                   int nuovaPosizione[2], Point deployCoords[],
725                   Point packsCoords[])
726 {
727
728   pthread_mutex_lock(&MatrixMutex);
729   griglia[nuovaPosizione[0]][nuovaPosizione[1]] = 'P';
730   if (eraUnPuntoDepo(vecchiaPosizione, deployCoords))
731     griglia[vecchiaPosizione[0]][vecchiaPosizione[1]] = '_';
732   else if (eraUnPacco(vecchiaPosizione, packsCoords))
733     griglia[vecchiaPosizione[0]][vecchiaPosizione[1]] = '$';
734   else
735     griglia[vecchiaPosizione[0]][vecchiaPosizione[1]] = '-';
736   pthread_mutex_unlock(&MatrixMutex);
737 }
738
739 PlayerStats gestisciW(char grigliaDiGioco[ROWS][COLUMNS],
740                       char grigliaOstacoli[ROWS][COLUMNS],
741                       PlayerStats giocatore, Obstacles *listaOstacoli,
742                       Point deployCoords[], Point packsCoords[])
743 {
744   if (giocatore == NULL)
745     return NULL;
746   int nuovaPosizione[2];
747   nuovaPosizione[1] = giocatore->position[1];
748   // Aggiorna la posizione vecchia spostando il player avanti di 1
749   nuovaPosizione[0] = (giocatore->position[0]) - 1;
750   int nuovoScore = giocatore->score;
751   int nuovoDeploy[2];
752   nuovoDeploy[0] = giocatore->deploy[0];
753   nuovoDeploy[1] = giocatore->deploy[1];
754   if (nuovaPosizione[0] >= 0 && nuovaPosizione[0] < ROWS)
755   {
```

```
    if (casellaVuotaOValida(grigliaDiGioco, grigliaOstacoli, nuovaPosizione))
    {
      spostaPlayer(grigliaDiGioco, giocatore->position, nuovaPosizione,
                   deployCoords, packsCoords);
    }
    else if (colpitoOstacolo(grigliaOstacoli, nuovaPosizione))
    {
      *listaOstacoli =
          addObstacle(*listaOstacoli, nuovaPosizione[0], nuovaPosizione[1]);
      nuovaPosizione[0] = giocatore->position[0];
      nuovaPosizione[1] = giocatore->position[1];
    }
    else if (colpitoPlayer(grigliaDiGioco, nuovaPosizione))
    {
      nuovaPosizione[0] = giocatore->position[0];
      nuovaPosizione[1] = giocatore->position[1];
    }
    giocatore->deploy[0] = nuovoDeploy[0];
    giocatore->deploy[1] = nuovoDeploy[1];
    giocatore->score = nuovoScore;
    giocatore->position[0] = nuovaPosizione[0];
    giocatore->position[1] = nuovaPosizione[1];
  }
  return giocatore;
}

