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 Traccia

Descrizione Sintetica

Realizzare un sistema client-server che consenta a più utenti di prendere e portare oggetti da una locazione di partenza ad una destinazione indicata. Scopo del gioco è consegnare più oggetti alla destinazione.

Si utilizzi il linguaggio C su piattaforma UNIX. I processi dovranno comunicare tramite socket TCP. Corredare l'implementazione di adeguata documentazione.

Descrizione Dettagliata

Il server manterrà una rappresentazione dell'ambiente in cui verranno posizionati degli oggetti, delle locazioni e degli ostacoli. L'ambiente sarà rappresentato da una matrice in cui gli utenti si potranno spostare di un passo alla volta nelle quattro direzioni: S, N, E, O oppure prendere o depositare oggetti (es. con azioni P, D). Il server posizionerà nella matrice locazioni, oggetti ed ostacoli in posizioni random. Ogni oggetto avrà associata una locazione in cui portarlo (indicata da un opportuno nome simbolico, es. L1, L2, etc.). Ogni utente, una volta connesso al server, verrà posizionato in una posizione random della matrice. All'inizio del gioco gli ostacoli sulla mappa saranno nascosta per l'utente, saranno invece visibili le posizioni degli altri utenti, degli oggetti e delle possibili locazioni. Il gioco durerà un tempo fissato a partire dal primo utente che inizierà a giocare. Gli utenti potranno inserirsi nel gioco anche a gioco già iniziato. Dopo ogni passo l'utente riceverà l'informazione sull'effetto proprio movimento: se lo spostamento porterà ad una collisione con un ostacolo oppure con un altro utente, il movimento avrà effetto nullo. In corrispondenza di un oggetto l'utente potrà prendere tale oggetto e leggere la locazione di destinazione. Dovrà quindi portarlo in tale locazione muovendosi sulla mappa per poi posarlo. Quando uno degli utenti avrà consegnato un numero massimo di pacchi, o alla scadenza di un limite di tempo fissato, il server notificherà agli utenti la fine della sessione e ne genererà una nuova.

Per accedere al servizio ogni utente dovrà prima registrarsi al sito indicando password e nickname.

Non c'è un limite a priori al numero di utenti che si possono collegare con il server.
Il client consentirà all'utente di collegarsi ad un server di comunicazione, indicando tramite riga di comando il nome o l'indirizzo IP di tale server e la porta da utilizzare. Una volta collegato ad un server l'utente potrà: registrarsi come nuovo utente o accedere al servizio come utente registrato. Il servizio permetterà all'utente di: spostarsi di una posizione, disconnettersi, vedere la lista degli utenti collegati, vedere il tempo mancante, vedere gli ostacoli incontrati e la posizione degli altri utenti.

Il server dovrà supportare tutte le funzionalità descritte nella sezione relativa al client. All'avvio del server, sarà possibile specificare tramite riga di comando la porta TCP sulla quale mettersi in ascolto.
Il server sarà di tipo concorrente, ovvero e in grado di servire più client simultanemente.

Durante il suo regolare funzionamento, il server effettuerà logging delle attivita principali in un file apposito. Ad esempio, memorizzando data e ora di connessione dei client, il loro nome simbolico (se disponibile, altrimenti l'indirizzo IP), data e ora della consegna dei pacchi insieme al nome dell'utente che ha consegnato.

Regole generali

Il server ed il client vanno realizzati in linguaggio C su piattaforma UNIX/Linux. Le comunicazioni tra client e server si svolgono tramite socket TCP. Oltre alle system call UNIX, i programmi possono utilizzare solo la libreria standard del C. E' sconsigliato l'uso di primitive non coperte dal corso (ad es., code di messaggi) al posto di quelle studiate.

2 Istruzioni preliminari

2.1 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
```

3 Guida all'uso

3.1 Server

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

