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 dei gioco si 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 protocoli 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

```
1  struct sockaddr_in configuraIndirizzo() {
2    struct sockaddr_in mio_indirizzo;
3    mio_indirizzo.sin_family = AF_INET;
4    mio_indirizzo.sin_port = htons(5200);
5    mio_indirizzo.sin_addr.s_addr = htonl(INADDR_ANY);
6    printf("Indirizzo socket configurato\n");
7    return mio_indirizzo;
8  }
```

Listato 2: Configurazione socket del server

```
1  void configuraSocket(struct sockaddr_in mio_indirizzo) {
2    if ((socketDesc = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
3      perror("Impossibile creare socket");
4      exit(-1);
5    }
6    if (setsockopt(socketDesc, SOL_SOCKET, SO_REUSEADDR, &(int){1}, sizeof(int)) <
7        0)
8      perror("Impossibile impostare il riutilizzo dell'indirizzo ip e della "
9             "porta\n");
10   if ((bind(socketDesc, (struct sockaddr *)&mio_indirizzo,
11             sizeof(mio_indirizzo))) < 0) {
12     perror("Impossibile effettuare bind");
13     exit(-1);
14   }
15 }
```

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. Una volta aver configurato il socket, infatti, il server si mette in ascolto per nuove connessioni in entrata ed ogni volta che viene stabilita una nuova connessione viene avviato un thread per gestire tale connessione. Di seguito il relativo codice:

Listato 3: Procedura di ascolto del server

```
1  void startListening() {
2    pthread_t tid;
3    int clientDesc;
4    int *puntClientDesc;
5    while (1 == 1) {
6      if (listen(socketDesc, 10) < 0)
7        perror("Impossibile mettersi in ascolto"), exit(-1);
8      printf("In ascolto..\n");
9      if ((clientDesc = accept(socketDesc, NULL, NULL)) < 0) {
10       perror("Impossibile effettuare connessione\n");
11       exit(-1);
12     }
13     printf("Nuovo client connesso\n");
14     struct sockaddr_in address;
15     socklen_t size = sizeof(struct sockaddr_in);
16     if (getpeername(clientDesc, (struct sockaddr *)&address, &size) < 0) {
17       perror("Impossibile ottenere l'indirizzo del client");
18       exit(-1);
19     }
20     char clientAddr[20];
21     strcpy(clientAddr, inet_ntoa(address.sin_addr));
22     Args args = (Args)malloc(sizeof(struct argsToSend));
23     args->userName = (char *)calloc(MAX_BUF, 1);
24     strcpy(args->userName, clientAddr);
25     args->flag = 2;
26     pthread_t tid;
27     pthread_create(&tid, NULL, fileWriter, (void *)args);
28
29     puntClientDesc = (int *)malloc(sizeof(int));
30     *puntClientDesc = clientDesc;
31     pthread_create(&tid, NULL, gestisci, (void *)puntClientDesc);
32   }
33   close(clientDesc);
34   quitServer();
35 }
```

In particolare al rigo 31 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 16 al rigo 27, 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 (argv[2]) dal tipo stringa al tipo della porta (uint16_t ovvero unsigned long integer).
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 decidiamo di ritoranre soltanto 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    int bufferReceive[2] = {1};
3    int client_sd = *(int *)descriptor;
4    int continua = 1;
5    char name[MAX_BUF];
6    while (continua) {
7      read(client_sd, bufferReceive, sizeof(bufferReceive));
8      if (bufferReceive[0] == 2)
9        registraClient(client_sd);
10     else if (bufferReceive[0] == 1)
11       if (tryLogin(client_sd, name)) {
12         play(client_sd, name);
13         continua = 0;
14       } else if (bufferReceive[0] == 3)
15         disconnettiClient(client_sd);
16       else {
17         printf("Input invalido, uscita...\n");
18         disconnettiClient(client_sd);
19       }
20   }
21   pthread_exit(0);
22 }
```

Si noti come il server riceva, al rigo 7, 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 26)
- Il player con le relative informazioni (Righi 28 a 31)
- Un messaggio che notifica al client se è iniziato un nuovo turno oppure no (Righi 40,43,46,53)

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. 13 Rigo 430 e List. 15 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.

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 eccezzione 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. 12 rigo 59).

Listato 8: Funzione play del server

```
void play(int clientDesc, char name[]) {
  int true = 1;
  int turnoFinito = 0;
  int turnoGiocatore = turno;
  int posizione[2];
  int destinazione[2] = {-1, -1};
  PlayerStats giocatore = initStats(destinazione, 0, posizione, 0);
  Obstacles listaOstacoli = NULL;
  char inputFromClient;
  if (timer != 0) {
    inserisciPlayerNellaGrigliaInPosizioneCasuale(
        grigliaDiGiocoConPacchiSenzaOstacoli, grigliaOstacoliSenzaPacchi,
        giocatore->position);
    playerGenerati++;
  }
  while (true) {
    if (clientDisconnesso(clientDesc)) {
      freeObstacles(listaOstacoli);
      disconnettiClient(clientDesc);
      return;
    }
    char grigliaTmp[ROWS][COLUMNS];
    clonaGriglia(grigliaTmp, grigliaDiGiocoConPacchiSenzaOstacoli);
    mergeGridAndList(grigliaTmp, listaOstacoli);
    // invia la griglia
    write(clientDesc, grigliaTmp, sizeof(grigliaTmp));
    // invia la struttura del player
    write(clientDesc, giocatore->deploy, sizeof(giocatore->deploy));
    write(clientDesc, giocatore->position, sizeof(giocatore->position));
    write(clientDesc, &giocatore->score, sizeof(giocatore->score));
    write(clientDesc, &giocatore->hasApack, sizeof(giocatore->hasApack));
    // legge l'input
    if (read(clientDesc, &inputFromClient, sizeof(char)) > 0)
      numMosse++;
    if (inputFromClient == 'e' || inputFromClient == 'E') {
      freeObstacles(listaOstacoli);
      listaOstacoli = NULL;
      disconnettiClient(clientDesc);
    } else if (inputFromClient == 't' || inputFromClient == 'T') {
      write(clientDesc, &turnoFinito, sizeof(int));
      sendTimerValue(clientDesc);
    } else if (inputFromClient == 'l' || inputFromClient == 'L') {
      write(clientDesc, &turnoFinito, sizeof(int));
      sendPlayerList(clientDesc);
    } else if (turnoGiocatore == turno) {
      write(clientDesc, &turnoFinito, sizeof(int));
      giocatore =
          gestisciInput(grigliaDiGiocoConPacchiSenzaOstacoli,
                        grigliaOstacoliSenzaPacchi, inputFromClient, giocatore,
                        &listaOstacoli, deployCoords, packsCoords, name);
    } else {
      turnoFinito = 1;
      write(clientDesc, &turnoFinito, sizeof(int));
      freeObstacles(listaOstacoli);
      listaOstacoli = NULL;
      inserisciPlayerNellaGrigliaInPosizioneCasuale(
          grigliaDiGiocoConPacchiSenzaOstacoli, grigliaOstacoliSenzaPacchi,
          giocatore->position);
      giocatore->score = 0;
      giocatore->hasApack = 0;
      giocatore->deploy[0] = -1;
      giocatore->deploy[1] = -1;
      turnoGiocatore = turno;
      turnoFinito = 0;
      playerGenerati++;
    }
  }
}
```

Listato 9: Funzione play del client

