

UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
SCUOLA POLITECNICA E DELLE SCIENZE DI BASE
DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE



CORSO DI LAUREA IN INFORMATICA
INSEGNAMENTO DI **LABORATORIO DI SISTEMI OPERATIVI**
ANNO ACCADEMICO 2019/2020

Progettazione e sviluppo di un'applicazione Client – Server, multithread, con l'ausilio delle system call del sistema UNIX.

Autori:

Gennaro **SORRENTINO**

MATRICOLA **N86/2351**

gennaro.sorrentino5@studenti.unina.it

Gianluca **L'ARCO**

MATRICOLA **N86/2799**

g.larco@studenti.unina.it

Docente:

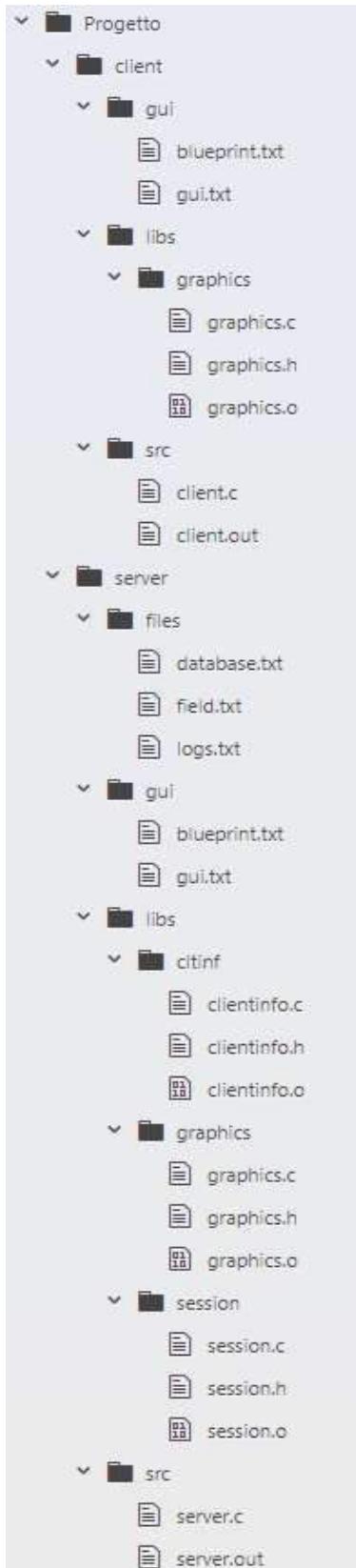
Prof. Alberto **FINZI**

Guida d'uso

- ◆ **Compilazione**
 - ◆ Compilazione manuale
 - ◆ Script di compilazione
- ◆ **Introduzione al Server**
 - ◆ Avvio del programma
 - ◆ Interfaccia grafica
- ◆ **Introduzione al Client**
 - ◆ Avvio del programma
 - ◆ Interfaccia grafica

Compilazione

Prima di procedere alla compilazione del codice prendiamo confidenza con la struttura del Progetto:



Chiaramente la cartella principale è **Progetto**, al suo interno risiedono la cartella **client** e quella **server**, oltre al file **comp.out** utilizzato nella compilazione “automatica”.

All'interno della cartella **client** troviamo:

- ♦ **gui**: All'interno di questa cartella risiedono i file **blueprint.txt** e **gui.txt**. Il primo viene utilizzato nella mappatura del terminale con la grafica (individuazione delle coordinate del cursore), mentre il secondo viene impiegato per l'inizializzazione dell'interfaccia grafica del client.
- ♦ **libs**: All'interno di questa cartella risiedono le librerie (personal) utilizzate all'interno del programma Client. In questo caso all'interno di **libs** troviamo la sola cartella **graphics** (la quale a sua volta contiene la libreria **graphics.h** → la libreria **graphics.h** contiene funzioni utili alla gestione dell'interfaccia grafica, oltre a molte costanti).
- ♦ **src**: All'interno di questa cartella risiede il codice sorgente del Client **client.c**.

All'interno della cartella **server** troviamo:

- ♦ **files**: All'interno di questa cartella risiedono i file **database.txt** (contiene le credenziali degli utenti registrati), **field.txt** (file utilizzato nella generazione della mappa di ogni sessione di gioco) e **logs.txt** (file contenente i logs del programma server, ossia informazioni quali: connessione, disconnessione, registrazione, login, generazione mappa, vincitore partita, trasporto pacchetti).
- ♦ **gui**: All'interno di questa cartella risiedono i file **blueprint.txt** e **gui.txt**. I due hanno lo stesso utilizzo di quelli del client.
- ♦ **libs**: Anche nel Server nella cartella **libs** risiedono le librerie (personal) utilizzate all'interno del programma Server. Oltre alla libreria **graphics**, troviamo altre due librerie: **cltinf** (librerie **clientinfo.h** → utilizzata per gestire le informazioni relative ai client connessi) e **session** (libreria **session.h** → utilizzata per la creazione e gestione di una sessione, contiene principalmente strutture).
- ♦ **src**: All'interno di questa cartella risiede il codice sorgente del Server **server.c**.

Possiamo ora procedere alla compilazione del codice, la quale può avvenire in due differenti modi:

♦ Compilazione manuale

Innanzitutto posizioniamoci nella cartella principale **Progetto**, nel nostro caso supponiamo che la cartella risieda in **Downloads**.

```
cd /home/<username>/Downloads/Progetto
```

Procediamo quindi con la compilazione del Client:

```
# Innanzitutto compiliamo la libreria graphics
cd client/libs/graphics
gcc -c graphics.c
# Ci spostiamo quindi nella cartella src e compiliamo il codice sorgente del Client.
cd ..; cd ..; cd src
gcc -o client.out client.c ../libs/graphics/graphics.o -pthread
# Infine ci riposizioniamo nella cartella Progetto
cd ..; cd ..
```

Procediamo ora con la compilazione del Server:

```
# Compiliamo la libreria grafica
cd server/libs/graphics
gcc -c graphics.c
# Compiliamo la libreria session
cd ..; cd session
gcc -c session.c
# Compiliamo la libreria cltinf
cd ..; cd cltinf
gcc -c clientinfo.c
# Ci spostiamo quindi nella cartella src e compiliamo il codice sorgente del Server.
cd ..; cd ..; cd src
gcc -o server.out server.c ../libs/session/session.o ../libs/cltinf/clientinfo.o
../libs/graphics/graphics.o -pthread
```

♦ Compilazione mediante script

Innanzitutto posizioniamoci nella cartella principale **Progetto**, nel nostro caso supponiamo che la cartella risieda in **Downloads**.

```
cd /home/<username>/Downloads/Progetto
```

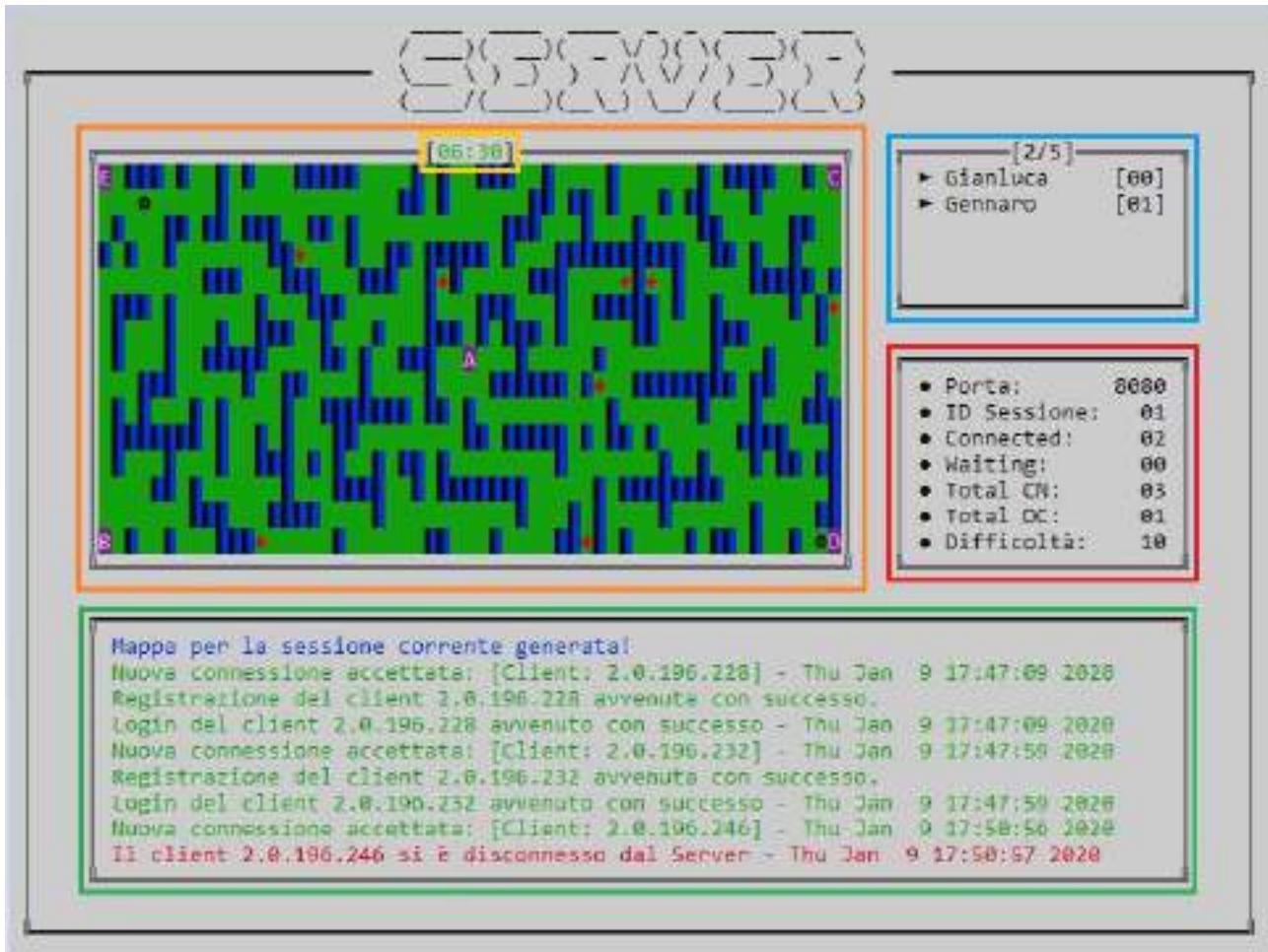
A questo punto basterà eseguire lo script **comp.out**.

```
./comp.out
```

Al termine di una delle due fasi di compilazione entrambi i programmi Client e Server saranno pronti per essere avviati. Per farlo basterà recarsi nelle rispettive cartelle **Progetto/client/src** → **./client.out** e **Progetto/server/src** → **./server.out**.

Introduzione al Server

Una volta avviato, il Server chiederà di inserire la porta su cui mettersi in ascolto e la difficoltà del gioco la quale si estende da un livello pari ad 1 (nessun ostacolo) fino ad arrivare a 10 (possibilità di generazione degli ostacoli massima). È anche possibile impostare una difficoltà randomica per ogni sessione, inserendo come difficoltà 0. A questo punto verrà avviata la schermata principale:



Si elencano ora le funzionalità delle varie componenti:

- ◆ **Giocatori connessi:** La componente individuata dal rettangolo celeste contiene i giocatori attualmente connessi alla sessione di gioco (ossia coloro che sono in partita). Ogni giocatore è individuato dal suo nome (che ricordiamo essere univoco) e dal numero di pacchetti consegnati.
- ◆ **Status del Server:** La componente individuata dal rettangolo rosso contiene informazioni riguardanti lo stato attuale del Server:
 - ◆ **Porta:** Porta sul quale il Server è in ascolto.
 - ◆ **ID Sessione:** Numero della sessione corrente (parte da 1).
 - ◆ **Connected:** Numero di client attualmente connessi al Server.
 - ◆ **Waiting:** Numero di client attualmente in attesa di entrare nella sessione di gioco.
 - ◆ **Total CN:** Numero di connessioni totali.
 - ◆ **Total DC:** Numero di disconnessioni totali.
 - ◆ **Difficoltà:** Difficoltà della sessione di gioco corrente.

- ◆ **Chat Logs:** La componente individuata dal rettangolo verde contiene alcuni dei logs generati durante l'esecuzione del Server. Le informazioni visualizzate dalla chat logs sono: generazione mappe, connessione di un client, registrazione di un client, login di un client, disconnessione di un client ed eventuali messaggi di errore.
- ◆ **Timer:** La componente individuata dal rettangolo giallo contiene il timer di sessione. Il timer di sessione è inizializzato a 10 minuti ed è avviato non appena un Client fa accesso nella sessione di gioco.
- ◆ **Campo di gioco:** La componente individuata dal rettangolo arancione contiene la mappa generata dal Server per la sessione di gioco corrente. I movimenti e le azioni effettuate dai Clients sono visibili dal Server mediante questa componente.

I blocchi blu e neri indicano gli ostacoli, mentre i rombi rossi indicano i pacchetti da raccogliere. Le emoji indicano i giocatori, infine i blocchi magenta rappresentano invece le locazioni in cui trasportare i pacchetti.

La generazione della mappa è randomica:

- ◆ A seconda della difficoltà inserita le mura hanno una certa probabilità di generazione, si parte da un'assenza totale (difficoltà 1), ad una presenza totale (difficoltà 10).
- ◆ La posizione, la quantità dei pacchetti e la locazione in cui trasportare il pacchetto sono anche esse randomiche. Ad ogni sessione di gioco è garantita la generazione di almeno quattro pacchetti.
- ◆ Le locazioni hanno posizioni fisse, tuttavia, il loro nome è randomico. Quindi ad esempio, nella sessione 1 potremmo avere la locazione A al centro, mentre, nella sessione 2, la stessa locazione A potrebbe trovarsi in alto a destra.
- ◆ Lo "spawn" di un giocatore è randomico e si estende su una scelta di otto possibili posizioni di spawn presenti sulla mappa. Lo schema delle posizioni è simmetrico e colloca i giocatori in posizioni tecnicamente bilanciate, anche se ciò dipende fortemente dalla casualità del gioco stesso.

Introduzione al Client

Una volta avviato, il Client chiederà di inserire l'indirizzo e la porta del Server a cui connettersi. Nel caso in cui la connessione dovesse andare a buon fine verrà avviata la schermata principale:



Il Client presenta pressoché le stesse componenti del Server (a differenza di quest'ultimo non possiede lo status) tuttavia in aggiunta possiede una textfield utilizzata per ricevere l'input utente, in più la Chat (che nel Server è una Chat logs, ossia una chat di sole notifiche) può essere utilizzata anche per le comunicazioni client-client, come vedremo più avanti.

A differenza del Server, chiaramente, il Client inizialmente non ha visibilità degli ostacoli, ma solo dei pacchetti, delle locazioni e degli altri giocatori.

Comunicazione Client - Server

- ◆ Comandi Client - Server
 - ◆ Comandi GUEST
 - ◆ Comandi REGISTERED
 - ◆ Comandi LOGGED
 - ◆ Comandi PLAYING
- ◆ Comunicazioni Server - Client
 - ◆ Giocatore (dis)connesso
 - ◆ Timer
 - ◆ Effetto del movimento
- ◆ Chat client - client
 - ◆ Chat globale
 - ◆ Chat di sessione
- ◆ Comunicazione dettagliata
 - ◆ Connessione
 - ◆ Registrazione | Login
 - ◆ Gioco

Comandi Client → Server

La comunicazione Client → Server avviene mediante appositi comandi. Quest'ultimi possono essere suddivisi per fasi: comandi **GUEST** (il client si è appena connesso al Server), comandi **REGISTERED** (il client, precedentemente **GUEST**, ha effettuato la registrazione), comandi **LOGGED** (il client ha effettuato il login) e infine i comandi **PLAYING** (il client è in una sessione di gioco, ossia sta giocando una partita).

Analizziamo i vari comandi:

♦ Comandi GUEST

Una volta connesso al Server, il client assume lo stato di **GUEST** (ospite) e dispone dei seguenti comandi:

- ◆ **@cmd**: Il comando @cmd, fornisce la lista dei comandi attualmente disponibili.

```
Server: Benvenuto/a! Digita @cmd per conoscere i comandi.  
Me: @cmd  
Server: Comandi disponibili -> [@cmd - @login - @signin - @exit]
```

- ◆ **@login**: Il comando @login fa in modo che il Server richieda l'username e la password dell'utente.

```
Me: @login  
Server: Inserisci l'username.  
Me: prova  
Server: Inserisci la password.  
Me: password  
Server: Username o password errati, riprovare.
```

- ◆ **@signin**: Il comando @signin fa in modo che il Server richieda l'username e la password dell'utente per la registrazione dello stesso.

```
Me: @signin  
Server: Inserisci l'username - massimo 10 caratteri.  
Me: prova  
Server: Inserisci la password - massimo 10 caratteri.  
Me: password  
Server: Registrazione avvenuta con successo, ora puoi effettuare il login.
```

- ◆ **@exit**: Il comando @exit effettua la disconnessione del Client dal Server. Nel caso in cui quest'ultimo dovesse essere utilizzato nella fase di login oppure di signin allora l'effetto è quello di uscire dalla fase corrente.

```
Me: @login  
Server: Inserisci l'username.  
Me: @exit  
Server: Comandi disponibili -> [@cmd - @login - @exit]  
Me: @signin  
Server: Comando non disponibile, riprova!
```

```
Server: Benvenuto/a! Digita @cmd per conoscere i comandi.  
Me: @exit  
Server: Torna presto! Disconnessione in corso...  
Server: Sei stato disconnesso dal server...
```

♦ Comandi REGISTERED

Nel caso in cui il client dovesse effettuare la registrazione, il suo stato passerebbe a **REGISTERED**. In tal caso i comandi disponibili sarebbero:

- ◆ **@cmd**: È praticamente uguale al comando **@cmd GUEST**, con l'unica differenza che il comando **@signin** non è ulteriormente specificato:

Me: **@cmd**
Server: Comandi disponibili -> [@cmd - @login - @exit]

- ◆ **@login**: Praticamente invariato.
- ◆ **@exit**: Praticamente invariato.

♦ Comandi LOGGED

Nel caso in cui il client dovesse effettuare il login, il suo stato passerebbe a **LOGGED**. In tal caso i comandi disponibili sarebbero:

- ◆ **@cmd**: Anche in questo caso **@cmd** fornisce la lista dei comandi disponibili.

Me: **@cmd**
Server: Comandi disponibili -> [@cmd - @time - @exit]

- ◆ **@time**: Fornisce il tempo stimato entro il quale, il Client che ha lanciato il comando, giocherà una partita.

Me: **@time**
Server: Giocherai all'incirca tra: 09:31 minuti.

- ◆ **@exit**: Praticamente invariato.

♦ Comandi PLAYING

Se il client dovesse trovarsi in una sessione di gioco, i comandi sarebbero:

- ◆ **@cmd**: Anche in questo caso **@cmd** fornisce la lista dei comandi disponibili.

Me: **@cmd**
Server: Comandi disponibili -> [@S - @N - @E - @O - @P(S/N/E/O) - @D(S/N/E/O)]

- ◆ **@S**: Il comando **@S** richiede lo spostamento del giocatore verso Sud.
- ◆ **@N**: Il comando **@N** richiede lo spostamento del giocatore verso Nord.
- ◆ **@E**: Il comando **@E** richiede lo spostamento del giocatore verso Est.
- ◆ **@O**: Il comando **@O** richiede lo spostamento del giocatore verso Ovest.

Me: **@N**
Server: Wow! Hai trovato un pacco, prendilo!

Me: **@O**
Server: Alt! Un altro giocatore ti impedisce il passaggio!

Me: **@S**
Server: Attento, stavi per cadere giù!

Me: **@E**
Server: Hei, hai un pacco per me?

