

# 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

<b>1</b>	<b>Traccia</b>	<b>1</b>
<b>2</b>	<b>Istruzioni preliminari</b>	<b>2</b>
2.1	Modalità di compilazione . . . . .	2
<b>3</b>	<b>Guida all'uso</b>	<b>2</b>
3.1	Server . . . . .	2
3.2	Client . . . . .	2
<b>4</b>	<b>Comunicazione tra client e server</b>	<b>5</b>
4.1	Configurazione del server . . . . .	5
4.2	Configurazione del client . . . . .	6
4.3	Comunicazione tra client e server . . . . .	7
4.3.1	Esempio: la prima comunicazione . . . . .	7
<b>5</b>	<b>Comunicazione durante la partita</b>	<b>8</b>
5.1	Funzione core del server . . . . .	8
5.2	Funzione core del client . . . . .	9
<b>6</b>	<b>Dettagli implementativi degni di nota</b>	<b>10</b>
6.1	Timer . . . . .	10
6.2	Gestione del file di Log . . . . .	11
6.3	Modifica della mappa di gioco da parte di più thread . . . . .	11
6.4	Gestione del login . . . . .	12
<b>A</b>	<b>Codici sorgente</b>	<b>14</b>
A.1	Codice sorgente del client . . . . .	14
A.2	Codice sorgente del server . . . . .	17
A.3	Codice sorgente boardUtility . . . . .	26
A.4	Codice sorgente list . . . . .	30
A.5	Codice sorgente parser . . . . .	32



# 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 nascosti 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 è in grado di servire più client simultaneamente.

Durante il suo regolare funzionamento, il server effettuerà logging delle attività 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. È 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 manualmente 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

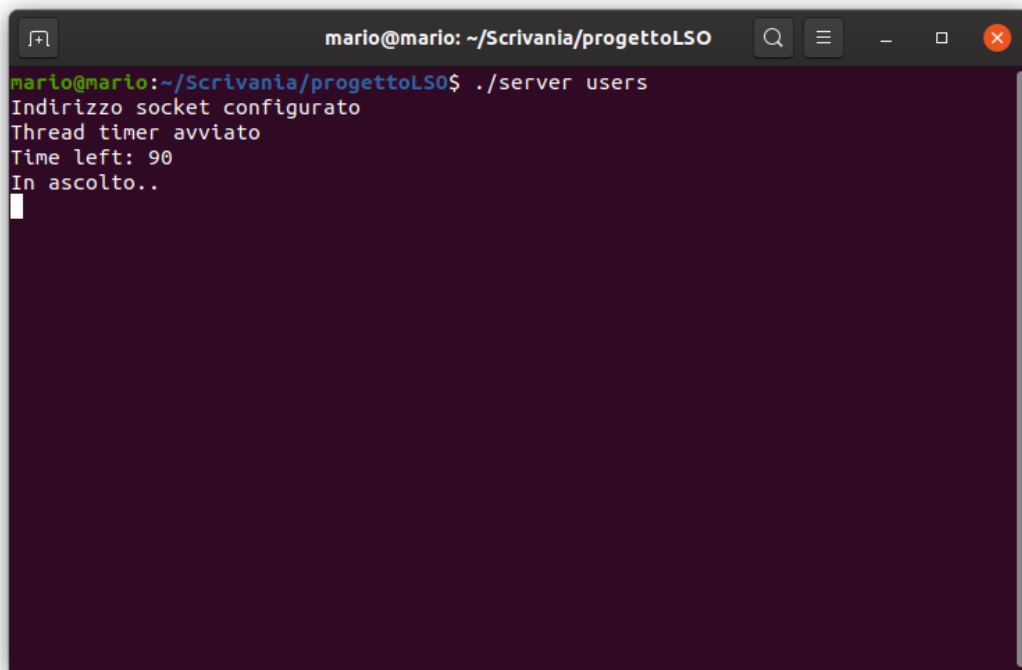
### 3.1 Server

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

```
./server users
```

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*.