```
./server users porta
```

L'identificativo *users* si 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*.
L'identificativo *porta* si riferisce alla porta che vogliamo usare per il server.

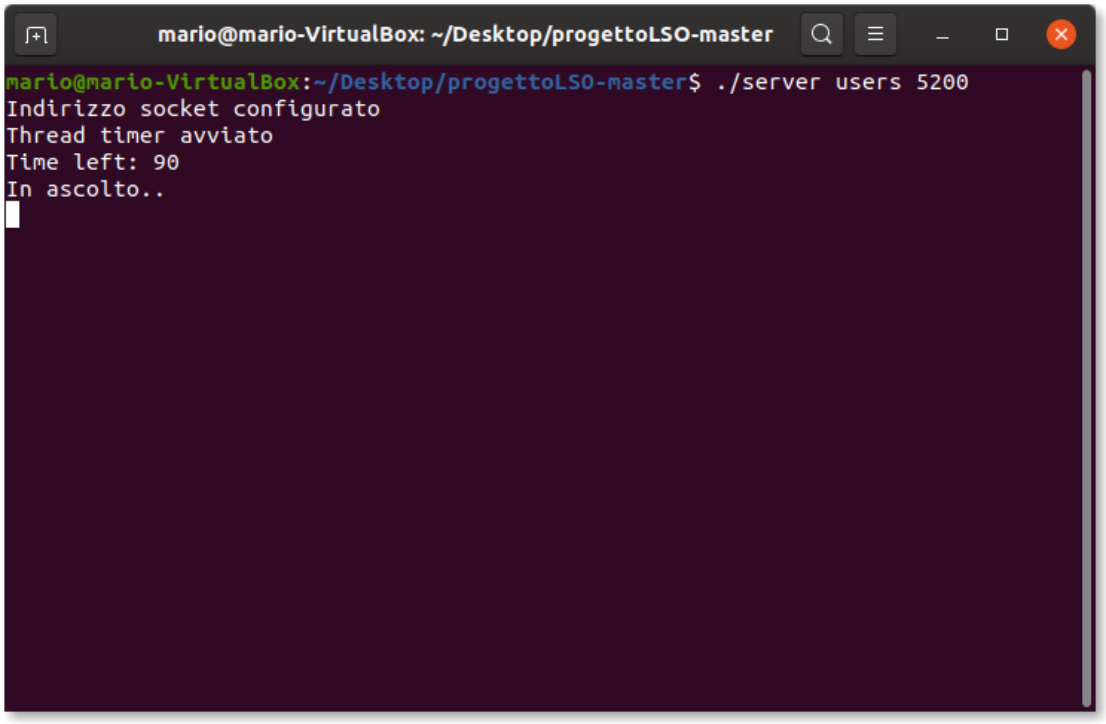

Figura 1: Menù del Server

3.2 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: gioca, registrati ed esci.

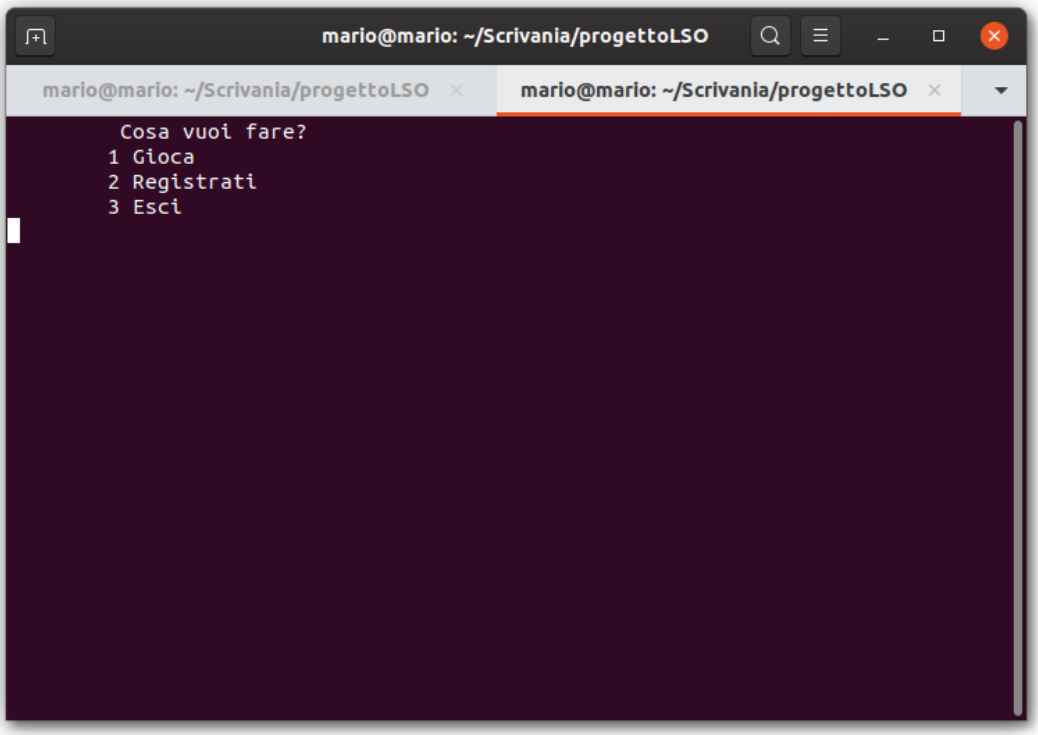

Figura 2: Menù del client.

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.

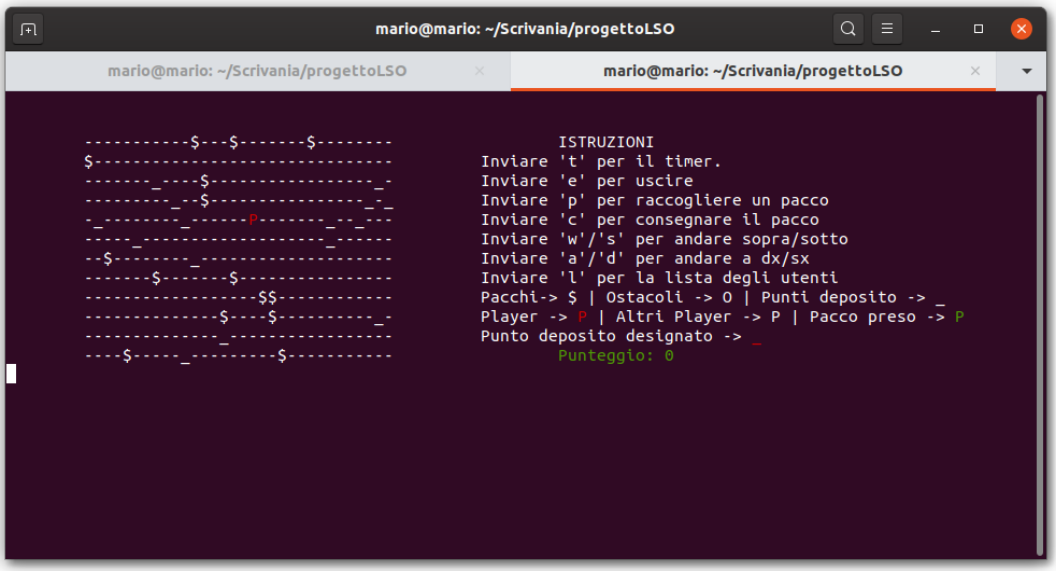

Figura 3: Il player sarà evidenziato da una P rossa.

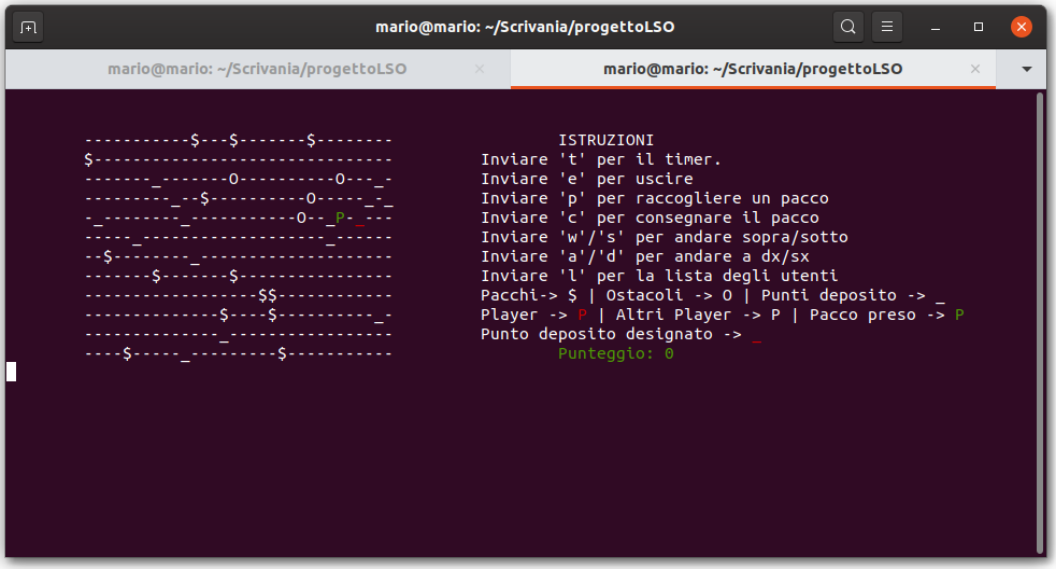

Figura 4: Una volta preso un pacco la P del player diventerà verde mentre il punto di deposito sarà evidenziato in rosso.

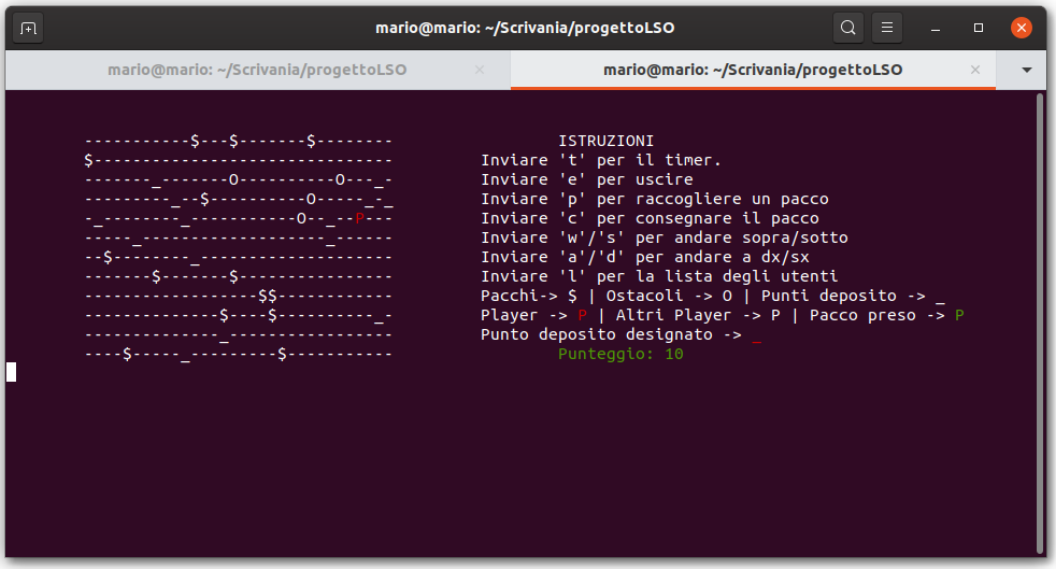

Figura 5: Una volta consegnato il pacco, la P tornerà rossa ed il punteggio verrà incrementato. Da notare alcuni ostacoli scoperti marcati con una 'O'.

Lo scopo del gioco è quello di raccogliere il numero massimo di pacchi prima dello scadere del tempo. Il server comunicherà agli utenti l'inizio di un nuovo turno allo scadere del tempo oppure quando un utente avrà consegnato il numero massimo di pacchi.

4 Comunicazione tra client e server

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

4.1 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(int port) {
  struct sockaddr_in mio_indirizzo;
  mio_indirizzo.sin_family = AF_INET;
  mio_indirizzo.sin_port = htons(port);
  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);
    }
    // Estrapolazione indirizzo ip del client
    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;
    // avvio thread di scrittura dell'indirizzo sul file di Log
    pthread_create(&tid, NULL, fileWriter, (void *)args);

    puntClientDesc = (int *)malloc(sizeof(int));
    *puntClientDesc = clientDesc;
    // avvio del thread di gestione del client
    pthread_create(&tid, NULL, gestisci, (void *)puntClientDesc);
  }
  close(clientDesc);
```

```
38   quitServer();
39 }
```

In particolare al rigo 34 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.