- ◆ @PS: Il comando @PS tenta di raccogliere un pacchetto verso Sud.
- ◆ @PN: Il comando @PN tenta di raccogliere un pacchetto verso Nord.
- ◆ @PE: Il comando @PE tenta di raccogliere un pacchetto verso Est.
- ◆ @PO: Il comando @PO tenta di raccogliere un pacchetto verso Ovest.

Me: @PE
Server: Trasporta il pacchetto fino alla locazione A!

Me: @PN
Server: Non c'è nulla qui!

- ◆ @DS: Il comando @DS tenta di depositare il pacchetto verso Sud.
- ◆ @DN: Il comando @DN, tenta di depositare il pacchetto verso Nord.
- ◆ @DE: Il comando @DE tenta di depositare il pacchetto verso Est.
- ◆ @DO: Il comando @DO tenta di depositare il pacchetto verso Ovest.

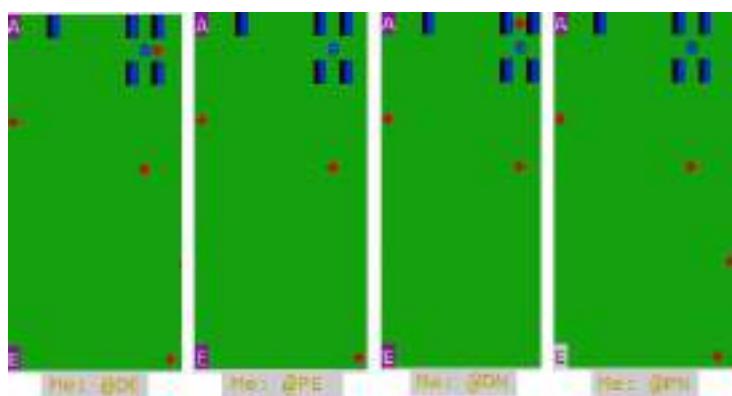
Me: @DE
Server: Non vorrai mica distruggere un muro con un pacchetto?!

Me: @DE
Server: Non puoi lanciare pacchetti contro gli altri giocatori!

Me: @DN
Server: Non puoi posizionare il pacco qui!

- ◆ @exit: Praticamente invariato.

Si osservi che un giocatore, una volta raccolto un pacchetto, può depositarlo in ogni momento, in qualunque direzione lui voglia (sempre se possibile). Nel caso in cui il rilascio di un pacchetto dovesse avvenire in punti della mappa non consentiti (giocatori, mura, fuori mappa, altri pacchetti etc..) verrebbe mostrato un adeguato messaggio di errore, inoltre se l'ostacolo incontrato dal pacchetto dovesse essere un muro allora quest'ultimo verrebbe rivelato al giocatore.



Chiaramente, anche depositando il pacchetto, la locazione di destinazione rimane invariata. Il Server comunica al Client la locazione di destinazione con un messaggio e mediante un comando Server attiva il blink (lampeggio) della stessa, come vedremo più avanti.

Comandi Server → Client

Anche la comunicazione Server → Client avviene mediante opportuni comandi. I comandi impartiti dal Server ai Clients connessi ad esso iniziano con il prefisso **\$**. Chiaramente i Clients non posso inviare messaggi con il prefisso **\$**, quest'ultimo viene infatti eliminato al momento dell'invio.

I comandi Server sono quindi:

- ◆ **\$time SECONDS**: Fornisce ai client connessi alla sessione di gioco, il tempo rimanente.
- ◆ **\$add**: Questo comando è utilizzato per impartire direttive di aggiornamento del campo di gioco e si ramifica in:
 - ◆ **\$add packet ROW COL**: Indica che alla posizione ROW COL della matrice (campo di gioco) è presente un pacchetto.
 - ◆ **\$add location ROW COL NAME**: Indica che alla posizione ROW COL della matrice (campo di gioco) è presente una locazione per il deposito dei pacchetti, il cui identificativo è NAME.
 - ◆ **\$add you ROW COL**: Indica ad un determinato Client, la sua posizione nella matrice.
 - ◆ **\$add player ROW COL**: Indica che alla posizione ROW COL della matrice è presente un giocatore.
 - ◆ **\$add wall ROW COL**: Indica che alla posizione ROW COL della matrice è presente un ostacolo.
 - ◆ **\$add blink ROW COL NAME**: Impartisce al Client l'ordine di far lampeggiare la locazione di destinazione di un pacchetto.
- ◆ **\$update ROW COL**: Anche questo comando fornisce direttive di aggiornamento del campo di gioco, a differenza del comando **\$add** tuttavia il suo scopo è semplicemente quello di aggiornare la posizione ROW COL al valore **GRASS** (ossia erba).
- ◆ **\$remove blink**: Impartisce al Client l'ordine di annullare il blink della locazione precedentemente interessata dal comando **\$add blink ROW COL NAME**.
- ◆ **\$reset**: Indica al Client di ripulire l'intero campo di gioco.
- ◆ **\$disconnected NAME**: Indica al Client che il giocatore NAME si è disconnesso dalla sessione di gioco.
- ◆ **\$connected NAME PACKETS**: Indica al Client che il giocatore NAME si è connesso alla sessione di gioco. Il numero di pacchetti è indicato dal momento che il comando **connected** è utilizzato ogni qualvolta il client si connette ad una sessione di gioco, nel senso che il Client in questione riceve tanti comandi di connessione quanti sono i giocatori già connessi (e che quindi possono avere pacchetti già consegnati).
- ◆ **\$delivered NAME**: Indica al Client che il giocatore NAME ha consegnato un pacchetto.
- ◆ **\$MSG**: Indica al Client che il messaggio MSG è una notifica Server. Il colore delle notifiche Server è il blu.

Chat Client ↔ Client

Il gioco consente ai Clients di scambiare messaggi mediante la Chat. Principalmente un messaggio Client può essere globale (tutti i Client che hanno effettuato il login hanno visione del messaggio) e di sessione (solo i client attualmente connessi alla sessione di gioco hanno visione del messaggio).

♦ Chat Globale



L'immagine di sopra rappresenta un esempio di comunicazione tra Clients mediante Chat Globale. I primi due si trovano in una sessione di gioco, mentre il terzo (Pluff) è in attesa e può comunicare con tutti gli altri Clients mediante la Chat Globale.

Come detto in precedenza, solo i Clients che sono almeno loggati al Server, possono utilizzare la Chat Globale:

```
Server: Benvenuto/a! Digita @cmd per conoscere i comandi.  
Me: ciao  
Server: Per utilizzare la chat devi prima effettuare il login!
```

♦ Chat di Sessione



L'immagine di sopra rappresenta un esempio di comunicazione tra Clients mediante Chat di Sessione. Per utilizzare quest'ultima è necessario utilizzare il prefisso % per ogni messaggio. I messaggi di sessione si differenziano da quelli globali essendo di colore magenta.

Comunicazione dettagliata

Analizziamo ora come avviene la comunicazione Client – Server nel dettaglio. Possiamo suddividere la comunicazione in tre fasi: connessione, registrazione/login e gioco.

◆ Connessione

Nella fase di connessione la comunicazione Client – Server non è particolarmente accentuata. Il Client, una volta connesso al Server riceverà un messaggio di benvenuto e potrà far uso dei comandi GUEST.

◆ Registrazione | Login

Il Client una volta connesso (e dunque in stato GUEST) può scegliere di effettuare la registrazione oppure il login:

Nel caso della registrazione il Server risponde chiedendo dapprima l'username e successivamente la password , il Server dunque verifica se le credenziali inserite sono valide (lunghezza, username disponibile etc.), in caso di successo notifica il Client dell'avvenuta registrazione.

Anche nel login, il Server risponde chiedendo dapprima l'username e successivamente la password e ne verifica l'attendibilità. In caso di successo, notifica il Client dell'avvenuto login e lo mette in coda per entrare in una sessione di gioco.

◆ Gioco

La fase di gioco presenta un ampio scambio di messaggi tra Client e Server. Al fine di mostrare tutte le interazioni ci poniamo in un tipico caso d'uso.

Supponiamo quindi di connetterci al Server, effettuare il login ed entrare in una sessione di gioco già avviata, con all'interno tre giocatori:



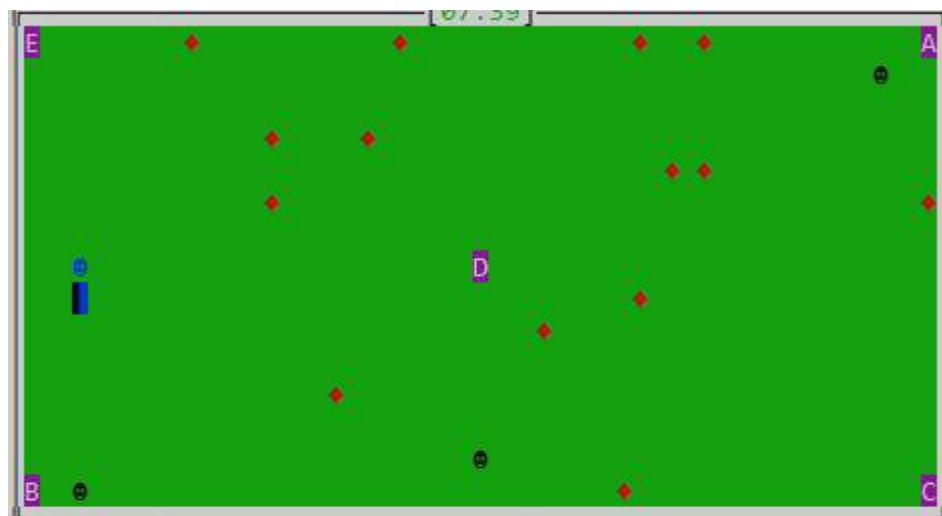
I comandi ricevuti dal Server, per il giocatore appena connesso (Pluff), sono (non necessariamente in ordine):

```
$In attesa che si liberi una sessione...
$reset
$connected Gennaro 0
$connected Gianluca 2
$connected Walter 0
$connected Pluff 0
$timer 312
$add packet 0 23
$add packet 5 45
$add player 13 0
$add player 13 28
$add player 7 26
$add you 1 3
$add location 0 0
$add location 0 56
$add location 7 27
$add location 14 0
$add location 14 56
$Che il gioco abbia inizio! Buon divertimento!
```

I comandi ricevuti dal Server, per gli altri giocatori, sono (non necessariamente in ordine):

```
$Tremate, Pluff si è unito alla sessione!
$connected Pluff 0
$add player 1 3
```

A questo è possibile cominciare a giocare, proviamo quindi ad effettuare uno spostamento verso Sud:



```
Server: Generazione di una nuova sessione in corso...
Server: In attesa che si liberi una sessione...
Server: Che il gioco abbia inizio! Buon divertimento!
Server: Tremate, Walter si è unito alla sessione!
Server: Tremate, Pluff si è unito alla sessione!
Me: @S
Server: Ouch! Hai sbattuto contro un muro!
```

Il Server risponderà con diversi comandi a seconda dell'effetto del movimento:

- ◆ **GRASS:** In questo caso l'effetto del movimento è positivo, viene dunque effettuato lo spostamento del giocatore. L'effetto sarà visibile a tutti i giocatori presenti nella sessione:

I comandi impartiti dal Server al giocatore che ha effettuato il movimento sono:

```
$update OLD_ROW OLD_COL  
$add you NEW_ROW OLD_COL
```

I comandi impartiti dal Server agli altri giocatori sono:

```
$update OLD_ROW OLD_COL  
$add player NEW_ROW OLD_COL
```

- ◆ **WALL:** In questo caso lo spostamento è nullo e il Server rivela la posizione del muro al solo Client che ha tentato lo spostamento:

```
$add wall ATTEMPT_ROW ATTEMPT_COL
```

- ◆ **LOCATION:** In questo caso lo spostamento è nullo e il Server invia al Client che ha tentato lo spostamento un semplice messaggio:

```
$Hei, hai un pacco per me?
```

- ◆ **PLAYER:** In questo caso lo spostamento è nullo e il Server invia al Client che ha tentato lo spostamento un semplice messaggio:

```
$Alt! Un giocatore di impedisce il passaggio.
```

- ◆ **FUORI MAPPA:** In questo caso lo spostamento è nullo e il Server invia al Client che ha tentato lo spostamento un semplice messaggio:

```
$Attento, stavi per cadere giù!
```

Quando si prova a raccogliere un pacchetto (con uno tra i comandi **@PS**, **@PN**, **@PE**, **@PO**), a seconda della presenza o meno di quest'ultimo i messaggi inviati dal Server al Client sono diversi:

- ◆ **Pacchetto assente:** In tal caso il Server invia al Client un semplice messaggio:

```
$Non c'è nulla qui!
```

- ◆ **Pacchetto presente:** In tal caso il comando ha effetto positivo. L'effetto sarà visibile a tutti i giocatori presenti nella sessione:

I comandi impartiti dal Server al giocatore che raccolto il pacchetto sono:

```
$update PACKET_ROW PACKET_COL  
$Trasporta il pacchetto fino alla locazione LOCATION_NAME.  
// Attiva il blink alla locazione in cui trasportare il pacchetto  
$add blink LOCATION_ROW LOCATION_COL LOCATION_NAME
```

I comandi impartiti dal Server agli altri giocatori sono:

```
$update PACKET_ROW PACKET_COL
```

Quando si prova a depositare un pacchetto (con uno tra i comandi **@DS**, **@DN**, **@DE**, **@DO**), i comandi impartiti dal Server variano a seconda del valore della matrice nel punto in cui si vuole depositare il pacchetto:

- ◆ **GRASS:** In questo caso l'azione di deposito è positiva. L'effetto sarà visibile a tutti i giocatori presenti nella sessione:

I comandi impartiti dal Server al giocatore che ha effettuato l'azione sono:

```
$remove blink  
$add packet ROW COL
```

I comandi impartiti dal Server agli altri giocatori sono:

```
$add packet ROW COL
```

- ◆ **WALL:** In questo caso l'azione ha effetto nullo. Il Server risponde al Client rivelando la posizione del muro.

```
$add wall ATTEMPT_ROW ATTEMPT_COL
```

- ◆ **LOCATION:** Se la locazione è errata il Server invia un messaggio al Client che indica l'errore commesso:

```
$Attento! Hai sbagliato locazione, riprova!
```

Nel caso in cui la locazione è esatta:

I comandi impartiti dal Server al giocatore che ha effettuato l'azione sono:

```
$remove blink  
$delivered NAME_PLAYER  
$PLAYER_NAME ha consegnato un pacco, complimenti!
```

I comandi impartiti dal Server agli altri giocatori sono:

```
$delivered NAME_PLAYER  
$PLAYER_NAME ha consegnato un pacco, complimenti!
```

- ◆ **PLAYER:** In questo caso l'azione è nulla e il Server invia al Client che ha tentato il deposito un semplice messaggio:

```
$Non puoi lanciare pacchetti contro gli altri giocatori!
```

- ◆ **FUORI MAPPA:** In questo caso l'azione è nulla e il Server invia al Client che ha tentato il deposito un semplice messaggio:

```
$Non puoi posizionare il pacco qui!
```

Provare a depositare un pacchetto senza averne raccolto prima uno, causerà l'invio da parte del Server di un messaggio di errore:

```
$Devi prima raccogliere un pacco!
```

Quando un giocatore si disconnette il Server invia tale informazione (mendiate il comando **disconnected**) a tutti i giocatori della sessione:

```
$disconnected PLAYER_NAME
```

Infine, il gioco termina quando il tempo esaurisce oppure quando tutti i pacchi vengono consegnati. A quel punto il Server decreta il vincitore della sessione di gioco.

Dettagli Implementativi

- ◆ ConnectionRequestManagement
- ◆ timeProvider
- ◆ disconnectionManagement e matchmaker
- ◆ sessionManagement
- ◆ signalHandler
- ◆ La gestione del cursore
- ◆ Lo scambio dei messaggi

◆ La funzione ConnectionRequestManagement [Server]

La funzione ConnectionRequestManagement, eseguita da un thread, gestisce le connessioni in entrata sulla porta nella quale il Server è in ascolto. Il compito principale di questa funzione è quello di allocare un nuovo nodo ClientInfo (un nodo ClientInfo è una struttura contenente diverse informazioni riguardanti il client) e creare un thread per l'ascolto dei messaggi in arrivo il quale esegue la funzione **listenerClient**.

```
/*
 * @attribute tidHandler: TID del thread principale che gestisce un determinato client.
 * @attribute clientSocket: Socket descriptor della socket associata ad un determinato client.
 * @attribute clientAddressIPv4: IPv4 del client.
 * @attribute username: Username del client.
 * @attribute timestamp: Ora della connessione
 * @attribute stato: LOGGED, REGISTERED, GUEST.
 * @attribute prevClientInfo: Puntatore al nodo clientInfo precedente.
 * @attribute nextClientInfo: Puntatore al nodo clientInfo successiva.
 * @attribute mutexSocket: Mutex associato alla socket.
 */

struct ClientInfo{
    pthread_t tidHandler;
    int clientSocket;
    char clientAddressIPv4[INET_ADDRSTRLEN];
    char username[CLTINF_USERNAME_STRLEN];
    time_t timestamp;
    int packetsDelivered;
    int status;
    int currRows;
    int currCols;
    int havePacket;
    pthread_mutex_t mutexSocket;
    struct ClientInfo* prevClientInfo;
    struct ClientInfo* nextClientInfo;
};
typedef struct ClientInfo ClientInfo;
typedef ClientInfo* LpClientInfo;
```

Essenzialmente il corpo della funzione è un ciclo infinito, il quale rimane in attesa di richieste di connessione al Server. Quando un Client si connette al Server, la funzione prova ad allocare, come detto in precedenza, un nodo ClientInfo, lo inserisce nella lista dei Clients connessi al Server e inizializza il thread di gestione associato al Client.

```
pthread_mutex_lock(&mutexClientInfo);
insertClientInfo(&listClientInfo, clientInfo);
pthread_mutex_unlock(&mutexClientInfo);
/* Crea un thread che gestirà l'accesso del client al Server */
pthread_create(&tidListenerClient, NULL, listenerClient, clientInfo);
clientInfo->tidHandler = tidListenerClient;
sprintf(msg, "Nuova connessione accettata: [Client: %s] - %s", clientAddressIPv4, ctime(&(clientInfo->timestamp)));
```

Oltre a fare ciò, ogni qualvolta un Client effettua con successo la connessione al Server, la funzione ConnectionRequestManagement aggiorna i vari status del Server (connessione totali e client attualmente connessi).

◆ La funzione timeProvider [Server]

La funzione timeProvider (eseguita da un thread apposito), gestisce il timer della sessione di gioco. Il tempo è gestito mediante la chiamata alarm. Al suo avvio la funzione imposta la sveglia del processo a TIME (costante del Server). Il corpo principale della funzione è costituito da un while la cui condizione verifica se il tempo è scaduto o se sono stati raccolti tutti i pacchetti.

Ad ogni iterazione, viene effettuata una chiamata ad alarm con parametro 0, in modo da ricavare i secondi rimanenti. Viene poi richiamata la funzione alarm con i secondi ricavati dalla precedente chiamata, ed infine viene inviato il comando \$time SECONDS a tutti i Clients connessi alla sessione.

```
while((totalSeconds = alarm(0)) >= 0 && sessionNumOfPackets > 0 && !finished){

    if(totalSeconds == 0){
        finished = true;
    }

    pthread_mutex_lock(&mutexTimer);
    alarm(totalSeconds);
    sprintf(msg, "$time %d", totalSeconds);
    sendMsgToSession(NULL, msg);
    pthread_mutex_unlock(&mutexTimer);
```

Come si evince dal frammento di codice il timer dispone di un mutex personale, questo perché il comando @time impartito dai Clients loggati al Server fa anch'esso uso della sveglia condivisa.