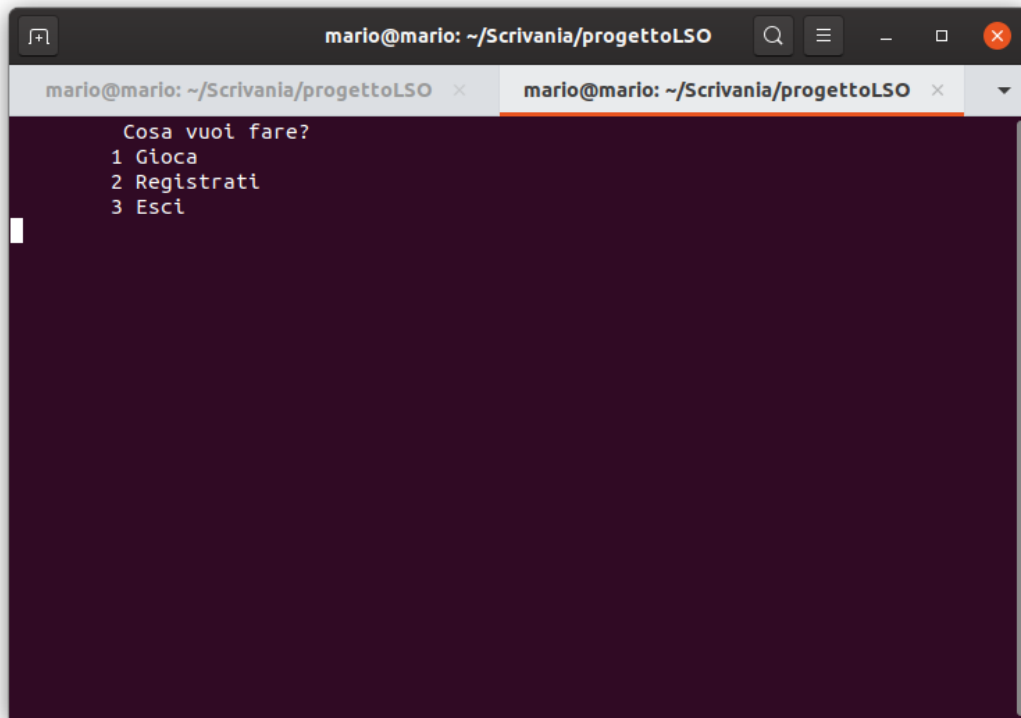
**Figura 1:** Menù del Server

### 3.2 Client

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

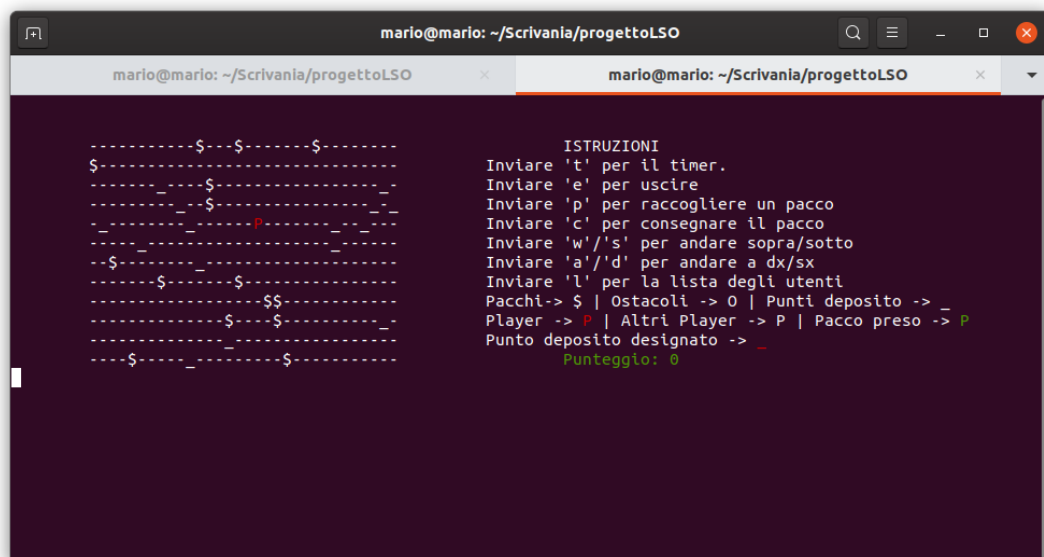
```
./client ip porta
```

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 comparirà il menu con le scelte 3 possibili: gioca, registrati ed esci.

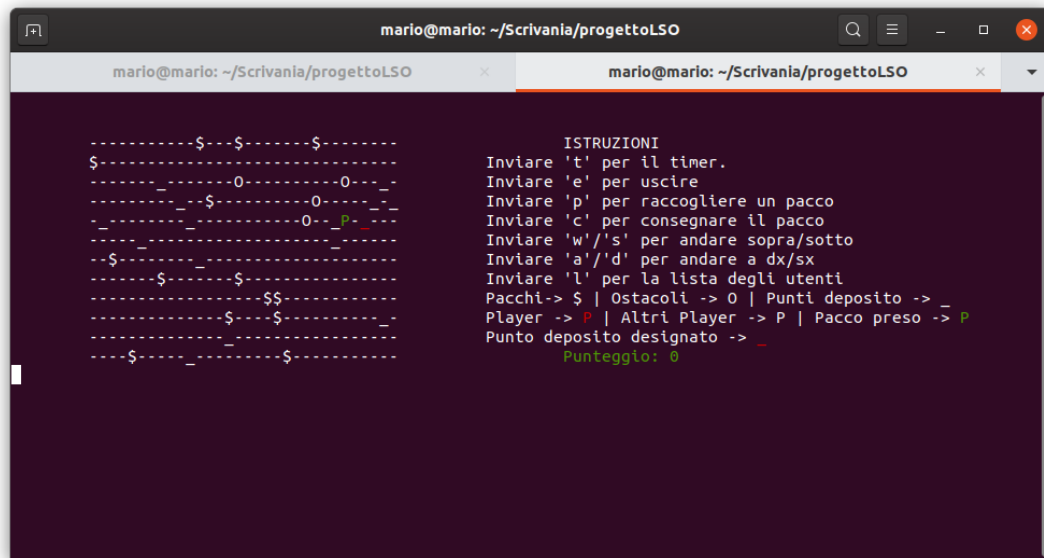


**Figura 2:** Menù del client.

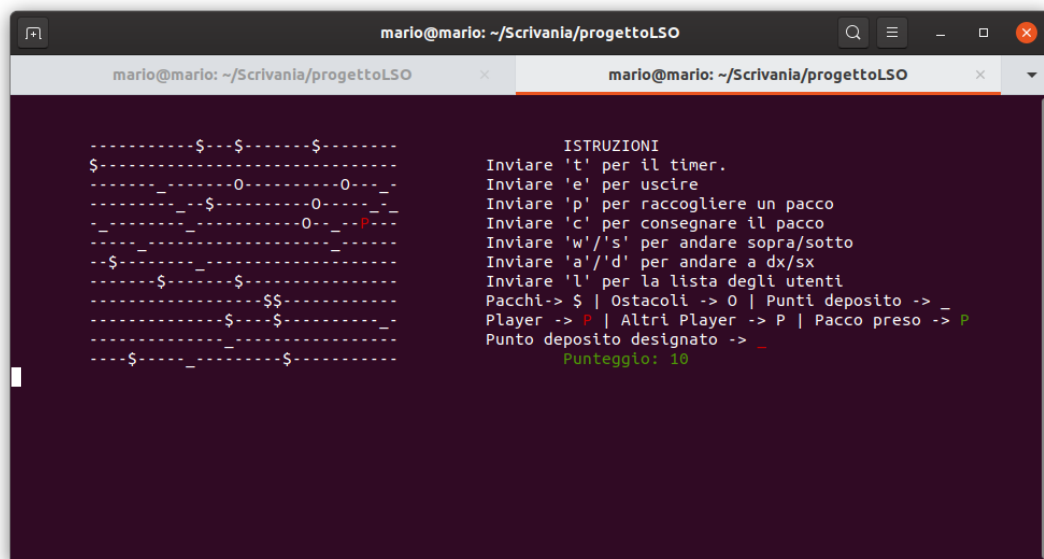
Una volta effettuata la registrazione dell'utente è possibile effettuare l'accesso al programma al seguito del quale verranno mostrate sia la mappa sia le istruzioni di gioco.



**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 protocolli PF\_INET, con tipo di trasmissione dati SOCK\_STREAM e con protocollo TCP. Mostriamo di seguito il codice sorgente:

**Listato 1:** Configurazione indirizzo del server

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

**Listato 2:** Configurazione socket del server

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

È importante notare anche come il server riesca a gestire in modo concorrente più client tramite l'uso di un thread dedicato ad ogni client 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

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

```
38 }
```

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 relative 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 rigi 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 più 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 relazione a ciò che è stato studiato durante il corso di sistemi operativi.