```
void play() {
  PlayerStats giocatore = NULL;
  int score, deploy[2], position[2], timer;
  int turnoFinito = 0;
  int exitFlag = 0, hasApack = 0;
  while (!exitFlag) {
    if (serverCaduto())
      serverCrashHandler();
    if (read(socketDesc, grigliaDiGioco, sizeof(grigliaDiGioco)) < 1)
      printf("Impossibile comunicare con il server\n"), exit(-1);
    if (read(socketDesc, deploy, sizeof(deploy)) < 1)
      printf("Impossibile comunicare con il server\n"), exit(-1);
    if (read(socketDesc, position, sizeof(position)) < 1)
      printf("Impossibile comunicare con il server\n"), exit(-1);
    if (read(socketDesc, &score, sizeof(score)) < 1)
      printf("Impossibile comunicare con il server\n"), exit(-1);
    if (read(socketDesc, &hasApack, sizeof(hasApack)) < 1)
      printf("Impossibile comunicare con il server\n"), exit(-1);
    giocatore = initStats(deploy, score, position, hasApack);
    printGrid(grigliaDiGioco, giocatore);
    char send = getUserInput();
    if (send == 'e' || send == 'E') {
      esciDalServer();
      exit(0);
    }
    write(socketDesc, &send, sizeof(char));
    read(socketDesc, &turnoFinito, sizeof(turnoFinito));
    if (turnoFinito) {
      system("clear");
      printf("Turno finito\n");
      sleep(1);
    } else {
      if (send == 't' || send == 'T')
        printTimer();
      else if (send == 'l' || send == 'L')
        printPlayerList();
    }
  }
}
```

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. 13 rigo 89 e 144) 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    int cambiato = 1;
3    while (1) {
4      if (almenoUnClientConnesso() && valoreTimerValido() &&
5          almenoUnPlayerGenerato() && almenoUnaMossaFatta()) {
6        cambiato = 1;
7        sleep(1);
8        timerCount--;
9        fprintf(stdout, "Time left: %d\n", timerCount);
10     } else if (numeroClientLoggati == 0) {
11       timerCount = TIME_LIMIT_IN_SECONDS;
12       if (cambiato) {
13         fprintf(stdout, "Time left: %d\n", timerCount);
14         cambiato = 0;
15       }
16     }
17     if (timerCount == 0 || scoreMassimo == packageLimitNumber) {
18       playerGenerati = 0;
19       numMosse = 0;
20       printf("Reset timer e generazione nuova mappa..\n");
21       startProceduraGenrazioneMappa();
22       pthread_join(tidGeneratoreMappa, NULL);
23       turno++;
24       timerCount = TIME_LIMIT_IN_SECONDS;
25     }
26   }
27 }
```

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]

[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 2 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

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

Listato 11: Generazione nuova mappa e posizione players

```
1  void *threadGenerazioneMappa(void *args) {
2    fprintf(stdout, "Rigenerazione mappa\n");
3    inizializzaGrigliaVuota(grigliaDiGiocoConPacchiSenzaOstacoli);
4    generaPosizioniRaccolta(grigliaDiGiocoConPacchiSenzaOstacoli,
5                            grigliaOstacoliSenzaPacchi, deployCoords);
6    riempiGrigliaConPacchiInPosizioniGenerateCasualmente(
7        grigliaDiGiocoConPacchiSenzaOstacoli, packsCoords);
8    generaPosizioneOstacoli(grigliaDiGiocoConPacchiSenzaOstacoli,
9                            grigliaOstacoliSenzaPacchi);
10   printf("Mappa generata\n");
11   pthread_exit(NULL);
12 }
```

A Codici sorgente

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

A.1 Codice sorgente del client

Listato 12: 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);
```

```
64  }
65  int connettiAlServer(char **argv) {
66    char *indirizzoServer;
67    uint16_t porta = strtoul(argv[2], NULL, 10);
68    indirizzoServer = ipResolver(argv);
69    struct sockaddr_in mio_indirizzo;
70    mio_indirizzo.sin_family = AF_INET;
71    mio_indirizzo.sin_port = htons(porta);
72    inet_aton(indirizzoServer, &mio_indirizzo.sin_addr);
73    if ((socketDesc = socket(PF_INET, SOCK_STREAM, 0)) < 0)
74      perror("Impossibile creare socket"), exit(-1);
75    else
76      printf("Socket creato\n");
77    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() {
```

```
  char c;
  c = getchar();
  int daIgnorare;
  while ((daIgnorare = getchar()) != '\n' && daIgnorare != EOF) {
  }
  return c;
}
```

A.2 Codice sorgente del server

Listato 13: Codice sorgente del server

```
#include "boardUtility.h"
#include "list.h"
#include "parser.h"
#include <arpa/inet.h>
#include <errno.h>
#include <fcntl.h>
#include <netinet/in.h> //conversioni
#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>

struct argsToSend {
  char *userName;
  int flag;
};
typedef struct argsToSend *Args;
void sendPlayerList(int clientDesc);
PlayerStats gestisciC(char grigliaDiGioco[ROWS][COLUMNS], PlayerStats giocatore,
                      Point deployCoords[], Point packsCoords[], char name[]);
PlayerStats gestisciInput(char grigliaDiGioco[ROWS][COLUMNS],
                          char grigliaOstacoli[ROWS][COLUMNS], char input,
                          PlayerStats giocatore, Obstacles *listaOstacoli,
                          Point deployCoords[], Point packsCoords[],
                          char name[]);
void clonaGriglia(char destinazione[ROWS][COLUMNS], char source[ROWS][COLUMNS]);
int almenoUnClientConnesso();
int valoreTimerValido();
int almenoUnPlayerGenerato();
int almenoUnaMossaFatta();
void sendTimerValue(int clientDesc);
void startProceduraGenrazioneMappa();
void *threadGenerazioneMappa(void *args);
void *fileWriter(void *);
int tryLogin(int clientDesc, char name[]);
void disconnettiClient(int);
int registraClient(int);
void *timer(void *args);
void *gestisci(void *descriptor);
void quitServer();
void clientCrashHandler(int signalNum);
void startTimer();
void configuraSocket(struct sockaddr_in mio_indirizzo);
struct sockaddr_in configuraIndirizzo();
void startListening();
int clientDisconnesso(int clientSocket);
void play(int clientDesc, char name[]);

char grigliaDiGiocoConPacchiSenzaOstacoli[ROWS][COLUMNS];
char grigliaOstacoliSenzaPacchi[ROWS][COLUMNS];
int numeroClientLoggati = 0;
int playerGenerati = 0;
```