◆ Le funzioni disconnectionManagement e matchmaker [Server]

La funzione disconnectionManagement gestisce la disconnessione di un Client dal Server. In particolare si occupa della deallocazione del nodo ClientInfo, della chiusura della socket, dell'aggiornamento della matrice, dei player connessi alla sessione di gioco (nel caso in cui il Client disconnesso stesse giocando), del posizionamento del pacchetto detenuto dal client sconnesso e dell'attivazione del matchmaker di sessione mediante la variabile di condizione condMatchmaker.

```
if(clientInfo->status == CLTINF_PLAYING){
    if(clientInfo->havePacket != -1){
        sprintf(msg, "$add packet %d %d", clientInfo->currRows, clientInfo->currCols);
        sendMsgToSession(NULL, msg);
        sessionNumOfPackets += 1;
        session->field[clientInfo->currRows][clientInfo->currCols] = SESSION_PACKET_VALUE;
        session->packets[clientInfo->havePacket].currRows = clientInfo->currRows;
        session->packets[clientInfo->havePacket].currCols = clientInfo->currCols;
        pthread_mutex_lock(&mutexCursor);
        moveto(field[clientInfo->currRows][clientInfo->currCols].posX, field[clientInfo->currRows][clientInfo->currCols].posY);
        setColor(GRAPHICS_FG_COLOR_RED, GRAPHICS_BG_COLOR_GREEN);
        printf("%s", SYMBOL_PACKET);
        reset();
        pthread_mutex_unlock(&mutexCursor);
    } else{
        session->field[clientInfo->currRows][clientInfo->currCols] = SESSION_GRASS_VALUE;
        sprintf(msg, "$update %d %d", clientInfo->currRows, clientInfo->currCols);
        sendMsgToSession(NULL, msg);
        pthread_mutex_lock(&mutexCursor);
        moveto(field[clientInfo->currRows][clientInfo->currCols].posX, field[clientInfo->currRows][clientInfo->currCols].posY);
        setColor(GRAPHICS_FG_COLOR_GREEN, GRAPHICS_BG_COLOR_GREEN);
        printf("%s", SYMBOL_GRASS);
        reset();
        pthread_mutex_unlock(&mutexCursor);
    }
}
```

```

        deleteClientFromSession(session, clientInfo);
        sprintf(msg, "$disconnected %s", clientInfo->username);
        dynamicBuffer = (char*)calloc(GRAPHICS_CHAT_WIDTH, sizeof(char));
        strcpy(dynamicBuffer, msg);
        pthread_create(&tidUpdatePlayersInfo, NULL, updatePlayersInfo, dynamicBuffer);
        sendMsgToSession(NULL, msg);
        pthread_cond_signal(&condMatchmaker);
    }
    pthread_mutex_unlock(&(session->mutexSession));
}

```

In pratica, quando il Server genera una nuova sessione, avvia il thread che si occupa del matchmaking (il thread in questione esegue la funzione matchmaker). Questa funzione cicla fino a quando la sessione non si riempie. Una volta riempita, la variabile di condizione fa sì che il thread entri in stato di wait.

La funzione disconnectionManagement, al verificarsi della disconnessione di un Client in partita (in sessione di gioco), risveglia il matchmaker (mediante la funzione pthread_cond_signal) il quale prende il primo Client in coda di attesa, per poi rimettersi in wait.

```

while(true){

    pthread_mutex_lock(&session->mutexSession);
    while(session->numOfPlayers > SESSION_PLAYERS_STRLEN-1 && session->joinable != false){
        pthread_cond_wait(&condMatchmaker, &(session->mutexSession));
    }
}

```

La funzione matchmaker, oltre a gestire la coda di attesa, invia i comandi di “inizializzazione” al Client che si appena unito alla sessione (reset, connected).

```

if(session->joinable){
    pthread_mutex_lock(&mutexClientInfo);
    if(headClientInfoToJoinQueue != NULL){
        clientToJoin = dequeueClientInfoToJoin(&headClientInfoToJoinQueue, &tailClientInfoToJoinQueue);
        waitingClients -= 1;
        pthread_mutex_lock(&mutexCursor);
        moveTo(status[3].posX, status[3].posY);
        printf(" \u25cf Waiting: %8.2d", waitingClients);
        pthread_mutex_unlock(&mutexCursor);
        clientToJoin->status = CLTINF_PLAYING;
        clientToJoin->packetsDelivered = 0;
        sendMsg(clientToJoin, "$remove blink");
        sendMsg(clientToJoin, "$reset");
        sleep(2);
        sendMsg(clientToJoin, "$Server: Che il gioco abbia inizio! Buon divertimento!");
        insertClientInSession(session, clientToJoin);
        joined = true;
        pthread_cond_signal(&condSession);
    }
    pthread_mutex_unlock(&mutexClientInfo);
}

```

◆ La funzione sessionManagement [Server]

La funzione sessionManagement si occupa della gestione della sessione creata ed inizializzata da initSession. Una sessione è costituita da una struttura contenente i seguenti attributi:

```
struct Session{
    LpClientInfo clients[SESSION_PLAYERS_STRLEN];
    int field[GRAPHICS_FIELD_HEIGHT][GRAPHICS_FIELD_WIDTH];
    Packet packets[SESSION_PACKETS_STRLEN];
    Location locations[SESSION_LOCATIONS_STRLEN];
    Spawn spawn[SESSION_SPAWN_STRLEN];
    bool joinable;
    int numOfPlayers;
    int numOfPackets;
    pthread_mutex_t mutexSession;
};

typedef struct Session Session;
typedef Session* lpSession;
```

Inizialmente la funzione sessionManagement rende la sessione accessibile, impostando il suo attributo joinable a true, quindi, crea il thread matchmaker e attende di ricevere un signal mettendosi in attesa sulla variabile numOfPlayers (in pratica sta aspettando che almeno un Client entri nella sessione prima di far partire il gioco).

```
session->joinable = true;
pthread_create(&tidMatchmaker, NULL, matchmaker, NULL);

pthread_mutex_lock(&(session->mutexSession));
while(session->numOfPlayers < 1){
    pthread_cond_wait(&condSession, &(session->mutexSession));
}
pthread_mutex_unlock(&(session->mutexSession));
```

Non appena un Client accede alla sessione di gioco, la funzione crea un thread per l'esecuzione della funzione timeProvider (la quale come visto precedentemente gestisce il timer del gioco) e ne fa la join. Quindi sblocca il thread matchmaker, ne effettua la join che rientra immediatamente, ripulisce le strutture e riaccoda tutti i Clients presenti nella sessione appena terminata.

```
pthread_create(&tidTimeProvider, NULL, timeProvider, NULL);
pthread_join(tidTimeProvider, NULL);

session->joinable = false;

pthread_cond_signal(&condMatchmaker);
pthread_join(tidMatchmaker, NULL);

pthread_mutex_lock(&mutexClientInfo);
for(int i=0; i<SESSION_PLAYERS_STRLEN; i++){
    if(session->clients[i] != NULL){
        if(session->clients[i]->packetsDelivered > maxPackets){
            maxPackets = session->clients[i]->packetsDelivered;
            occurrence = 1;
            client = i;
        } else if(session->clients[i]->packetsDelivered == maxPackets){
            occurrence += 1;
        }
    }
}
pthread_mutex_unlock(&mutexClientInfo);
```

◆ La funzione signalHandler [Server]

La funzione signalHandler gestisce i segnali SIGINT e SIGSTOP. In entrambi i casi stampa il messaggio di disconnessione, chiude la socket dove si trova in ascolto, chiude i file logs, database e fieldGenerator, chiude tutte le socket Clients e fa terminare il programma.

```
void signalHandler(int signal){

    char msg[GRAPHICS_CHAT_WIDTH];
    LpMsg msgChat;
    pthread_t tidUpdateChat;
    LpClientInfo tmp;

    switch(signal){
        case SIGINT:
            sprintf(msg, "Disconnessione del server in corso!");
            if((msgChat = allocMsg(msg, GRAPHICS_FG_COLOR_RED, true))!=NULL){
                pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
            }
            close(listenerSocket);
            close(logs);
            close(database);
            close(fieldGenerator);
            tmp = listClientInfo;
            while(tmp != NULL){
                close(tmp->clientSocket);
                tmp = tmp->nextClientInfo;
            }
            sleep(2);
            clean();
            raise(SIGTERM);
            break;
    }
}
```

◆ La gestione del cursore [Client] [Server]

L'aggiornamento della grafica non avviene mediante una clean del terminale, ma mediante uno spostamento del cursore e una stampa. Ovviamente, il movimento del cursore è protetto da mutex opportuni.

Nel Client il focus principale del cursore è la textfield. Quando l'utente digita dell'input nella textfield il programma tiene conto della posizione dello stesso in una variabile globale (tale variabile viene modificata solo quando il cursore opera sulla textfield). Ogni qualvolta è richiesto un aggiornamento, viene effettuato lo spostamento del cursore, la stampa e il riposizionamento del cursore nell'ultima posizione abbandonata dal cursore sulla textfield, prima di essere spostato.

In alcuni casi il solo mutex del cursore non basta., ad esempio anche la chat dispone del proprio mutex, così come l'info box dei player connessi alla sessione di gioco.

```
pthread_mutex_lock(&mutexChat);
/* Controlla se la chat è piena */
if(chat.currMsgPosition == GRAPHICS_CHAT_HEIGHT-1 && chat.msgBox[chat.currMsgPosition].isEmpty == false){
    /* Effettua lo shift della chat */
    for(int currMsg = 0; currMsg < GRAPHICS_CHAT_HEIGHT-1; currMsg++){
        strcpy(chat.msgBox[currMsg].msg, chat.msgBox[currMsg+1].msg);
        chat.msgBox[currMsg].color = chat.msgBox[currMsg+1].color;
    }
    /* Copia il nuovo messaggio nella prima posizione a partire dal basso */
    strcpy(chat.msgBox[GRAPHICS_CHAT_HEIGHT-1].msg, msg);
    chat.msgBox[GRAPHICS_CHAT_HEIGHT-1].color = color;

    /* Ristampa l'intera chat shiftata */
    for(int currMsg = 0; currMsg < GRAPHICS_CHAT_HEIGHT; currMsg++){
        /* Pattern creato per gestire la stampa e il movimento del cursore dove stampare. */
        pthread_mutex_lock(&mutexCursor);

```

◆ Lo scambio dei messaggi tra Client - Server

Lo scambio dei messaggi tra Client e Server segue determinate regole. I messaggi in uscita dal Client hanno sempre lunghezza pari a 70, mentre quelli in entrata 82, chiaramente vale il viceversa per il Server. Ciò è dovuto non solo a motivi grafici (grandezza della chat box) ma anche a una standardizzazione del formato dei messaggi.

Quando un Client invia un messaggio al Server, la funzione che si occupa di tale operazione (`sendMsg`) inizializza un buffer di lunghezza 70 con '\0', e all'interno copia il messaggio che il Client vuole inoltrare al Server. Non c'è pericolo di uscire fuori dal buffer dal momento che la `textfield` impedisce la scrittura di più di 70 caratteri. La stessa operazione è effettuata dal Server, dalla medesima funzione.

```
/*
 * @param msg: messaggio da inoltrare al server.
 *
 * @return: void.
 *
 * La funzione sendMsg inoltra il messaggio catturato
 * dal listenerTextField al server.
 *
 */
void sendMsg(char* msg){

    pthread_t tidUpdateChat;
    char buffer[GRAPHICS_TEXTFIELD_STRLEN];
    memset(buffer, '\0', GRAPHICS_TEXTFIELD_STRLEN);
    strcpy(buffer, msg);

    write(clientSocket, buffer, GRAPHICS_TEXTFIELD_STRLEN);

}
```

Si osservi che non è necessario verificare se la `write` ha riscontrato un errore (SIGPIPE) dal momento che i thread `listenerClient` (nel Server) e `listenerServer` (nel Client) sono in continua attesa di messaggi. L'eventuale chiusura della connessione, o di errori della stessa, è dunque catturata da questi due thread.

Codice Sorgente

- ◆ Codice Client
- ◆ Codice Server
- ◆ Libreria Graphics
- ◆ Libreria Session
- ◆ Libreria Cltinf

```

1  //*****
2  **** INIT CLIENT
3  *****/
4
5  //
6  _____ LIBRARIES _____
7
8
9  #include "../libs/graphics/graphics.h"
10 #include <sys/socket.h>
11 #include <sys/types.h>
12 #include <netinet/in.h>
13 #include <errno.h>
14 #include <arpa/inet.h>
15 #include <pthread.h>
16 #include <stdbool.h>
17 #include <string.h>
18 #include <stdio.h>
19 #include <signal.h>
20 #include <unistd.h>
21 #include <time.h>
22 #include <fcntl.h>
23 #include <ctype.h>
24 #include <stdlib.h>
25
26 //____ COSTANTS _____
27
28
29 #define SERVER_NOTIFY          '$'
30 #define SESSION_NOTIFY         '%'
31 #define SYMBOL_GRASS           " "
32 #define SYMBOL_PLAYER          "\u263b"
33 #define SYMBOL_PACKET          "\u2666"
34 #define SYMBOL_WALL             "\u2590"
35
36 //____ GLOBAL
37 _____ VARIABLES _____
38
39
40
41
42
43
44
45
46
47
48
49
50
51

```

```

52     int colToBlink;
53     char* locationToBlink;
54
55     bool connected = false;
56
57 //____FUNCTIONS
58 #DECLARATION_____
59
60     int initGUI(char*, char*);
61     int mapping(char* );
62     void goToTheTextField(void);
63     void cleanTextField(void);
64     void* listenerTextField(void* );
65     bool isAlphabet(char);
66     int connection(void);
67     void* updateChat(void* );
68     int trimMsg(char[]);
69     LpMsg allocMsg(char*, int, bool);
70     void sendMsg(char* );
71     void* listenerServer(void* );
72     int takeAction(char[]);
73     void* updateTimer(void* );
74     void* updatePlayersInfo(void* );
75     void* updateField(void* );
76     void signalHandler(int);
77
78
79     int main(void){
80
81         pthread_t tidListenerTextField;
82         pthread_t tidListenerServer;
83
84         srand(time(NULL));
85         signal(SIGPIPE, SIG_IGN);
86
87         clientSocket = socket(PF_INET, SOCK_STREAM, 0);
88         if(connection() == 1){
89             return 1;
90         }
91
92         signal(SIGSTOP, signalHandler);
93         signal(SIGINT, signalHandler);
94
95         if(initGUI("../gui/blueprint.txt", "../gui/gui.txt")){
96             beep();
97             setColor(GRAPHICS_FG_COLOR_RED, GRAPHICS_BG_COLOR_WHITE);
98             printf("\n <!> Errore durante l'inizializzazione dell'interfaccia grafica.\n\n");
99             reset();
100            return 1;
101        }
102
103        pthread_create(&tidListenerTextField, NULL, listenerTextField, NULL);
104        pthread_create(&tidListenerServer, NULL, listenerServer, NULL);
105        pthread_join(tidListenerServer, NULL);
106        pthread_cancel(tidListenerTextField);
107
108        sleep(2);

```

```

109     setCursor(true);
110     clean();
111
112     close(clientSocket);
113
114     return 0;
115
116 }
117
118 //____FUNCTION FOR MANAGEMENT
* GRAPHICS_____
119
120 /**
121 * @param blueprint: path del file blueprint
122 * che inizializza la matrice per la grafica finale.
123 * @param gui: path del file della grafica finale
124 * da andare a stampare.
125 *
126 * @return: 1 in caso di errore 0 altrimenti.
127 *
128 * La funzione initGUI apre in sola lettura il file che
129 * contiene le grafica che poi stamperà sul rispettivo terminale.
130 *
131 */
132 int initGUI(char* blueprint, char* gui){
133
134     char basicComponent[1];
135     int bytesReaded;
136     int guiFile;
137
138     setCursor(true);
139
140     if(mapping(blueprint)){
141         return 1;
142     }
143
144     if((guiFile = open(gui, O_RDONLY)) == -1){
145         return 1;
146     }
147
148     clean();
149     while((bytesReaded = read(guiFile, basicComponent, 1)) > 0){
150         basicComponent[bytesReaded] = '\0';
151         if(basicComponent[0] == 'x'){
152             setColor(GRAPHICS_FG_COLOR_GREEN, GRAPHICS_BG_COLOR_GREEN);
153             printf(" ");
154             reset();
155         } else{
156             printf("%s", basicComponent);
157         }
158     }
159
160     cursor posX = textField posX;
161     cursor posY = textField posY;
162     goToTheTextField();
163
164 }
165
---
```

```

166
167 /**
168 * @param blueprint: path del file che inizializza
169 * La matrice per poter stampare poi sopra la grafica.
170 *
171 * @return: return 1 in caso di errore e 0 altrimenti.
172 *
173 * La funzione mapping apre in sola lettura lettura
174 * il file che inizializza la matrice per la grafica.
175 *
176 */
177 int mapping(char* blueprint){
178
179     char basicComponent[1];
180     int bytesReaded;
181     int blueprintFile;
182     int currRowsScreen = 1;
183     int currColsScreen = 1;
184     int currRowsField = 0;
185     int currColsField = 0;
186     int currPlayerPosition = 0;
187     int currMsgPosition = 0;
188
189     if((blueprintFile = open(blueprint, O_RDONLY)) == -1){
190         return 1;
191     }
192
193     while((bytesReaded = read(blueprintFile, basicComponent, 1)) > 0){
194
195         basicComponent[bytesReaded] = '\0';
196
197         switch (basicComponent[0]) {
198             case GRAPHICS_PIN_CHAT:
199                 chat.msgBox[currMsgPosition].coords.posX = currColsScreen;
200                 chat.msgBox[currMsgPosition].coords.posY = currRowsScreen;
201                 currMsgPosition += 1;
202                 break;
203             case GRAPHICS_PIN_FIELD:
204                 field[currRowsField][currColsField].posX = currColsScreen;
205                 field[currRowsField][currColsField].posY = currRowsScreen;
206                 currColsField += 1;
207                 if(currColsField == GRAPHICS_FIELD_WIDTH){
208                     currRowsField += 1;
209                     currColsField = 0;
210                 }
211                 break;
212             case GRAPHICS_PIN_TIMER:
213                 timer.coords posX = currColsScreen;
214                 timer.coords posY = currRowsScreen;
215                 break;
216             case GRAPHICS_PIN_COUNTER:
217                 playersCounter.coords posX = currColsScreen;
218                 playersCounter.coords posY = currRowsScreen;
219                 break;
220             case GRAPHICS_PIN_PLAYERS:
221                 playersInfo.playerBox[currPlayerPosition].coords posX = currColsScreen;
222                 playersInfo.playerBox[currPlayerPosition].coords posY = currRowsScreen;
223                 currPlayerPosition += 1;
224             ...