### 6.1 Timer

Lo svolgimento della partita è legato al timer: ogni round durerà un numero finito di secondi od oppure terminerà 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 conto alla rovescia.<sup>1</sup>

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);
10 }
```

<sup>1</sup>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

```

10 printf("Mappa generata\n");
11 pthread_exit(NULL);
12 }

```

## 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'implementazione di questa funzione piuttosto interessante poiché, 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

```

1 void *fileWriter(void *args) {
2     int fDes = open("Log", O_RDWR | O_CREAT | O_APPEND, S_IWUSR | S_IRUSR);
3     if (fDes < 0) {
4         perror("Error while opening log file");
5         exit(-1);
6     }
7     Args info = (Args)args;
8     char dateAndTime[64];
9     putCurrentDateAndTimeInString(dateAndTime);
10    if (logDelPacco(info->flag)) {
11        char message[MAX_BUF] = "";
12        prepareMessageForPackDelivery(message, info->userName, dateAndTime);
13        pthread_mutex_lock(&LogMutex);
14        write(fDes, message, strlen(message));
15        pthread_mutex_unlock(&LogMutex);
16    } else if (logDelLogin(info->flag)) {
17        char message[MAX_BUF] = "";
18        prepareMessageForLogin(message, info->userName, dateAndTime);
19        pthread_mutex_lock(&LogMutex);
20        write(fDes, message, strlen(message));
21        pthread_mutex_unlock(&LogMutex);
22    } else if (logDellaConnessione(info->flag)) {
23        char message[MAX_BUF] = "";
24        prepareMessageForConnection(message, info->userName, dateAndTime);
25        pthread_mutex_lock(&LogMutex);
26        write(fDes, message, strlen(message));
27        pthread_mutex_unlock(&LogMutex);
28    }
29    close(fDes);
30    free(info);
31    pthread_exit(NULL);
32 }

```

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

```

1 void spostaPlayer(char griglia[ROWS][COLUMNS], int vecchiaPosizione[2],
2                 int nuovaPosizione[2], Point deployCoords[],
3                 Point packsCoords[]) {
4
5     pthread_mutex_lock(&MatrixMutex);
6     griglia[nuovaPosizione[0]][nuovaPosizione[1]] = 'P';
7     if (eraUnPuntoDepo(vecchiaPosizione, deployCoords))
8         griglia[vecchiaPosizione[0]][vecchiaPosizione[1]] = '_';
9     else if (eraUnPacco(vecchiaPosizione, packsCoords))
10        griglia[vecchiaPosizione[0]][vecchiaPosizione[1]] = '$';
11    else
12        griglia[vecchiaPosizione[0]][vecchiaPosizione[1]] = '-';
13    pthread_mutex_unlock(&MatrixMutex);
14 }

```

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 posizione.

Il secondo accorgimento è stato quello di far in modo che un player possa raccogliere un pacco solo quando si trova nella posizione 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 giudicato 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"

```

1 int isRegistered(char *name, char *file) {
2     char command[MAX_BUF] = "cat ";
3     strcat(command, file);
4     char toApp[] = " |cut -d\" \" -f1|grep \"^\"";
5     strcat(command, toApp);
6     strcat(command, name);
7     char toApp2[] = "$\">tmp";
8     strcat(command, toApp2);
9     int ret = 0;
10    system(command);
11    int fileDes = openFileRDON("tmp");
12    struct stat info;
13    fstat(fileDes, &info);
14    if ((int)info.st_size > 0)
15        ret = 1;
16    close(fileDes);
17    system("rm tmp");
18    return ret;
19 }
```

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"

```

1 int validateLogin(char *name, char *pwd, char *file) {
2     if (!isRegistered(name, file))
3         return 0;
4     char command[MAX_BUF] = "cat ";
5     strcat(command, file);
6     char toApp[] = " |grep \"^\"";
7     strcat(command, toApp);
8     strcat(command, name);
9     strcat(command, " ");
10    strcat(command, pwd);
11    char toApp2[] = "$\">tmp";
12    strcat(command, toApp2);
13    int ret = 0;
14    system(command);
15    int fileDes = openFileRDON("tmp");
16    struct stat info;
17    fstat(fileDes, &info);
18    if ((int)info.st_size > 0)
19        ret = 1;
20    close(fileDes);
21    system("rm tmp");
```



```
22 | return ret;  
23 | }
```

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

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

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

## A Codici sorgente

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

### A.1 Codice sorgente del client

**Listato 16:** Codice sorgente del client

```

1  #include "boardUtility.h"
2  #include "list.h"
3  #include "parser.h"
4  #include <arpa/inet.h>
5  #include <fcntl.h>
6  #include <netdb.h>
7  #include <netinet/in.h> //conversioni
8  #include <netinet/in.h>
9  #include <netinet/ip.h> //struttura
10 #include <pthread.h>
11 #include <signal.h>
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <string.h>
15 #include <sys/socket.h>
16 #include <sys/stat.h>
17 #include <sys/types.h>
18 #include <time.h>
19 #include <unistd.h>
20
21 void printPlayerList();
22 int getTimer();
23 void printTimer();
24 void play();
25 int tryLogin();
26 void printMenu();
27 int connettiAlServer(char **argv);
28 char *ipResolver(char **argv);
29 int registrati();
30 int gestisci();
31 char getUserInput();
32 void clientCrashHandler();
33 void serverCrashHandler();
34 int serverCaduto();
35 void esciDalServer();
36 int isCorrect(char);
37
38 int socketDesc;
39 char grigliaDiGioco[ROWS][COLUMNS];
40
41 int main(int argc, char **argv) {
42     signal(SIGINT, clientCrashHandler); /* CTRL-C */
43     signal(SIGHUP, clientCrashHandler); /* Chiusura della console */
44     signal(SIGQUIT, clientCrashHandler);
45     signal(SIGTSTP, clientCrashHandler); /* CTRL-Z */
46     signal(SIGTERM, clientCrashHandler); /* generato da 'kill' */
47     signal(SIGPIPE, serverCrashHandler);
48     char bufferReceive[2];
49     if (argc != 3) {
50         perror("Inserire indirizzo ip/url e porta (./client 127.0.0.1 5200)");
51         exit(-1);
52     }
53     if ((socketDesc = connettiAlServer(argv)) < 0)
54         exit(-1);
55     gestisci(socketDesc);
56     close(socketDesc);
57     exit(0);
58 }
59 void esciDalServer() {
60     int msg = 3;
61     printf("Uscita in corso\n");
62     write(socketDesc, &msg, sizeof(msg));
63     close(socketDesc);
64 }
65 int connettiAlServer(char **argv) {
66     char *indirizzoServer;
67     uint16_t porta = strtoul(argv[2], NULL, 10);
68     indirizzoServer = ipResolver(argv);
69     struct sockaddr_in mio_indirizzo;
70     mio_indirizzo.sin_family = AF_INET;
71     mio_indirizzo.sin_port = htons(porta);
72     inet_aton(indirizzoServer, &mio_indirizzo.sin_addr);
73     if ((socketDesc = socket(PF_INET, SOCK_STREAM, 0)) < 0)
74         perror("Impossibile creare socket"), exit(-1);
75     else
76         printf("Socket creato\n");
77     if (connect(socketDesc, (struct sockaddr *)&mio_indirizzo,

```