4.2 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.

4.3 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.

4.3.1 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 le quali vanno 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      if (read(client_sd, bufferReceive, sizeof(bufferReceive)) < 1) {
8        continua = 0;
9        break;
10     }
11     if (bufferReceive[0] == 2)
12       registraClient(client_sd);
13     else if (bufferReceive[0] == 1) {
14       if (tryLogin(client_sd, name)) {
15         play(client_sd, name);
16         continua = 0;
17       }
18     } else if (bufferReceive[0] == 3) {
19       disconnettiClient(client_sd, NULL);
20       continua = 0;
21     } else {
22       printf("Input invalido\n");
23     }
24   }
25   pthread_exit(0);
26 }
```

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      choice = getUserInput();
6      system("clear");
7      if (choice == '3') {
8        esciDalServer();
9        return (0);
10     } else if (choice == '2') {
11       registrati();
12     } else if (choice == '1') {
13       if (tryLogin())
14         play();
15     } else
16       printf("Input errato, inserire 1,2 o 3\n");
17   }
18 }
```

5 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.

5.1 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
- Il player con le relative informazioni
- Un messaggio che notifica al client se è iniziato un nuovo turno oppure no

Dopodichè il thread del server rimane in attesa di ricevere l'input del client per spostare il giocatore sulla mappa tramite la relativa funzione. (Rigo 35) 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 73) quando viene effettuato il cambio di mappa ed inviare il tempo rimanente o la lista degli utente loggati al client su sua richiesta.

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 viene mandata una copia temporanea della matrice di gioco a cui vengono aggiunti gli ostacoli già scoperti dal quello specifico client (dai righi 24 a 26),
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    int true = 1;
3    int turnoFinito = 0;
4    int turnoGiocatore = turno;
5    int posizione[2];
6    int destinazione[2] = {-1, -1};
7    PlayerStats giocatore = initStats(destinazione, 0, posizione, 0);
8    Obstacles listaOstacoli = NULL;
9    char inputFromClient;
10   if (timer != 0) {
11     inserisciPlayerNellaGrigliaInPosizioneCasuale(
12         grigliaDiGiocoConPacchiSenzaOstacoli, grigliaOstacoliSenzaPacchi,
13         giocatore->position);
14     pthread_mutex_lock(&PlayerGeneratiMutex);
15     playerGenerati++;
16     pthread_mutex_unlock(&PlayerGeneratiMutex);
17   }
18   while (true) {
19     if (clientDisconnesso(clientDesc)) {
20       freeObstacles(listaOstacoli);
21       disconnettiClient(clientDesc, giocatore);
22       return;
23     }
24     char grigliaTmp[ROWS][COLUMNS];
25     clonaGriglia(grigliaTmp, grigliaDiGiocoConPacchiSenzaOstacoli);
26     mergeGridAndList(grigliaTmp, listaOstacoli);
27     // invia la griglia
28     write(clientDesc, grigliaTmp, sizeof(grigliaTmp));
29     // invia la struttura del player
30     write(clientDesc, giocatore->deploy, sizeof(giocatore->deploy));
31     write(clientDesc, giocatore->position, sizeof(giocatore->position));
32     write(clientDesc, &giocatore->score, sizeof(giocatore->score));
33     write(clientDesc, &giocatore->hasApack, sizeof(giocatore->hasApack));
34     // legge l'input
35     if (read(clientDesc, &inputFromClient, sizeof(char)) > 0) {
36       pthread_mutex_lock(&numMosseMutex);
37       numMosse++;
38       pthread_mutex_unlock(&numMosseMutex);
39     }
40     if (inputFromClient == 'e' || inputFromClient == 'E') {
41       freeObstacles(listaOstacoli);
42       listaOstacoli = NULL;
43       disconnettiClient(clientDesc, giocatore);
44     } else if (inputFromClient == 't' || inputFromClient == 'T') {
45       write(clientDesc, &turnoFinito, sizeof(int));
46       sendTimerValue(clientDesc);
```

```
47      } else if (inputFromClient == 'l' || inputFromClient == 'L') {
48        write(clientDesc, &turnoFinito, sizeof(int));
49        sendPlayerList(clientDesc);
50      } else if (turnoGiocatore == turno) {
51        write(clientDesc, &turnoFinito, sizeof(int));
52        giocatore =
53            gestisciInput(grigliaDiGiocoConPacchiSenzaOstacoli,
54                          grigliaOstacoliSenzaPacchi, inputFromClient, giocatore,
55                          &listaOstacoli, deployCoords, packsCoords, name);
56      } else {
57        turnoFinito = 1;
58        write(clientDesc, &turnoFinito, sizeof(int));
59        freeObstacles(listaOstacoli);
60        listaOstacoli = NULL;
61        inserisciPlayerNellaGrigliaInPosizioneCasuale(
62            grigliaDiGiocoConPacchiSenzaOstacoli, grigliaOstacoliSenzaPacchi,
63            giocatore->position);
64        giocatore->score = 0;
65        giocatore->hasApack = 0;
66        giocatore->deploy[0] = -1;
67        giocatore->deploy[1] = -1;
68        turnoGiocatore = turno;
69        turnoFinito = 0;
70        pthread_mutex_lock(&PlayerGeneratiMutex);
71        playerGenerati++;
72        pthread_mutex_unlock(&PlayerGeneratiMutex);
73      }
74    }
75  }
```

5.2 Funzione core del client

Dall'altro lato, la funzione play del client, è stata mantenuta il può semplice possibile. Lo scopo del client è unicamente quello di ricevere dal server la mappa di gioco e le informazioni sul player, 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.

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  }
```

6 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.

6.1 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 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       pthread_mutex_lock(&PlayerGeneratiMutex);
19       playerGenerati = 0;
20       pthread_mutex_unlock(&PlayerGeneratiMutex);
21       pthread_mutex_lock(&numMosseMutex);
22       numMosse = 0;
23       pthread_mutex_unlock(&numMosseMutex);
24       printf("Reset timer e generazione nuova mappa..\n");
25       startProceduraGenrazioneMappa();
26       pthread_join(tidGeneratoreMappa, NULL);
27       turno++;
28       pthread_mutex_lock(&ScoreMassimoMutex);
29       scoreMassimo = 0;
30       pthread_mutex_unlock(&ScoreMassimoMutex);
31       timerCount = TIME_LIMIT_IN_SECONDS;
32     }
33   }
34 }
```

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    fprintf(stdout, "Rigenerazione mappa\n");
3    inizializzaGrigliaVuota(grigliaDiGiocoConPacchiSenzaOstacoli);
4    generaPosizioniRaccolta(grigliaDiGiocoConPacchiSenzaOstacoli,
5                            grigliaOstacoliSenzaPacchi, deployCoords);
6    riempiGrigliaConPacchiInPosizioniGenerateCasualmente(
7        grigliaDiGiocoConPacchiSenzaOstacoli, packsCoords);
8    generaPosizioneOstacoli(grigliaDiGiocoConPacchiSenzaOstacoli,
9                            grigliaOstacoliSenzaPacchi);
```

[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 6 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

```
  printf("Mappa generata\n");
  pthread_exit(NULL);
}
```

6.2 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.

6.3 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.