```
int timerCount = TIME_LIMIT_IN_SECONDS;
int turno = 0;
pthread_t tidTimer;
pthread_t tidGeneratoreMappa;
int socketDesc;
Players onLineUsers = NULL;
char *users;
int scoreMassimo = 0;
int numMosse = 0;
Point deployCoords[numberOfPackages];
Point packsCoords[numberOfPackages];
pthread_mutex_t LogMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t RegMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t PlayerMutex = PTHREAD_MUTEX_INITIALIZER;

int main(int argc, char **argv) {
  if (argc != 2) {
    printf("Wrong parameters number(Usage: ./server usersFile)\n");
    exit(-1);
  } else if (strcmp(argv[1], "Log") == 0) {
    printf("Cannot use the Log file as a UserList \n");
    exit(-1);
  }
  users = argv[1];
  struct sockaddr_in mio_indirizzo = configuraIndirizzo();
  configuraSocket(mio_indirizzo);
  signal(SIGPIPE, clientCrashHandler);
  signal(SIGINT, quitServer);
  signal(SIGHUP, quitServer);
  startTimer();
  inizializzaGiocoSenzaPlayer(grigliaDiGiocoConPacchiSenzaOstacoli,
                              grigliaOstacoliSenzaPacchi, packsCoords);
  generaPosizioniRaccolta(grigliaDiGiocoConPacchiSenzaOstacoli,
                          grigliaOstacoliSenzaPacchi, deployCoords);
  startListening();
  return 0;
}
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));
    *puntClientDesc = clientDesc;
    pthread_create(&tid, NULL, gestisci, (void *)puntClientDesc);
  }
  close(clientDesc);
  quitServer();
}
struct sockaddr_in configuraIndirizzo() {
  struct sockaddr_in mio_indirizzo;
```

```
134   mio_indirizzo.sin_family = AF_INET;
135   mio_indirizzo.sin_port = htons(5200);
136   mio_indirizzo.sin_addr.s_addr = htonl(INADDR_ANY);
137   printf("Indirizzo socket configurato\n");
138   return mio_indirizzo;
139 }
140 void startProceduraGenrazioneMappa() {
141   printf("Inizio procedura generazione mappa\n");
142   pthread_create(&tidGeneratoreMappa, NULL, threadGenerazioneMappa, NULL);
143 }
144 void startTimer() {
145   printf("Thread timer avviato\n");
146   pthread_create(&tidTimer, NULL, timer, NULL);
147 }
148 int tryLogin(int clientDesc, char name[]) {
149   char *userName = (char *)calloc(MAX_BUF, 1);
150   char *password = (char *)calloc(MAX_BUF, 1);
151   int dimName, dimPwd;
152   read(clientDesc, &dimName, sizeof(int));
153   read(clientDesc, &dimPwd, sizeof(int));
154   read(clientDesc, userName, dimName);
155   read(clientDesc, password, dimPwd);
156   int ret = 0;
157   if (validateLogin(userName, password, users) &&
158       !isAlreadyLogged(onLineUsers, userName)) {
159     ret = 1;
160     numeroClientLoggati++;
161     write(clientDesc, "y", 1);
162     strcpy(name, userName);
163     Args args = (Args)malloc(sizeof(struct argsToSend));
164     args->userName = (char *)calloc(MAX_BUF, 1);
165     strcpy(args->userName, name);
166     args->flag = 0;
167     pthread_t tid;
168     pthread_create(&tid, NULL, fileWriter, (void *)args);
169     printf("Nuovo client loggato, client loggati : %d\n", numeroClientLoggati);
170     pthread_mutex_lock(&PlayerMutex);
171     onLineUsers = addPlayer(onLineUsers, userName, clientDesc);
172     pthread_mutex_unlock(&PlayerMutex);
173     printPlayers(onLineUsers);
174     printf("\n");
175   } else {
176     write(clientDesc, "n", 1);
177   }
178   return ret;
179 }
180 void *gestisci(void *descriptor) {
181   int bufferReceive[2] = {1};
182   int client_sd = *(int *)descriptor;
183   int continua = 1;
184   char name[MAX_BUF];
185   while (continua) {
186     read(client_sd, bufferReceive, sizeof(bufferReceive));
187     if (bufferReceive[0] == 2)
188       registraClient(client_sd);
189     else if (bufferReceive[0] == 1)
190       if (tryLogin(client_sd, name)) {
191         play(client_sd, name);
192         continua = 0;
193       } else if (bufferReceive[0] == 3)
194         disconnettiClient(client_sd);
195       else {
196         printf("Input invalido, uscita...\n");
197         disconnettiClient(client_sd);
198       }
199   }
200   pthread_exit(0);
201 }
202 void play(int clientDesc, char name[]) {
203   int true = 1;
204   int turnoFinito = 0;
205   int turnoGiocatore = turno;
206   int posizione[2];
207   int destinazione[2] = {-1, -1};
```

```
208   PlayerStats giocatore = initStats(destinazione, 0, posizione, 0);
209   Obstacles listaOstacoli = NULL;
210   char inputFromClient;
211   if (timer != 0) {
212     inserisciPlayerNellaGrigliaInPosizioneCasuale(
213         grigliaDiGiocoConPacchiSenzaOstacoli, grigliaOstacoliSenzaPacchi,
214         giocatore->position);
215     playerGenerati++;
216   }
217   while (true) {
218     if (clientDisconnesso(clientDesc)) {
219       freeObstacles(listaOstacoli);
220       disconnettiClient(clientDesc);
221       return;
222     }
223     char grigliaTmp[ROWS][COLUMNS];
224     clonaGriglia(grigliaTmp, grigliaDiGiocoConPacchiSenzaOstacoli);
225     mergeGridAndList(grigliaTmp, listaOstacoli);
226     // invia la griglia
227     write(clientDesc, grigliaTmp, sizeof(grigliaTmp));
228     // invia la struttura del player
229     write(clientDesc, giocatore->deploy, sizeof(giocatore->deploy));
230     write(clientDesc, giocatore->position, sizeof(giocatore->position));
231     write(clientDesc, &giocatore->score, sizeof(giocatore->score));
232     write(clientDesc, &giocatore->hasApack, sizeof(giocatore->hasApack));
233     // legge l'input
234     if (read(clientDesc, &inputFromClient, sizeof(char)) > 0)
235       numMosse++;
236     if (inputFromClient == 'e' || inputFromClient == 'E') {
237       freeObstacles(listaOstacoli);
238       listaOstacoli = NULL;
239       disconnettiClient(clientDesc);
240     } else if (inputFromClient == 't' || inputFromClient == 'T') {
241       write(clientDesc, &turnoFinito, sizeof(int));
242       sendTimerValue(clientDesc);
243     } else if (inputFromClient == 'l' || inputFromClient == 'L') {
244       write(clientDesc, &turnoFinito, sizeof(int));
245       sendPlayerList(clientDesc);
246     } else if (turnoGiocatore == turno) {
247       write(clientDesc, &turnoFinito, sizeof(int));
248       giocatore =
249           gestisciInput(grigliaDiGiocoConPacchiSenzaOstacoli,
250                         grigliaOstacoliSenzaPacchi, inputFromClient, giocatore,
251                         &listaOstacoli, deployCoords, packsCoords, name);
252     } else {
253       turnoFinito = 1;
254       write(clientDesc, &turnoFinito, sizeof(int));
255       freeObstacles(listaOstacoli);
256       listaOstacoli = NULL;
257       inserisciPlayerNellaGrigliaInPosizioneCasuale(
258           grigliaDiGiocoConPacchiSenzaOstacoli, grigliaOstacoliSenzaPacchi,
259           giocatore->position);
260       giocatore->score = 0;
261       giocatore->hasApack = 0;
262       giocatore->deploy[0] = -1;
263       giocatore->deploy[1] = -1;
264       turnoGiocatore = turno;
265       turnoFinito = 0;
266       playerGenerati++;
267     }
268   }
269 }
270 void sendTimerValue(int clientDesc) {
271   if (!clientDisconnesso(clientDesc))
272     write(clientDesc, &timerCount, sizeof(timerCount));
273 }
274 void clonaGriglia(char destinazione[ROWS][COLUMNS],
275                   char source[ROWS][COLUMNS]) {
276   int i = 0, j = 0;
277   for (i = 0; i < ROWS; i++) {
278     for (j = 0; j < COLUMNS; j++) {
279       destinazione[i][j] = source[i][j];
280     }
281   }
```