```

78         sizeof(mio_indirizzo)) < 0)
79     perror("Impossibile connettersi"), exit(-1);
80 else
81     printf("Connesso a %s\n", indirizzoServer);
82     return socketDesc;
83 }
84 int gestisci() {
85     char choice;
86     while (1) {
87         printMenu();
88         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 già 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();
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 scoreMassimo = 0; // mutex
98 | int numMosse = 0; // mutex
99 | Point deployCoords[numberOfPackages];
100 | Point packsCoords[numberOfPackages];
101 | pthread_mutex_t LogMutex = PTHREAD_MUTEX_INITIALIZER;
102 | pthread_mutex_t RegMutex = PTHREAD_MUTEX_INITIALIZER;
103 | pthread_mutex_t PlayerMutex = PTHREAD_MUTEX_INITIALIZER;
104 | pthread_mutex_t MatrixMutex = PTHREAD_MUTEX_INITIALIZER;
105 | pthread_mutex_t PlayerGeneratiMutex = PTHREAD_MUTEX_INITIALIZER;
106 | pthread_mutex_t ScoreMassimoMutex = PTHREAD_MUTEX_INITIALIZER;
107 | pthread_mutex_t numMosseMutex = PTHREAD_MUTEX_INITIALIZER;
108 |
109 | int main(int argc, char **argv) {
110 |     if (argc != 2) {
111 |         printf("Wrong parameters number(Usage: ./server usersFile)\n");
112 |         exit(-1);
113 |     } else if (strcmp(argv[1], "Log") == 0) {
114 |         printf("Cannot use the Log file as a UserList \n");
115 |         exit(-1);
116 |     }
117 |     users = argv[1];
118 |     struct sockaddr_in mio_indirizzo = configuraIndirizzo();
119 |     configuraSocket(mio_indirizzo);
120 |     signal(SIGPIPE, SIG_IGN);
121 |     signal(SIGINT, quitServer);
122 |     signal(SIGHUP, quitServer);
123 |     startTimer();
124 |     inizializzaGiocoSenzaPlayer(grigliaDiGiocoConPacchiSenzaOstacoli,
125 |                                 grigliaOstacoliSenzaPacchi, packsCoords);
126 |     generaPosizioniRaccolta(grigliaDiGiocoConPacchiSenzaOstacoli,
127 |                             grigliaOstacoliSenzaPacchi, deployCoords);
128 |     startListening();
129 |     return 0;
130 | }
131 | void startListening() {
132 |     pthread_t tid;
133 |     int clientDesc;
134 |     int *puntClientDesc;
135 |     while (1 == 1) {
136 |         if (listen(socketDesc, 10) < 0)
137 |             perror("Impossibile mettersi in ascolto"), exit(-1);
138 |         printf("In ascolto..\n");
139 |         if ((clientDesc = accept(socketDesc, NULL, NULL)) < 0) {

```

```

140     perror("Impossibile effettuare connessione\n");
141     exit(-1);
142 }
143 printf("Nuovo client connesso\n");
144 struct sockaddr_in address;
145 socklen_t size = sizeof(struct sockaddr_in);
146 if (getpeername(clientDesc, (struct sockaddr *)&address, &size) < 0) {
147     perror("Impossibile ottenere l'indirizzo del client");
148     exit(-1);
149 }
150 //Estrapolazione indirizzo ip del client
151 char clientAddr[20];
152 strcpy(clientAddr, inet_ntoa(address.sin_addr));
153 Args args = (Args)malloc(sizeof(struct argsToSend));
154 args->userName = (char *)calloc(MAX_BUF, 1);
155 strcpy(args->userName, clientAddr);
156 args->flag = 2;
157 pthread_t tid;
158 //avvio thread di scrittura dell'indirizzo sul file di Log
159 pthread_create(&tid, NULL, fileWriter, (void *)args);
160
161 puntClientDesc = (int *)malloc(sizeof(int));
162 *puntClientDesc = clientDesc;
163 //avvio del thread di gestione del client
164 pthread_create(&tid, NULL, gestisci, (void *)puntClientDesc);
165 }
166 close(clientDesc);
167 quitServer();
168 }
169 struct sockaddr_in configuraIndirizzo() {
170     struct sockaddr_in mio_indirizzo;
171     mio_indirizzo.sin_family = AF_INET;
172     mio_indirizzo.sin_port = htons(5200);
173     mio_indirizzo.sin_addr.s_addr = htonl(INADDR_ANY);
174     printf("Indirizzo socket configurato\n");
175     return mio_indirizzo;
176 }
177 void startProceduraGenerazioneMappa() {
178     printf("Inizio procedura generazione mappa\n");
179     pthread_create(&tidGeneratoreMappa, NULL, threadGenerazioneMappa, NULL);
180 }
181 void startTimer() {
182     printf("Thread timer avviato\n");
183     pthread_create(&tidTimer, NULL, timer, NULL);
184 }
185 int tryLogin(int clientDesc, char name[]) {
186     char *userName = (char *)calloc(MAX_BUF, 1);
187     char *password = (char *)calloc(MAX_BUF, 1);
188     int dimName, dimPwd;
189     read(clientDesc, &dimName, sizeof(int));
190     read(clientDesc, &dimPwd, sizeof(int));
191     read(clientDesc, userName, dimName);
192     read(clientDesc, password, dimPwd);
193     int ret = 0;
194     pthread_mutex_lock(&PlayerMutex);
195     if (validateLogin(userName, password, users) &&
196         !isAlreadyLogged(onLineUsers, userName)) {
197         ret = 1;
198         write(clientDesc, "y", 1);
199         strcpy(name, userName);
200         Args args = (Args)malloc(sizeof(struct argsToSend));
201         args->userName = (char *)calloc(MAX_BUF, 1);
202         strcpy(args->userName, name);
203         args->flag = 0;
204         pthread_t tid;
205         pthread_create(&tid, NULL, fileWriter, (void *)args);
206         numeroClientLoggati++;
207         printf("Nuovo client loggato, client loggati : %d\n", numeroClientLoggati);
208         onLineUsers = addPlayer(onLineUsers, userName, clientDesc);
209         printPlayers(onLineUsers);
210         printf("\n");
211     } else {
212         printf("Non validato\n");
213         write(clientDesc, "n", 1);
214         ret = 0;
215     }
216     pthread_mutex_unlock(&PlayerMutex);
217     return ret;
218 }
219 void *gestisci(void *descriptor) {
220     int bufferReceive[2] = {1};
221     int client_sd = *(int *)descriptor;
222     int continua = 1;
223     char name[MAX_BUF];
224     while (continua) {
225         if (read(client_sd, bufferReceive, sizeof(bufferReceive)) < 1) {
226             continua = 0;
227             break;
228         }

```