PlayerStats gestisciD(char grigliaDiGioco[ROWS][COLUMNS],
                      char grigliaOstacoli[ROWS][COLUMNS],
                      PlayerStats giocatore, Obstacles *listaOstacoli,
                      Point deployCoords[], Point packsCoords[])
{
  if (giocatore == NULL)
  {
    return NULL;
  }
  int nuovaPosizione[2];
  nuovaPosizione[1] = giocatore->position[1] + 1;
  nuovaPosizione[0] = giocatore->position[0];
  int nuovoScore = giocatore->score;
  int nuovoDeploy[2];
  nuovoDeploy[0] = giocatore->deploy[0];
  nuovoDeploy[1] = giocatore->deploy[1];
  if (nuovaPosizione[1] >= 0 && nuovaPosizione[1] < COLUMNS)
  {
    if (casellaVuotaOValida(grigliaDiGioco, grigliaOstacoli, nuovaPosizione))
    {
      spostaPlayer(grigliaDiGioco, giocatore->position, nuovaPosizione,
                   deployCoords, packsCoords);
    }
    else if (colpitoOstacolo(grigliaOstacoli, nuovaPosizione))
    {
      printf("Ostacolo\n");
      *listaOstacoli =
          addObstacle(*listaOstacoli, nuovaPosizione[0], nuovaPosizione[1]);
      nuovaPosizione[0] = giocatore->position[0];
      nuovaPosizione[1] = giocatore->position[1];
    }
    else if (colpitoPlayer(grigliaDiGioco, nuovaPosizione))
    {
      nuovaPosizione[0] = giocatore->position[0];
      nuovaPosizione[1] = giocatore->position[1];
    }
    giocatore->deploy[0] = nuovoDeploy[0];
    giocatore->deploy[1] = nuovoDeploy[1];
    giocatore->score = nuovoScore;
    giocatore->position[0] = nuovaPosizione[0];
    giocatore->position[1] = nuovaPosizione[1];
  }
  return giocatore;
}
PlayerStats gestisciA(char grigliaDiGioco[ROWS][COLUMNS],
                      char grigliaOstacoli[ROWS][COLUMNS],
                      PlayerStats giocatore, Obstacles *listaOstacoli,
                      Point deployCoords[], Point packsCoords[])
{
  if (giocatore == NULL)
    return NULL;
  int nuovaPosizione[2];
  nuovaPosizione[0] = giocatore->position[0];
  // Aggiorna la posizione vecchia spostando il player avanti di 1
  nuovaPosizione[1] = (giocatore->position[1]) - 1;
  int nuovoScore = giocatore->score;
  int nuovoDeploy[2];
  nuovoDeploy[0] = giocatore->deploy[0];
  nuovoDeploy[1] = giocatore->deploy[1];
  if (nuovaPosizione[1] >= 0 && nuovaPosizione[1] < COLUMNS)
  {
    if (casellaVuotaOValida(grigliaDiGioco, grigliaOstacoli, nuovaPosizione))
    {
```

```
845       printf("Casella vuota \n");
846       spostaPlayer(grigliaDiGioco, giocatore->position, nuovaPosizione,
847                    deployCoords, packsCoords);
848     }
849     else if (colpitoOstacolo(grigliaOstacoli, nuovaPosizione))
850     {
851       printf("Ostacolo\n");
852       *listaOstacoli =
853           addObstacle(*listaOstacoli, nuovaPosizione[0], nuovaPosizione[1]);
854       nuovaPosizione[0] = giocatore->position[0];
855       nuovaPosizione[1] = giocatore->position[1];
856     }
857     else if (colpitoPlayer(grigliaDiGioco, nuovaPosizione))
858     {
859       printf("colpito player\n");
860       nuovaPosizione[0] = giocatore->position[0];
861       nuovaPosizione[1] = giocatore->position[1];
862     }
863     giocatore->deploy[0] = nuovoDeploy[0];
864     giocatore->deploy[1] = nuovoDeploy[1];
865     giocatore->score = nuovoScore;
866     giocatore->position[0] = nuovaPosizione[0];
867     giocatore->position[1] = nuovaPosizione[1];
868   }
869   return giocatore;
870 }
871 PlayerStats gestisciS(char grigliaDiGioco[ROWS][COLUMNS],
872                       char grigliaOstacoli[ROWS][COLUMNS],
873                       PlayerStats giocatore, Obstacles *listaOstacoli,
874                       Point deployCoords[], Point packsCoords[])
875 {
876   if (giocatore == NULL)
877   {
878     return NULL;
879   }
880   // crea le nuove statistiche
881   int nuovaPosizione[2];
882   nuovaPosizione[1] = giocatore->position[1];
883   nuovaPosizione[0] = (giocatore->position[0]) + 1;
884   int nuovoScore = giocatore->score;
885   int nuovoDeploy[2];
886   nuovoDeploy[0] = giocatore->deploy[0];
887   nuovoDeploy[1] = giocatore->deploy[1];
888   // controlla che le nuove statistiche siano corrette
889   if (nuovaPosizione[0] >= 0 && nuovaPosizione[0] < ROWS)
890   {
891     if (casellaVuotaOValida(grigliaDiGioco, grigliaOstacoli, nuovaPosizione))
892     {
893       spostaPlayer(grigliaDiGioco, giocatore->position, nuovaPosizione,
894                    deployCoords, packsCoords);
895     }
896     else if (colpitoOstacolo(grigliaOstacoli, nuovaPosizione))
897     {
898       printf("Ostacolo\n");
899       *listaOstacoli =
900           addObstacle(*listaOstacoli, nuovaPosizione[0], nuovaPosizione[1]);
901       nuovaPosizione[0] = giocatore->position[0];
902       nuovaPosizione[1] = giocatore->position[1];
903     }
904     else if (colpitoPlayer(grigliaDiGioco, nuovaPosizione))
905     {
906       nuovaPosizione[0] = giocatore->position[0];
907       nuovaPosizione[1] = giocatore->position[1];
908     }
909     giocatore->deploy[0] = nuovoDeploy[0];
910     giocatore->deploy[1] = nuovoDeploy[1];
911     giocatore->score = nuovoScore;
912     giocatore->position[0] = nuovaPosizione[0];
913     giocatore->position[1] = nuovaPosizione[1];
914   }
915   return giocatore;
916 }
917
918 int logDelPacco(int flag)
919 {
920   if (flag == 1)
921     return 1;
922   return 0;
923 }
924 int logDelLogin(int flag)
925 {
926   if (flag == 0)
927     return 1;
928   return 0;
929 }
930 int logDellaConnessione(int flag)
931 {
932   if (flag == 2)
933     return 1;
```