```
282 }
283 void clientCrashHandler(int signalNum) {
284   char msg[0];
285   int socketClientCrashato;
286   int flag = 1;
287   // TODO eliminare la lista degli ostacoli dell'utente
288   if (onLineUsers != NULL) {
289     Players prec = onLineUsers;
290     Players top = prec->next;
291     while (top != NULL && flag) {
292       if (write(top->sockDes, msg, sizeof(msg)) < 0) {
293         socketClientCrashato = top->sockDes;
294         printPlayers(onLineUsers);
295         disconnettiClient(socketClientCrashato);
296         flag = 0;
297       }
298       top = top->next;
299     }
300   }
301   signal(SIGPIPE, SIG_IGN);
302 }
303 void disconnettiClient(int clientDescriptor) {
304   if (numeroClientLoggati > 0)
305     numeroClientLoggati--;
306   pthread_mutex_lock(&PlayerMutex);
307   onLineUsers = removePlayer(onLineUsers, clientDescriptor);
308   pthread_mutex_unlock(&PlayerMutex);
309   printPlayers(onLineUsers);
310   int msg = 1;
311   printf("Client disconnesso (client attualmente loggati: %d)\n",
312          numeroClientLoggati);
313   write(clientDescriptor, &msg, sizeof(msg));
314   close(clientDescriptor);
315 }
316 int clientDisconnesso(int clientSocket) {
317   char msg[1] = {'u'}; // UP?
318   if (write(clientSocket, msg, sizeof(msg)) < 0)
319     return 1;
320   if (read(clientSocket, msg, sizeof(char)) < 0)
321     return 1;
322   else
323     return 0;
324 }
325 int registraClient(int clientDesc) {
326   char *userName = (char *)calloc(MAX_BUF, 1);
327   char *password = (char *)calloc(MAX_BUF, 1);
328   int dimName, dimPwd;
329   read(clientDesc, &dimName, sizeof(int));
330   read(clientDesc, &dimPwd, sizeof(int));
331   read(clientDesc, userName, dimName);
332   read(clientDesc, password, dimPwd);
333   pthread_mutex_lock(&RegMutex);
334   int ret = appendPlayer(userName, password, users);
335   pthread_mutex_unlock(&RegMutex);
336   char risposta;
337   if (!ret) {
338     risposta = 'n';
339     write(clientDesc, &risposta, sizeof(char));
340     printf("Impossibile registrare utente, riprovare\n");
341   } else {
342     risposta = 'y';
343     write(clientDesc, &risposta, sizeof(char));
344     printf("Utente registrato con successo\n");
345   }
346   return ret;
347 }
348 void quitServer() {
349   printf("Chiusura server in corso..\n");
350   close(socketDesc);
351   exit(-1);
352 }
353 void *threadGenerazioneMappa(void *args) {
354   fprintf(stdout, "Rigenerazione mappa\n");
355   inizializzaGrigliaVuota(grigliaDiGiocoConPacchiSenzaOstacoli);
```

```
356   generaPosizioniRaccolta(grigliaDiGiocoConPacchiSenzaOstacoli,
357                           grigliaOstacoliSenzaPacchi, deployCoords);
358   riempiGrigliaConPacchiInPosizioniGenerateCasualmente(
359       grigliaDiGiocoConPacchiSenzaOstacoli, packsCoords);
360   generaPosizioneOstacoli(grigliaDiGiocoConPacchiSenzaOstacoli,
361                           grigliaOstacoliSenzaPacchi);
362   printf("Mappa generata\n");
363   pthread_exit(NULL);
364 }
365 int almenoUnaMossaFatta() {
366   if (numMosse > 0)
367     return 1;
368   return 0;
369 }
370 int almenoUnClientConnesso() {
371   if (numeroClientLoggati > 0)
372     return 1;
373   return 0;
374 }
375 int valoreTimerValido() {
376   if (timerCount > 0 && timerCount <= TIME_LIMIT_IN_SECONDS)
377     return 1;
378   return 0;
379 }
380 int almenoUnPlayerGenerato() {
381   if (playerGenerati > 0)
382     return 1;
383   return 0;
384 }
385 void *timer(void *args) {
386   int cambiato = 1;
387   while (1) {
388     if (almenoUnClientConnesso() && valoreTimerValido() &&
389         almenoUnPlayerGenerato() && almenoUnaMossaFatta()) {
390       cambiato = 1;
391       sleep(1);
392       timerCount--;
393       fprintf(stdout, "Time left: %d\n", timerCount);
394     } else if (numeroClientLoggati == 0) {
395       timerCount = TIME_LIMIT_IN_SECONDS;
396       if (cambiato) {
397         fprintf(stdout, "Time left: %d\n", timerCount);
398         cambiato = 0;
399       }
400     }
401     if (timerCount == 0 || scoreMassimo == packageLimitNumber) {
402       playerGenerati = 0;
403       numMosse = 0;
404       printf("Reset timer e generazione nuova mappa..\n");
405       startProceduraGenrazioneMappa();
406       pthread_join(tidGeneratoreMappa, NULL);
407       turno++;
408       timerCount = TIME_LIMIT_IN_SECONDS;
409     }
410   }
411 }
412
413 void configuraSocket(struct sockaddr_in mio_indirizzo) {
414   if ((socketDesc = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
415     perror("Impossibile creare socket");
416     exit(-1);
417   }
418   if (setsockopt(socketDesc, SOL_SOCKET, SO_REUSEADDR, &(int){1}, sizeof(int)) <
419       0)
420     perror("Impossibile impostare il riutilizzo dell'indirizzo ip e della "
421            "porta\n");
422   if ((bind(socketDesc, (struct sockaddr *)&mio_indirizzo,
423             sizeof(mio_indirizzo))) < 0) {
424     perror("Impossibile effettuare bind");
425     exit(-1);
426   }
427 }
428
429 PlayerStats gestisciInput(char grigliaDiGioco[ROWS][COLUMNS],
```