```

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

```



```

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

```

```

407     if (timerCount > 0 && timerCount <= TIME_LIMIT_IN_SECONDS)
408         return 1;
409     return 0;
410 }
411 int almenoUnPlayerGenerato() {
412     if (playerGenerati > 0)
413         return 1;
414     return 0;
415 }
416 void *timer(void *args) {
417     int cambiato = 1;
418     while (1) {
419         if (almenoUnClientConnesso() && valoreTimerValido() &&
420             almenoUnPlayerGenerato() && almenoUnaMossaFatta()) {
421             cambiato = 1;
422             sleep(1);
423             timerCount--;
424             fprintf(stdout, "Time left: %d\n", timerCount);
425         } else if (numeroClientLoggati == 0) {
426             timerCount = TIME_LIMIT_IN_SECONDS;
427             if (cambiato) {
428                 fprintf(stdout, "Time left: %d\n", timerCount);
429                 cambiato = 0;
430             }
431         }
432         if (timerCount == 0 || scoreMassimo == packageLimitNumber) {
433             pthread_mutex_lock(&PlayerGeneratiMutex);
434             playerGenerati = 0;
435             pthread_mutex_unlock(&PlayerGeneratiMutex);
436             pthread_mutex_lock(&numMosseMutex);
437             numMosse = 0;
438             pthread_mutex_unlock(&numMosseMutex);
439             printf("Reset timer e generazione nuova mappa..\n");
440             startProceduraGenrazioneMappa();
441             pthread_join(tidGeneratoreMappa, NULL);
442             turno++;
443             pthread_mutex_lock(&ScoreMassimoMutex);
444             scoreMassimo = 0;
445             pthread_mutex_unlock(&ScoreMassimoMutex);
446             timerCount = TIME_LIMIT_IN_SECONDS;
447         }
448     }
449 }
450
451 void configuraSocket(struct sockaddr_in mio_indirizzo) {
452     if ((socketDesc = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
453         perror("Impossibile creare socket");
454         exit(-1);
455     }
456     if (setsockopt(socketDesc, SOL_SOCKET, SO_REUSEADDR, &(int){1}, sizeof(int)) <
457         0)
458         perror("Impossibile impostare il riutilizzo dell'indirizzo ip e della "
459             "porta\n");
460     if ((bind(socketDesc, (struct sockaddr *)&mio_indirizzo,
461         sizeof(mio_indirizzo))) < 0) {
462         perror("Impossibile effettuare bind");
463         exit(-1);
464     }
465 }
466
467 PlayerStats gestisciInput(char grigliaDiGioco[ROWS][COLUMNS],
468     char grigliaOstacoli[ROWS][COLUMNS], char input,
469     PlayerStats giocatore, Obstacles *listaOstacoli,
470     Point deployCoords[], Point packsCoords[],
471     char name[]) {
472     if (giocatore == NULL) {
473         return NULL;
474     }
475     if (input == 'w' || input == 'W') {
476         giocatore = gestisciW(grigliaDiGioco, grigliaOstacoli, giocatore,
477             listaOstacoli, deployCoords, packsCoords);
478     } else if (input == 's' || input == 'S') {
479         giocatore = gestisciS(grigliaDiGioco, grigliaOstacoli, giocatore,
480             listaOstacoli, deployCoords, packsCoords);
481     } else if (input == 'a' || input == 'A') {
482         giocatore = gestisciA(grigliaDiGioco, grigliaOstacoli, giocatore,
483             listaOstacoli, deployCoords, packsCoords);
484     } else if (input == 'd' || input == 'D') {
485         giocatore = gestisciD(grigliaDiGioco, grigliaOstacoli, giocatore,
486             listaOstacoli, deployCoords, packsCoords);
487     } else if (input == 'p' || input == 'P') {
488         giocatore = gestisciP(grigliaDiGioco, giocatore, deployCoords, packsCoords);
489     } else if (input == 'c' || input == 'C') {
490         giocatore =
491             gestisciC(grigliaDiGioco, giocatore, deployCoords, packsCoords, name);
492     }
493     // aggiorna la posizione dell'utente
494     return giocatore;
495 }

```

```

496 }
497
498 PlayerStats gestisciC(char grigliaDiGioco[ROWS][COLUMNS], PlayerStats giocatore,
499                     Point deployCoords[], Point packsCoords[], char name[]) {
500     pthread_t tid;
501     if (giocatore->hasApack == 0) {
502         return giocatore;
503     } else {
504         if (isOnCorrectDeployPoint(giocatore, deployCoords)) {
505             Args args = (Args)malloc(sizeof(struct argsToSend));
506             args->userName = (char *)calloc(MAX_BUF, 1);
507             strcpy(args->userName, name);
508             args->flag = 1;
509             pthread_create(&tid, NULL, fileWriter, (void *)args);
510             giocatore->score += 10;
511             if (giocatore->score > scoreMassimo) {
512                 pthread_mutex_lock(&ScoreMassimoMutex);
513                 scoreMassimo = giocatore->score;
514                 fprintf(stdout, "Score massimo: %d\n", scoreMassimo);
515                 pthread_mutex_unlock(&ScoreMassimoMutex);
516             }
517             giocatore->deploy[0] = -1;
518             giocatore->deploy[1] = -1;
519             giocatore->hasApack = 0;
520         } else {
521             if (!isOnAPack(giocatore, packsCoords) &&
522                 !isOnADeployPoint(giocatore, deployCoords)) {
523                 int index = getHiddenPack(packsCoords);
524                 if (index >= 0) {
525                     packsCoords[index]->x = giocatore->position[0];
526                     packsCoords[index]->y = giocatore->position[1];
527                     giocatore->hasApack = 0;
528                     giocatore->deploy[0] = -1;
529                     giocatore->deploy[1] = -1;
530                 }
531             } else
532                 return giocatore;
533         }
534     }
535     return giocatore;
536 }
537
538 void sendPlayerList(int clientDesc) {
539     int lunghezza = 0;
540     char name[100];
541     Players tmp = onLineUsers;
542     int numeroClientLoggati = dimensioneLista(tmp);
543     printf("%d ", numeroClientLoggati);
544     if (!clientDisconnesso(clientDesc)) {
545         write(clientDesc, &numeroClientLoggati, sizeof(numeroClientLoggati));
546         while (numeroClientLoggati > 0 && tmp != NULL) {
547             strcpy(name, tmp->name);
548             lunghezza = strlen(tmp->name);
549             write(clientDesc, &lunghezza, sizeof(lunghezza));
550             write(clientDesc, name, lunghezza);
551             tmp = tmp->next;
552             numeroClientLoggati--;
553         }
554     }
555 }
556
557 void prepareMessageForPackDelivery(char message[], char username[],
558                                   char date[]) {
559     strcat(message, "Pack delivered by \"");
560     strcat(message, username);
561     strcat(message, "\" at ");
562     strcat(message, date);
563     strcat(message, "\n");
564 }
565
566 void prepareMessageForLogin(char message[], char username[], char date[]) {
567     strcat(message, username);
568     strcat(message, "\" logged in at ");
569     strcat(message, date);
570     strcat(message, "\n");
571 }
572
573 void prepareMessageForConnection(char message[], char ipAddress[],
574                                  char date[]) {
575     strcat(message, ipAddress);
576     strcat(message, "\" connected at ");
577     strcat(message, date);
578     strcat(message, "\n");
579 }
580
581 void putCurrentDateAndTimeInString(char dateAndTime[]) {
582     time_t t = time(NULL);
583     struct tm *infoTime = localtime(&t);
584     strftime(dateAndTime, 64, "%X %x", infoTime);

```