```
934   return 0;
935 }
```

A.3 Codice sorgente boardUtility

Listato 18: Codice header utility del gioco 1

```
1  #include "list.h"
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <time.h>
5  #include <unistd.h>
6  #define ROWS 10
7  #define COLUMNS 30
8  #define numberOfObstacles 35
9  #define numberOfPackages 15
10 #define TIME_LIMIT_IN_SECONDS 30
11 #define packageLimitNumber 4
12 #define MATRIX_DIMENSION sizeof(char) * ROWS *COLUMNS
13 #define RED_COLOR "\x1b[31m"
14 #define GREEN_COLOR "\x1b[32m"
15 #define RESET_COLOR "\x1b[0m"
16
17 struct Coord {
18   int x;
19   int y;
20 };
21 typedef struct Coord *Point;
22 void rimuoviPaccoDaArray(int posizione[2], Point packsCoords[]);
23 void printMenu();
24 int getHiddenPack(Point packsCoords[]);
25 int casellaVuotaOValida(char grigliaDiGioco[ROWS][COLUMNS],
26                         char grigliaOstacoli[ROWS][COLUMNS], int posizione[2]);
27 void stampaIstruzioni(int i);
28 int isOnADeployPoint(PlayerStats giocatore, Point deployCoords[]);
29 int isOnCorrectDeployPoint(PlayerStats giocatore, Point deployCoords[]);
30 PlayerStats gestisciP(char grigliaDiGioco[ROWS][COLUMNS], PlayerStats giocatore,
31                       Point deployCoords[], Point packsCoords[]);
32 void inizializzaGiocoSenzaPlayer(char grigliaDiGioco[ROWS][COLUMNS],
33                                  char grigliaConOstacoli[ROWS][COLUMNS],
34                                  Point packsCoords[]);
35 void inserisciPlayerNellaGrigliaInPosizioneCasuale(
36     char grigliaDiGioco[ROWS][COLUMNS], char grigliaOstacoli[ROWS][COLUMNS],
37     int posizione[2]);
38 void inizializzaGrigliaVuota(char grigliaDiGioco[ROWS][COLUMNS]);
39 void generaPosizioneOstacoli(char grigliaDiGioco[ROWS][COLUMNS],
40                              char grigliaOstacoli[ROWS][COLUMNS]);
41 void riempiGrigliaConPacchiInPosizioniGenerateCasualmente(
42     char grigliaDiGioco[ROWS][COLUMNS], Point packsCoords[]);
43 void printGrid(char grigliaDaStampare[ROWS][COLUMNS], PlayerStats stats);
44 void start(char grigliaDiGioco[ROWS][COLUMNS],
45            char grigliaOstacoli[ROWS][COLUMNS]);
46 void riempiGrigliaConGliOstacoli(char grigliaDiGioco[ROWS][COLUMNS],
47                                  char grigliaOstacoli[ROWS][COLUMNS]);
48 void generaPosizioniRaccolta(char grigliaDiGioco[ROWS][COLUMNS],
49                              char grigliaOstacoli[ROWS][COLUMNS],
50                              Point coord[]);
51 void mergeGridAndList(char grid[ROWS][COLUMNS], Obstacles top);
52 void scegliPosizioneRaccolta(Point coord[], int deploy[]);
53 int colpitoOstacolo(char grigliaOstacoli[ROWS][COLUMNS], int posizione[2]);
54 int colpitoPacco(Point packsCoords[], int posizione[2]);
55 int colpitoPlayer(char grigliaDiGioco[ROWS][COLUMNS], int posizione[2]);
56 int casellaVuota(char grigliaDiGioco[ROWS][COLUMNS],
57                  char grigliaOstacoli[ROWS][COLUMNS], int posizione[2]);
58 int arrivatoADestinazione(int posizione[2], int destinazione[2]);
59 int eraUnPuntoDepo(int vecchiaPosizione[2], Point depo[]);
60 int eraUnPacco(int vecchiaPosizione[2], Point packsCoords[]);
61 int isOnAPack(PlayerStats giocatore, Point packsCoords[]);
```

Listato 19: Codice sorgente utility del gioco 1