```
                          char grigliaOstacoli[ROWS][COLUMNS], char input,
                          PlayerStats giocatore, Obstacles *listaOstacoli,
                          Point deployCoords[], Point packsCoords[],
                          char name[]) {
  if (giocatore == NULL) {
    return NULL;
  }
  if (input == 'w' || input == 'W') {
    giocatore = gestisciW(grigliaDiGioco, grigliaOstacoli, giocatore,
                          listaOstacoli, deployCoords, packsCoords);
  } else if (input == 's' || input == 'S') {
    giocatore = gestisciS(grigliaDiGioco, grigliaOstacoli, giocatore,
                          listaOstacoli, deployCoords, packsCoords);
  } else if (input == 'a' || input == 'A') {
    giocatore = gestisciA(grigliaDiGioco, grigliaOstacoli, giocatore,
                          listaOstacoli, deployCoords, packsCoords);
  } else if (input == 'd' || input == 'D') {
    giocatore = gestisciD(grigliaDiGioco, grigliaOstacoli, giocatore,
                          listaOstacoli, deployCoords, packsCoords);
  } else if (input == 'p' || input == 'P') {
    giocatore = gestisciP(grigliaDiGioco, giocatore, deployCoords, packsCoords);
  } else if (input == 'c' || input == 'C') {
    giocatore =
        gestisciC(grigliaDiGioco, giocatore, deployCoords, packsCoords, name);
  }

  // aggiorna la posizione dell'utente
  return giocatore;
}

PlayerStats gestisciC(char grigliaDiGioco[ROWS][COLUMNS], PlayerStats giocatore,
                      Point deployCoords[], Point packsCoords[], char name[]) {
  pthread_t tid;
  if (giocatore->hasApack == 0) {
    return giocatore;
  } else {
    if (isOnCorrectDeployPoint(giocatore, deployCoords)) {
      Args args = (Args)malloc(sizeof(struct argsToSend));
      args->userName = (char *)calloc(MAX_BUF, 1);
      strcpy(args->userName, name);
      args->flag = 1;
      pthread_create(&tid, NULL, fileWriter, (void *)args);
      giocatore->score += 10;
      if (giocatore->score > scoreMassimo)
        scoreMassimo = giocatore->score;
      giocatore->deploy[0] = -1;
      giocatore->deploy[1] = -1;
      giocatore->hasApack = 0;
    } else {
      if (!isOnAPack(giocatore, packsCoords) &&
          !isOnADeployPoint(giocatore, deployCoords)) {
        int index = getHiddenPack(packsCoords);
        if (index >= 0) {
          packsCoords[index]->x = giocatore->position[0];
          packsCoords[index]->y = giocatore->position[1];
          giocatore->hasApack = 0;
          giocatore->deploy[0] = -1;
          giocatore->deploy[1] = -1;
        }
      } else
        return giocatore;
    }
  }
  return giocatore;
}

void sendPlayerList(int clientDesc) {
  int lunghezza = 0;
  char name[100];
  Players tmp = onLineUsers;
  int numeroClientLoggati = dimensioneLista(tmp);
  printf("%d ", numeroClientLoggati);
  if (!clientDisconnesso(clientDesc)) {
    write(clientDesc, &numeroClientLoggati, sizeof(numeroClientLoggati));
```

```
    while (numeroClientLoggati > 0 && tmp != NULL) {
      strcpy(name, tmp->name);
      lunghezza = strlen(tmp->name);
      write(clientDesc, &lunghezza, sizeof(lunghezza));
      write(clientDesc, name, lunghezza);
      tmp = tmp->next;
      numeroClientLoggati--;
    }
  }
}

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;
  time_t t = time(NULL);
  struct tm *infoTime = localtime(&t);
  char toPrint[64];
  strftime(toPrint, sizeof(toPrint), "%X %x", infoTime);
  if (info->flag == 1) {
    char message[MAX_BUF] = "Pack delivered by \"";
    strcat(message, info->userName);
    char at[] = "\" at ";
    strcat(message, at);
    strcat(message, toPrint);
    strcat(message, "\n");
    pthread_mutex_lock(&LogMutex);
    write(fDes, message, strlen(message));
    pthread_mutex_unlock(&LogMutex);
  } else if (info->flag == 0) {
    char message[MAX_BUF] = "\"";
    strcat(message, info->userName);
    strcat(message, "\" logged in at ");
    strcat(message, toPrint);
    strcat(message, "\n");
    pthread_mutex_lock(&LogMutex);
    write(fDes, message, strlen(message));
    pthread_mutex_unlock(&LogMutex);
  } else if (info->flag == 2) {
    char message[MAX_BUF] = "\"";
    strcat(message, info->userName);
    strcat(message, "\" connected at ");
    strcat(message, toPrint);
    strcat(message, "\n");
    pthread_mutex_lock(&LogMutex);
    write(fDes, message, strlen(message));
    pthread_mutex_unlock(&LogMutex);
  }
  close(fDes);
  free(info);
  pthread_exit(NULL);
}
```

A.3 Codice sorgente boardUtility

Listato 14: Codice header utility del gioco 1

```
#include "list.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#define ROWS 10
#define COLUMNS 30
#define numberOfObstacles 35
#define numberOfPackages 15
#define TIME_LIMIT_IN_SECONDS 30
#define packageLimitNumber 4
```

```c
#define MATRIX_DIMENSION sizeof(char) * ROWS *COLUMNS
#define RED_COLOR "\x1b[31m"
#define GREEN_COLOR "\x1b[32m"
#define RESET_COLOR "\x1b[0m"