```

585 }
586
587 void *fileWriter(void *args) {
588     int fDes = open("Log", O_RDWR | O_CREAT | O_APPEND, S_IWUSR | S_IRUSR);
589     if (fDes < 0) {
590         perror("Error while opening log file");
591         exit(-1);
592     }
593     Args info = (Args)args;
594     char dateAndTime[64];
595     putCurrentDateAndTimeInString(dateAndTime);
596     if (logDelPacco(info->flag)) {
597         char message[MAX_BUF] = "";
598         prepareMessageForPackDelivery(message, info->userName, dateAndTime);
599         pthread_mutex_lock(&LogMutex);
600         write(fDes, message, strlen(message));
601         pthread_mutex_unlock(&LogMutex);
602     } else if (logDelLogin(info->flag)) {
603         char message[MAX_BUF] = "";
604         prepareMessageForLogin(message, info->userName, dateAndTime);
605         pthread_mutex_lock(&LogMutex);
606         write(fDes, message, strlen(message));
607         pthread_mutex_unlock(&LogMutex);
608     } else if (logDellaConnessione(info->flag)) {
609         char message[MAX_BUF] = "";
610         prepareMessageForConnection(message, info->userName, dateAndTime);
611         pthread_mutex_lock(&LogMutex);
612         write(fDes, message, strlen(message));
613         pthread_mutex_unlock(&LogMutex);
614     }
615     close(fDes);
616     free(info);
617     pthread_exit(NULL);
618 }
619
620 void spostaPlayer(char griglia[ROWS][COLUMNS], int vecchiaPosizione[2],
621                 int nuovaPosizione[2], Point deployCoords[],
622                 Point packsCoords[]) {
623
624     pthread_mutex_lock(&MatrixMutex);
625     griglia[nuovaPosizione[0]][nuovaPosizione[1]] = 'P';
626     if (eraUnPuntoDepo(vecchiaPosizione, deployCoords))
627         griglia[vecchiaPosizione[0]][vecchiaPosizione[1]] = '_';
628     else if (eraUnPacco(vecchiaPosizione, packsCoords))
629         griglia[vecchiaPosizione[0]][vecchiaPosizione[1]] = '$';
630     else
631         griglia[vecchiaPosizione[0]][vecchiaPosizione[1]] = '-';
632     pthread_mutex_unlock(&MatrixMutex);
633 }
634
635 PlayerStats gestisciW(char grigliaDiGioco[ROWS][COLUMNS],
636                      char grigliaOstacoli[ROWS][COLUMNS],
637                      PlayerStats giocatore, Obstacles *listaOstacoli,
638                      Point deployCoords[], Point packsCoords[]) {
639     if (giocatore == NULL)
640         return NULL;
641     int nuovaPosizione[2];
642     nuovaPosizione[1] = giocatore->position[1];
643     // Aggiorna la posizione vecchia spostando il player avanti di 1
644     nuovaPosizione[0] = (giocatore->position[0]) - 1;
645     int nuovoScore = giocatore->score;
646     int nuovoDeploy[2];
647     nuovoDeploy[0] = giocatore->deploy[0];
648     nuovoDeploy[1] = giocatore->deploy[1];
649     if (nuovaPosizione[0] >= 0 && nuovaPosizione[0] < ROWS) {
650         if (casellaVuotaOValida(grigliaDiGioco, grigliaOstacoli, nuovaPosizione)) {
651             spostaPlayer(grigliaDiGioco, giocatore->position, nuovaPosizione,
652                         deployCoords, packsCoords);
653         } else if (colpitoOstacolo(grigliaOstacoli, nuovaPosizione)) {
654             *listaOstacoli =
655                 addObstacle(*listaOstacoli, nuovaPosizione[0], nuovaPosizione[1]);
656             nuovaPosizione[0] = giocatore->position[0];
657             nuovaPosizione[1] = giocatore->position[1];
658         } else if (colpitoPlayer(grigliaDiGioco, nuovaPosizione)) {
659             nuovaPosizione[0] = giocatore->position[0];
660             nuovaPosizione[1] = giocatore->position[1];
661         }
662         giocatore->deploy[0] = nuovoDeploy[0];
663         giocatore->deploy[1] = nuovoDeploy[1];
664         giocatore->score = nuovoScore;
665         giocatore->position[0] = nuovaPosizione[0];
666         giocatore->position[1] = nuovaPosizione[1];
667     }
668     return giocatore;
669 }
670
671 PlayerStats gestisciD(char grigliaDiGioco[ROWS][COLUMNS],
672                      char grigliaOstacoli[ROWS][COLUMNS],
673                      PlayerStats giocatore, Obstacles *listaOstacoli,

```

```

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

```

```

763     spostaPlayer(grigliaDiGioco, giocatore->position, nuovaPosizione,
764                 deployCoords, packsCoords);
765 } else if (colpitoOstacolo(grigliaOstacoli, nuovaPosizione)) {
766     printf("Ostacolo\n");
767     *listaOstacoli =
768         addObstacle(*listaOstacoli, nuovaPosizione[0], nuovaPosizione[1]);
769     nuovaPosizione[0] = giocatore->position[0];
770     nuovaPosizione[1] = giocatore->position[1];
771 } else if (colpitoPlayer(grigliaDiGioco, nuovaPosizione)) {
772     nuovaPosizione[0] = giocatore->position[0];
773     nuovaPosizione[1] = giocatore->position[1];
774 }
775 giocatore->deploy[0] = nuovoDeploy[0];
776 giocatore->deploy[1] = nuovoDeploy[1];
777 giocatore->score = nuovoScore;
778 giocatore->position[0] = nuovaPosizione[0];
779 giocatore->position[1] = nuovaPosizione[1];
780 }
781 return giocatore;
782 }
783
784 int logDelPacco(int flag) {
785     if (flag == 1)
786         return 1;
787     return 0;
788 }
789
790 int logDelLogin(int flag) {
791     if (flag == 0)
792         return 1;
793     return 0;
794 }
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 rimuoviPaccoFromArray(int posizione[2], Point packsCoords[]);
23 void printMenu();
24 int getHiddenPack(Point packsCoords[]);
25 int casellaVuotaOValida(char grigliaDiGioco[ROWS][COLUMNS],
26                         char grigliaOstacoli[ROWS][COLUMNS], int posizione[2]);
27 void stampaIstruzioni(int i);
28 int isOnADeployPoint(PlayerStats giocatore, Point deployCoords[]);
29 int isOnCorrectDeployPoint(PlayerStats giocatore, Point deployCoords[]);
30 PlayerStats gestisciP(char grigliaDiGioco[ROWS][COLUMNS], PlayerStats giocatore,
31                      Point deployCoords[], Point packsCoords[]);

```