```

```

224         break;
225     case GRAPHICS_PIN_TEXTFIELD:
226         textField posX = currColsScreen;
227         textField posY = currRowsScreen;
228         break;
229     case GRAPHICS_PIN_START:
230         clean();
231         break;
232     }
233
234     currColsScreen += 1;
235
236     if(basicComponent[0] == '\n'){
237         currColsScreen = 1;
238         currRowsScreen += 1;
239     }
240
241 }
242
243     if(bytesReaded == -1){
244         return 1;
245     }
246
247     chat.currMsgPosition = 0;
248     playersInfo.currPlayerPosition = -1;
249     playersCounter.value = 0;
250
251     for(int msg = 0; msg < GRAPHICS_CHAT_HEIGHT; msg++){
252         chat.msgBox[msg].isEmpty = true;
253     }
254
255     close(blueprintFile);
256     return 0;
257
258 }
259
260
261 /**
262 * @param arg: msgChat -> messaggio da inserire in chat .
263 *
264 * @return: puntatore a void.
265 *
266 * La funzione updateChat aggiorna la chat stampando
267 * eventuali nuovi messaggi. Nel caso in cui la chat fosse
268 * piena viene effettuato lo shit della stessa dal basso
269 * verso l'alto.
270 *
271 */
272 void* updateChat(void* arg){
273
274     LpMsg msgChat = (LpMsg)arg;
275     char msg[GRAPHICS_CHAT_WIDTH];
276     strcpy(msg, msgChat->msg);
277     int color = msgChat->color;
278
279     pthread_mutex_lock(&mutexChat);
280     /* Controllo se la chat è piena */
281     if(chat.currMsgPosition == GRAPHICS_CHAT_HEIGHT-1 &&
282         chat.msgBox[chat.currMsgPosition].isEmpty == false)

```

```

chat.msgBox[chat.currMsgPosition].isEmpty == false){
    /* Effettuo lo shift della chat */
    for(int currMsg = 0; currMsg < GRAPHICS_CHAT_HEIGHT-1; currMsg++){
        strcpy(chat.msgBox[currMsg].msg, chat.msgBox[currMsg+1].msg);
        chat.msgBox[currMsg].color = chat.msgBox[currMsg+1].color;
    }
    /* Copio il nuovo messaggio nella prima posizione a partire dal basso */
    strcpy(chat.msgBox[GRAPHICS_CHAT_HEIGHT-1].msg, msg);
    chat.msgBox[GRAPHICS_CHAT_HEIGHT-1].color = color;

    /* Ristampo l'intera chat shiftata */
    for(int currMsg = 0; currMsg < GRAPHICS_CHAT_HEIGHT; currMsg++){
        /* Pattern creato per gestire la stampa e il movimento del cursore dove
        stampare. */
        pthread_mutex_lock(&mutexCursor);
        setCursor(false);
        moveto(chat.msgBox[currMsg].coords.posX, chat.msgBox[currMsg].coords.posY);
        printf("      |      ");
        fflush(stdout);
        moveto(chat.msgBox[currMsg].coords.posX, chat.msgBox[currMsg].coords.posY);
        setColor(chat.msgBox[currMsg].color, 0);
        printf("%s", chat.msgBox[currMsg].msg);
        fflush(stdout);
        reset();
        moveto(cursor.posX, cursor.posY);
        setCursor(true);
        pthread_mutex_unlock(&mutexCursor);
    }
}

} else{
    /* Imposta il msgBox relativo al messaggio a pieno */
    chat.msgBox[chat.currMsgPosition].isEmpty = false;
    chat.msgBox[chat.currMsgPosition].color = color;
    strcpy(chat.msgBox[chat.currMsgPosition].msg, msg);

    /* Pattern creato per gestire la stampa e il movimento del cursore dove stampare.
    */
    pthread_mutex_lock(&mutexCursor);
    setCursor(false);
    moveto(chat.msgBox[chat.currMsgPosition].coords.posX,
    chat.msgBox[chat.currMsgPosition].coords.posY);
    printf("      |      ");
    fflush(stdout);
    moveto(chat.msgBox[chat.currMsgPosition].coords.posX,
    chat.msgBox[chat.currMsgPosition].coords.posY);
    setColor(color, 0);
    printf("%s", msg);
    fflush(stdout);
    reset();
    moveto(cursor.posX, cursor.posY);
    setCursor(true);
    pthread_mutex_unlock(&mutexCursor);

    /* Incremento l'indice della chat */
    if(chat.currMsgPosition != GRAPHICS_CHAT_HEIGHT-1){
        chat.currMsgPosition += 1;
    }
}

```

```

334     }
335 }
336
337 pthread_mutex_unlock(&mutexChat);
338
339 if(msgChat->error == true){
340     beep();
341 }
342
343 free(msgChat);
344
345 }
346
347
348 /**
349 * @param msg: arg -> Secondi totali.
350 *
351 * @return: puntatore a void.
352 *
353 * La funzione updateTimer aggiorna il timer di fine
354 * partita.
355 *
356 */
357 void* updateTimer(void* arg){
358
359     int totalSeconds = *((int*)arg);
360
361     int minutes = totalSeconds / 60;      /* Calcolo i minuti */
362     int seconds = totalSeconds % 60;      /* Calcolo i secondi */
363
364     pthread_mutex_lock(&mutexCursor);
365     setCursor(false);
366     moveTo(timer.posX, timer.posY);
367
368     /* Effettuo l'aggiornamento del timer */
369     if(totalSeconds >= 300 && totalSeconds <= 600){
370         setColor(GRAPHICS_FG_COLOR_GREEN, 0);
371         printf("%.2d:%.2d", minutes, seconds);
372         reset();
373     } else if(totalSeconds >= 60 && totalSeconds <= 300){
374         setColor(GRAPHICS_FG_COLOR_YELLOW, 0);
375         printf("%.2d:%.2d", minutes, seconds);
376         reset();
377     } else{
378         if(totalSeconds <= 10 && totalSeconds%2 == 0){
379             beep();
380             setColor(GRAPHICS_FG_COLOR_RED, 0);
381             printf("%.2d:%.2d", minutes, seconds);
382             reset();
383         } else if(totalSeconds <= 10 && totalSeconds%2 != 0){
384             beep();
385             setColor(GRAPHICS_FG_COLOR_YELLOW, 0);
386             printf("%.2d:%.2d", minutes, seconds);
387             reset();
388         } else{
389             setColor(GRAPHICS_FG_COLOR_RED, 0);
390             printf("%.2d:%.2d", minutes, seconds);
391             reset();

```

```

392     }
393 }
394
395     fflush(stdout);
396     moveto(cursor.posX, cursor.posY);
397     setCursor(true);
398     pthread_mutex_unlock(&mutexCursor);
399     free(arg);
400
401 }
402
403
404 /**
405 * @param msg: arg -> Comando [$(dis)connected Username (packagesDelivered)].
406 *
407 * @return: puntatore a void.
408 *
409 * La funzione updatePlayersInfo aggiorna il counter
410 * e la lista dei players attualmente connessi alla
411 * sessione corrente.
412 *
413 */
414 void* updatePlayersInfo(void* arg){
415
416     char* msg = (char*)arg;                                /* Messaggio in entrata
417     */
418     char* saveptr;
419     char* cmd = strtok_r(msg, " ", &saveptr);           /* Comando effettivo
420     */
421     char* username;                                     /* Username dell'utente (dis)connesso
422     */
423     int packetsDelivered;                             /* Numero di pacchetti consegnati
424     */
425     char buffer[GRAPHICS_TEXTFIELD_STRLEN];
426     int focusedPlayer = 0;                            /* Posizione del player puntato
427     */
428
429     pthread_mutex_lock(&mutexUpdate);
430     /* Se il comando è di tipo disconnectione allora */
431     if(!strcmp(cmd, "$disconnected") && playersCounter.value > 0){
432         /* Estraggo l'username */
433         username = strtok_r(NULL, " ", &saveptr);
434         /* Aggiorno il counter */
435         playersCounter.value -= 1;
436         pthread_mutex_lock(&mutexCursor);
437         setCursor(false);
438         moveto(playersCounter.coords.posX, playersCounter.coords.posY);
439         printf("%d", playersCounter.value);
440         fflush(stdout);
441         moveto(cursor.posX, cursor.posY);
442         setCursor(true);
443         pthread_mutex_unlock(&mutexCursor);
444
445         /* Aggiorno la lista dei players */
446         /* Estraggo l'indice del player disconnesso */
447         while(strcmp(username, playersInfo.playerBox[focusedPlayer].username) != 0){
448             focusedPlayer += 1;
449         }

```

```

445
446     /* Scorro verso l'alto a partire dall'utente eliminato tutti gli utenti presenti in
447     * sessione(a livello grafico), */
448     /* mentre nell'array viene fatto un semplice shift a sinistra di tutti gli utenti
449     * che si trovano dopo */
450     /* L'utente
451     * eliminato
452     */
453
454     for(int player = focusedPlayer; player < playersInfo.currPlayerPosition; player++){
455         strcpy(playersInfo.playerBox[player].username,
456             playersInfo.playerBox[player+1].username);
457         playersInfo.playerBox[player].packetsDelivered =
458             playersInfo.playerBox[player+1].packetsDelivered;
459         pthread_mutex_lock(&mutexCursor);
460         setCursor(false);
461         moveto(playersInfo.playerBox[player].coords.posX,
462             playersInfo.playerBox[player].coords.posY);
463         printf("           |   |");
464         fflush(stdout);
465         memset(buffer, '\0', GRAPHICS_TEXTFIELD_STRLEN);
466         moveto(playersInfo.playerBox[player].coords.posX,
467             playersInfo.playerBox[player].coords.posY);
468         sprintf(buffer, " \u25ba %-13s[%d] ", playersInfo.playerBox[player].username,
469             playersInfo.playerBox[player].packetsDelivered);
470         printf("%s", buffer);
471         fflush(stdout);
472         moveto(cursor.posX, cursor.posY);
473         setCursor(true);
474         pthread_mutex_unlock(&mutexCursor);
475     }
476
477     pthread_mutex_lock(&mutexCursor);
478     setCursor(false);
479     moveto(playersInfo.playerBox[playersInfo.currPlayerPosition].coords.posX,
480             playersInfo.playerBox[playersInfo.currPlayerPosition].coords.posY);
481     printf("           |   |");
482     fflush(stdout);
483     playersInfo.currPlayerPosition -= 1;
484     moveto(cursor.posX, cursor.posY);
485     setCursor(true);
486     pthread_mutex_unlock(&mutexCursor);
487
488     /* Se il comando è di tipo connessione allora */
489 } else if(!strcmp(cmd, "$connected")){
490     /* Estraggo l'username */
491     username = strtok_r(NULL, " ", &saveptr);
492     packetsDelivered = atoi(strtok_r(NULL, " ", &saveptr));
493     memset(buffer, '\0', GRAPHICS_TEXTFIELD_STRLEN);
494     /* Unisco nel buffer info la stringa da stampare nella lista degli utenti in
495     * sessione */
496     sprintf(buffer, " \u25ba %-13s[%d] ", username, packetsDelivered);
497     /* Aggiorno il counter */
498     playersCounter.value += 1;
499     pthread_mutex_lock(&mutexCursor);
500     setCursor(false);
501     moveto(playersCounter.coords.posX, playersCounter.coords.posY);
502     printf("%d", playersCounter.value);
503     fflush(stdout);

```

```

492     moveto(cursor.posX, cursor.posY);
493     setCursor(true);
494     pthread_mutex_unlock(&mutexCursor);
495     playersInfo.currPlayerPosition += 1;
496     strcpy(playersInfo.playerBox[playersInfo.currPlayerPosition].username, username);
497     playersInfo.playerBox[playersInfo.currPlayerPosition].packetsDelivered =
498         packetsDelivered;
499     pthread_mutex_lock(&mutexCursor);
500     setCursor(false);
501     moveto(playersInfo.playerBox[playersInfo.currPlayerPosition].coords.posX,
502         * playersInfo.playerBox[playersInfo.currPlayerPosition].coords.posY);
503     printf("%s", buffer);
504     fflush(stdout);
505     moveto(cursor.posX, cursor.posY);
506     setCursor(true);
507     pthread_mutex_unlock(&mutexCursor);

508     /* Se il comando è di tipo Aggiornamento pacchetti allora */
509 } else if(!strcmp(cmd, "$delivered")){
510     /* Estraggo l'username */
511     username = strtok_r(NULL, " ", &saveptr);

512     /* Aggiorno la lista dei players */
513     /* Estraggo l'indice del player che ha consegnato il pacchetto */
514     while(strcmp(username, playersInfo.playerBox[focusedPlayer].username) != 0){
515         focusedPlayer += 1;
516     }

517     /* Aggiorno il numero di pacchetti consegnati */
518     playersInfo.playerBox[focusedPlayer].packetsDelivered += 1;
519     memset(buffer, '\0', GRAPHICS_TEXTFIELD_STRLEN);
520     /* Unisco nel buffer info la stringa da stampare nella lista degli utenti in
521      * sessione */
522     sprintf(buffer, "\u25ba %-13s[%d] ", username,
523             playersInfo.playerBox[focusedPlayer].packetsDelivered);
524     pthread_mutex_lock(&mutexCursor);
525     setCursor(false);
526     moveto(playersInfo.playerBox[focusedPlayer].coords.posX,
527         * playersInfo.playerBox[focusedPlayer].coords.posY);
528     printf(" | |");
529     fflush(stdout);
530     moveto(playersInfo.playerBox[focusedPlayer].coords.posX,
531         * playersInfo.playerBox[focusedPlayer].coords.posY);
532     printf("%s", buffer);
533     fflush(stdout);
534     moveto(cursor.posX, cursor.posY);
535     setCursor(true);
536     pthread_mutex_unlock(&mutexCursor);
537 }
538
539
540 /**
541  * @param arg: -> NULL.
542  *
543  * @return: puntatore a void.

```

```

544 *
545 * La funzione updateField gestisce le funzioni di aggiunta e di aggiornamento
546 * di oggetti della mappa (Locazioni, pacchi, ostacoli, player);
547 *
548 */
549 void* updateField(void* arg){
550
551     int currRow = 0;
552     int currColumn = 0;
553     char* saveptr;
554     char* cmd; /* Messaggio di comando */ 
555     char* type; /* Messaggio di tipo di oggetto */ 
556     char* msg = (char*)arg; /* Messaggio in entrata */ 
557
558     /* Estraggo il tipo di comando */
559     cmd = strtok_r(msg, " ", &saveptr);
560
561     /* Controllo che tipo di comando sia */
562     if(!strcmp("$add", cmd)){
563         type = strtok_r(NULL, " ", &saveptr);
564         /* Se il comando è di tipo add e l'oggetto da aggiungere è un packet allora */
565         if(!strcmp("packet", type)){
566             currRow = atoi(strtok_r(NULL, " ", &saveptr));
567             currColumn = atoi(strtok_r(NULL, " ", &saveptr));
568             /* Pattern creato per gestire la stampa e il movimento del cursore dove
569             stampare. */
570             pthread_mutex_lock(&mutexCursor);
571             setCursor(false);
572             moveto(field[currRow][currColumn].posX, field[currRow][currColumn].posY);
573             setColor(GRAPHICS_FG_COLOR_RED, GRAPHICS_BG_COLOR_GREEN);
574             printf("%s", SYMBOL_PACKET);
575             reset();
576             fflush(stdout);
577             moveto(cursor.posX, cursor.posY);
578             setCursor(true);
579             pthread_mutex_unlock(&mutexCursor);
580             /* Se il comando è di tipo add e l'oggetto da aggiungere è una location allora */
581         } else if(!strcmp("location", type)){
582             currRow = atoi(strtok_r(NULL, " ", &saveptr));
583             currColumn = atoi(strtok_r(NULL, " ", &saveptr));
584             /* Pattern creato per gestire la stampa e il movimento del cursore dove
585             stampare. */
586             pthread_mutex_lock(&mutexCursor);
587             setCursor(false);
588             moveto(field[currRow][currColumn].posX, field[currRow][currColumn].posY);
589             setColor(GRAPHICS_FG_COLOR_WHITE, GRAPHICS_BG_COLOR_MAGENTA);
590             printf("%s", strtok_r(NULL, " ", &saveptr));
591             reset();
592             fflush(stdout);
593             moveto(cursor.posX, cursor.posY);
594             setCursor(true);
595             pthread_mutex_unlock(&mutexCursor);
596             /* Se il comando è di tipo add e l'oggetto da aggiungere è il mio omino allora */
597         } else if(!strcmp("you", type)){
598             currRow = atoi(strtok_r(NULL, " ", &saveptr));
599             currColumn = atoi(strtok_r(NULL, " ", &saveptr));
600             /* Pattern creato per gestire la stampa e il movimento del cursore dove
601             stampare. */

```

```

599     pthread_mutex_lock(&mutexCursor);
600     setCursor(false);
601     moveto(field[currRow][currColumn].posX, field[currRow][currColumn].posY);
602     setColor(GRAPHICS_FG_COLOR_BLUE, GRAPHICS_BG_COLOR_GREEN);
603     printf("%s", SYMBOL_PLAYER);
604     reset();
605     fflush(stdout);
606     moveto(cursor.posX, cursor.posY);
607     setCursor(true);
608     pthread_mutex_unlock(&mutexCursor);
609     /* Se il comando è di tipo add e l'oggetto da aggiungere è un player allora */
610 } else if(!strcmp("player", type)){
611     currRow = atoi(strtok_r(NULL, " ", &saveptr));
612     currColumn = atoi(strtok_r(NULL, " ", &saveptr));
613     /* Pattern creato per gestire la stampa e il movimento del cursore dove
614      stampare. */
615     pthread_mutex_lock(&mutexCursor);
616     setCursor(false);
617     moveto(field[currRow][currColumn].posX, field[currRow][currColumn].posY);
618     setColor(GRAPHICS_FG_COLOR_BLACK, GRAPHICS_BG_COLOR_GREEN);
619     printf("%s", SYMBOL_PLAYER);
620     reset();
621     fflush(stdout);
622     moveto(cursor.posX, cursor.posY);
623     setCursor(true);
624     pthread_mutex_unlock(&mutexCursor);
625     /* Se il comando è di tipo add e l'oggetto da aggiungere è un wall allora */
626 } else if(!strcmp("wall", type)){
627     currRow = atoi(strtok_r(NULL, " ", &saveptr));
628     currColumn = atoi(strtok_r(NULL, " ", &saveptr));
629     /* Pattern creato per gestire la stampa e il movimento del cursore dove
630      stampare. */
631     pthread_mutex_lock(&mutexCursor);
632     setCursor(false);
633     moveto(field[currRow][currColumn].posX, field[currRow][currColumn].posY);
634     setColor(GRAPHICS_FG_COLOR_BLUE, GRAPHICS_BG_COLOR_BLACK);
635     printf("%s", SYMBOL_WALL);
636     reset();
637     fflush(stdout);
638     moveto(cursor.posX, cursor.posY);
639     setCursor(true);
640     pthread_mutex_unlock(&mutexCursor);
641     /* Se il comando è di tipo add e l'oggetto da aggiungere è un blink allora */
642 } else if(!strcmp("blink", type)){
643     rowToBlink = atoi(strtok_r(NULL, " ", &saveptr));
644     colToBlink = atoi(strtok_r(NULL, " ", &saveptr));
645     locationToBlink = strtok_r(NULL, " ", &saveptr);
646     tidBlink = pthread_self();
647     bool status = true;
648     while(true){
649         pthread_mutex_lock(&mutexCursor);
650         setCursor(false);
651         moveto(field[rowToBlink][colToBlink].posX,
652               field[rowToBlink][colToBlink].posY);
653         if(status){
654             setColor(GRAPHICS_FG_COLOR_MAGENTA, GRAPHICS_BG_COLOR_WHITE);
655         } else{
656             setColor(GRAPHICS_FG_COLOR_WHITE, GRAPHICS_BG_COLOR_MAGENTA);
657         }
658     }
659 }