6.4 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: "Gestione 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: "Gestione 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);
  strcat(command, name);
  strcat(command, " ");
  strcat(command, pwd);
  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");
```

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

```
167   }
168 }
169 void printTimer() {
170   int timer;
171   if (!serverCaduto(socketDesc)) {
172     read(socketDesc, &timer, sizeof(timer));
173     printf("\t\tTempo restante: %d...\n", timer);
174     sleep(1);
175   }
176 }
177 int getTimer() {
178   int timer;
179   if (!serverCaduto(socketDesc))
180     read(socketDesc, &timer, sizeof(timer));
181   return timer;
182 }
183 int tryLogin() {
184   int msg = 1;
185   write(socketDesc, &msg, sizeof(int));
186   system("clear");
187   printf("Inserisci i dati per il Login\n");
188   char username[20];
189   char password[20];
190   printf("Inserisci nome utente(MAX 20 caratteri): ");
191   scanf(" %s", username);
192   printf("\nInserisci password(MAX 20 caratteri):");
193   scanf(" %s", password);
194   int dimUname = strlen(username), dimPwd = strlen(password);
195   if (write(socketDesc, &dimUname, sizeof(dimUname)) < 0)
196     serverCrashHandler();
197   if (write(socketDesc, &dimPwd, sizeof(dimPwd)) < 0)
198     serverCrashHandler();
199   if (write(socketDesc, username, dimUname) < 0)
200     serverCrashHandler();
201   if (write(socketDesc, password, dimPwd) < 0)
202     serverCrashHandler();
203   char validate;
204   int ret;
205   if (read(socketDesc, &validate, 1) < 0)
206     serverCrashHandler();
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   printf("\nChiusura client...\n");
265   write(socketDesc, &msg, sizeof(msg));
266   close(socketDesc);
267   signal(SIGINT, SIG_IGN);
268   signal(SIGQUIT, SIG_IGN);
269   signal(SIGTERM, SIG_IGN);
270   signal(SIGTSTP, SIG_IGN);
271   exit(0);
272 }
273 void serverCrashHandler() {
274   system("clear");
275   printf("Il server á stato spento o á irraggiungibile\n");
276   close(socketDesc);
277   signal(SIGPIPE, SIG_IGN);
278   premiEnterPerContinuare();
279   exit(0);
280 }
281 char getUserInput() {
282   char line[MAX_BUF];
283   fgets(line, sizeof(line), stdin);
284   return line[0];
285 }
```

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   char *userName;
23   int flag;
24 };
25
26 typedef struct argsToSend *Args;
27 void prepareMessageForLogin(char message[], char username[], char date[]);
28 void sendPlayerList(int clientDesc);
29 PlayerStats gestisciC(char grigliaDiGioco[ROWS][COLUMNS], PlayerStats giocatore,
30                       Point deployCoords[], Point packsCoords[], char name[]);
31 PlayerStats gestisciInput(char grigliaDiGioco[ROWS][COLUMNS],
32                           char grigliaOstacoli[ROWS][COLUMNS], char input,
33                           PlayerStats giocatore, Obstacles *listaOstacoli,
34                           Point deployCoords[], Point packsCoords[],
35                           char name[]);
36 void clonaGriglia(char destinazione[ROWS][COLUMNS], char source[ROWS][COLUMNS]);
37 int almenoUnClientConnesso();
38 void prepareMessageForConnection(char message[], char ipAddress[], char date[]);
39 void spostaPlayer(char griglia[ROWS][COLUMNS], int vecchiaPosizione[2],
40                   int nuovaPosizione[2], Point deployCoords[],
41                   Point packsCoords[]);
42 int valoreTimerValido();
43 PlayerStats gestisciW(char grigliaDiGioco[ROWS][COLUMNS],
44                       char grigliaOstacoli[ROWS][COLUMNS],
45                       PlayerStats giocatore, Obstacles *listaOstacoli,
46                       Point deployCoords[], Point packsCoords[]);
47 PlayerStats gestisciA(char grigliaDiGioco[ROWS][COLUMNS],
48                       char grigliaOstacoli[ROWS][COLUMNS],
49                       PlayerStats giocatore, Obstacles *listaOstacoli,
50                       Point deployCoords[], Point packsCoords[]);
```

```
51  PlayerStats gestisciD(char grigliaDiGioco[ROWS][COLUMNS],
52                        char grigliaOstacoli[ROWS][COLUMNS],
53                        PlayerStats giocatore, Obstacles *listaOstacoli,
54                        Point deployCoords[], Point packsCoords[]);
55  PlayerStats gestisciS(char grigliaDiGioco[ROWS][COLUMNS],
56                        char grigliaOstacoli[ROWS][COLUMNS],
57                        PlayerStats giocatore, Obstacles *listaOstacoli,
58                        Point deployCoords[], Point packsCoords[]);
59  void rimuoviPlayerDallaMappa(PlayerStats);
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 clientDescriptor, PlayerStats giocatore);
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(int);
77  void startListening();
78  int clientDisconnesso(int clientSocket);
79  void play(int clientDesc, char name[]);
80  void prepareMessageForPackDelivery(char message[], char username[],
81                                     char date[]);
82  int logDelPacco(int flag);
83  int logDelLogin(int flag);
84  int logDellaConnessione(int flag);
85
86  char grigliaDiGiocoConPacchiSenzaOstacoli[ROWS][COLUMNS]; // protetta
87  char grigliaOstacoliSenzaPacchi[ROWS][COLUMNS];           // protetta
88  int numeroClientLoggati = 0;                              // protetto
89  int playerGenerati = 0;                                   // mutex
90  int timerCount = TIME_LIMIT_IN_SECONDS;
91  int turno = 0; // lo cambia solo timer
92  pthread_t tidTimer;
93  pthread_t tidGeneratoreMappa;
94  int socketDesc;
95  Players onLineUsers = NULL; // protetto
96  char *users;
97  int port;
98  int scoreMassimo = 0; // mutex
99  int numMosse = 0;     // mutex
100 Point deployCoords[numberOfPackages];
101 Point packsCoords[numberOfPackages];
102 pthread_mutex_t LogMutex = PTHREAD_MUTEX_INITIALIZER;
103 pthread_mutex_t RegMutex = PTHREAD_MUTEX_INITIALIZER;
104 pthread_mutex_t PlayerMutex = PTHREAD_MUTEX_INITIALIZER;
105 pthread_mutex_t MatrixMutex = PTHREAD_MUTEX_INITIALIZER;
106 pthread_mutex_t PlayerGeneratiMutex = PTHREAD_MUTEX_INITIALIZER;
107 pthread_mutex_t ScoreMassimoMutex = PTHREAD_MUTEX_INITIALIZER;
108 pthread_mutex_t numMosseMutex = PTHREAD_MUTEX_INITIALIZER;
109
110 int main(int argc, char **argv) {
111   if (argc != 3) {
112     printf("Wrong parameters number(Usage: ./server usersFile port)\n");
113     exit(-1);
114   } else if (strcmp(argv[1], "Log") == 0) {
115     printf("Cannot use the Log file as a UserList \n");
116     exit(-1);
117   }
118   users = argv[1];
119   port = atoi(argv[2]);
120   struct sockaddr_in mio_indirizzo = configuraIndirizzo(port);
121   configuraSocket(mio_indirizzo);
122   signal(SIGPIPE, SIG_IGN);
123   signal(SIGINT, quitServer);
124   signal(SIGHUP, quitServer);
125   startTimer();
126   inizializzaGiocoSenzaPlayer(grigliaDiGiocoConPacchiSenzaOstacoli,
127                               grigliaOstacoliSenzaPacchi, packsCoords);
128   generaPosizioniRaccolta(grigliaDiGiocoConPacchiSenzaOstacoli,
129                           grigliaOstacoliSenzaPacchi, deployCoords);
130   startListening();
131   return 0;
132 }
133 void startListening() {
134   pthread_t tid;
135   int clientDesc;
136   int *puntClientDesc;
137   while (1 == 1) {
138     if (listen(socketDesc, 10) < 0)
139       perror("Impossibile mettersi in ascolto"), exit(-1);
```