```

32 void inizializzaGiocoSenzaPlayer(char grigliaDiGioco[ROWS][COLUMNS],
33                                 char grigliaConOstacoli[ROWS][COLUMNS],
34                                 Point packsCoords[]);
35 void inserisciPlayerNellaGrigliaInPosizioneCasuale(
36     char grigliaDiGioco[ROWS][COLUMNS], char grigliaOstacoli[ROWS][COLUMNS],
37     int posizione[2]);
38 void inizializzaGrigliaVuota(char grigliaDiGioco[ROWS][COLUMNS]);
39 void generaPosizioneOstacoli(char grigliaDiGioco[ROWS][COLUMNS],
40                              char grigliaOstacoli[ROWS][COLUMNS]);
41 void riempiGrigliaConPacchiInPosizioniGenerateCasualmente(
42     char grigliaDiGioco[ROWS][COLUMNS], Point packsCoords[]);
43 void printGrid(char grigliaDaStampare[ROWS][COLUMNS], PlayerStats stats);
44 void start(char grigliaDiGioco[ROWS][COLUMNS],
45            char grigliaOstacoli[ROWS][COLUMNS]);
46 void riempiGrigliaConGliOstacoli(char grigliaDiGioco[ROWS][COLUMNS],
47                                  char grigliaOstacoli[ROWS][COLUMNS]);
48 void generaPosizioniRaccolta(char grigliaDiGioco[ROWS][COLUMNS],
49                              char grigliaOstacoli[ROWS][COLUMNS],
50                              Point coord[]);
51 void mergeGridAndList(char grid[ROWS][COLUMNS], Obstacles top);
52 void scegliPosizioneRaccolta(Point coord[], int deploy[]);
53 int colpitoOstacolo(char grigliaOstacoli[ROWS][COLUMNS], int posizione[2]);
54 int colpitoPacco(Point packsCoords[], int posizione[2]);
55 int colpitoPlayer(char grigliaDiGioco[ROWS][COLUMNS], int posizione[2]);
56 int casellaVuota(char grigliaDiGioco[ROWS][COLUMNS],
57                  char grigliaOstacoli[ROWS][COLUMNS], int posizione[2]);
58 int arrivatoADestinazione(int posizione[2], int destinazione[2]);
59 int eraUnPuntoDepo(int vecchiaPosizione[2], Point depo[]);
60 int eraUnPacco(int vecchiaPosizione[2], Point packsCoords[]);
61 int isOnAPack(PlayerStats giocatore, Point packsCoords[]);

```

Listato 19: Codice sorgente utility del gioco 1

```

1  #include "boardUtility.h"
2  #include "list.h"
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6  #include <unistd.h>
7  void printMenu() {
8      system("clear");
9      printf("\t Cosa vuoi fare?\n");
10     printf("\t1 Gioca\n");
11     printf("\t2 Registrati\n");
12     printf("\t3 Esci\n");
13 }
14 int colpitoOstacolo(char grigliaOstacoli[ROWS][COLUMNS], int posizione[2]) {
15     if (grigliaOstacoli[posizione[0]][posizione[1]] == 'O')
16         return 1;
17     return 0;
18 }
19 int colpitoPacco(Point packsCoords[], int posizione[2]) {
20     int i = 0;
21     for (i = 0; i < numberOfPackages; i++) {
22         if (packsCoords[i]->x == posizione[0] && packsCoords[i]->y == posizione[1])
23             return 1;
24     }
25     return 0;
26 }
27 int casellaVuotaOValida(char grigliaDiGioco[ROWS][COLUMNS],
28                          char grigliaOstacoli[ROWS][COLUMNS], int posizione[2]) {
29     if (grigliaDiGioco[posizione[0]][posizione[1]] == '-' || // casella vuota
30         grigliaDiGioco[posizione[0]][posizione[1]] == '_' || // punto deploy
31         grigliaDiGioco[posizione[0]][posizione[1]] == '$') // pacco
32         if (grigliaOstacoli[posizione[0]][posizione[1]] == '-' ||
33             grigliaOstacoli[posizione[0]][posizione[1]] == '_' ||
34             grigliaOstacoli[posizione[0]][posizione[1]] == '$')
35             return 1;
36     return 0;
37 }
38 int colpitoPlayer(char grigliaDiGioco[ROWS][COLUMNS], int posizione[2]) {
39     if (grigliaDiGioco[posizione[0]][posizione[1]] == 'P')
40         return 1;
41     return 0;
42 }
43 int isOnCorrectDeployPoint(PlayerStats giocatore, Point deployCoords[]) {
44     int i = 0;
45     for (i = 0; i < numberOfPackages; i++) {
46         if (giocatore->deploy[0] == deployCoords[i]->x &&
47             giocatore->deploy[1] == deployCoords[i]->y) {
48             if (deployCoords[i]->x == giocatore->position[0] &&
49                 deployCoords[i]->y == giocatore->position[1])
50                 return 1;
51         }
52     }
53     return 0;
54 }
55 int getHiddenPack(Point packsCoords[]) {

```

```

56     int i = 0;
57     for (i = 0; i < numberOfPackages; i++) {
58         if (packsCoords[i]->x == -1 && packsCoords[i]->y == -1)
59             return i;
60     }
61     return -1;
62 }
63 int isOnAPack(PlayerStats giocatore, Point packsCoords[]) {
64     int i = 0;
65     for (i = 0; i < numberOfPackages; i++) {
66         if (giocatore->position[0] == packsCoords[i]->x &&
67             giocatore->position[1] == packsCoords[i]->y)
68             return 1;
69     }
70     return 0;
71 }
72 int isOnADeployPoint(PlayerStats giocatore, Point deployCoords[]) {
73     int i = 0;
74     for (i = 0; i < numberOfPackages; i++) {
75         if (giocatore->position[0] == deployCoords[i]->x &&
76             giocatore->position[1] == deployCoords[i]->y)
77             return 1;
78     }
79     return 0;
80 }
81 void inizializzaGrigliaVuota(char griglia[ROWS][COLUMNS]) {
82     int i = 0, j = 0;
83     for (i = 0; i < ROWS; i++) {
84         for (j = 0; j < COLUMNS; j++) {
85             griglia[i][j] = '-';
86         }
87     }
88 }
89 PlayerStats gestisciP(char grigliaDiGioco[ROWS][COLUMNS], PlayerStats giocatore,
90                      Point deployCoords[], Point packsCoords[]) {
91     int nuovoDeploy[2];
92     if (colpitoPacco(packsCoords, giocatore->position) &&
93         giocatore->hasApack == 0) {
94         scegliPosizioneRaccolta(deployCoords, nuovoDeploy);
95         giocatore->hasApack = 1;
96         rimuoviPaccoFromArray(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\t Inviare 't' per il timer.");
133     if (i == 2)
134         printf("\t\t Inviare 'e' per uscire");
135     if (i == 3)
136         printf("\t\t Inviare 'p' per raccogliere un pacco");
137     if (i == 4)
138         printf("\t\t Inviare 'c' per consegnare il pacco");
139     if (i == 5)
140         printf("\t\t Inviare 'w'/'s' per andare sopra/sotto");
141     if (i == 6)
142         printf("\t\t Inviare 'a'/'d' per andare a dx/sx");
143     if (i == 7)
144         printf("\t\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));
232     for (i = 0; i < numberOfPackages; i++) {
233         x = rand() % COLUMNS;

```