```

```

654         }
655         printf("%s", locationToBlink);
656         reset();
657         moveto(cursor.posX, cursor.posY);
658         setCursor(true);
659         status = !status;
660         pthread_mutex_unlock(&mutexCursor);
661         sleep(1);
662     }
663 }
/* Se il comando è di tipo update allora */
664 } else if(!strcmp("$update", cmd)){
665     currRow = atoi(strtok_r(NULL, " ", &saveptr));
666     currColumn = atoi(strtok_r(NULL, " ", &saveptr));
667     /* Pattern creato per gestire la stampa e il movimento del cursore dove stampare.
668      */
669     pthread_mutex_lock(&mutexCursor);
670     setCursor(false);
671     moveto(field[currRow][currColumn].posX, field[currRow][currColumn].posY);
672     setColor(GRAPHICS_FG_COLOR_GREEN, GRAPHICS_BG_COLOR_GREEN);
673     printf("%s", SYMBOL_GRASS);
674     reset();
675     fflush(stdout);
676     moveto(cursor.posX, cursor.posY);
677     setCursor(true);
678     pthread_mutex_unlock(&mutexCursor);
679 }
680
681 free(msg);
682 }
683 }
684
685
686 /**
687 * @param void.
688 *
689 * @return: void.
690 *
691 * La funzione cleanTextField ripulisce la text field
692 * (write:).
693 *
694 */
695 void cleanTextField(void){
696     pthread_mutex_lock(&mutexCursor);
697     moveto(textField.posX, textField.posY);
698     printf("|
699     |");
700     fflush(stdout);
701     pthread_mutex_unlock(&mutexCursor);
702     goToTheTextField();
703 }
704
705 /**
706 * @param void.
707 *
708 * @return: void.
709 *
710 * La funzione goToTheTextField muove il cursore sulla

```

```

710     * La funzione goToTheTextField muove il cursore sulla
711     * text field (Write:).
712     *
713     */
714 void goToTheTextField(void){
715     pthread_mutex_lock(&mutexCursor);
716     cursor.posX = textField.posX;
717     moveto(textField.posX, textField.posY);
718     pthread_mutex_unlock(&mutexCursor);
719 }
720
721
722 /**
723 * @param arg: -> NULL.
724 *
725 * @return: puntatore a void.
726 *
727 * La funzione ListenerTextField si mette un ascolto di input
728 * dalla text field (Write:). L'input "catturato" verrà poi
729 * processato da un'apposita funzione.
730 *
731 * Time wasted: 7h 49m >.<
732 *
733 */
734 void* listenerTextField(void* arg){
735
736     pthread_t tidUpdateChat;
737     char msg[GRAPHICS_TEXTFIELD_STRLEN];
738     char msgUser[GRAPHICS_CHAT_WIDTH];
739     LpMsg msgChat;
740     char keyTyped;
741     int currMsgCharacter = 0;
742
743     while(true){
744         currMsgCharacter = 0;
745         while((keyTyped = getch()) != GRAPHICS_KEY_ENTER){
746             if(isAlphabet(keyTyped)){
747                 if(keyTyped != GRAPHICS_KEY_BACKSPACE){
748                     if(currMsgCharacter < GRAPHICS_TEXTFIELD_STRLEN){
749                         msg[currMsgCharacter] = keyTyped;
750                         currMsgCharacter += 1;
751                         pthread_mutex_lock(&mutexCursor);
752                         cursor.posX += 1;
753                         printf("%c", keyTyped);
754                         fflush(stdout);
755                         pthread_mutex_unlock(&mutexCursor);
756                     } else{
757                         beep();
758                     }
759                 } else if(keyTyped == GRAPHICS_KEY_BACKSPACE && currMsgCharacter != 0){
760                     msg[currMsgCharacter] = '\0';
761                     pthread_mutex_lock(&mutexCursor);
762                     cursor.posX -= 1;
763                     moveto(cursor.posX, cursor.posY);
764                     printf(" ");
765                     moveto(cursor.posX, cursor.posY);
766                     fflush(stdout);
767                     pthread_mutex_unlock(&mutexCursor);
768                     currMsgCharacter -= 1;
769                 }
770             }
771         }
772     }

```

```

766         currMsgCharacter -- ;
767     } else if(keyTyped == GRAPHICS_KEY_BACKSPACE && currMsgCharacter == 0){
768         beep();
769     }
770     } else{
771         beep();
772     }
773 }
774 }
775 }
776 msg[currMsgCharacter] = '\0';
777 cleanTextField();
778
779 if(trimMsg(msg) != 1){
780     memset(msgUser, '\0', GRAPHICS_CHAT_WIDTH);
781     strcat(msgUser, "Me: \0");
782     strcat(msgUser, msg);
783     if((msgChat = allocMsg(msgUser, GRAPHICS_FG_COLOR_YELLOW, false)) != NULL){
784         pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
785     }
786     sendMsg(msg);
787 }
788 memset(msg, '\0', GRAPHICS_TEXTFIELD_STRLEN);
789 }
790 }
791
792
793 //_____FUNCTION FOR MANAGEMENT OPERATION CLIENT
794
795 /**
796 * @param keyTyped: Tasto premuto.
797 *
798 * @return: true se il carattere è consentito, false altrimenti.
799 *
800 * La funzione isAlphabet verifica se il carattere premuto
801 * dall'utente è consentito.
802 *
803 */
804 bool isAlphabet(char keyTyped){
805     if((keyTyped >= 32 && keyTyped <= 255) || keyTyped == GRAPHICS_KEY_ENTER){
806         return true;
807     }
808
809     return false;
810 }
811
812
813
814 /**
815 * @param void.
816 *
817 * @return: 1 in caso di errore, 0 in caso successo.
818 *
819 * La funzione connection effettua la connessione al server
820 * mediante le specifiche fornite dall'utente.
821 *
822 */
823 int connection(void){
824     int bytesReaded = 0;                                /* Numero di bytes letti dalla
825     * read           */

```

```

825     int port;                                /* Porta inserita
826     * dal'utente                      */
827     char addressBuffer[GRAPHICS_TEXTFIELD_STRLEN]; /* Buffer per l'indirizzo del
828     * server          */
829     char portBuffer[GRAPHICS_TEXTFIELD_STRLEN];   /* Buffer per la porta del
830     * server          */
831     bool error = false;                     /* Flag che indica la presenza di
832     * errori      */
833
834
835
836     do{
837         clean();
838         /* Riabilito il cursore nel caso fosse disabilitato */
839         setCursor(true);
840         /* Richiedo l'indirizzo del server */
841         printf("\n » Inserire l'indirizzo del server: ");
842         fflush(stdout);
843         do{
844             error = false;
845             if((bytesReaded = read(STDIN_FILENO, addressBuffer, GRAPHICS_TEXTFIELD_STRLEN)) == -1){
846                 setColor(GRAPHICS_FG_COLOR_RED, 0);
847                 perror("\n    <!> Errore read");
848                 reset();
849                 return 1;
850             }
851             addressBuffer[bytesReaded-1] = '\0';
852             /* Verifico se l'indirizzo fornito è valido */
853             if/inet_aton(addressBuffer, &address.sin_addr) == 0){
854                 setColor(GRAPHICS_FG_COLOR_RED, 0);
855                 printf("\n    <!> Indirizzo non valido - riprovare: ");
856                 reset();
857                 fflush(stdout);
858                 error = true;
859             } else{
860                 setColor(GRAPHICS_FG_COLOR_GREEN, 0);
861                 printf("\n    » Indirizzo %s inserito con successo.\n", addressBuffer);
862                 reset();
863             }
864         }while(error != false);
865
866         /* Richiedo la porta del server */
867         printf("\n » Inserire la porta del server: ");
868         fflush(stdout);
869         do{
870             error = false;
871             if((bytesReaded = read(STDIN_FILENO, portBuffer, GRAPHICS_TEXTFIELD_STRLEN)) == -1){
872                 setColor(GRAPHICS_FG_COLOR_RED, 0);
873                 perror("\n    <!> Errore read");
874                 reset();
875                 return 1;
876             }

```

```

877     portBuffer[bytesReaded] = '\0';
878     /* Verifico se la porta fornita è valida */
879     if((port = atoi(portBuffer)) == 0 || !(port >= 0 && port <= 65535)){
880         setColor(GRAPHICS_FG_COLOR_RED, 0);
881         printf("\n    <!> Porta non valida - riprovare: ");
882         reset();
883         fflush(stdout);
884         error = true;
885     } else{
886         setColor(GRAPHICS_FG_COLOR_GREEN, 0);
887         printf("\n    » Porta %d inserita con successo.\n", port);
888         reset();
889         address.sin_port = htons(atoi(portBuffer));
890     }
891 }while(error != false);

893     /* Provo ad effettuare la connessione al server */
894     if(connect(clientSocket, (struct sockaddr*)&address, sizeof(address)) == -1){
895         setColor(GRAPHICS_FG_COLOR_RED, 0);
896         perror("\n <!> Errore connect, impossibile raggiungere il server - "
897             "riprovare.");
898         reset();
899         error = true;
900     } else{
901         connected = true;
902         setColor(GRAPHICS_FG_COLOR_BLUE, 0);
903         printf("\n » Connessione al server [ADDR: %s - PORT: %d] effettuata con "
904             "successo.\n", addressBuffer, port);
905         reset();
906     }
907
908     /* Disabilito il cursore */
909     setCursor(false);
910     sleep(1);

911 }while(error != false);

912
913 return 0;
914 }

915 /**
916 * @param msg[]: Messaggio da regolare.
917 *
918 * @return: 1 se il messaggio ottenuto è vuoto 0 altrimenti.
919 *
920 * La funzione trimMsg elimina eventuali spazi iniziali
921 * e di coda e inoltre verifica se il messaggio è vuoto oppure
922 * no.
923 *
924 */
925
926 int trimMsg(char msg[]){
927
928     int index = strlen(msg)-1;
929     int tmp = 0;
930     int len;
931
932     /* Elimino gli spazi finali */

```

```

933     while(msg[index] == ' '){
934         msg[index] = '\0';
935         index -= 1;
936     }
937
938     /* Elimino gli spazi iniziali */
939     index = 0;
940     while(msg[index] == ' '){
941         index += 1;
942     }
943     if(index != 0){
944         tmp = 0;
945         while(msg[tmp+index] != '\0'){
946             msg[tmp] = msg[tmp+index];
947             tmp++;
948         }
949         msg[tmp] = '\0';
950     }
951
952     /* $ è l'intestazione dei messaggi del Server */
953     while(msg[0] == SERVER_NOTIFY){
954         index = 0;
955         for(index=0; index<strlen(msg)-1; index++){
956             msg[index] = msg[index+1];
957         }
958         msg[index] = '\0';
959     }
960
961     /* Verifico se la stringa ottenuta è vuota */
962     len = strlen(msg);
963     if(len == 0){
964         return 1;
965     }
966
967     return 0;
968
969 }
970
971 /**
972 * @param msg: Messaggio da scrivere sulla chat.
973 * @param colorPair: Colore con cui stampare il messaggio.
974 * @param error: Indica se il messaggio Ã" di errore.
975 *
976 * @return: una struttura di tipo LpMsg.
977 *
978 * La funzione allocMsg alloca un nodo Msg per il thread
979 * updateChat.
980 *
981 */
982
983 LpMsg allocMsg(char* msg, int color, bool error){
984     LpMsg msgChat;
985     if((msgChat = (LpMsg)malloc(sizeof(Msg))) != NULL){
986         strcpy(msgChat->msg, msg);
987         msgChat->color = color;
988         msgChat->error = error;
989     }
990     return msgChat;

```

```

991
992     }
993
994
995 /**
996  * @param msg: messaggio da inoltrare al server.
997  *
998  * @return: void.
999  *
1000 * La funzione sendMsg inoltra il messaggio catturato
1001 * dal textField al server.
1002 *
1003 */
1004 void sendMsg(char* msg){
1005
1006     pthread_t tidUpdateChat;
1007     char buffer[GRAPHICS_TEXTFIELD_STRLEN];
1008     memset(buffer, '\0', GRAPHICS_TEXTFIELD_STRLEN);
1009     strcpy(buffer, msg);
1010
1011     write(clientSocket, buffer, GRAPHICS_TEXTFIELD_STRLEN);
1012
1013 }
1014
1015
1016 /**
1017  * @param arg -> NULL.
1018  *
1019  * @return: puntatore a void.
1020  *
1021 * La funzione ListenerServer si mette un ascolto di messaggi
1022 * dal Server.
1023 *
1024 */
1025 void* listenerServer(void* arg){
1026
1027     pthread_t tidUpdateChat;
1028     LpMsg msgChat;
1029     char incomingMsg[GRAPHICS_CHAT_WIDTH];
1030     int bytesReaded;
1031     int action;
1032     int index;
1033
1034     memset(incomingMsg, '\0', GRAPHICS_CHAT_WIDTH);
1035
1036     while((bytesReaded = read(clientSocket, incomingMsg, GRAPHICS_CHAT_WIDTH)) > 0){
1037         incomingMsg[bytesReaded] = '\0';
1038         if(incomingMsg[0] == SERVER_NOTIFY){
1039             if((action = takeAction(incomingMsg)) == 0){
1040                 trimMsg(incomingMsg);
1041                 if((msgChat = allocMsg(incomingMsg, GRAPHICS_FG_COLOR_BLUE, false)) != NULL){
1042                     pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
1043                 }
1044             } else if(action == -1){
1045                 if((msgChat = allocMsg("<!> Errore malloc: impossibile allocare
1046                 * memoria.\0", GRAPHICS_FG_COLOR_RED, true)) != NULL){
1047                     pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
1048                 }
1049             }
1050         }
1051     }

```

```

1048         sleep(2);
1049         return NULL;
1050     }
1051 } else if(incomingMsg[0] == SESSION_NOTIFY){
1052     index = 0;
1053     for(index=0; index<strlen(incomingMsg)-1; index++){
1054         incomingMsg[index] = incomingMsg[index+1];
1055     }
1056     incomingMsg[index] = '\0';
1057
1058     if((msgChat = allocMsg(incomingMsg, GRAPHICS_FG_COLOR_MAGENTA, false)) !=
1059     * NULL){
1060         pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
1061     }
1062 } else{
1063     if((msgChat = allocMsg(incomingMsg, GRAPHICS_FG_COLOR_BLACK, false)) != NULL){
1064         pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
1065     }
1066 }
1067 memset(incomingMsg, '\0', GRAPHICS_CHAT_WIDTH);
1068 }
1069
1070 /* In caso di fallimento esco */
1071 if((msgChat = allocMsg("Server: Sei stato disconnesso dal server...\0",
1072 * GRAPHICS_FG_COLOR_RED, false)) != NULL){
1073     pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
1074 }
1075
1076 return NULL;
1077 }
1078
1079 /**
1080 * @param msg: Messaggio in entrata.
1081 *
1082 * @return: 1 nel caso il messaggio richiedeva azioni, 0 altrimenti, -1 in caso di
1083 * errore.
1084 *
1085 * La funzione takeAction verifica se il messaggio in entrata
1086 * richiede di effettuare delle determinate azioni. In caso
1087 * affermativo provvede ad effettuarle.
1088 *
1089 */
1090 int takeAction(char msg[]){
1091     pthread_t tid;                      /* TID del thread che gestisce il comando
1092     */
1093     char* dynamicBuffer;                /* Messaggio da passare ai thread
1094     */
1095     char* saveptr;
1096     char buffer[GRAPHICS_CHAT_WIDTH];   /* Buffer utilizzato per la tokenization
1097     */
1098     int action;
1099
1100     if((dynamicBuffer = (char*)calloc(GRAPHICS_CHAT_WIDTH, sizeof(char))) == NULL){
1101         return -1;
1102     }

```

```

1100
1101     strcpy(dynamicBuffer, msg);
1102     strcpy(buffer, msg);
1103     char* cmd = strtok_r(buffer, " ", &saveptr);
1104
1105     /* Estraggo la prima stringa del messaggio (comando) e verifico che comando sia */
1106     if(!strcmp(cmd, "$time")){
1107         int* time = (int*)malloc(sizeof(int));
1108         *time = atoi(strtok_r(NULL, " ", &saveptr));
1109         pthread_create(&tid, NULL, updateTimer, time);
1110         return 1;
1111     } else if(!strcmp(cmd, "$add") || !strcmp(cmd, "$update")){
1112         pthread_create(&tid, NULL, updateField, dynamicBuffer);
1113         return 1;
1114     } else if(!strcmp(cmd, "$disconnected") || !strcmp(cmd, "$connected") ||
1115     !strcmp(cmd, "$delivered")){
1116         pthread_create(&tid, NULL, updatePlayersInfo, dynamicBuffer);
1117         return 1;
1118     } else if(!strcmp(cmd, "$remove")){
1119         char* type = strtok_r(NULL, " ", &saveptr);
1120         if(!strcmp(type, "blink")){
1121             if(tidBlink != 0){
1122                 pthread_cancel(tidBlink);
1123                 pthread_mutex_lock(&mutexCursor);
1124                 setCursor(false);
1125                 moveto(field[rowToBlink][colToBlink].posX,
1126                     field[rowToBlink][colToBlink].posY);
1127                 setColor(GRAPHICS_FG_COLOR_WHITE, GRAPHICS_BG_COLOR_MAGENTA);
1128                 printf("%s", locationToBlink);
1129                 reset();
1130                 moveto(cursor.posX, cursor.posY);
1131                 setCursor(true);
1132                 pthread_mutex_unlock(&mutexCursor);
1133                 tidBlink = 0;
1134             }
1135         }
1136         return 1;
1137     } else if(!strcmp(cmd, "$reset")){
1138         pthread_mutex_lock(&mutexCursor);
1139         setCursor(false);
1140         for(int i=0; i<GRAPHICS_FIELD_HEIGHT; i++){
1141             for(int j=0; j<GRAPHICS_FIELD_WIDTH; j++){
1142                 moveto(field[i][j].posX, field[i][j].posY);
1143                 setColor(GRAPHICS_FG_COLOR_GREEN, GRAPHICS_BG_COLOR_GREEN);
1144                 printf(" ");
1145                 reset();
1146             }
1147         }
1148         moveto(cursor.posX, cursor.posY);
1149         setCursor(true);
1150         pthread_mutex_unlock(&mutexCursor);
1151         return 1;
1152     }
1153 }
1154
1155

```

```
1156 /**
1157 * @param signal: segnale catturato
1158 *
1159 * @return void.
1160 *
1161 * La funzione signalHandler cattura i segnali in questione
1162 * e ne detta il comportamento al loro verificarsi.
1163 *
1164 */
1165 void signalHandler(int signal){
1166     switch(signal){
1167         case SIGINT:
1168             sendMsg("@exit");
1169             break;
1170         case SIGSTOP:
1171             sendMsg("@exit");
1172             break;
1173     }
1174 }
1175
1176
1177 //***** END CLIENT
1178 /******/
```

```

1  | **** INIT SERVER ****
2  |
3  // _____ LIBRARIES _____
4  |
5  |
6  #include "../libs/session/session.h"
7  #include <sys/socket.h>
8  #include <sys/types.h>
9  #include <netinet/in.h>
10 #include <errno.h>
11 #include <arpa/inet.h>
12 #include <pthread.h>
13 #include <stdbool.h>
14 #include <string.h>
15 #include <stdio.h>
16 #include <signal.h>
17 #include <unistd.h>
18 #include <time.h>
19 #include <fcntl.h>
20 #include <ctype.h>
21 #include <stdlib.h>
22
23 // _____ COSTANTS _____
24
25 #define SERVER_NOTIFY          '$'
26 #define SESSION_NOTIFY         '%'
27 #define SYMBOL_GRASS           " "
28 #define SYMBOL_PLAYER          "\u263b"
29 #define SYMBOL_PACKET          "\u2666"
30 #define SYMBOL_WALL             "\u2590"
31 #define LISTENER_QUEUE_STRLEN  50
32 #define INCOMING_MSG_STRLEN   70
33 #define OUTCOMING_MSG_STRLEN  82
34 #define TIME                   600
35 #define BUFFER_STRLEN          150
36
37 // _____ GLOBAL
38 // _____ VARIABLES _____
39
40 Coord timer;
41 Coord status[GRAPHICS_STATUS_HEIGHT];
42 Coord field[GRAPHICS_FIELD_HEIGHT][GRAPHICS_FIELD_WIDTH];
43 PlayersCounter playersCounter;
44 PlayersInfo playersInfo;
45 Chat chat;
46 LpSession session = NULL;
47 int listenerSocket;
48 int listenerPort;
49
50 int logs;
51 int database;
52 int fieldGenerator;