```
1  #include "boardUtility.h"
2  #include "list.h"
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6  #include <unistd.h>
7  void printMenu() {
8    system("clear");
9    printf("\t Cosa vuoi fare?\n");
10   printf("\t1 Gioca\n");
11   printf("\t2 Registrati\n");
12   printf("\t3 Esci\n");
13 }
```

```
14 int colpitoOstacolo(char grigliaOstacoli[ROWS][COLUMNS], int posizione[2]) {
15   if (grigliaOstacoli[posizione[0]][posizione[1]] == 'O')
16     return 1;
17   return 0;
18 }
19 int colpitoPacco(Point packsCoords[], int posizione[2]) {
20   int i = 0;
21   for (i = 0; i < numberOfPackages; i++) {
22     if (packsCoords[i]->x == posizione[0] && packsCoords[i]->y == posizione[1])
23       return 1;
24   }
25   return 0;
26 }
27 int casellaVuotaOValida(char grigliaDiGioco[ROWS][COLUMNS],
28                         char grigliaOstacoli[ROWS][COLUMNS], int posizione[2]) {
29   if (grigliaDiGioco[posizione[0]][posizione[1]] == '-' || // casella vuota
30       grigliaDiGioco[posizione[0]][posizione[1]] == '_' || // punto deploy
31       grigliaDiGioco[posizione[0]][posizione[1]] == '$')   // pacco
32     if (grigliaOstacoli[posizione[0]][posizione[1]] == '-' ||
33         grigliaOstacoli[posizione[0]][posizione[1]] == '_' ||
34         grigliaOstacoli[posizione[0]][posizione[1]] == '$')
35       return 1;
36   return 0;
37 }
38 int colpitoPlayer(char grigliaDiGioco[ROWS][COLUMNS], int posizione[2]) {
39   if (grigliaDiGioco[posizione[0]][posizione[1]] == 'P')
40     return 1;
41   return 0;
42 }
43 int isOnCorrectDeployPoint(PlayerStats giocatore, Point deployCoords[]) {
44   int i = 0;
45   for (i = 0; i < numberOfPackages; i++) {
46     if (giocatore->deploy[0] == deployCoords[i]->x &&
47         giocatore->deploy[1] == deployCoords[i]->y) {
48       if (deployCoords[i]->x == giocatore->position[0] &&
49           deployCoords[i]->y == giocatore->position[1])
50         return 1;
51     }
52   }
53   return 0;
54 }
55 int getHiddenPack(Point packsCoords[]) {
56   int i = 0;
57   for (i = 0; i < numberOfPackages; i++) {
58     if (packsCoords[i]->x == -1 && packsCoords[i]->y == -1)
59       return i;
60   }
61   return -1;
62 }
63 int isOnAPack(PlayerStats giocatore, Point packsCoords[]) {
64   int i = 0;
65   for (i = 0; i < numberOfPackages; i++) {
66     if (giocatore->position[0] == packsCoords[i]->x &&
67         giocatore->position[1] == packsCoords[i]->y)
68       return 1;
69   }
70   return 0;
71 }
72 int isOnADeployPoint(PlayerStats giocatore, Point deployCoords[]) {
73   int i = 0;
74   for (i = 0; i < numberOfPackages; i++) {
75     if (giocatore->position[0] == deployCoords[i]->x &&
76         giocatore->position[1] == deployCoords[i]->y)
77       return 1;
78   }
79   return 0;
80 }
81 void inizializzaGrigliaVuota(char griglia[ROWS][COLUMNS]) {
82   int i = 0, j = 0;
83   for (i = 0; i < ROWS; i++) {
84     for (j = 0; j < COLUMNS; j++) {
85       griglia[i][j] = '-';
86     }
87   }
88 }
89 PlayerStats gestisciP(char grigliaDiGioco[ROWS][COLUMNS], PlayerStats giocatore,
90                       Point deployCoords[], Point packsCoords[]) {
91   int nuovoDeploy[2];
92   if (colpitoPacco(packsCoords, giocatore->position) &&
93       giocatore->hasApack == 0) {
94     scegliPosizioneRaccolta(deployCoords, nuovoDeploy);
95     giocatore->hasApack = 1;
96     rimuoviPaccoDaArray(giocatore->position, packsCoords);
97   }
98   giocatore->deploy[0] = nuovoDeploy[0];
99   giocatore->deploy[1] = nuovoDeploy[1];
100  return giocatore;
101 }
102
```