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

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

```
318       pthread_mutex_unlock(&PlayerGeneratiMutex);
319     }
320   }
321 }
322 void sendTimerValue(int clientDesc) {
323   if (!clientDisconnesso(clientDesc))
324     write(clientDesc, &timerCount, sizeof(timerCount));
325 }
326 void clonaGriglia(char destinazione[ROWS][COLUMNS],
327                   char source[ROWS][COLUMNS]) {
328   int i = 0, j = 0;
329   for (i = 0; i < ROWS; i++) {
330     for (j = 0; j < COLUMNS; j++) {
331       destinazione[i][j] = source[i][j];
332     }
333   }
334 }
335 void clientCrashHandler(int signalNum) { signal(SIGPIPE, SIG_IGN); }
336 void disconnettiClient(int clientDescriptor, PlayerStats giocatore) {
337   pthread_mutex_lock(&PlayerMutex);
338   if (numeroClientLoggati > 0)
339     numeroClientLoggati--;
340   rimuoviPlayerDallaMappa(giocatore);
341   onLineUsers = removePlayer(onLineUsers, clientDescriptor);
342   pthread_mutex_unlock(&PlayerMutex);
343   printPlayers(onLineUsers);
344   int msg = 1;
345   printf("Client disconnesso (client attualmente loggati: %d)\n",
346          numeroClientLoggati);
347   close(clientDescriptor);
348 }
349 int clientDisconnesso(int clientSocket) {
350   char msg[1] = {'u'}; // UP?
351   if (write(clientSocket, msg, sizeof(msg)) < 0)
352     return 1;
353   if (read(clientSocket, msg, sizeof(char)) < 0)
354     return 1;
355   else
356     return 0;
357 }
358 int registraClient(int clientDesc) {
359   char *userName = (char *)calloc(MAX_BUF, 1);
360   char *password = (char *)calloc(MAX_BUF, 1);
361   int dimName, dimPwd;
362   read(clientDesc, &dimName, sizeof(int));
363   read(clientDesc, &dimPwd, sizeof(int));
364   read(clientDesc, userName, dimName);
365   read(clientDesc, password, dimPwd);
366   pthread_mutex_lock(&RegMutex);
367   int ret = appendPlayer(userName, password, users);
368   pthread_mutex_unlock(&RegMutex);
369   char risposta;
370   if (!ret) {
371     risposta = 'n';
372     write(clientDesc, &risposta, sizeof(char));
373     printf("Impossibile registrare utente, riprovare\n");
374   } else {
375     risposta = 'y';
376     write(clientDesc, &risposta, sizeof(char));
377     printf("Utente registrato con successo\n");
378   }
379   return ret;
380 }
381 void quitServer() {
382   printf("Chiusura server in corso..\n");
383   close(socketDesc);
384   exit(-1);
385 }
386 void *threadGenerazioneMappa(void *args) {
387   fprintf(stdout, "Rigenerazione mappa\n");
388   inizializzaGrigliaVuota(grigliaDiGiocoConPacchiSenzaOstacoli);
389   generaPosizioniRaccolta(grigliaDiGiocoConPacchiSenzaOstacoli,
390                           grigliaOstacoliSenzaPacchi, deployCoords);
391   riempiGrigliaConPacchiInPosizioniGenerateCasualmente(
392       grigliaDiGiocoConPacchiSenzaOstacoli, packsCoords);
393   generaPosizioneOstacoli(grigliaDiGiocoConPacchiSenzaOstacoli,
394                           grigliaOstacoliSenzaPacchi);
395   printf("Mappa generata\n");
396   pthread_exit(NULL);
397 }
398 int almenoUnaMossaFatta() {
399   if (numMosse > 0)
400     return 1;
401   return 0;
402 }
403 int almenoUnClientConnesso() {
404   if (numeroClientLoggati > 0)
405     return 1;
406   return 0;
```

```
}
int valoreTimerValido() {
  if (timerCount > 0 && timerCount <= TIME_LIMIT_IN_SECONDS)
    return 1;
  return 0;
}
int almenoUnPlayerGenerato() {
  if (playerGenerati > 0)
    return 1;
  return 0;
}
void *timer(void *args) {
  int cambiato = 1;
  while (1) {
    if (almenoUnClientConnesso() && valoreTimerValido() &&
        almenoUnPlayerGenerato() && almenoUnaMossaFatta()) {
      cambiato = 1;
      sleep(1);
      timerCount--;
      fprintf(stdout, "Time left: %d\n", timerCount);
    } else if (numeroClientLoggati == 0) {
      timerCount = TIME_LIMIT_IN_SECONDS;
      if (cambiato) {
        fprintf(stdout, "Time left: %d\n", timerCount);
        cambiato = 0;
      }
    }
    if (timerCount == 0 || scoreMassimo == packageLimitNumber) {
      pthread_mutex_lock(&PlayerGeneratiMutex);
      playerGenerati = 0;
      pthread_mutex_unlock(&PlayerGeneratiMutex);
      pthread_mutex_lock(&numMosseMutex);
      numMosse = 0;
      pthread_mutex_unlock(&numMosseMutex);
      printf("Reset timer e generazione nuova mappa..\n");
      startProceduraGenrazioneMappa();
      pthread_join(tidGeneratoreMappa, NULL);
      turno++;
      pthread_mutex_lock(&ScoreMassimoMutex);
      scoreMassimo = 0;
      pthread_mutex_unlock(&ScoreMassimoMutex);
      timerCount = TIME_LIMIT_IN_SECONDS;
    }
  }
}

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);
  }
}

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);
  }