struct Coord {
  int x;
  int y;
};
typedef struct Coord *Point;
void rimuoviPaccoDaArray(int posizione[2], Point packsCoords[]);
void printMenu();
int getHiddenPack(Point packsCoords[]);
void stampaIstruzioni(int i);
int isOnADeployPoint(PlayerStats giocatore, Point deployCoords[]);
int isOnCorrectDeployPoint(PlayerStats giocatore, Point deployCoords[]);
PlayerStats gestisciP(char grigliaDiGioco[ROWS][COLUMNS], PlayerStats giocatore,
                      Point deployCoords[], Point packsCoords[]);
PlayerStats gestisciA(char grigliaDiGioco[ROWS][COLUMNS],
                      char grigliaOstacoli[ROWS][COLUMNS],
                      PlayerStats giocatore, Obstacles *listaOstacoli,
                      Point deployCoords[], Point packsCoords[]);
PlayerStats gestisciD(char grigliaDiGioco[ROWS][COLUMNS],
                      char grigliaOstacoli[ROWS][COLUMNS],
                      PlayerStats giocatore, Obstacles *listaOstacoli,
                      Point deployCoords[], Point packsCoords[]);
PlayerStats gestisciS(char grigliaDiGioco[ROWS][COLUMNS],
                      char grigliaOstacoli[ROWS][COLUMNS],
                      PlayerStats giocatore, Obstacles *listaOstacoli,
                      Point deployCoords[], Point packsCoords[]);
void inizializzaGiocoSenzaPlayer(char grigliaDiGioco[ROWS][COLUMNS],
                                 char grigliaConOstacoli[ROWS][COLUMNS],
                                 Point packsCoords[]);
void inserisciPlayerNellaGrigliaInPosizioneCasuale(
    char grigliaDiGioco[ROWS][COLUMNS], char grigliaOstacoli[ROWS][COLUMNS],
    int posizione[2]);
void inizializzaGrigliaVuota(char grigliaDiGioco[ROWS][COLUMNS]);
void generaPosizioneOstacoli(char grigliaDiGioco[ROWS][COLUMNS],
                             char grigliaOstacoli[ROWS][COLUMNS]);
void riempiGrigliaConPacchiInPosizioniGenerateCasualmente(
    char grigliaDiGioco[ROWS][COLUMNS], Point packsCoords[]);
void printGrid(char grigliaDaStampare[ROWS][COLUMNS], PlayerStats stats);
void start(char grigliaDiGioco[ROWS][COLUMNS],
           char grigliaOstacoli[ROWS][COLUMNS]);
void riempiGrigliaConGliOstacoli(char grigliaDiGioco[ROWS][COLUMNS],
                                 char grigliaOstacoli[ROWS][COLUMNS]);
void generaPosizioniRaccolta(char grigliaDiGioco[ROWS][COLUMNS],
                             char grigliaOstacoli[ROWS][COLUMNS],
                             Point coord[]);
PlayerStats gestisciW(char grigliaDiGioco[ROWS][COLUMNS],
                      char grigliaOstacoli[ROWS][COLUMNS],
                      PlayerStats giocatore, Obstacles *listaOstacoli,
                      Point deployCoords[], Point packsCoords[]);
void mergeGridAndList(char grid[ROWS][COLUMNS], Obstacles top);
void scegliPosizioneRaccolta(Point coord[], int deploy[]);
int colpitoOstacolo(char grigliaOstacoli[ROWS][COLUMNS], int posizione[2]);
int colpitoPacco(Point packsCoords[], int posizione[2]);
int colpitoPlayer(char grigliaDiGioco[ROWS][COLUMNS], int posizione[2]);
int casellaVuota(char grigliaDiGioco[ROWS][COLUMNS],
                 char grigliaOstacoli[ROWS][COLUMNS], int posizione[2]);
void spostaPlayer(char griglia[ROWS][COLUMNS], int vecchiaPosizione[2],
                  int nuovaPosizione[2], Point deployCoords[],
                  Point packsCoords[]);
int arrivatoADestinazione(int posizione[2], int destinazione[2]);
int eraUnPuntoDepo(int vecchiaPosizione[2], Point depo[]);
int eraUnPacco(int vecchiaPosizione[2], Point packsCoords[]);
int isOnAPack(PlayerStats giocatore, Point packsCoords[]);
```

Listato 15: Codice sorgente utility del gioco 1

```c
#include "boardUtility.h"
#include "list.h"
#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]] == '-' ||
30       grigliaDiGioco[posizione[0]][posizione[1]] == '_' ||
31       grigliaDiGioco[posizione[0]][posizione[1]] == '$')
32     if (grigliaOstacoli[posizione[0]][posizione[1]] == '-' ||
33         grigliaOstacoli[posizione[0]][posizione[1]] == '_' ||
34         grigliaOstacoli[posizione[0]][posizione[1]] == '$')
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 void printGrid(char grigliaDaStampare[ROWS][COLUMNS], PlayerStats stats) {
103   system("clear");
104   printf("\n\n");
105   int i = 0, j = 0;
106   for (i = 0; i < ROWS; i++) {
107     printf("\t");
108     for (j = 0; j < COLUMNS; j++) {
109       if (stats != NULL) {
110         if ((i == stats->deploy[0] && j == stats->deploy[1]) ||
111             (i == stats->position[0] && j == stats->position[1]))
112           if (grigliaDaStampare[i][j] == 'P' && stats->hasApack == 1)
113             printf(GREEN_COLOR "%c" RESET_COLOR, grigliaDaStampare[i][j]);
114           else
115             printf(RED_COLOR "%c" RESET_COLOR, grigliaDaStampare[i][j]);
116         else
117           printf("%c", grigliaDaStampare[i][j]);
118       } else
119         printf("%c", grigliaDaStampare[i][j]);
120     }
121     stampaIstruzioni(i);
122     if (i == 8)
123       printf(GREEN_COLOR "\t\t Punteggio: %d" RESET_COLOR, stats->score);
124     printf("\n");
125   }
126 }
127 void stampaIstruzioni(int i) {
128   if (i == 0)
129     printf("\t \t ISTRUZIONI ");
130   if (i == 1)
131     printf("\t Inviare 't' per il timer.");
132   if (i == 2)
133     printf("\t Inviare 'e' per uscire");
134   if (i == 3)
135     printf("\t Inviare 'p' per raccogliere un pacco");
136   if (i == 4)
137     printf("\t Inviare 'c' per consegnare il pacco");
138   if (i == 5)
139     printf("\t Inviare 'w'/'s' per andare sopra/sotto");
140   if (i == 6)
141     printf("\t Inviare 'a'/'d' per andare a dx/sx");
142   if (i == 7)
143     printf("\t Inviare 'l' per la lista degli utenti ");
144 }
145 // aggiunge alla griglia gli ostacoli visti fino ad ora dal client
146 void mergeGridAndList(char grid[ROWS][COLUMNS], Obstacles top) {
147   while (top) {
148     grid[top->x][top->y] = 'O';
149     top = top->next;
150   }
151 }
```