```
103 void printGrid(char grigliaDaStampare[ROWS][COLUMNS], PlayerStats stats) {
104   system("clear");
105   printf("\n\n");
106   int i = 0, j = 0;
107   for (i = 0; i < ROWS; i++) {
108     printf("\t");
109     for (j = 0; j < COLUMNS; j++) {
110       if (stats != NULL) {
111         if ((i == stats->deploy[0] && j == stats->deploy[1]) ||
112             (i == stats->position[0] && j == stats->position[1]))
113           if (grigliaDaStampare[i][j] == 'P' && stats->hasApack == 1)
114             printf(GREEN_COLOR "%c" RESET_COLOR, grigliaDaStampare[i][j]);
115           else
116             printf(RED_COLOR "%c" RESET_COLOR, grigliaDaStampare[i][j]);
117         else
118           printf("%c", grigliaDaStampare[i][j]);
119       } else
120         printf("%c", grigliaDaStampare[i][j]);
121     }
122     stampaIstruzioni(i);
123     if (i == 8)
124       printf(GREEN_COLOR "\t\t Punteggio: %d" RESET_COLOR, stats->score);
125     printf("\n");
126   }
127 }
128 void stampaIstruzioni(int i) {
129   if (i == 0)
130     printf("\t \t ISTRUZIONI ");
131   if (i == 1)
132     printf("\t Inviare 't' per il timer.");
133   if (i == 2)
134     printf("\t Inviare 'e' per uscire");
135   if (i == 3)
136     printf("\t Inviare 'p' per raccogliere un pacco");
137   if (i == 4)
138     printf("\t Inviare 'c' per consegnare il pacco");
139   if (i == 5)
140     printf("\t Inviare 'w'/'s' per andare sopra/sotto");
141   if (i == 6)
142     printf("\t Inviare 'a'/'d' per andare a dx/sx");
143   if (i == 7)
144     printf("\t Inviare 'l' per la lista degli utenti ");
145 }
146 // aggiunge alla griglia gli ostacoli visti fino ad ora dal client
147 void mergeGridAndList(char grid[ROWS][COLUMNS], Obstacles top) {
148   while (top) {
149     grid[top->x][top->y] = 'O';
150     top = top->next;
151   }
152 }
153 /* Genera la posizione degli ostacoli */
154 void generaPosizioneOstacoli(char grigliaDiGioco[ROWS][COLUMNS],
155                              char grigliaOstacoli[ROWS][COLUMNS]) {
156   int x, y, i;
157   inizializzaGrigliaVuota(grigliaOstacoli);
158   srand(time(0));
159   for (i = 0; i < numberOfObstacles; i++) {
160     x = rand() % COLUMNS;
161     y = rand() % ROWS;
162     if (grigliaDiGioco[y][x] == '-')
163       grigliaOstacoli[y][x] = 'O';
164     else
165       i--;
166   }
167 }
168 void rimuoviPaccoDaArray(int posizione[2], Point packsCoords[]) {
169   int i = 0, found = 0;
170   while (i < numberOfPackages && !found) {
171     if ((packsCoords[i])->x == posizione[0] &&
172         (packsCoords[i])->y == posizione[1]) {
173       (packsCoords[i])->x = -1;
174       (packsCoords[i])->y = -1;
175       found = 1;
176     }
177     i++;
178   }
179 }
180 // sceglie una posizione di raccolta tra quelle disponibili
181 void scegliPosizioneRaccolta(Point coord[], int deploy[]) {
182   int index = 0;
183   srand(time(NULL));
184   index = rand() % numberOfPackages;
185   deploy[0] = coord[index]->x;
186   deploy[1] = coord[index]->y;
187 }
188 /*genera posizione di raccolta di un pacco*/
189 void generaPosizioniRaccolta(char grigliaDiGioco[ROWS][COLUMNS],
190                              char grigliaOstacoli[ROWS][COLUMNS],
191                              Point coord[]) {
```