```

```
496   // aggiorna la posizione dell'utente
497   return giocatore;
498 }
499
500 PlayerStats gestisciC(char grigliaDiGioco[ROWS][COLUMNS], PlayerStats giocatore,
501                       Point deployCoords[], Point packsCoords[], char name[]) {
502   pthread_t tid;
503   if (giocatore->hasApack == 0) {
504     return giocatore;
505   } else {
506     if (isOnCorrectDeployPoint(giocatore, deployCoords)) {
507       Args args = (Args)malloc(sizeof(struct argsToSend));
508       args->userName = (char *)calloc(MAX_BUF, 1);
509       strcpy(args->userName, name);
510       args->flag = 1;
511       pthread_create(&tid, NULL, fileWriter, (void *)args);
512       giocatore->score += 10;
513       if (giocatore->score > scoreMassimo) {
514         pthread_mutex_lock(&ScoreMassimoMutex);
515         scoreMassimo = giocatore->score;
516         fprintf(stdout, "Score massimo: %d\n", scoreMassimo);
517         pthread_mutex_unlock(&ScoreMassimoMutex);
518       }
519       giocatore->deploy[0] = -1;
520       giocatore->deploy[1] = -1;
521       giocatore->hasApack = 0;
522     } else {
523       if (!isOnAPack(giocatore, packsCoords) &&
524           !isOnADeployPoint(giocatore, deployCoords)) {
525         int index = getHiddenPack(packsCoords);
526         if (index >= 0) {
527           packsCoords[index]->x = giocatore->position[0];
528           packsCoords[index]->y = giocatore->position[1];
529           giocatore->hasApack = 0;
530           giocatore->deploy[0] = -1;
531           giocatore->deploy[1] = -1;
532         }
533       } else
534         return giocatore;
535     }
536   }
537   return giocatore;
538 }
539
540 void sendPlayerList(int clientDesc) {
541   int lunghezza = 0;
542   char name[100];
543   Players tmp = onLineUsers;
544   int numeroClientLoggati = dimensioneLista(tmp);
545   printf("%d ", numeroClientLoggati);
546   if (!clientDisconnesso(clientDesc)) {
547     write(clientDesc, &numeroClientLoggati, sizeof(numeroClientLoggati));
548     while (numeroClientLoggati > 0 && tmp != NULL) {
549       strcpy(name, tmp->name);
550       lunghezza = strlen(tmp->name);
551       write(clientDesc, &lunghezza, sizeof(lunghezza));
552       write(clientDesc, name, lunghezza);
553       tmp = tmp->next;
554       numeroClientLoggati--;
555     }
556   }
557 }
558
559 void prepareMessageForPackDelivery(char message[], char username[],
560                                    char date[]) {
561   strcat(message, "Pack delivered by \"");
562   strcat(message, username);
563   strcat(message, "\" at ");
564   strcat(message, date);
565   strcat(message, "\n");
566 }
567
568 void prepareMessageForLogin(char message[], char username[], char date[]) {
569   strcat(message, username);
570   strcat(message, "\" logged in at ");
571   strcat(message, date);
572   strcat(message, "\n");
573 }
574
575 void prepareMessageForConnection(char message[], char ipAddress[],
576                                  char date[]) {
577   strcat(message, ipAddress);
578   strcat(message, "\" connected at ");
579   strcat(message, date);
580   strcat(message, "\n");
581 }
582
583 void putCurrentDateAndTimeInString(char dateAndTime[]) {
584   time_t t = time(NULL);
```

```
585   struct tm *infoTime = localtime(&t);
586   strftime(dateAndTime, 64, "%X %x", infoTime);
587 }
588
589 void *fileWriter(void *args) {
590   int fDes = open("Log", O_RDWR | O_CREAT | O_APPEND, S_IWUSR | S_IRUSR);
591   if (fDes < 0) {
592     perror("Error while opening log file");
593     exit(-1);
594   }
595   Args info = (Args)args;
596   char dateAndTime[64];
597   putCurrentDateAndTimeInString(dateAndTime);
598   if (logDelPacco(info->flag)) {
599     char message[MAX_BUF] = "";
600     prepareMessageForPackDelivery(message, info->userName, dateAndTime);
601     pthread_mutex_lock(&LogMutex);
602     write(fDes, message, strlen(message));
603     pthread_mutex_unlock(&LogMutex);
604   } else if (logDelLogin(info->flag)) {
605     char message[MAX_BUF] = "\"";
606     prepareMessageForLogin(message, info->userName, dateAndTime);
607     pthread_mutex_lock(&LogMutex);
608     write(fDes, message, strlen(message));
609     pthread_mutex_unlock(&LogMutex);
610   } else if (logDellaConnessione(info->flag)) {
611     char message[MAX_BUF] = "\"";
612     prepareMessageForConnection(message, info->userName, dateAndTime);
613     pthread_mutex_lock(&LogMutex);
614     write(fDes, message, strlen(message));
615     pthread_mutex_unlock(&LogMutex);
616   }
617   close(fDes);
618   free(info);
619   pthread_exit(NULL);
620 }
621
622 void spostaPlayer(char griglia[ROWS][COLUMNS], int vecchiaPosizione[2],
623                   int nuovaPosizione[2], Point deployCoords[],
624                   Point packsCoords[]) {
625
626   pthread_mutex_lock(&MatrixMutex);
627   griglia[nuovaPosizione[0]][nuovaPosizione[1]] = 'P';
628   if (eraUnPuntoDepo(vecchiaPosizione, deployCoords))
629     griglia[vecchiaPosizione[0]][vecchiaPosizione[1]] = '_';
630   else if (eraUnPacco(vecchiaPosizione, packsCoords))
631     griglia[vecchiaPosizione[0]][vecchiaPosizione[1]] = '$';
632   else
633     griglia[vecchiaPosizione[0]][vecchiaPosizione[1]] = '-';
634   pthread_mutex_unlock(&MatrixMutex);
635 }
636
637 PlayerStats gestisciW(char grigliaDiGioco[ROWS][COLUMNS],
638                       char grigliaOstacoli[ROWS][COLUMNS],
639                       PlayerStats giocatore, Obstacles *listaOstacoli,
640                       Point deployCoords[], Point packsCoords[]) {
641   if (giocatore == NULL)
642     return NULL;
643   int nuovaPosizione[2];
644   nuovaPosizione[1] = giocatore->position[1];
645   // Aggiorna la posizione vecchia spostando il player avanti di 1
646   nuovaPosizione[0] = (giocatore->position[0]) - 1;
647   int nuovoScore = giocatore->score;
648   int nuovoDeploy[2];
649   nuovoDeploy[0] = giocatore->deploy[0];
650   nuovoDeploy[1] = giocatore->deploy[1];
651   if (nuovaPosizione[0] >= 0 && nuovaPosizione[0] < ROWS) {
652     if (casellaVuotaOValida(grigliaDiGioco, grigliaOstacoli, nuovaPosizione)) {
653       spostaPlayer(grigliaDiGioco, giocatore->position, nuovaPosizione,
654                    deployCoords, packsCoords);
655     } else if (colpitoOstacolo(grigliaOstacoli, nuovaPosizione)) {
656       *listaOstacoli =
657           addObstacle(*listaOstacoli, nuovaPosizione[0], nuovaPosizione[1]);
658       nuovaPosizione[0] = giocatore->position[0];
659       nuovaPosizione[1] = giocatore->position[1];