```

```

52
53 LpClientInfo listClientInfo = NULL;
54 LpClientInfoToJoin headClientInfoToJoinQueue = NULL;
55 LpClientInfoToJoin tailClientInfoToJoinQueue = NULL;
56
57 int sessionNumOfPackets = 0;
58
59 int idSession = 0;
60 int totalConnections = 0;
61 int totalDisconnects = 0;
62 int waitingClients = 0;
63 int connectedClients = 0;
64 int difficulty = 0;
65
66 pthread_mutex_t mutexCursor = PTHREAD_MUTEX_INITIALIZER;
67 pthread_mutex_t mutexClientInfo = PTHREAD_MUTEX_INITIALIZER;
68 pthread_mutex_t mutexDatabase = PTHREAD_MUTEX_INITIALIZER;
69 pthread_mutex_t mutexUpdate = PTHREAD_MUTEX_INITIALIZER;
70 pthread_mutex_t mutexTimer = PTHREAD_MUTEX_INITIALIZER;
71 pthread_mutex_t mutexLogs = PTHREAD_MUTEX_INITIALIZER;
72
73 pthread_cond_t condMatchmaker = PTHREAD_COND_INITIALIZER;
74 pthread_cond_t condSession = PTHREAD_COND_INITIALIZER;
75
76 //____FUNCTIONS
77 * DECLARATION
78
79 int initGUI(char*, char*);
80 int mapping(char* );
81 int initFile(void);
82 int connection(void);
83 void* updateChat(void* );
84 void* connectionRequestsManagement(void* );
85 void* listenerClient(void* );
86 void* noecho(void* );
87 LpMsg allocMsg(char*, int, bool);
88 void sendMsg(LpClientInfo, char* );
89 void sendMsgToAll(LpClientInfo, char* );
90 void sendMsgToSession(LpClientInfo, char* );
91 bool login(LpClientInfo);
92 bool signin(LpClientInfo);
93 int initField(void);
94 void shuffleLocations(char[], int);
95 void shufflePackets(Packet[], int);
96 void shuffleSpawn(Spawn[], int);
97 int initSession(void);
98 void* sessionManagement(void* );
99 void* timeProvider(void* );
100 void* matchmaker(void* );
101 void sendBasicInformation(LpClientInfo);
102 void disconnectionManagement(LpClientInfo);
103 void movePlayer(LpClientInfo, char[]);
104 void actionPlayer(LpClientInfo, char[]);
105 void signalHandler(int);
106 void* updatePlayersInfo(void* );
107
108

```

```

109     int main(void){
110
111     pthread_t tidConnectionRequestsManagement;
112     pthread_t tidSessionManagement;
113
114     srand(time(NULL));
115
116     signal(SIGPIPE, SIG_IGN);
117     signal(SIGALRM, SIG_IGN);
118
119     if(initFile()){
120         beep();
121         setColor(GRAPHICS_FG_COLOR_RED, GRAPHICS_BG_COLOR_WHITE);
122         printf("\n <!> Errore durante l'inizializzazione del Server.\n\n");
123         reset();
124         return 1;
125     }
126
127     if(connection()){
128         return 1;
129     }
130
131     signal(SIGSTOP, signalHandler);
132     signal(SIGINT, signalHandler);
133
134     if(initGUI("../gui/blueprint.txt", "../gui/gui.txt")){
135         beep();
136         setColor(GRAPHICS_FG_COLOR_RED, GRAPHICS_BG_COLOR_WHITE);
137         printf("\n <!> Errore durante l'inizializzazione dell'interfaccia grafica.\n\n");
138         reset();
139         return 1;
140     }
141
142     pthread_create(&tidConnectionRequestsManagement, NULL,
143     connectionRequestsManagement, NULL);
144
145     while(true){
146         if(initSession()){
147             char msg[GRAPHICS_CHAT_WIDTH];
148             LpMsg msgChat;
149             pthread_t tidUpdateChat;
150             sprintf(msg, "Errore durante l'inizializzazione della sessione!");
151             if((msgChat = allocMsg(msg, GRAPHICS_FG_COLOR_RED, true)) != NULL){
152                 pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
153             }
154             pthread_join(tidUpdateChat, NULL);
155             raise(SIGINT);
156         }
157         pthread_create(&tidSessionManagement, NULL, sessionManagement, NULL);
158         pthread_join(tidSessionManagement, NULL);
159     }
160
161     return 0;
162 }
163
164 //____FUNCTION FOR MANAGEMENT GRAPHICS AND ACTION
165 * CLIENT_____
```

```

165
166 /**
167 * @param blueprint: path del file blueprint
168 * che inizializza la matrice per la grafica finale.
169 * @param gui: path del file della grafica finale
170 * da andare a stampare.
171 *
172 * @return: 1 in caso di errore 0 altrimenti.
173 *
174 * La funzione initGUI apre in sola lettura il file che
175 * contiene le grafica che poi stamperà sul rispettivo terminale.
176 *
177 */
178 int initGUI(char* blueprint, char* gui){
179
180     char basicComponent[1];
181     int bytesReaded;
182     int guiFile;
183     pthread_t tidNoEcho;
184     pthread_t tidUpdateChat;
185     LpMsg msgChat;
186     char msg[GRAPHICS_CHAT_WIDTH];
187
188     setCursor(false);
189     pthread_create(&tidNoEcho, NULL, noecho, NULL);
190
191     if(mapping(blueprint)){
192         return 1;
193     }
194
195     if((guiFile = open(gui, O_RDONLY)) == -1){
196         return 1;
197     }
198
199     clean();
200     while((bytesReaded = read(guiFile, basicComponent, 1)) > 0){
201         basicComponent[bytesReaded] = '\0';
202         printf("%s", basicComponent);
203     }
204
205     sprintf(msg, "Sono in ascolto sulla porta %d!", listenerPort);
206     if((msgChat = allocMsg(msg, GRAPHICS_FG_COLOR_BLUE, false)) != NULL){
207         pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
208     }
209
210     pthread_mutex_lock(&mutexCursor);
211     moveto(status[0].posX, status[0].posY);
212     printf(" \u25cf Porta: %10d", listenerPort);
213     moveto(status[1].posX, status[1].posY);
214     printf(" \u25cf ID Sessione: %4.2d", idSession);
215     moveto(status[2].posX, status[2].posY);
216     printf(" \u25cf Connected: %6.2d", connectedClients);
217     moveto(status[3].posX, status[3].posY);
218     printf(" \u25cf Waiting: %8.2d", waitingClients);
219     moveto(status[4].posX, status[4].posY);
220     printf(" \u25cf Total CN: %7.2d", totalConnections);
221     moveto(status[5].posX, status[5].posY);
222     printf(" \u25cf Total DC: %7.2d", totalDisconnections);
223     moveto(status[6].posX, status[6].posY);

```

```

223     moveTo(statusObj posX, statusObj posY);
224     printf(" \u25cf Difficoltà: %5.2d", difficulty);
225     pthread_mutex_unlock(&mutexCursor);
226
227 }
228
229
230 /**
231 * @param blueprint: path del file che inizializza
232 * La matrice per poter stampare poi sopra la grafica.
233 *
234 * @return: return 1 in caso di errore e 0 altrimenti.
235 *
236 * La funzione mapping apre in sola Lettura Lettura
237 * il file che inizializza la matrice per la grafica.
238 *
239 */
240 int mapping(char* blueprint){
241
242     char basicComponent[1];
243     int bytesReaded;
244     int blueprintFile;
245     int currRowsScreen = 1;
246     int currColsScreen = 1;
247     int currRowsField = 0;
248     int currColsField = 0;
249     int currPlayerPosition = 0;
250     int currMsgPosition = 0;
251     int currStatusPosition = 0;
252
253     if((blueprintFile = open(blueprint, O_RDONLY)) == -1){
254         return 1;
255     }
256
257     while((bytesReaded = read(blueprintFile, basicComponent, 1)) > 0){
258         basicComponent[bytesReaded] = '\0';
259         switch (basicComponent[0]) {
260             case GRAPHICS_PIN_CHAT:
261                 chat.msgBox[currMsgPosition].coords.posX = currColsScreen;
262                 chat.msgBox[currMsgPosition].coords.posY = currRowsScreen;
263                 currMsgPosition += 1;
264                 break;
265             case GRAPHICS_PIN_FIELD:
266                 field[currRowsField][currColsField].posX = currColsScreen;
267                 field[currRowsField][currColsField].posY = currRowsScreen;
268                 currColsField += 1;
269                 if(currColsField == GRAPHICS_FIELD_WIDTH){
270                     currRowsField += 1;
271                     currColsField = 0;
272                 }
273                 break;
274             case GRAPHICS_PIN_TIMER:
275                 timer.coords posX = currColsScreen;
276                 timer.coords posY = currRowsScreen;
277                 break;
278             case GRAPHICS_PIN_COUNTER:
279                 playersCounter.coords posX = currColsScreen;
280                 playersCounter.coords posY = currRowsScreen;
281                 break;

```

```

        direction,
282     case GRAPHICS_PIN_PLAYERS:
283         playersInfo.playerBox[currPlayerPosition].coords posX = currColsScreen;
284         playersInfo.playerBox[currPlayerPosition].coords posY = currRowsScreen;
285         currPlayerPosition += 1;
286         break;
287     case GRAPHICS_PIN_STATUS:
288         status[currStatusPosition].posX = currColsScreen;
289         status[currStatusPosition].posY = currRowsScreen;
290         currStatusPosition += 1;
291         break;
292     case GRAPHICS_PIN_START:
293         clean();
294         break;
295     }
296
297     currColsScreen += 1;
298     if(basicComponent[0] == '\n'){
299         currColsScreen = 1;
300         currRowsScreen += 1;
301     }
302 }
303
304 if(bytesReaded == -1){
305     return 1;
306 }
307
308 chat.currMsgPosition = 0;
309 playersInfo.currPlayerPosition = -1;
310 playersCounter.value = 0;
311 for(int msg = 0; msg < GRAPHICS_CHAT_HEIGHT; msg++){
312     chat.msgBox[msg].isEmpty = true;
313 }
314
315 close(blueprintFile);
316 return 0;
317
318 }
319
320
321 /**
322 * @param arg: NULL.
323 *
324 * @return: return 1 in caso di errore e 0 altrimenti.
325 *
326 * La funzione noecho blocca l'input utente.
327 *
328 */
329 void* noecho(void* arg){
330     while(true){
331         getch();
332     }
333 }
334
335
336 /**
337 * @param void.
338 *
339 * @return: 1 in caso di errore 0 altrimenti

```

```

340     *
341     * La funzione initField Legge i valori dal file fieldGenerator
342     * e inizializza i valori della matrice con quelli letti dal file.
343     *
344     */
345 int initField(void){
346
347     int bytesReaded;
348     char character[1];
349     int currRows = 0;
350     int curCols = 0;
351     int gameDifficulty;
352     char msg[GRAPHICS_CHAT_WIDTH];
353     LpMsg msgChat;
354     pthread_t tidUpdateChat;
355     char logsBuffer[BUFFER_STRLEN];
356
357     lseek(fieldGenerator, 0, SEEK_SET);
358
359     do{
360         /*Leggo carattere per carattere il file fieldGenerator per inizializzare la matrice
361         */
362         if((bytesReaded = read(fieldGenerator, character, 1)) > 0){
363             if(character[0] != '\n'){
364                 session->field[curCols][currRows++] = atoi(character);
365             } else{
366                 curCols++;
367                 currRows=0;
368             }
369         }
370     }while(bytesReaded != 0);
371
372     if(bytesReaded == -1){
373         return 1;
374     } else{
375
376         char locationName[5] = {'A', 'B', 'C', 'D', 'E'};
377         int currLocation = 0;
378         int currPacket = 0;
379         int currSpawn = 0;
380         int wallSpawnPercentage = 0;
381
382         /* Assegno la percentuale di spawn di muri a seconda della difficoltà*/
383         if(difficulty == 0){
384             gameDifficulty = rand()%10+1;
385             pthread_mutex_lock(&mutexCursor);
386             moveto(status[6].posX, status[6].posY);
387             printf(" \u25cf Difficolta: %5.2d", gameDifficulty);
388             pthread_mutex_unlock(&mutexCursor);
389         } else{
390             gameDifficulty = difficulty;
391         }
392         switch (gameDifficulty){
393             case 0:
394                 wallSpawnPercentage = rand()%10+1;
395                 break;
396             case 1:
397                 wallSpawnPercentage = 1;

```

```

397     break;
398     case 2:
399         wallSpawnPercentage = 2;
400     break;
401     case 3:
402         wallSpawnPercentage = 3;
403     break;
404     case 4:
405         wallSpawnPercentage = 4;
406     break;
407     case 5:
408         wallSpawnPercentage = 5;
409     break;
410     case 6:
411         wallSpawnPercentage = 6;
412     break;
413     case 7:
414         wallSpawnPercentage = 7;
415     break;
416     case 8:
417         wallSpawnPercentage = 8;
418     break;
419     case 9:
420         wallSpawnPercentage = 9;
421     break;
422     case 10:
423         wallSpawnPercentage = 10;
424     break;
425 }
426 sprintf(msg, "Difficoltà della nuova sessione: %d.", gameDifficulty);
427 if((msgChat = allocMsg(msg, GRAPHICS_FG_COLOR_BLUE, false))){
428     pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
429 }
430
431 pthread_mutex_lock(&mutexLogs);
432 sprintf(logsBuffer, "Difficoltà della sessione [%d]: %d\n.", idSession,
433 * gameDifficulty);
434 if(write(logs, logsBuffer, strlen(logsBuffer)) == -1){
435     if((msgChat = allocMsg("Errore durante la scrittura nel file di logs.",
436 * GRAPHICS_FG_COLOR_RED, true)) != NULL){
437         pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
438     }
439 }
440 pthread_mutex_unlock(&mutexLogs);
441 /*Inizializzo i valori e le posizioni di ogni oggetto nella matrice della sessione*/
442 for(int i=0; i<GRAPHICS_FIELD_HEIGHT; i++){
443     for(int j=0; j<GRAPHICS_FIELD_WIDTH; j++){
444         switch (session->field[i][j]) {
445             case 1:
446                 if(rand()%10+1 < wallSpawnPercentage){
447                     session->field[i][j] = SESSION_WALL_VALUE;
448                 } else{
449                     session->field[i][j] = SESSION_GRASS_VALUE;
450                 }
451             break;
452             case 2:
453                 session->field[i][j] = SESSION_LOCATION_VALUE;
454                 session->locations[currLocation].currRows = i;

```

```

453         session->locations[currLocation].currCols = j;
454         currLocation += 1;
455         break;
456     case 3:
457         session->field[i][j] = SESSION_PACKET_VALUE;
458         session->packets[currPacket].currRows = i;
459         session->packets[currPacket].currCols = j;
460         session->packets[currPacket].location = rand()%5;
461         currPacket += 1;
462         break;
463     case 4:
464         session->spawn[currSpawn].currRows = i;
465         session->spawn[currSpawn].currCols = j;
466         currSpawn += 1;
467         break;
468     }
469 }
470 }
471 }
472
473 /* Randomizzo il numero di packet che verrà stampato*/
474 session->numOfPackets = rand()%18+4;
475 sessionNumOfPackets = session->numOfPackets;
476 /* Randomizzo le posizioni delle locazioni da stampare */
477 shuffleLocations(locationName, SESSION_LOCATIONS_STRLEN);
478 for(int i=0; i<SESSION_LOCATIONS_STRLEN; i++){
479     session->locations[i].name = locationName[i];
480 }
481
482 /* Randomizzo la posizione dei packet da stampare */
483 shufflePackets(session->packets, SESSION_PACKETS_STRLEN);
484 for(int i=session->numOfPackets; i<SESSION_PACKETS_STRLEN; i++){
485     session->field[session->packets[i].currRows][session->packets[i].currCols] =
486     SESSION_GRASS_VALUE;
487 }
488 /* Randomizzo la posizione degli spawn da stampare */
489 for(int i=0; i<SESSION_SPAWN_STRLEN; i++){
490     session->field[session->spawn[i].currRows][session->spawn[i].currCols] =
491     SESSION_GRASS_VALUE;
492 }
493 }
494
495
496 /**
497 * @param arg: msgChat -> messaggio da inserire in chat .
498 *
499 * @return: puntatore a void.
500 *
501 * La funzione updateChat aggiorna la chat stampando
502 * eventuali nuovi messaggi. Nel caso in cui la chat fosse
503 * piena viene effettuato lo shit della stessa dal basso
504 * verso l'alto.
505 *
506 */
507 void* updateChat(void* arg){
508

```

```

509 LpMsg msgChat = (LpMsg)arg;
510 char msg[GRAPHICS_CHAT_WIDTH];
511 strcpy(msg, msgChat->msg);
512 int color = msgChat->color;
513
514 pthread_mutex_lock(&mutexUpdate);
515
516 if(chat.currMsgPosition == GRAPHICS_CHAT_HEIGHT-1 &&
517 chat.msgBox[chat.currMsgPosition].isEmpty == false){
518     for(int currMsg = 0; currMsg < GRAPHICS_CHAT_HEIGHT-1; currMsg++){
519         strcpy(chat.msgBox[currMsg].msg, chat.msgBox[currMsg+1].msg);
520         chat.msgBox[currMsg].color = chat.msgBox[currMsg+1].color;
521     }
522     strcpy(chat.msgBox[GRAPHICS_CHAT_HEIGHT-1].msg, msg);
523     chat.msgBox[GRAPHICS_CHAT_HEIGHT-1].color = color;
524     for(int currMsg = 0; currMsg < GRAPHICS_CHAT_HEIGHT; currMsg++){
525         pthread_mutex_lock(&mutexCursor);
526         moveto(chat.msgBox[currMsg].coords.posX, chat.msgBox[currMsg].coords.posY);
527         printf("      |      |");
528         fflush(stdout);
529         moveto(chat.msgBox[currMsg].coords.posX, chat.msgBox[currMsg].coords.posY);
530         setColor(chat.msgBox[currMsg].color, 0);
531         printf("%s", chat.msgBox[currMsg].msg);
532         fflush(stdout);
533         reset();
534         pthread_mutex_unlock(&mutexCursor);
535     }
536 } else{
537     chat.msgBox[chat.currMsgPosition].isEmpty = false;
538     chat.msgBox[chat.currMsgPosition].color = color;
539     strcpy(chat.msgBox[chat.currMsgPosition].msg, msg);
540     pthread_mutex_lock(&mutexCursor);
541     moveto(chat.msgBox[chat.currMsgPosition].coords.posX,
542             chat.msgBox[chat.currMsgPosition].coords.posY);
543     printf("      |      |");
544     fflush(stdout);
545     moveto(chat.msgBox[chat.currMsgPosition].coords.posX,
546             chat.msgBox[chat.currMsgPosition].coords.posY);
547     setColor(color, 0);
548     printf("%s", msg);
549     fflush(stdout);
550     reset();
551     pthread_mutex_unlock(&mutexCursor);
552 }
553
554 pthread_mutex_unlock(&mutexUpdate);
555
556 if(msgChat->error == true){
557     beep();
558 }
559
560 free(msgChat);
561