```
192   int x, y;
193   srand(time(0));
194   int i = 0;
195   for (i = 0; i < numberOfPackages; i++) {
196     coord[i] = (Point)malloc(sizeof(struct Coord));
197   }
198   i = 0;
199   for (i = 0; i < numberOfPackages; i++) {
200     x = rand() % COLUMNS;
201     y = rand() % ROWS;
202     if (grigliaDiGioco[y][x] == '-' && grigliaOstacoli[y][x] == '-') {
203       coord[i]->x = y;
204       coord[i]->y = x;
205       grigliaDiGioco[y][x] = '_';
206       grigliaOstacoli[y][x] = '_';
207     } else
208       i--;
209   }
210 }
211 /*Inserisci dei pacchi nella griglia di gioco nella posizione casuale */
212 void riempiGrigliaConPacchiInPosizioniGenerateCasualmente(
213     char grigliaDiGioco[ROWS][COLUMNS], Point packsCoords[]) {
214   int x, y, i = 0;
215   for (i = 0; i < numberOfPackages; i++) {
216     packsCoords[i] = (Point)malloc(sizeof(struct Coord));
217   }
218   srand(time(0));
219   for (i = 0; i < numberOfPackages; i++) {
220     x = rand() % COLUMNS;
221     y = rand() % ROWS;
222     if (grigliaDiGioco[y][x] == '-') {
223       grigliaDiGioco[y][x] = '$';
224       packsCoords[i]->x = y;
225       packsCoords[i]->y = x;
226     } else
227       i--;
228   }
229 }
230 /*Inserisci gli ostacoli nella griglia di gioco */
231 void riempiGrigliaConGliOstacoli(char grigliaDiGioco[ROWS][COLUMNS],
232                                  char grigliaOstacoli[ROWS][COLUMNS]) {
233   int i, j = 0;
234   for (i = 0; i < ROWS; i++) {
235     for (j = 0; j < COLUMNS; j++) {
236       if (grigliaOstacoli[i][j] == 'O')
237         grigliaDiGioco[i][j] = 'O';
238     }
239   }
240 }
241 void inserisciPlayerNellaGrigliaInPosizioneCasuale(
242     char grigliaDiGioco[ROWS][COLUMNS], char grigliaOstacoli[ROWS][COLUMNS],
243     int posizione[2]) {
244   int x, y;
245   srand(time(0));
246   printf("Inserisco player\n");
247   do {
248     x = rand() % COLUMNS;
249     y = rand() % ROWS;
250   } while (grigliaDiGioco[y][x] != '-' && grigliaOstacoli[y][x] != '-');
251   grigliaDiGioco[y][x] = 'P';
252   posizione[0] = y;
253   posizione[1] = x;
254 }
255 void inizializzaGiocoSenzaPlayer(char grigliaDiGioco[ROWS][COLUMNS],
256                                  char grigliaConOstacoli[ROWS][COLUMNS],
257                                  Point packsCoords[]) {
258   inizializzaGrigliaVuota(grigliaDiGioco);
259   riempiGrigliaConPacchiInPosizioniGenerateCasualmente(grigliaDiGioco,
260                                                        packsCoords);
261   generaPosizioneOstacoli(grigliaDiGioco, grigliaConOstacoli);
262   return;
263 }
264
265 int eraUnPuntoDepo(int vecchiaPosizione[2], Point depo[]) {
266   int i = 0, ret = 0;
267   while (ret == 0 && i < numberOfPackages) {
268     if ((depo[i])->y == vecchiaPosizione[1] &&
269         (depo[i])->x == vecchiaPosizione[0]) {
270       ret = 1;
271     }
272     i++;
273   }
274   return ret;
275 }
276 int eraUnPacco(int vecchiaPosizione[2], Point packsCoords[]) {
277   int i = 0, ret = 0;
278   while (ret == 0 && i < numberOfPackages) {
279     if ((packsCoords[i])->y == vecchiaPosizione[1] &&
280         (packsCoords[i])->x == vecchiaPosizione[0]) {
```