660     } else if (colpitoPlayer(grigliaDiGioco, nuovaPosizione)) {
661       nuovaPosizione[0] = giocatore->position[0];
662       nuovaPosizione[1] = giocatore->position[1];
663     }
664     giocatore->deploy[0] = nuovoDeploy[0];
665     giocatore->deploy[1] = nuovoDeploy[1];
666     giocatore->score = nuovoScore;
667     giocatore->position[0] = nuovaPosizione[0];
668     giocatore->position[1] = nuovaPosizione[1];
669   }
670   return giocatore;
671 }
672
673 PlayerStats gestisciD(char grigliaDiGioco[ROWS][COLUMNS],
```

```
674                     char grigliaOstacoli[ROWS][COLUMNS],
675                     PlayerStats giocatore, Obstacles *listaOstacoli,
676                     Point deployCoords[], Point packsCoords[]) {
677   if (giocatore == NULL) {
678     return NULL;
679   }
680   int nuovaPosizione[2];
681   nuovaPosizione[1] = giocatore->position[1] + 1;
682   nuovaPosizione[0] = giocatore->position[0];
683   int nuovoScore = giocatore->score;
684   int nuovoDeploy[2];
685   nuovoDeploy[0] = giocatore->deploy[0];
686   nuovoDeploy[1] = giocatore->deploy[1];
687   if (nuovaPosizione[1] >= 0 && nuovaPosizione[1] < COLUMNS) {
688     if (casellaVuotaOValida(grigliaDiGioco, grigliaOstacoli, nuovaPosizione)) {
689       spostaPlayer(grigliaDiGioco, giocatore->position, nuovaPosizione,
690                    deployCoords, packsCoords);
691     } else if (colpitoOstacolo(grigliaOstacoli, nuovaPosizione)) {
692       printf("Ostacolo\n");
693       *listaOstacoli =
694           addObstacle(*listaOstacoli, nuovaPosizione[0], nuovaPosizione[1]);
695       nuovaPosizione[0] = giocatore->position[0];
696       nuovaPosizione[1] = giocatore->position[1];
697     } else if (colpitoPlayer(grigliaDiGioco, nuovaPosizione)) {
698       nuovaPosizione[0] = giocatore->position[0];
699       nuovaPosizione[1] = giocatore->position[1];
700     }
701     giocatore->deploy[0] = nuovoDeploy[0];
702     giocatore->deploy[1] = nuovoDeploy[1];
703     giocatore->score = nuovoScore;
704     giocatore->position[0] = nuovaPosizione[0];
705     giocatore->position[1] = nuovaPosizione[1];
706   }
707   return giocatore;
708 }
709 PlayerStats gestisciA(char grigliaDiGioco[ROWS][COLUMNS],
710                     char grigliaOstacoli[ROWS][COLUMNS],
711                     PlayerStats giocatore, Obstacles *listaOstacoli,
712                     Point deployCoords[], Point packsCoords[]) {
713   if (giocatore == NULL)
714     return NULL;
715   int nuovaPosizione[2];
716   nuovaPosizione[0] = giocatore->position[0];
717   // Aggiorna la posizione vecchia spostando il player avanti di 1
718   nuovaPosizione[1] = (giocatore->position[1]) - 1;
719   int nuovoScore = giocatore->score;
720   int nuovoDeploy[2];
721   nuovoDeploy[0] = giocatore->deploy[0];
722   nuovoDeploy[1] = giocatore->deploy[1];
723   if (nuovaPosizione[1] >= 0 && nuovaPosizione[1] < COLUMNS) {
724     if (casellaVuotaOValida(grigliaDiGioco, grigliaOstacoli, nuovaPosizione)) {
725       printf("Casella vuota \n");
726       spostaPlayer(grigliaDiGioco, giocatore->position, nuovaPosizione,
727                    deployCoords, packsCoords);
728     } else if (colpitoOstacolo(grigliaOstacoli, nuovaPosizione)) {
729       printf("Ostacolo\n");
730       *listaOstacoli =
731           addObstacle(*listaOstacoli, nuovaPosizione[0], nuovaPosizione[1]);
732       nuovaPosizione[0] = giocatore->position[0];
733       nuovaPosizione[1] = giocatore->position[1];
734     } else if (colpitoPlayer(grigliaDiGioco, nuovaPosizione)) {
735       printf("colpito player\n");
736       nuovaPosizione[0] = giocatore->position[0];
737       nuovaPosizione[1] = giocatore->position[1];
738     }
739     giocatore->deploy[0] = nuovoDeploy[0];
740     giocatore->deploy[1] = nuovoDeploy[1];
741     giocatore->score = nuovoScore;
742     giocatore->position[0] = nuovaPosizione[0];
743     giocatore->position[1] = nuovaPosizione[1];
744   }
745   return giocatore;
746 }
747 PlayerStats gestisciS(char grigliaDiGioco[ROWS][COLUMNS],
748                     char grigliaOstacoli[ROWS][COLUMNS],
749                     PlayerStats giocatore, Obstacles *listaOstacoli,
750                     Point deployCoords[], Point packsCoords[]) {
751   if (giocatore == NULL) {
752     return NULL;
753   }
754   // crea le nuove statistiche
755   int nuovaPosizione[2];
756   nuovaPosizione[1] = giocatore->position[1];
757   nuovaPosizione[0] = (giocatore->position[0]) + 1;
758   int nuovoScore = giocatore->score;
759   int nuovoDeploy[2];
760   nuovoDeploy[0] = giocatore->deploy[0];
761   nuovoDeploy[1] = giocatore->deploy[1];
762   // controlla che le nuove statistiche siano corrette
```

```
763   if (nuovaPosizione[0] >= 0 && nuovaPosizione[0] < ROWS) {
764     if (casellaVuotaOValida(grigliaDiGioco, grigliaOstacoli, nuovaPosizione)) {
765       spostaPlayer(grigliaDiGioco, giocatore->position, nuovaPosizione,
766                    deployCoords, packsCoords);
767     } else if (colpitoOstacolo(grigliaOstacoli, nuovaPosizione)) {
768       printf("Ostacolo\n");
769       *listaOstacoli =
770           addObstacle(*listaOstacoli, nuovaPosizione[0], nuovaPosizione[1]);
771       nuovaPosizione[0] = giocatore->position[0];
772       nuovaPosizione[1] = giocatore->position[1];
773     } else if (colpitoPlayer(grigliaDiGioco, nuovaPosizione)) {
774       nuovaPosizione[0] = giocatore->position[0];
775       nuovaPosizione[1] = giocatore->position[1];
776     }
777     giocatore->deploy[0] = nuovoDeploy[0];
778     giocatore->deploy[1] = nuovoDeploy[1];
779     giocatore->score = nuovoScore;
780     giocatore->position[0] = nuovaPosizione[0];
781     giocatore->position[1] = nuovaPosizione[1];
782   }
783   return giocatore;
784 }
785
786 int logDelPacco(int flag) {
787   if (flag == 1)
788     return 1;
789   return 0;
790 }
791 int logDelLogin(int flag) {
792   if (flag == 0)
793     return 1;
794   return 0;
795 }
796 int logDellaConnessione(int flag) {
797   if (flag == 2)
798     return 1;
799   return 0;
800 }
801
802 void rimuoviPlayerDallaMappa(PlayerStats giocatore) {
803   if (giocatore == NULL)
804     return;
805   int x = giocatore->position[1];
806   int y = giocatore->position[0];
807   if (eraUnPacco(giocatore->position, packsCoords))
808     grigliaDiGiocoConPacchiSenzaOstacoli[y][x] = '$';
809   else if (eraUnPuntoDepo(giocatore->position, deployCoords))
810     grigliaDiGiocoConPacchiSenzaOstacoli[y][x] = '_';
811   else
812     grigliaDiGiocoConPacchiSenzaOstacoli[y][x] = '-';
813 }
```