```
152 /* Genera la posizione degli ostacoli */
153 void generaPosizioneOstacoli(char grigliaDiGioco[ROWS][COLUMNS],
154                              char grigliaOstacoli[ROWS][COLUMNS]) {
155   int x, y, i;
156   inizializzaGrigliaVuota(grigliaOstacoli);
157   srand(time(0));
158   for (i = 0; i < numberOfObstacles; i++) {
159     x = rand() % COLUMNS;
160     y = rand() % ROWS;
161     if (grigliaDiGioco[y][x] == '-')
162       grigliaOstacoli[y][x] = 'O';
163     else
164       i--;
165   }
166 }
167 void rimuoviPaccoDaArray(int posizione[2], Point packsCoords[]) {
168   int i = 0, found = 0;
169   while (i < numberOfPackages && !found) {
170     if ((packsCoords[i])->x == posizione[0] &&
171         (packsCoords[i])->y == posizione[1]) {
172       (packsCoords[i])->x = -1;
173       (packsCoords[i])->y = -1;
174       found = 1;
175     }
176     i++;
177   }
178 }
179 // sceglie una posizione di raccolta tra quelle disponibili
180 void scegliPosizioneRaccolta(Point coord[], int deploy[]) {
181   int index = 0;
182   srand(time(NULL));
183   index = rand() % numberOfPackages;
184   deploy[0] = coord[index]->x;
185   deploy[1] = coord[index]->y;
186 }
187 /*genera posizione di raccolta di un pacco*/
188 void generaPosizioniRaccolta(char grigliaDiGioco[ROWS][COLUMNS],
189                              char grigliaOstacoli[ROWS][COLUMNS],
190                              Point coord[]) {
191   int x, y;
192   srand(time(0));
193   int i = 0;
194   for (i = 0; i < numberOfPackages; i++) {
195     coord[i] = (Point)malloc(sizeof(struct Coord));
196   }
197   i = 0;
198   for (i = 0; i < numberOfPackages; i++) {
199     x = rand() % COLUMNS;
200     y = rand() % ROWS;
201     if (grigliaDiGioco[y][x] == '-' && grigliaOstacoli[y][x] == '-') {
202       coord[i]->x = y;
203       coord[i]->y = x;
204       grigliaDiGioco[y][x] = '_';
205       grigliaOstacoli[y][x] = '_';
206     } else
207       i--;
208   }
209 }
210 /*Inserisci dei pacchi nella griglia di gioco nella posizione casuale */
211 void riempiGrigliaConPacchiInPosizioniGenerateCasualmente(
212     char grigliaDiGioco[ROWS][COLUMNS], Point packsCoords[]) {
213   int x, y, i = 0;
214   for (i = 0; i < numberOfPackages; i++) {
215     packsCoords[i] = (Point)malloc(sizeof(struct Coord));
216   }
217   srand(time(0));
218   for (i = 0; i < numberOfPackages; i++) {
219     x = rand() % COLUMNS;
220     y = rand() % ROWS;
221     if (grigliaDiGioco[y][x] == '-') {
222       grigliaDiGioco[y][x] = '$';
223       packsCoords[i]->x = y;
224       packsCoords[i]->y = x;
225     } else
```

```
226       i--;
227     }
228   }
229   /*Inserisci gli ostacoli nella griglia di gioco */
230   void riempiGrigliaConGliOstacoli(char grigliaDiGioco[ROWS][COLUMNS],
231                                    char grigliaOstacoli[ROWS][COLUMNS]) {
232     int i, j = 0;
233     for (i = 0; i < ROWS; i++) {
234       for (j = 0; j < COLUMNS; j++) {
235         if (grigliaOstacoli[i][j] == 'O')
236           grigliaDiGioco[i][j] = 'O';
237       }
238     }
239   }
240   void inserisciPlayerNellaGrigliaInPosizioneCasuale(
241       char grigliaDiGioco[ROWS][COLUMNS], char grigliaOstacoli[ROWS][COLUMNS],
242       int posizione[2]) {
243     int x, y;
244     srand(time(0));
245     printf("Inserisco player\n");
246     do {
247       x = rand() % COLUMNS;
248       y = rand() % ROWS;
249     } while (grigliaDiGioco[y][x] != '-' && grigliaOstacoli[y][x] != '-');
250     grigliaDiGioco[y][x] = 'P';
251     posizione[0] = y;
252     posizione[1] = x;
253   }
254   void inizializzaGiocoSenzaPlayer(char grigliaDiGioco[ROWS][COLUMNS],
255                                    char grigliaConOstacoli[ROWS][COLUMNS],
256                                    Point packsCoords[]) {
257     inizializzaGrigliaVuota(grigliaDiGioco);
258     riempiGrigliaConPacchiInPosizioniGenerateCasualmente(grigliaDiGioco,
259                                                          packsCoords);
260     generaPosizioneOstacoli(grigliaDiGioco, grigliaConOstacoli);
261     return;
262   }
263   void spostaPlayer(char griglia[ROWS][COLUMNS], int vecchiaPosizione[2],
264                     int nuovaPosizione[2], Point deployCoords[],
265                     Point packsCoords[]) {
266     griglia[nuovaPosizione[0]][nuovaPosizione[1]] = 'P';
267     if (eraUnPuntoDepo(vecchiaPosizione, deployCoords))
268       griglia[vecchiaPosizione[0]][vecchiaPosizione[1]] = '_';
269     else if (eraUnPacco(vecchiaPosizione, packsCoords))
270       griglia[vecchiaPosizione[0]][vecchiaPosizione[1]] = '$';
271     else
272       griglia[vecchiaPosizione[0]][vecchiaPosizione[1]] = '-';
273   }
274   int eraUnPuntoDepo(int vecchiaPosizione[2], Point depo[]) {
275     int i = 0, ret = 0;
276     while (ret == 0 && i < numberOfPackages) {
277       if ((depo[i])->y == vecchiaPosizione[1] &&
278           (depo[i])->x == vecchiaPosizione[0]) {
279         ret = 1;
280       }
281       i++;
282     }
283     return ret;
284   }
285   int eraUnPacco(int vecchiaPosizione[2], Point packsCoords[]) {
286     int i = 0, ret = 0;
287     while (ret == 0 && i < numberOfPackages) {
288       if ((packsCoords[i])->y == vecchiaPosizione[1] &&
289           (packsCoords[i])->x == vecchiaPosizione[0]) {
290         ret = 1;
291       }
292       i++;
293     }
294     return ret;
295   }
296   PlayerStats gestisciW(char grigliaDiGioco[ROWS][COLUMNS],
297                         char grigliaOstacoli[ROWS][COLUMNS],
298                         PlayerStats giocatore, Obstacles *listaOstacoli,
299                         Point deployCoords[], Point packsCoords[]) {
```

```
  if (giocatore == NULL)
    return NULL;
  int nuovaPosizione[2];
  nuovaPosizione[1] = giocatore->position[1];
  // Aggiorna la posizione vecchia spostando il player avanti di 1
  nuovaPosizione[0] = (giocatore->position[0]) - 1;
  int nuovoScore = giocatore->score;
  int nuovoDeploy[2];
  nuovoDeploy[0] = giocatore->deploy[0];
  nuovoDeploy[1] = giocatore->deploy[1];
  if (nuovaPosizione[0] >= 0 && nuovaPosizione[0] < ROWS) {
    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];
```