```
      ret = 1;
    }
    i++;
  }
  return ret;
}

int arrivatoADestinazione(int posizione[2], int destinazione[2]) {
  if (posizione[0] == destinazione[0] && posizione[1] == destinazione[1])
    return 1;
  return 0;
}
```

A.4 Codice sorgente list

Listato 20: Codice header utility del gioco 2

```
#ifndef DEF_LIST_H
#define DEF_LIST_H
#define MAX_BUF 200
#include <pthread.h>
// players
struct TList {
  char *name;
  struct TList *next;
  int sockDes;
} TList;

struct Data {
  int deploy[2];
  int score;
  int position[2];
  int hasApack;
} Data;

// Obstacles
struct TList2 {
  int x;
  int y;
  struct TList2 *next;
} TList2;

typedef struct Data *PlayerStats;
typedef struct TList *Players;
typedef struct TList2 *Obstacles;

// calcola e restituisce il numero di player commessi dalla lista L
int dimensioneLista(Players L);

// inizializza un giocatore
Players initPlayerNode(char *name, int sockDes);

// Crea un nodo di Stats da mandare a un client
PlayerStats initStats(int deploy[], int score, int position[], int flag);

// Inizializza un nuovo nodo
Players initNodeList(char *name, int sockDes);

// Aggiunge un nodo in testa alla lista
// La funzione ritorna sempre la testa della lista
Players addPlayer(Players L, char *name, int sockDes);

// Rimuove solo un occorrenza di un nodo con il socket Descriptor
//  specificato dalla lista
// La funzione ritorna sempre la testa della lista
Players removePlayer(Players L, int sockDes);

// Dealloca la lista interamente
void freePlayers(Players L);

// Stampa la lista
void printPlayers(Players L);

// Controlla se un utente á giá loggato
int isAlreadyLogged(Players L, char *name);

// Dealloca la lista degli ostacoli
void freeObstacles(Obstacles L);

// Stampa la lista degli ostacoli
void printObstacles(Obstacles L);

// Aggiunge un ostacolo in testa
Obstacles addObstacle(Obstacles L, int x, int y);

```

```
69 // Inizializza un nuovo nodo ostacolo
70 Obstacles initObstacleNode(int x, int y);
71 #endif
```

Listato 21: Codice sorgente utility del gioco 2

```
1  #include "list.h"
2  #include <pthread.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6
7  Players initPlayerNode(char *name, int sockDes) {
8    Players L = (Players)malloc(sizeof(struct TList));
9    L->name = (char *)malloc(MAX_BUF);
10   strcpy(L->name, name);
11   L->sockDes = sockDes;
12   L->next = NULL;
13   return L;
14 }
15 PlayerStats initStats(int deploy[], int score, int position[], int flag) {
16   PlayerStats L = (PlayerStats)malloc(sizeof(struct Data));
17   L->deploy[0] = deploy[0];
18   L->deploy[1] = deploy[1];
19   L->score = score;
20   L->hasApack = flag;
21   L->position[0] = position[0];
22   L->position[1] = position[1];
23   return L;
24 }
25 Obstacles initObstacleNode(int x, int y) {
26   Obstacles L = (Obstacles)malloc(sizeof(struct TList2));
27   L->x = x;
28   L->y = y;
29   L->next = NULL;
30   return L;
31 }
32 Obstacles addObstacle(Obstacles L, int x, int y) {
33   Obstacles tmp = initObstacleNode(x, y);
34   if (L != NULL)
35     tmp->next = L;
36   return tmp;
37 }
38 int dimensioneLista(Players L) {
39   int size = 0;
40   Players tmp = L;
41   while (tmp != NULL) {
42     size++;
43     tmp = tmp->next;
44   }
45   return size;
46 }
47 int isAlreadyLogged(Players L, char *name) {
48   int ret = 0;
49   if (L != NULL) {
50     if (strcmp(L->name, name) == 0)
51       return 1;
52     ret = isAlreadyLogged(L->next, name);
53   }
54   return ret;
55 }
56 Players addPlayer(Players L, char *name, int sockDes) {
57   Players tmp = initPlayerNode(name, sockDes);
58   if (L != NULL)
59     tmp->next = L;
60   return tmp;
61 }
62 Players removePlayer(Players L, int sockDes) {
63   if (L != NULL) {
64     if (L->sockDes == sockDes) {
65       Players tmp = L->next;
66       free(L);
67       return tmp;
68     }
69     L->next = removePlayer(L->next, sockDes);
70   }
71   return L;
72 }
73 void freePlayers(Players L) {
74   if (L != NULL) {
75     freePlayers(L->next);
76     free(L);
77   }
78 }
79 void freeObstacles(Obstacles L) {
80   if (L != NULL) {
81     freeObstacles(L->next);
82     free(L);
```