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 12
7  #define COLUMNS 32
8  #define numberOfObstacles 35
9  #define numberOfPackages 15
10 #define TIME_LIMIT_IN_SECONDS 90
11 #define packageLimitNumber 40
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[]);
```

```
PlayerStats gestisciP(char grigliaDiGioco[ROWS][COLUMNS], PlayerStats giocatore,
                      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[]);
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]);
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 19: Codice sorgente utility del gioco 1

```
#include "boardUtility.h"
#include "list.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
void printMenu() {
  system("clear");
  printf("\t Cosa vuoi fare?\n");
  printf("\t1 Gioca\n");
  printf("\t2 Registrati\n");
  printf("\t3 Esci\n");
}
int colpitoOstacolo(char grigliaOstacoli[ROWS][COLUMNS], int posizione[2]) {
  if (grigliaOstacoli[posizione[0]][posizione[1]] == 'O')
    return 1;
  return 0;
}
int colpitoPacco(Point packsCoords[], int posizione[2]) {
  int i = 0;
  for (i = 0; i < numberOfPackages; i++) {
    if (packsCoords[i]->x == posizione[0] && packsCoords[i]->y == posizione[1])
      return 1;
  }
  return 0;
}
int casellaVuotaOValida(char grigliaDiGioco[ROWS][COLUMNS],
                        char grigliaOstacoli[ROWS][COLUMNS], int posizione[2]) {
  if (grigliaDiGioco[posizione[0]][posizione[1]] == '-' || // casella vuota
      grigliaDiGioco[posizione[0]][posizione[1]] == '_' || // punto deploy
      grigliaDiGioco[posizione[0]][posizione[1]] == '$')   // pacco
    if (grigliaOstacoli[posizione[0]][posizione[1]] == '-' ||
        grigliaOstacoli[posizione[0]][posizione[1]] == '_' ||
        grigliaOstacoli[posizione[0]][posizione[1]] == '$')
      return 1;
  return 0;
}
int colpitoPlayer(char grigliaDiGioco[ROWS][COLUMNS], int posizione[2]) {
  if (grigliaDiGioco[posizione[0]][posizione[1]] == 'P')
    return 1;
  return 0;
}
int isOnCorrectDeployPoint(PlayerStats giocatore, Point deployCoords[]) {
  int i = 0;
  for (i = 0; i < numberOfPackages; i++) {
    if (giocatore->deploy[0] == deployCoords[i]->x &&
        giocatore->deploy[1] == deployCoords[i]->y) {
      if (deployCoords[i]->x == giocatore->position[0] &&
          deployCoords[i]->y == giocatore->position[1])
        return 1;
    }
  }
  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 == ROWS - 1)
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   if (i == 8)
146     printf("\t Pacchi-> $ | Ostacoli -> O | Punti deposito -> _ ");
147   if (i == 9) {
148     printf("\t Player ->");
149     printf(RED_COLOR " P" RESET_COLOR);
150     printf(" | Altri Player -> P");
151     printf(" | Pacco preso -> ");
152     printf(GREEN_COLOR "P" RESET_COLOR);
153   }
154   if (i == 10) {
155     printf("\t Punto deposito designato -> ");
156     printf(RED_COLOR "_" RESET_COLOR);
157   }
158 }
159 // aggiunge alla griglia gli ostacoli visti fino ad ora dal client
160 void mergeGridAndList(char grid[ROWS][COLUMNS], Obstacles top) {
161   while (top) {
162     grid[top->x][top->y] = 'O';
163     top = top->next;
164   }
165 }
166 /* Genera la posizione degli ostacoli */
167 void generaPosizioneOstacoli(char grigliaDiGioco[ROWS][COLUMNS],
168                              char grigliaOstacoli[ROWS][COLUMNS]) {
169   int x, y, i;
170   inizializzaGrigliaVuota(grigliaOstacoli);
171   srand(time(0));
172   for (i = 0; i < numberOfObstacles; i++) {
173     x = rand() % COLUMNS;
174     y = rand() % ROWS;
175     if (grigliaDiGioco[y][x] == '-')
176       grigliaOstacoli[y][x] = 'O';
177     else
178       i--;
179   }
180 }
181 void rimuoviPaccoDaArray(int posizione[2], Point packsCoords[]) {
182   int i = 0, found = 0;
183   while (i < numberOfPackages && !found) {
184     if ((packsCoords[i])->x == posizione[0] &&
185         (packsCoords[i])->y == posizione[1]) {
186       (packsCoords[i])->x = -1;
187       (packsCoords[i])->y = -1;
188       found = 1;
189     }
190     i++;
191   }
192 }
193 // sceglie una posizione di raccolta tra quelle disponibili
194 void scegliPosizioneRaccolta(Point coord[], int deploy[]) {
195   int index = 0;
196   srand(time(NULL));
197   index = rand() % numberOfPackages;
198   deploy[0] = coord[index]->x;
199   deploy[1] = coord[index]->y;
200 }
201 /*genera posizione di raccolta di un pacco*/
202 void generaPosizioniRaccolta(char grigliaDiGioco[ROWS][COLUMNS],
203                              char grigliaOstacoli[ROWS][COLUMNS],
204                              Point coord[]) {
205   int x, y;
206   srand(time(0));
207   int i = 0;
208   for (i = 0; i < numberOfPackages; i++) {
209     coord[i] = (Point)malloc(sizeof(struct Coord));
210   }
211   i = 0;
212   for (i = 0; i < numberOfPackages; i++) {
213     x = rand() % COLUMNS;
214     y = rand() % ROWS;
215     if (grigliaDiGioco[y][x] == '-' && grigliaOstacoli[y][x] == '-') {
216       coord[i]->x = y;
217       coord[i]->y = x;
218       grigliaDiGioco[y][x] = '_';
219       grigliaOstacoli[y][x] = '_';
220     } else
221       i--;
222   }
223 }
224 /*Inserisci dei pacchi nella griglia di gioco nella posizione casuale */
225 void riempiGrigliaConPacchiInPosizioniGenerateCasualmente(
226     char grigliaDiGioco[ROWS][COLUMNS], Point packsCoords[]) {
227   int x, y, i = 0;
228   for (i = 0; i < numberOfPackages; i++) {
229     packsCoords[i] = (Point)malloc(sizeof(struct Coord));
230   }
231   srand(time(0));
```

```
  for (i = 0; i < numberOfPackages; i++) {
    x = rand() % COLUMNS;
    y = rand() % ROWS;
    if (grigliaDiGioco[y][x] == '-') {
      grigliaDiGioco[y][x] = '$';
      packsCoords[i]->x = y;
      packsCoords[i]->y = x;
    } else
      i--;
  }
}
/*Inserisci gli ostacoli nella griglia di gioco */
void riempiGrigliaConGliOstacoli(char grigliaDiGioco[ROWS][COLUMNS],
                                 char grigliaOstacoli[ROWS][COLUMNS]) {
  int i, j = 0;
  for (i = 0; i < ROWS; i++) {
    for (j = 0; j < COLUMNS; j++) {
      if (grigliaOstacoli[i][j] == 'O')
        grigliaDiGioco[i][j] = 'O';
    }
  }
}
void inserisciPlayerNellaGrigliaInPosizioneCasuale(
    char grigliaDiGioco[ROWS][COLUMNS], char grigliaOstacoli[ROWS][COLUMNS],
    int posizione[2]) {
  int x, y;
  srand(time(0));
  printf("Inserisco player\n");
  do {
    x = rand() % COLUMNS;
    y = rand() % ROWS;
  } while (grigliaDiGioco[y][x] != '-' && grigliaOstacoli[y][x] != '-');
  grigliaDiGioco[y][x] = 'P';
  posizione[0] = y;
  posizione[1] = x;
}
void inizializzaGiocoSenzaPlayer(char grigliaDiGioco[ROWS][COLUMNS],
                                 char grigliaConOstacoli[ROWS][COLUMNS],
                                 Point packsCoords[]) {
  inizializzaGrigliaVuota(grigliaDiGioco);
  riempiGrigliaConPacchiInPosizioniGenerateCasualmente(grigliaDiGioco,
                                                       packsCoords);
  generaPosizioneOstacoli(grigliaDiGioco, grigliaConOstacoli);
  return;
}

int eraUnPuntoDepo(int vecchiaPosizione[2], Point depo[]) {
  int i = 0, ret = 0;
  while (ret == 0 && i < numberOfPackages) {
    if ((depo[i])->y == vecchiaPosizione[1] &&
        (depo[i])->x == vecchiaPosizione[0]) {
      ret = 1;
    }
    i++;
  }
  return ret;
}
int eraUnPacco(int vecchiaPosizione[2], Point packsCoords[]) {
  int i = 0, ret = 0;
  while (ret == 0 && i < numberOfPackages) {
    if ((packsCoords[i])->y == vecchiaPosizione[1] &&
        (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 {
```

```
7   char *name;
8   struct TList *next;
9   int sockDes;
10 } TList;
11
12 struct Data {
13   int deploy[2];
14   int score;
15   int position[2];
16   int hasApack;
17 } Data;
18
19 // Obstacles
20 struct TList2 {
21   int x;
22   int y;
23   struct TList2 *next;
24 } TList2;
25
26 typedef struct Data *PlayerStats;
27 typedef struct TList *Players;
28 typedef struct TList2 *Obstacles;
29
30 // calcola e restituisce il numero di player commessi dalla lista L
31 int dimensioneLista(Players L);
32
33 // inizializza un giocatore
34 Players initPlayerNode(char *name, int sockDes);
35
36 // Crea un nodo di Stats da mandare a un client
37 PlayerStats initStats(int deploy[], int score, int position[], int flag);
38
39 // Inizializza un nuovo nodo
40 Players initNodeList(char *name, int sockDes);
41
42 // Aggiunge un nodo in testa alla lista
43 // La funzione ritorna sempre la testa della lista
44 Players addPlayer(Players L, char *name, int sockDes);
45
46 // Rimuove solo un occorrenza di un nodo con il socket Descriptor
47 //  specificato dalla lista
48 // La funzione ritorna sempre la testa della lista
49 Players removePlayer(Players L, int sockDes);
50
51 // Dealloca la lista interamente
52 void freePlayers(Players L);
53
54 // Stampa la lista
55 void printPlayers(Players L);
56
57 // Controlla se un utente á giá loggato
58 int isAlreadyLogged(Players L, char *name);
59
60 // Dealloca la lista degli ostacoli
61 void freeObstacles(Obstacles L);
62
63 // Stampa la lista degli ostacoli
64 void printObstacles(Obstacles L);
65
66 // Aggiunge un ostacolo in testa
67 Obstacles addObstacle(Obstacles L, int x, int y);
68
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;
```

```
  L->position[0] = position[0];
  L->position[1] = position[1];
  return L;
}
Obstacles initObstacleNode(int x, int y) {
  Obstacles L = (Obstacles)malloc(sizeof(struct TList2));
  L->x = x;
  L->y = y;
  L->next = NULL;
  return L;
}
Obstacles addObstacle(Obstacles L, int x, int y) {
  Obstacles tmp = initObstacleNode(x, y);
  if (L != NULL)
    tmp->next = L;
  return tmp;
}
int dimensioneLista(Players L) {
  int size = 0;
  Players tmp = L;
  while (tmp != NULL) {
    size++;
    tmp = tmp->next;
  }
  return size;
}
int isAlreadyLogged(Players L, char *name) {
  int ret = 0;
  if (L != NULL) {
    if (strcmp(L->name, name) == 0)
      return 1;
    ret = isAlreadyLogged(L->next, name);
  }
  return ret;
}
Players addPlayer(Players L, char *name, int sockDes) {
  Players tmp = initPlayerNode(name, sockDes);
  if (L != NULL)
    tmp->next = L;
  return tmp;
}
Players removePlayer(Players L, int sockDes) {
  if (L != NULL) {
    if (L->sockDes == sockDes) {
      Players tmp = L->next;
      free(L);
      return tmp;
    }
    L->next = removePlayer(L->next, sockDes);
  }
  return L;
}
void freePlayers(Players L) {
  if (L != NULL) {
    freePlayers(L->next);
    free(L);
  }
}
void freeObstacles(Obstacles L) {
  if (L != NULL) {
    freeObstacles(L->next);
    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);
```

```
4  int validateLogin(char *name, char *pwd, char *file);
5  int openFileRDON(char *file);
6  void premiEnterPerContinuare();
```

Listato 23: 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