```
374   nuovaPosizione[0] = giocatore->position[0];
375   // Aggiorna la posizione vecchia spostando il player avanti di 1
376   nuovaPosizione[1] = (giocatore->position[1]) - 1;
377   int nuovoScore = giocatore->score;
378   int nuovoDeploy[2];
379   nuovoDeploy[0] = giocatore->deploy[0];
380   nuovoDeploy[1] = giocatore->deploy[1];
381   if (nuovaPosizione[1] >= 0 && nuovaPosizione[1] < COLUMNS) {
382     if (casellaVuotaOValida(grigliaDiGioco, grigliaOstacoli, nuovaPosizione)) {
383       printf("Casella vuota \n");
384       spostaPlayer(grigliaDiGioco, giocatore->position, nuovaPosizione,
385                    deployCoords, packsCoords);
386     } else if (colpitoOstacolo(grigliaOstacoli, nuovaPosizione)) {
387       printf("Ostacolo\n");
388       *listaOstacoli =
389           addObstacle(*listaOstacoli, nuovaPosizione[0], nuovaPosizione[1]);
390       nuovaPosizione[0] = giocatore->position[0];
391       nuovaPosizione[1] = giocatore->position[1];
392     } else if (colpitoPlayer(grigliaDiGioco, nuovaPosizione)) {
393       printf("colpito player\n");
394       nuovaPosizione[0] = giocatore->position[0];
395       nuovaPosizione[1] = giocatore->position[1];
396     }
397     giocatore->deploy[0] = nuovoDeploy[0];
398     giocatore->deploy[1] = nuovoDeploy[1];
399     giocatore->score = nuovoScore;
400     giocatore->position[0] = nuovaPosizione[0];
401     giocatore->position[1] = nuovaPosizione[1];
402   }
403   return giocatore;
404 }
405 PlayerStats gestisciS(char grigliaDiGioco[ROWS][COLUMNS],
406                       char grigliaOstacoli[ROWS][COLUMNS],
407                       PlayerStats giocatore, Obstacles *listaOstacoli,
408                       Point deployCoords[], Point packsCoords[]) {
409   if (giocatore == NULL) {
410     return NULL;
411   }
412   // crea le nuove statistiche
413   int nuovaPosizione[2];
414   nuovaPosizione[1] = giocatore->position[1];
415   nuovaPosizione[0] = (giocatore->position[0]) + 1;
416   int nuovoScore = giocatore->score;
417   int nuovoDeploy[2];
418   nuovoDeploy[0] = giocatore->deploy[0];
419   nuovoDeploy[1] = giocatore->deploy[1];
420   // controlla che le nuove statistiche siano corrette
421   if (nuovaPosizione[0] >= 0 && nuovaPosizione[0] < ROWS) {
422     if (casellaVuotaOValida(grigliaDiGioco, grigliaOstacoli, nuovaPosizione)) {
423       spostaPlayer(grigliaDiGioco, giocatore->position, nuovaPosizione,
424                    deployCoords, packsCoords);
425     } else if (colpitoOstacolo(grigliaOstacoli, nuovaPosizione)) {
426       printf("Ostacolo\n");
427       *listaOstacoli =
428           addObstacle(*listaOstacoli, nuovaPosizione[0], nuovaPosizione[1]);
429       nuovaPosizione[0] = giocatore->position[0];
430       nuovaPosizione[1] = giocatore->position[1];
431     } else if (colpitoPlayer(grigliaDiGioco, nuovaPosizione)) {
432       nuovaPosizione[0] = giocatore->position[0];
433       nuovaPosizione[1] = giocatore->position[1];
434     }
435     giocatore->deploy[0] = nuovoDeploy[0];
436     giocatore->deploy[1] = nuovoDeploy[1];
437     giocatore->score = nuovoScore;
438     giocatore->position[0] = nuovaPosizione[0];
439     giocatore->position[1] = nuovaPosizione[1];
440   }
441   return giocatore;
442 }
443 int arrivatoADestinazione(int posizione[2], int destinazione[2]) {
444   if (posizione[0] == destinazione[0] && posizione[1] == destinazione[1])
445     return 1;
446   return 0;
447 }
```

A.4 Codice sorgente list

Listato 16: 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);

// Inizializza un nuovo nodo ostacolo
```

```
70 Obstacles initObstacleNode(int x, int y);
71 #endif
```

Listato 17: 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);
83    }
84  }
85  void printPlayers(Players L) {
86    if (L != NULL) {
87      printf("%s ->", L->name);
88      printPlayers(L->next);
89    }
90    printf("\n");
91  }
92  void printObstacles(Obstacles L) {
93    if (L != NULL) {
94      printf("X:%d Y:%d ->", L->x, L->y);
95      printObstacles(L->next);
96    }
97  }
```

A.5 Codice sorgente parser

Listato 18: Codice header utility del gioco 3

```
1  int appendPlayer(char *name, char *pwd, char *file);
2  int isRegistered(char *name, char *file);
3  int openFileRDWRAPP(char *file);
4  int validateLogin(char *name, char *pwd, char *file);
5  int openFileRDON(char *file);
6  void premiEnterPerContinuare();
```

Listato 19: Codice sorgente utility del gioco 3

```
1  #include "parser.h"
2  #include <errno.h>
3  #include <fcntl.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7  #include <sys/stat.h>
8  #include <sys/types.h>
9  #include <unistd.h>
10 #define MAX_BUF 200
11 int openFileRDWRAPP(char *file) {
12   int fileDes = open(file, O_RDWR | O_CREAT | O_APPEND, S_IRUSR | S_IWUSR);
13   if (fileDes < 0)
14     perror("Errore apertura file\n"), exit(-1);
15   return fileDes;
16 }
17 int openFileRDON(char *file) {
18   int fileDes = open(file, O_RDONLY);
19   if (fileDes < 0)
20     perror("Errore apertura file\n"), exit(-1);
21   return fileDes;
22 }
23 int appendPlayer(char *name, char *pwd, char *file) {
24   if (isRegistered(name, file))
25     return 0;
26   int fileDes = openFileRDWRAPP(file);
```

```
27    write(fileDes, name, strlen(name));
28    write(fileDes, " ", 1);
29    write(fileDes, pwd, strlen(pwd));
30    write(fileDes, "\n", 1);
31    close(fileDes);
32    return 1;
33  }
34  int isRegistered(char *name, char *file) {
35    char command[MAX_BUF] = "cat  ";
36    strcat(command, file);
37    char toApp[] = " |cut -d\" \" -f1|grep \"^";
38    strcat(command, toApp);
39    strcat(command, name);
40    char toApp2[] = "$\">tmp";
41    strcat(command, toApp2);
42    int ret = 0;
43    system(command);
44    int fileDes = openFileRDON("tmp");
45    struct stat info;
46    fstat(fileDes, &info);
47    if ((int)info.st_size > 0)
48      ret = 1;
49    close(fileDes);
50    system("rm tmp");
51    return ret;
52  }
53  int validateLogin(char *name, char *pwd, char *file) {
54    if (!isRegistered(name, file))
55      return 0;
56    char command[MAX_BUF] = "cat  ";
57    strcat(command, file);
58    char toApp[] = " |grep \"^";
59    strcat(command, toApp);
60    strcat(command, name);
61    strcat(command, " ");
62    strcat(command, pwd);
63    char toApp2[] = "$\">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