```

234     y = rand() % ROWS;
235     if (grigliaDiGioco[y][x] == '-') {
236         grigliaDiGioco[y][x] = '$';
237         packsCoords[i]->x = y;
238         packsCoords[i]->y = x;
239     } else
240         i--;
241 }
242 }
243 /*Inserisci gli ostacoli nella griglia di gioco */
244 void riempiGrigliaConGliOstacoli(char grigliaDiGioco[ROWS][COLUMNS],
245                                char grigliaOstacoli[ROWS][COLUMNS]) {
246     int i, j = 0;
247     for (i = 0; i < ROWS; i++) {
248         for (j = 0; j < COLUMNS; j++) {
249             if (grigliaOstacoli[i][j] == 'O')
250                 grigliaDiGioco[i][j] = 'O';
251         }
252     }
253 }
254 void inserisciPlayerNellaGrigliaInPosizioneCasuale(
255     char grigliaDiGioco[ROWS][COLUMNS], char grigliaOstacoli[ROWS][COLUMNS],
256     int posizione[2]) {
257     int x, y;
258     srand(time(0));
259     printf("Inserisco player\n");
260     do {
261         x = rand() % COLUMNS;
262         y = rand() % ROWS;
263     } while (grigliaDiGioco[y][x] != '-' && grigliaOstacoli[y][x] != '-');
264     grigliaDiGioco[y][x] = 'P';
265     posizione[0] = y;
266     posizione[1] = x;
267 }
268 void inizializzaGiocoSenzaPlayer(char grigliaDiGioco[ROWS][COLUMNS],
269                                 char grigliaConOstacoli[ROWS][COLUMNS],
270                                 Point packsCoords[]) {
271     inizializzaGrigliaVuota(grigliaDiGioco);
272     riempiGrigliaConPacchiInPosizioniGenerateCasualmente(grigliaDiGioco,
273                                                            packsCoords);
274     generaPosizioneOstacoli(grigliaDiGioco, grigliaConOstacoli);
275     return;
276 }
277
278 int eraUnPuntoDepo(int vecchiaPosizione[2], Point depo[]) {
279     int i = 0, ret = 0;
280     while (ret == 0 && i < numberOfPackages) {
281         if ((depo[i]->y == vecchiaPosizione[1] &&
282             (depo[i]->x == vecchiaPosizione[0])) {
283             ret = 1;
284         }
285         i++;
286     }
287     return ret;
288 }
289 int eraUnPacco(int vecchiaPosizione[2], Point packsCoords[]) {
290     int i = 0, ret = 0;
291     while (ret == 0 && i < numberOfPackages) {
292         if ((packsCoords[i]->y == vecchiaPosizione[1] &&
293             (packsCoords[i]->x == vecchiaPosizione[0])) {
294             ret = 1;
295         }
296         i++;
297     }
298     return ret;
299 }
300
301 int arrivatoADestinazione(int posizione[2], int destinazione[2]) {
302     if (posizione[0] == destinazione[0] && posizione[1] == destinazione[1])
303         return 1;
304     return 0;
305 }

```

## A.4 Codice sorgente list

**Listato 20:** Codice header utility del gioco 2

```

1 #ifndef DEF_LIST_H
2 #define DEF_LIST_H
3 #define MAX_BUF 200
4 #include <pthread.h>
5 // players
6 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 è già 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
16 PlayerStats initStats(int deploy[], int score, int position[], int flag) {
17     PlayerStats L = (PlayerStats)malloc(sizeof(struct Data));
18     L->deploy[0] = deploy[0];
19     L->deploy[1] = deploy[1];
20     L->score = score;
21     L->hasApack = flag;
22     L->position[0] = position[0];
23     L->position[1] = position[1];

```

```

23     return L;
24 }
25 Obstacles initObstacleNode(int x, int y) {
26     Obstacles L = (Obstacles)malloc(sizeof(struct TList2));
27     L->x = x;
28     L->y = y;
29     L->next = NULL;
30     return L;
31 }
32 Obstacles addObstacle(Obstacles L, int x, int y) {
33     Obstacles tmp = initObstacleNode(x, y);
34     if (L != NULL)
35         tmp->next = L;
36     return tmp;
37 }
38 int dimensioneLista(Players L) {
39     int size = 0;
40     Players tmp = L;
41     while (tmp != NULL) {
42         size++;
43         tmp = tmp->next;
44     }
45     return size;
46 }
47 int isAlreadyLogged(Players L, char *name) {
48     int ret = 0;
49     if (L != NULL) {
50         if (strcmp(L->name, name) == 0)
51             return 1;
52         ret = isAlreadyLogged(L->next, name);
53     }
54     return ret;
55 }
56 Players addPlayer(Players L, char *name, int sockDes) {
57     Players tmp = initPlayerNode(name, sockDes);
58     if (L != NULL)
59         tmp->next = L;
60     return tmp;
61 }
62 Players removePlayer(Players L, int sockDes) {
63     if (L != NULL) {
64         if (L->sockDes == sockDes) {
65             Players tmp = L->next;
66             free(L);
67             return tmp;
68         }
69         L->next = removePlayer(L->next, sockDes);
70     }
71     return L;
72 }
73 void freePlayers(Players L) {
74     if (L != NULL) {
75         freePlayers(L->next);
76         free(L);
77     }
78 }
79 void freeObstacles(Obstacles L) {
80     if (L != NULL) {
81         freeObstacles(L->next);
82         free(L);
83     }
84 }
85 void printPlayers(Players L) {
86     if (L != NULL) {
87         printf("%s ->", L->name);
88         printPlayers(L->next);
89     }
90     printf("\n");
91 }
92 void printObstacles(Obstacles L) {
93     if (L != NULL) {
94         printf("X:%d Y:%d ->", L->x, L->y);
95         printObstacles(L->next);
96     }
97 }

```

## A.5 Codice sorgente parser

**Listato 22:** Codice header utility del gioco 3

```

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

```

```
6 void premiEnterPerContinuare();
```

**Listato 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

1	Configurazione indirizzo del server . . . . .	5
2	Configurazione socket del server . . . . .	5
3	Procedura di ascolto del server . . . . .	5
4	Configurazione e connessione del client . . . . .	6
5	Risoluzione url del client . . . . .	6
6	Prima comunicazione del server . . . . .	7
7	Prima comunicazione del client . . . . .	7
8	Funzione play del server . . . . .	8
9	Funzione play del client . . . . .	9
10	Funzione di gestione del timer . . . . .	10
11	Generazione nuova mappa e posizione players . . . . .	10
12	Funzione di log . . . . .	11
13	Funzione spostaPlayer . . . . .	11
14	"Gestione del login 1" . . . . .	12
15	"Gestione del login 2" . . . . .	12
16	Codice sorgente del client . . . . .	14
17	Codice sorgente del server . . . . .	17
18	Codice header utility del gioco 1 . . . . .	26
19	Codice sorgente utility del gioco 1 . . . . .	27
20	Codice header utility del gioco 2 . . . . .	30
21	Codice sorgente utility del gioco 2 . . . . .	31
22	Codice header utility del gioco 3 . . . . .	32
23	Codice sorgente utility del gioco 3 . . . . .	33