```
  }
}
void printPlayers(Players L) {
  if (L != NULL) {
    printf("%s ->", L->name);
    printPlayers(L->next);
  }
  printf("\n");
}
void printObstacles(Obstacles L) {
  if (L != NULL) {
    printf("X:%d Y:%d ->", L->x, L->y);
    printObstacles(L->next);
  }
}
```

A.5 Codice sorgente parser

Listato 22: Codice header utility del gioco 3

```
int appendPlayer(char *name, char *pwd, char *file);
int isRegistered(char *name, char *file);
int openFileRDWRAPP(char *file);
int validateLogin(char *name, char *pwd, char *file);
int openFileRDON(char *file);
void premiEnterPerContinuare();
```

Listato 23: Codice sorgente utility del gioco 3

```
#include "parser.h"
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#define MAX_BUF 200
int openFileRDWRAPP(char *file) {
  int fileDes = open(file, O_RDWR | O_CREAT | O_APPEND, S_IRUSR | S_IWUSR);
  if (fileDes < 0)
    perror("Errore apertura file\n"), exit(-1);
  return fileDes;
}
int openFileRDON(char *file) {
  int fileDes = open(file, O_RDONLY);
  if (fileDes < 0)
    perror("Errore apertura file\n"), exit(-1);
  return fileDes;
}
int appendPlayer(char *name, char *pwd, char *file) {
  if (isRegistered(name, file))
    return 0;
  int fileDes = openFileRDWRAPP(file);
  write(fileDes, name, strlen(name));
  write(fileDes, " ", 1);
  write(fileDes, pwd, strlen(pwd));
  write(fileDes, "\n", 1);
  close(fileDes);
  return 1;
}
int isRegistered(char *name, char *file) {
  char command[MAX_BUF] = "cat  ";
  strcat(command, file);
  char toApp[] = " |cut -d\" \" -f1|grep \"^";
  strcat(command, toApp);
  strcat(command, name);
  char toApp2[] = "$\">tmp";
  strcat(command, toApp2);
  int ret = 0;
  system(command);
  int fileDes = openFileRDON("tmp");
  struct stat info;
  fstat(fileDes, &info);
  if ((int)info.st_size > 0)
    ret = 1;
  close(fileDes);
  system("rm tmp");
  return ret;
}
int validateLogin(char *name, char *pwd, char *file) {
  if (!isRegistered(name, file))
    return 0;
```

```
56    char command[MAX_BUF] = "cat  ";
57    strcat(command, file);
58    char toApp[] = " |grep \"^";
59    strcat(command, toApp);
60    strcat(command, name);
61    strcat(command, " ");
62    strcat(command, pwd);
63    char toApp2[] = "$\">tmp";
64    strcat(command, toApp2);
65    int ret = 0;
66    system(command);
67    int fileDes = openFileRDON("tmp");
68    struct stat info;
69    fstat(fileDes, &info);
70    if ((int)info.st_size > 0)
71      ret = 1;
72    close(fileDes);
73    system("rm tmp");
74    return ret;
75  }
76  void premiEnterPerContinuare() {
77    fflush(stdin);
78    printf("Premi Invio per continuare\n");
79    char c = getchar();
80  }
```

Listati