```

```

562     }
563
564
565     /**
566      * @param msg: arg -> Secondi totali.
567      *
568      * @return: puntatore a void.
569      *
570      * La funzione timeProvider aggiorna e stampa il timer di fine
571      * partita.
572      *
573      */
574     void* timeProvider(void* arg){
575
576     int totalSeconds = TIME;
577     int seconds;
578     int minutes;
579     char msg[GRAPHICS_CHAT_WIDTH];
580     bool finished = false;
581
582     alarm(TIME);
583
584     /* Ciclo finche il tempo non è scaduto o i tutti packet sono stati consegnati nelle
585      * Locazioni */
586     while((totalSeconds = alarm(0)) >= 0 && sessionNumOfPackets > 0 && !finished){
587         if(totalSeconds == 0){
588             finished = true;
589         }
590
591         pthread_mutex_lock(&mutexTimer);
592         alarm(totalSeconds);
593         sprintf(msg, "$time %d", totalSeconds);
594         sendMsgToSession(NULL, msg);
595         pthread_mutex_unlock(&mutexTimer);
596         pthread_mutex_lock(&mutexCursor);
597         /* Calcolo minuti */
598         minutes = totalSeconds / 60;
599         /* Calcolo secondi */
600         seconds = totalSeconds % 60;
601         /* Muovo il cursore sulla posizione del timer e stampo il tempo ad ogni ciclo del
602          * while */
603         /* che cambia colore ad ogni range di secondi
604          *
605          */
606         moveto(timer.posX, timer.posY);
607         if(totalSeconds >= 300 && totalSeconds <= 600){
608             setColor(GRAPHICS_FG_COLOR_GREEN, 0);
609             printf("%.2d:%.2d", minutes, seconds);
610             reset();
611         } else if(totalSeconds >= 60 && totalSeconds <= 300){
612             setColor(GRAPHICS_FG_COLOR_YELLOW, 0);
613             printf("%.2d:%.2d", minutes, seconds);
614             reset();
615         } else{
616             if(totalSeconds <= 10 && totalSeconds%2 == 0){
617                 beep();
618                 setColor(GRAPHICS_FG_COLOR_RED, 0);
619                 printf("%.2d:%.2d", minutes, seconds);
620                 reset();
621             }
622         }
623     }
624 }
```

```

617         } else if(totalSeconds <= 10 && totalSeconds%2 != 0){
618             beep();
619             setColor(GRAPHICS_FG_COLOR_YELLOW, 0);
620             printf("%.2d:%.2d", minutes, seconds);
621             reset();
622         } else{
623             setColor(GRAPHICS_FG_COLOR_RED, 0);
624             printf("%.2d:%.2d", minutes, seconds);
625             reset();
626         }
627     }
628     fflush(stdout);
629     pthread_mutex_unlock(&mutexCursor);
630
631     sleep(1);
632 }
633 }
634
635
636 /**
637 * @param msg: arg -> Comando [$(dis)connected Username (packagesDelivered)].
638 *
639 * @return: puntatore a void.
640 *
641 * La funzione updatePlayersInfo aggiorna il counter
642 * e la lista dei players attualmente connessi alla
643 * sessione corrente.
644 *
645 */
646 void* updatePlayersInfo(void* arg){
647
648     char* msg = (char*)arg;
649     char* saveptr;
650     char* cmd = strtok_r(msg, " ", &saveptr);
651     char* username;
652     int packetsDelivered;
653     char buffer[GRAPHICS_TEXTFIELD_STRLEN];
654     int focusedPlayer = 0;
655
656     pthread_mutex_lock(&mutexUpdate);
657     if(!strcmp(cmd, "$disconnected") && playersCounter.value > 0){
658         username = strtok_r(NULL, " ", &saveptr);
659         playersCounter.value -= 1;
660         pthread_mutex_lock(&mutexCursor);
661         moveto(playersCounter.coords.posX, playersCounter.coords.posY);
662         printf("%d", playersCounter.value);
663         fflush(stdout);
664         pthread_mutex_unlock(&mutexCursor);
665
666         while(strcmp(username, playersInfo.playerBox[focusedPlayer].username) != 0){
667             focusedPlayer += 1;
668         }
669
670         for(int player = focusedPlayer; player < playersInfo.currPlayerPosition; player++){
671             strcpy(playersInfo.playerBox[player].username,
672             *playersInfo.playerBox[player+1].username);
673             playersInfo.playerBox[player].packetsDelivered =
674             playersInfo.playerBox[player+1].packetsDelivered;
675             ...
676         }
677     }

```

```

673     pthread_mutex_lock(&mutexCursor);
674     moveto(playersInfo.playerBox[player].coords.posX,
675             playersInfo.playerBox[player].coords.posY);
676     printf("           |   | ");
677     fflush(stdout);
678     memset(buffer, '\0', GRAPHICS_TEXTFIELD_STRLEN);
679     moveto(playersInfo.playerBox[player].coords.posX,
680             playersInfo.playerBox[player].coords.posY);
681     sprintf(buffer, " \u25ba %-13s[%.2d] ", playersInfo.playerBox[player].username,
682             playersInfo.playerBox[player].packetsDelivered);
683     printf("%s", buffer);
684     fflush(stdout);
685     pthread_mutex_unlock(&mutexCursor);
686 }
687
688 pthread_mutex_lock(&mutexCursor);
689 moveto(playersInfo.playerBox[playersInfo.currPlayerPosition].coords.posX,
690         playersInfo.playerBox[playersInfo.currPlayerPosition].coords.posY);
691 printf("           |   | ");
692 fflush(stdout);
693 playersInfo.currPlayerPosition -= 1;
694 pthread_mutex_unlock(&mutexCursor);
695
696 } else if(!strcmp(cmd, "$connected")){
697     username = strtok_r(NULL, " ", &saveptr);
698     packetsDelivered = atoi(strtok_r(NULL, " ", &saveptr));
699     memset(buffer, '\0', GRAPHICS_TEXTFIELD_STRLEN);
700     sprintf(buffer, " \u25ba %-13s[%.2d] ", username, packetsDelivered);
701     playersCounter.value += 1;
702     pthread_mutex_lock(&mutexCursor);
703     moveto(playersCounter.coords.posX, playersCounter.coords.posY);
704     printf("%d", playersCounter.value);
705     fflush(stdout);
706     pthread_mutex_unlock(&mutexCursor);
707     playersInfo.currPlayerPosition += 1;
708     strcpy(playersInfo.playerBox[playersInfo.currPlayerPosition].username, username);
709     playersInfo.playerBox[playersInfo.currPlayerPosition].packetsDelivered =
710     packetsDelivered;
711     pthread_mutex_lock(&mutexCursor);
712     moveto(playersInfo.playerBox[playersInfo.currPlayerPosition].coords.posX,
713             playersInfo.playerBox[playersInfo.currPlayerPosition].coords.posY);
714     printf("%s", buffer);
715     fflush(stdout);
716     pthread_mutex_unlock(&mutexCursor);
717 } else if(!strcmp(cmd, "$delivered")){
718     username = strtok_r(NULL, " ", &saveptr);
719
720     while(strcmp(username, playersInfo.playerBox[focusedPlayer].username) != 0){
721         focusedPlayer += 1;
722     }
723
724     playersInfo.playerBox[focusedPlayer].packetsDelivered += 1;
725     memset(buffer, '\0', GRAPHICS_TEXTFIELD_STRLEN);
726     sprintf(buffer, " \u25ba %-13s[%.2d] ", username,
727             playersInfo.playerBox[focusedPlayer].packetsDelivered);
728     pthread_mutex_lock(&mutexCursor);
729     moveto(playersInfo.playerBox[focusedPlayer].coords.posX,
730             playersInfo.playerBox[focusedPlayer].coords.posY);
731     printf("%s", buffer);
732     fflush(stdout);
733 }
```

```

723     printf(           |   | );
724     fflush(stdout);
725     moveto(playersInfo.playerBox[focusedPlayer].coords.posX,
726             *playersInfo.playerBox[focusedPlayer].coords.posY);
727     printf("%s", buffer);
728     fflush(stdout);
729     pthread_mutex_unlock(&mutexCursor);
730 }
731 pthread_mutex_unlock(&mutexUpdate);
732 free(arg);
733 }
734
735
736 /**
737 * @param clientInfo: puntatore della struttura che contiene informazioni
738 * del client.
739 *
740 * @return:void.
741 *
742 * La funzione sendBasicInformation si occupa di mandare a sendMsg
743 * Le informazioni riguardo alle posizioni dove stampare gli oggetti
744 * della mappa al client e alla stampa sulla grafica del server
745 * dell'oggetto del client entranto in game.
746 *
747 */
748 void sendBasicInformation(LpClientInfo clientInfo){
749
750     char msg[GRAPHICS_CHAT_WIDTH];
751     int spawnIndex = 0;
752
753     /* Mando al client le posizioni dei packet che poi stamperà graficamente sulla sua
754     * mappa */
755     for(int i=0; i<session->numOfPackets; i++){
756         if(!session->packets[i].delivered){
757             sprintf(msg, "$add packet %d %d", session->packets[i].currRows, session-
758                     >packets[i].currCols);
759             sendMsg(clientInfo, msg);
760         }
761     }
762
763     /* Mando al client le posizioni e il nome delle locazioni che poi stamperà
764     * graficamente sulla sua mappa */
765     for(int i=0; i<SESSION_LOCATIONS_STRLEN; i++){
766         sprintf(msg, "$add location %d %d %c", session->locations[i].currRows, session-
767                 >locations[i].currCols, session->locations[i].name);
768         sendMsg(clientInfo, msg);
769     }
770
771     /* Randomizzo gli spawn */
772     shuffleSpawn(session->spawn, SESSION_SPAWN_STRLEN);
773
774     /* Cerco uno spawn Libero */
775     while(spawnIndex < SESSION_SPAWN_STRLEN && session->field[session-
776             >spawn[spawnIndex].currRows][session->spawn[spawnIndex].currCols] !=
777             SESSION_GRASS_VALUE){
778         spawnIndex += 1;
779     }
780

```

```

775 /* Assegno alla posizione dello spawn presa il valore del player */
776 session->field[session->spawn[spawnIndex].currRows][session-
    * >spawn[spawnIndex].currCols] = SESSION_PLAYER_VALUE;
777
778 pthread_mutex_lock(&mutexClientInfo);
779 /* Assegno i valori della posizione di spawn del player nell'apposita struttura che
    * contiene le informazioni riguardo al client */
780 clientInfo->currRows = session->spawn[spawnIndex].currRows;
781 clientInfo->currCols = session->spawn[spawnIndex].currCols;
782 pthread_mutex_unlock(&mutexClientInfo);
783 /* Invio a tutti gli altri client la posizione del player appena entrato in modo da
    * aggiornare la mappa */
784 sprintf(msg, "$add player %d %d", session->spawn[spawnIndex].currRows, session-
    * >spawn[spawnIndex].currCols);
785 sendMsgToSession(clientInfo, msg);
786 pthread_mutex_lock(&mutexCursor);
787 moveto(field[session->spawn[spawnIndex].currRows][session-
    * >spawn[spawnIndex].currCols].posX, field[session-
    * >spawn[spawnIndex].currRows][session->spawn[spawnIndex].currCols].posY);
788 setColor(GRAPHICS_FG_COLOR_BLACK, GRAPHICS_BG_COLOR_GREEN);
789 printf("%s", SYMBOL_PLAYER);
790 reset();
791 pthread_mutex_unlock(&mutexCursor);
792 /* Mando al client in questione la sua posizione in modo che aggiorni la sua mappa */
793 sprintf(msg, "$add you %d %d", session->spawn[spawnIndex].currRows, session-
    * >spawn[spawnIndex].currCols);
794 sendMsg(clientInfo, msg);
795
796 /* Mando al client in questione la posizione di tutti i player in game per
    * aggiornare la sua mappa */
797 for(int i=0; i<SESSION_PLAYERS_STRLEN; i++){
798     if(session->clients[i] != NULL && session->clients[i] != clientInfo){
799         sprintf(msg, "$add player %d %d", session->clients[i]->currRows, session-
            * >clients[i]->currCols);
800         sendMsg(clientInfo, msg);
801     }
802 }
803 }
804
805
806 /**
807 * @param clientInfo: puntatore della struttura che contiene informazioni
808 * del client.
809 * @param incomingMsg: array che contiene il comando utilizzato dal client.
810 *
811 * @return: void.
812 *
813 * La funzione movePlayer si occupa di gestire i movimenti del client
814 * all'interno della mappa.
815 *
816 */
817 void movePlayer(LpClientInfo clientInfo, char incomingMsg[]){
818
819 LpClientInfo clientToJoin;
820 char msg[GRAPHICS_CHAT_WIDTH];
821
822 /* Verifico che il comando sia di tipo @S */
823 if(!strcmp(incomingMsg, "@S")){

```



```

870     }
871 }
872 /* Verifico che il comando sia di tipo @N */
873 } else if(!strcmp(incomingMsg, "@N")){
874     if(clientInfo->currRows-1 == -1){
875         sendMsg(clientInfo, "$Server: Attento, stavi per cadere giù!");
876     } else{
877         /* Se il comando è @N verifico cosa si trova nella posizione al suo nord */
878         switch(session->field[clientInfo->currRows-1][clientInfo->currCols]){
879             /* Se c'è dell'erba allora sposto il player di una posizione verso nord e
880             * invio il messaggio con le posizioni aggiornate ai client in sessione */
881             case SESSION_GRASS_VALUE:
882                 pthread_mutex_lock(&(session->mutexSession));
883                 session->field[clientInfo->currRows][clientInfo->currCols] =
884                     SESSION_GRASS_VALUE;
885                 session->field[clientInfo->currRows-1][clientInfo->currCols] =
886                     SESSION_PLAYER_VALUE;
887                 pthread_mutex_unlock(&(session->mutexSession));
888                 clientInfo->currRows -= 1;
889                 sprintf(msg, "$add you %d %d", clientInfo->currRows, clientInfo-
890                         >currCols);
891                 sendMsg(clientInfo, msg);
892                 sprintf(msg, "$add player %d %d", clientInfo->currRows, clientInfo-
893                         >currCols);
894                 sendMsgToSession(clientInfo, msg);
895                 sprintf(msg, "$update %d %d", clientInfo->currRows+1, clientInfo-
896                         >currCols);
897                 sendMsgToSession(NULL, msg);
898                 pthread_mutex_lock(&mutexCursor);
899                 moveto(field[clientInfo->currRows+1][clientInfo->currCols].posX,
900                         field[clientInfo->currRows+1][clientInfo->currCols].posY);
901                 setColor(GRAPHICS_FG_COLOR_GREEN, GRAPHICS_BG_COLOR_GREEN);
902                 printf("%s", SYMBOL_GRASS);
903                 moveto(field[clientInfo->currRows][clientInfo->currCols].posX,
904                         field[clientInfo->currRows][clientInfo->currCols].posY);
905                 setColor(GRAPHICS_FG_COLOR_BLACK, GRAPHICS_BG_COLOR_GREEN);
906                 printf("%s", SYMBOL_PLAYER);
907                 reset();
908                 pthread_mutex_unlock(&mutexCursor);
909             break;
910             /* Se c'è il muro invio al client il messaggio di presenza del muro , e
911             * delle coordinate del muro per poter aggiornare la sua mappa */
912             case SESSION_WALL_VALUE:
913                 sendMsg(clientInfo, "$Server: Ouch! Hai sbattuto contro un muro!");
914                 sprintf(msg, "$add wall %d %d", clientInfo->currRows-1, clientInfo-
915                         >currCols);
916                 sendMsg(clientInfo, msg);
917             break;
918             /* Se c'è una Locazione invio un messaggio di presenza di locazione */
919             case SESSION_LOCATION_VALUE:
920                 sendMsg(clientInfo, "$Server: Hei, hai un pacco per me?");
921             break;
922             /* Se c'è un packet invio un messaggio al client di presenza del packet */
923             case SESSION_PACKET_VALUE:
924                 sendMsg(clientInfo, "$Server: Wow! Hai trovato un pacco, prendilo!");
925             break;
926             /* Se c'è un player invio un messaggio al client della presenza di un player
927             */

```

```

917     case SESSION_PLAYER_VALUE:
918         sendMsg(clientInfo, "$Server: Alt! Un altro giocatore ti impedisce il
919             passaggio!");
920         break;
921     }
922     /* Verifico che il comando sia di tipo @E */
923 } else if(!strcmp(incomingMsg, "@E")){
924     if(clientInfo->currCols+1 == GRAPHICS_FIELD_WIDTH){
925         sendMsg(clientInfo, "$Server: Attento, stavi per cadere giù!");
926     } else{
927         /* Se il comando è @N verifico cosa si trova nella posizione al suo est */
928         switch(session->field[clientInfo->currRows][clientInfo->currCols+1]){
929             /* Se c'è dell'erba allora sposto il player di una posizione verso est e
930                 invio il messaggio con le posizioni aggiornate ai client in sessione */
931             case SESSION_GRASS_VALUE:
932                 pthread_mutex_lock(&(session->mutexSession));
933                 session->field[clientInfo->currRows][clientInfo->currCols] =
934                     SESSION_GRASS_VALUE;
935                 session->field[clientInfo->currRows][clientInfo->currCols+1] =
936                     SESSION_PLAYER_VALUE;
937                 pthread_mutex_unlock(&(session->mutexSession));
938                 clientInfo->currCols += 1;
939                 sprintf(msg, "$add you %d %d", clientInfo->currRows, clientInfo-
940                     >currCols);
941                 sendMsg(clientInfo, msg);
942                 sprintf(msg, "$add player %d %d", clientInfo->currRows, clientInfo-
943                     >currCols);
944                 sendMsgToSession(clientInfo, msg);
945                 sprintf(msg, "$update %d %d", clientInfo->currRows, clientInfo->currCols-
946                     1);
947                 sendMsgToSession(NULL, msg);
948                 pthread_mutex_lock(&mutexCursor);
949                 moveto(field[clientInfo->currRows][clientInfo->currCols-1].posX,
950                     field[clientInfo->currRows][clientInfo->currCols-1].posY);
951                 setColor(GRAPHICS_FG_COLOR_GREEN, GRAPHICS_BG_COLOR_GREEN);
952                 printf("%s", SYMBOL_GRASS);
953                 moveto(field[clientInfo->currRows][clientInfo->currCols].posX,
954                     field[clientInfo->currRows][clientInfo->currCols].posY);
955                 setColor(GRAPHICS_FG_COLOR_BLACK, GRAPHICS_BG_COLOR_GREEN);
956                 printf("%s", SYMBOL_PLAYER);
957                 reset();
958                 pthread_mutex_unlock(&mutexCursor);
959             break;
960             /* Se c'è il muro invio al client il messaggio di presenza del muro , e
961                 delle coordinate del muro per poter aggiornare la sua mappa */
962             case SESSION_WALL_VALUE:
963                 sendMsg(clientInfo, "$Server: Ouch! Hai sbattuto contro un muro!");
964                 sprintf(msg, "$add wall %d %d", clientInfo->currRows, clientInfo-
965                     >currCols+1);
966                 sendMsg(clientInfo, msg);
967             break;
968             /* Se c'è una locazione invio un messaggio di presenza di locazione */
969             case SESSION_LOCATION_VALUE:
970                 sendMsg(clientInfo, "$Server: Hei, hai un pacco per me?");
971             break;
972             /* Se c'è un packet invio un messaggio al client di presenza del packet */
973             case SESSION_PACKET_VALUE:

```

```

964         sendMsg(clientInfo, "$Server: Wow! Hai trovato un pacco, prendilo!");
965         break;
966         /* Se c'è un player invio un messaggio al client della presenza di un
967            player */
968         case SESSION_PLAYER_VALUE:
969             sendMsg(clientInfo, "$Server: Alt! Un altro giocatore ti impedisce il
970               passaggio!");
971             break;
972     }
973     /* Verifico che il comando sia di tipo @0 */
974 } else if(!strcmp(incomingMsg, "@0")){
975     if(clientInfo->currCols-1 == -1){
976         sendMsg(clientInfo, "$Server: Attento, stavi per cadere giù!");
977     } else{
978         /* Se il comando è @N verifico cosa si trova nella posizione al suo ovest */
979         switch(session->field[clientInfo->currRows][clientInfo->currCols-1]){
980             /* Se c'è dell'erba allora sposto il player di una posizione verso ovest e
981                invio il messaggio con le posizioni aggiornate ai client in sessione */
982             case SESSION_GRASS_VALUE:
983                 pthread_mutex_lock(&(session->mutexSession));
984                 session->field[clientInfo->currRows][clientInfo->currCols] =
985                     SESSION_GRASS_VALUE;
986                 session->field[clientInfo->currRows][clientInfo->currCols-1] =
987                     SESSION_PLAYER_VALUE;
988                 pthread_mutex_unlock(&(session->mutexSession));
989                 clientInfo->currCols -= 1;
990                 sprintf(msg, "$add you %d %d", clientInfo->currRows, clientInfo-
991                   >currCols);
992                 sendMsg(clientInfo, msg);
993                 sprintf(msg, "$add player %d %d", clientInfo->currRows, clientInfo-
994                   >currCols);
995                 sendMsgToSession(clientInfo, msg);
996                 sprintf(msg, "$update %d %d", clientInfo->currRows, clientInfo-
997                   >currCols+1);
998                 sendMsgToSession(NULL, msg);
999                 pthread_mutex_lock(&mutexCursor);
1000                 moveto(field[clientInfo->currRows][clientInfo->currCols+1].posX,
1001                   field[clientInfo->currRows][clientInfo->currCols+1].posY);
1002                 setColor(GRAPHICS_FG_COLOR_GREEN, GRAPHICS_BG_COLOR_GREEN);
1003                 printf("%s", SYMBOL_GRASS);
1004                 moveto(field[clientInfo->currRows][clientInfo->currCols].posX,
1005                   field[clientInfo->currRows][clientInfo->currCols].posY);
1006                 setColor(GRAPHICS_FG_COLOR_BLACK, GRAPHICS_BG_COLOR_GREEN);
1007                 printf("%s", SYMBOL_PLAYER);
1008                 reset();
1009                 pthread_mutex_unlock(&mutexCursor);
1010             break;
1011             /* Se c'è il muro invio al client il messaggio di presenza del muro , e
1012                delle coordinate del muro per poter aggiornare la sua mappa */
1013             case SESSION_WALL_VALUE:
1014                 sendMsg(clientInfo, "$Server: Ouch! Hai sbattuto contro un muro!");
1015                 sprintf(msg, "$add wall %d %d", clientInfo->currRows, clientInfo-
1016                   >currCols-1);
1017                 sendMsg(clientInfo, msg);
1018             break;
1019             /* Se c'è una Locazione invio un messaggio di presenza di locazione */
1020             case SESSION_LOCATION_VALUE:

```

```

1010         sendMsg(clientInfo, "$Server: Hei, hai un pacco per me?");  

1011     break;  

1012     /* Se c'è un packet invio un messaggio al client di presenza del packet */  

1013     case SESSION_PACKET_VALUE:  

1014         sendMsg(clientInfo, "$Server: Wow! Hai trovato un pacco, prendilo!");  

1015     break;  

1016     /* Se c'è un player invio un messaggio al client della presenza di un  

1017      * player */  

1018     case SESSION_PLAYER_VALUE:  

1019         sendMsg(clientInfo, "$Server: Alt! Un altro giocatore ti impedisce il  

1020           passaggio!");  

1021     break;  

1022     }  

1023 }  

1024  

1025  

1026 /**
1027 * @param clientInfo: puntatore della struttura che contiene informazioni  

1028 * del client.  

1029 * @param incomingMsg: array che contiene il comando utilizzato dal client.  

1030 *  

1031 * @return: void.  

1032 *  

1033 * La funzione actionPlayer si occupa di gestire il comando di presa e  

1034 * di deposito del pacco da parte del client .  

1035 *  

1036 */  

1037 void actionPlayer(LpClientInfo clientInfo, char incomingMsg[]){  

1038  

1039     LpClientInfo clientToJoin;  

1040     char msg[GRAPHICS_CHAT_WIDTH];  

1041     int currRows;  

1042     int currCols;  

1043     char locationName;  

1044     int location;  

1045     char* dynamicBuffer;  

1046     pthread_t tidUpdatePlayersInfo;  

1047     time_t timestamp;  

1048     char logsBuffer[BUFFER_STRLEN];  

1049     pthread_t tidUpdateChat;  

1050     LpMsg msgChat;  

1051  

1052     /* Controllo se il comando è di tipo P */  

1053     if(incomingMsg[1] == 'P'){

1054         switch (incomingMsg[2]) {
1055             /* Se la posizione dove prendere è S */
1056             case 'S':
1057                 /*Se non è presente un pacco nella posizione verso sud allora invio un
1058                  * messaggio al client della non presenza del packet */
1059                 if(clientInfo->currRows+1 == GRAPHICS_FIELD_HEIGHT || session-
1060                   >field[clientInfo->currRows+1][clientInfo->currCols] != SESSION_PACKET_VALUE){
1061                     sendMsg(clientInfo, "$Server: Non c'è nulla qui!");
1062                 } else{
1063                     if(session->field[clientInfo->currRows+1][clientInfo->currCols] ==
1064                       SESSION_PACKET_VALUE){

```

```

1063     /* Verifico se il client è già in possesso di un packet se si invio un
1064      * messaggio di segnalazione */
1065     if(clientInfo->havePacket != -1){
1066         sendMsg(clientInfo, "$Server: Non essere avido! Hai già un
1067          * pacchetto.");
1068     } else{
1069         /* Altrimenti se non possiede packet aggiorno il valore ad erba e
1070          * invio un messaggio ai client per aggiornare la mappa */
1071         pthread_mutex_lock(&(session->mutexSession));
1072         session->field[clientInfo->currRows+1][clientInfo->currCols] =
1073             SESSION_GRASS_VALUE;
1074         pthread_mutex_unlock(&(session->mutexSession));
1075         sprintf(msg, "$update %d %d", clientInfo->currRows+1, clientInfo-
1076           >currCols);
1077         sendMsgToSession(NULL, msg);
1078         pthread_mutex_lock(&mutexCursor);
1079         moveto(field[clientInfo->currRows+1][clientInfo->currCols].posX,
1080           field[clientInfo->currRows+1][clientInfo->currCols].posY);
1081         setColor(GRAPHICS_FG_COLOR_GREEN, GRAPHICS_BG_COLOR_GREEN);
1082         printf("%s", SYMBOL_GRASS);
1083         reset();
1084         pthread_mutex_unlock(&mutexCursor);
1085
1086         /* Ricerca la locazione dove portare il packet preso dal client */
1087         for(int i=0; i<session->numOfPackets; i++){
1088             if(session->packets[i].currRows == clientInfo->currRows+1 &&
1089               session->packets[i].currCols == clientInfo->currCols){
1090                 clientInfo->havePacket = i;
1091                 session->packets[i].delivered = true;
1092                 currRows = session->locations[session-
1093                   >packets[i].location].currRows;
1094                 currCols = session->locations[session-
1095                   >packets[i].location].currCols;
1096                 locationName = session->locations[session-
1097                   >packets[i].location].name;
1098                 break;
1099             }
1100         }
1101
1102         /* Invio le informazioni riguardo alla locazione al client per
1103          * permettere di aggiornare la mappa */
1104         sprintf(msg, "$add blink %d %d %c", currRows, currCols,
1105           locationName);
1106         sendMsg(clientInfo, msg);
1107         sprintf(msg, "$Server: Trasporta il pacchetto fino alla locazione
1108           %c!", locationName);
1109         sendMsg(clientInfo, msg);
1110     }
1111     break;
1112
1113     /* Se la posizione dove prendere è N */
1114     case 'N':
1115         /* Se non è presente un pacco nella posizione verso sud allora invio un
1116          * messaggio al client della non presenza del packet */
1117         if(clientInfo->currRows-1 == -1 || session->field[clientInfo->currRows-
1118           1][clientInfo->currCols] != SESSION_PACKET_VALUE){
1119             sendMsg(clientInfo, "$Server: Non c'è nulla qui!");
1120         }

```

```

1186     } else{
1187         if(session->field[clientInfo->currRows-1][clientInfo->currCols] ==
1188             * SESSION_PACKET_VALUE){
1189             /* Verifico se il client è già in possesso di un packet se si invio un
1190                * messaggio di segnalazione */
1191             if(clientInfo->havePacket != -1){
1192                 sendMsg(clientInfo, "$Server: Non essere avido! Hai già un
1193                 * pacchetto.");
1194             } else{
1195                 /* Altrimenti se non possiede packet aggiorno il valore ad erba e
1196                    invio un messaggio al client per aggiornare la mappa */
1197                 pthread_mutex_lock(&(session->mutexSession));
1198                 session->field[clientInfo->currRows-1][clientInfo->currCols] =
1199                     * SESSION_GRASS_VALUE;
1200                 pthread_mutex_unlock(&(session->mutexSession));
1201                 sprintf(msg, "$update %d %d", clientInfo->currRows-1, clientInfo-
1202                     >currCols);
1203                 sendMsgToSession(NULL, msg);
1204                 pthread_mutex_lock(&mutexCursor);
1205                 moveto(field[clientInfo->currRows-1][clientInfo->currCols].posX,
1206                     field[clientInfo->currRows-1][clientInfo->currCols].posY);
1207                 setColor(GRAPHICS_FG_COLOR_GREEN, GRAPHICS_BG_COLOR_GREEN);
1208                 printf("%s", SYMBOL_GRASS);
1209                 reset();
1210                 pthread_mutex_unlock(&mutexCursor);
1211
1212
1213             /* Ricerca la locazione dove portare il packet preso dal client */
1214             for(int i=0; i<session->numOfPackets; i++){
1215                 if(session->packets[i].currRows == clientInfo->currRows-1 && session-
1216                     >packets[i].currCols == clientInfo->currCols){
1217                     clientInfo->havePacket = i;
1218                     session->packets[i].delivered = true;
1219                     currRows = session->locations[session-
1220                         >packets[i].location].currRows;
1221                     currCols = session->locations[session-
1222                         >packets[i].location].currCols;
1223                     locationName = session->locations[session-
1224                         >packets[i].location].name;
1225                     break;
1226                 }
1227             }
1228
1229
1230             /* Invio le informazioni riguardo alla locazione al client per
1231                permettere di aggiornare la mappa */
1232             sprintf(msg, "$add blink %d %d %c", currRows, currCols, locationName);
1233             sendMsg(clientInfo, msg);
1234             sprintf(msg, "$Server: Trasporta il pacchetto fino alla locazione
1235                 %c!", locationName);
1236             sendMsg(clientInfo, msg);
1237         }
1238     }
1239
1240     */
1241     break;
1242
1243     /*
1244     */
1245     break;
1246
1247     /* Se la posizione dove prendere è E */
1248     case 'E':
1249         /*Se non è presente un pacco nella posizione verso sud allora invio un
1250            * messaggio al client della non presenza del packet */
1251         if(clientInfo->currCols+1 == GRAPHICS_FIELD_WIDTH || session->field[clientInfo-
1252             >currRows][clientInfo->currCols+1] == SESSION_PACKET_VALUE)

```

```

*
>currRows][clientInfo->currCols+1] != SESSION_PACKET_VALUE){\n
    sendMsg(clientInfo, "$Server: Non c'è nulla qui!");\n
} else{\n
    if(session->field[clientInfo->currRows][clientInfo->currCols+1] ==\n
        SESSION_PACKET_VALUE){\n
            /* Verifico se il client è già in possesso di un packet se si invio un\n
            messaggio di segnalazione */\n
            if(clientInfo->havePacket != -1){\n
                sendMsg(clientInfo, "$Server: Non essere avido! Hai già un\n
                pacchetto.");\n
            } else{\n
                /* Altrimenti se non possiede packet aggiorno il valore ad erba e\n
                invio un messaggio al client per aggiornare la mappa */\n
                pthread_mutex_lock(&(session->mutexSession));\n
                session->field[clientInfo->currRows][clientInfo->currCols+1] =\n
                    SESSION_GRASS_VALUE;\n
                pthread_mutex_unlock(&(session->mutexSession));\n
                sprintf(msg, "$update %d %d", clientInfo->currRows, clientInfo-\n
                    >currCols+1);\n
                sendMsgToSession(NULL, msg);\n
                pthread_mutex_lock(&mutexCursor);\n
                moveto(field[clientInfo->currRows][clientInfo->currCols+1].posX,\n
                    field[clientInfo->currRows][clientInfo->currCols+1].posY);\n
                setColor(GRAPHICS_FG_COLOR_GREEN, GRAPHICS_BG_COLOR_GREEN);\n
                printf("%s", SYMBOL_GRASS);\n
                reset();\n
                pthread_mutex_unlock(&mutexCursor);\n
\n
                /* Ricerco la locazione dove portare il packet preso dal client */\n
                for(int i=0; i<session->numOfPackets; i++){\n
                    if(session->packets[i].currRows == clientInfo->currRows && session-\n
                        >packets[i].currCols == clientInfo->currCols+1){\n
                        clientInfo->havePacket = i;\n
                        session->packets[i].delivered = true;\n
                        currRows = session->locations[session-\n
                            >packets[i].location].currRows;\n
                        currCols = session->locations[session-\n
                            >packets[i].location].currCols;\n
                        locationName = session->locations[session-\n
                            >packets[i].location].name;\n
                        break;\n
                    }\n
                }\n
\n
                /* Invio le informazioni riguardo alla locazione al client per\n
                permettere di aggiornare la mappa */\n
                sprintf(msg, "$add blink %d %d %c", currRows, currCols,\n
                    locationName);\n
                sendMsg(clientInfo, msg);\n
                sprintf(msg, "$Server: Trasporta il pacchetto fino alla locazione\n
                    %c!", locationName);\n
                sendMsg(clientInfo, msg);\n
            }\n
        }\n
    }\n
}\n
break;\n
/* Se la posizione dove prendere è 0 */\ncase '0':\n
    /* Se non è presente un pacchetto nella posizione verso sud allora invio un

```

```

1191     /* Se non c'è presente un packet nella posizione verso sua destra invio un
1192      messaggio al client della non presenza del packet */
1193     if(clientInfo->currCols-1 == -1 || session->field[clientInfo-
1194 >currRows][clientInfo->currCols-1] != SESSION_PACKET_VALUE){
1195         sendMsg(clientInfo, "$Server: Non c'è nulla qui!");
1196     } else{
1197         if(session->field[clientInfo->currRows][clientInfo->currCols-1] ==
1198 SESSION_PACKET_VALUE){
1199             /* Verifico se il client è già in possesso di un packet se si invia un
1200            messaggio di segnalazione */
1201             if(clientInfo->havePacket != -1){
1202                 sendMsg(clientInfo, "$Server: Non essere avido! Hai già un pacchetto.");
1203             } else{
1204                 /* Altrimenti se non possiede packet aggiorno il valore ad erba e
1205                invio un messaggio ai client per aggiornare La mappa */
1206                 pthread_mutex_lock(&(session->mutexSession));
1207                 session->field[clientInfo->currRows][clientInfo->currCols-1] =
1208 SESSION_GRASS_VALUE;
1209                 pthread_mutex_unlock(&(session->mutexSession));
1210                 sprintf(msg, "$update %d %d", clientInfo->currRows, clientInfo-
1211 >currCols-1);
1212                 sendMsgToSession(NULL, msg);
1213                 pthread_mutex_lock(&mutexCursor);
1214                 moveto(field[clientInfo->currRows][clientInfo->currCols-1].posX,
1215 field[clientInfo->currRows][clientInfo->currCols-1].posY);
1216                 setColor(GRAPHICS_FG_COLOR_GREEN, GRAPHICS_BG_COLOR_GREEN);
1217                 printf("%s", SYMBOL_GRASS);
1218                 reset();
1219                 pthread_mutex_unlock(&mutexCursor);

1220             /* Ricerca La Locazione dove portare il packet preso dal client */
1221             for(int i=0; i<session->numOfPackets; i++){
1222                 if(session->packets[i].currRows == clientInfo->currRows && session-
1223 >packets[i].currCols == clientInfo->currCols-1){
1224                     clientInfo->havePacket = i;
1225                     session->packets[i].delivered = true;
1226                     currRows = session->locations[session-
1227 >packets[i].location].currRows;
1228                     currCols = session->locations[session-
1229 >packets[i].location].currCols;
1230                     locationName = session->locations[session-
1231 >packets[i].location].name;
1232                     break;
1233                 }
1234             }
1235             /* Invio Le informazioni riguardo alla Locazione al client per
1236            permettere di aggiornare La mappa */
1237             sprintf(msg, "$add blink %d %d %c", currRows, currCols, locationName);
1238             sendMsg(clientInfo, msg);
1239             sprintf(msg, "$Server: Trasporta il pacchetto fino alla locazione
1240 %c!", locationName);
1241             sendMsg(clientInfo, msg);
1242         }
1243     }
1244     break;
1245 }
1246 /* Verifico se il comando è di tipo D */

```

```

1238 } else if(incomingMsg[1] == 'D'){
1239     /* Se la posizione dove prendere è S */
1240     switch (incomingMsg[2]) {
1241         case 'S':
1242             /* Verifico che il client sia in possesso di un packet, in caso negativo
1243                 invio al client un messaggio di segnalazione */
1244             if(clientInfo->havePacket == -1){
1245                 sendMsg(clientInfo, "$Server: Devi prima raccogliere un pacco!");
1246                 /* Se possiede un pacco verifico se nella posizione verso sud sia presente
1247                     un muro, in caso di esito positivo invio al client un messaggio di
1248                     segnalazione */
1249             } else if(clientInfo->currRows+1 == GRAPHICS_FIELD_HEIGHT || (session-
1250 >field[clientInfo->currRows+1][clientInfo->currCols] != SESSION_GRASS_VALUE
1251     && session->field[clientInfo->currRows+1][clientInfo->currCols] !=
1252     SESSION_LOCATION_VALUE)){
1253         if(session->field[clientInfo->currRows+1][clientInfo->currCols] ==
1254             SESSION_WALL_VALUE){
1255             sendMsg(clientInfo, "$Server: Non vorrai mica distruggere un muro con
1256                 un pacchetto?!");
1257             sprintf(msg, "$add wall %d %d", clientInfo->currRows+1, clientInfo-
1258 >currCols);
1259             sendMsg(clientInfo, msg);
1260             /* Se possiede un pacco verifico se nella posizione verso sud sia
1261                 presente un player, in caso di esito positivo invio al client un
1262                 messaggio di segnalazione */
1263         } else if(session->field[clientInfo->currRows+1][clientInfo->currCols]
1264             == SESSION_PLAYER_VALUE){
1265             sendMsg(clientInfo, "$Server: Non puoi lanciare pacchetti contro gli
1266                 altri giocatori!");
1267         } else{
1268             sendMsg(clientInfo, "$Server: Non puoi posizionare il pacco qui!");
1269         }
1270     } else{
1271         /* Verifico se nella posizione verso sud sia presente dell'erba , in caso
1272             di esito invio un messaggio al client delle nuove posizione per poter
1273             aggiornare la mappa */
1274         if(session->field[clientInfo->currRows+1][clientInfo->currCols] ==
1275             SESSION_GRASS_VALUE){
1276             pthread_mutex_lock(&(session->mutexSession));
1277             session->field[clientInfo->currRows+1][clientInfo->currCols] =
1278                 SESSION_PACKET_VALUE;
1279             session->packets[clientInfo->havePacket].currRows = clientInfo-
1280 >currRows+1;
1281             session->packets[clientInfo->havePacket].currCols = clientInfo-
1282 >currCols;
1283             pthread_mutex_unlock(&(session->mutexSession));
1284             clientInfo->havePacket = -1;
1285             sendMsg(clientInfo, "$remove blink");
1286             sprintf(msg, "$add packet %d %d", clientInfo->currRows+1, clientInfo-
1287 >currCols);
1288             sendMsgToSession(NULL, msg);
1289             pthread_mutex_lock(&mutexCursor);
1290             moveto(field[clientInfo->currRows+1][clientInfo->currCols].posX,
1291                 field[clientInfo->currRows+1][clientInfo->currCols].posY);
1292             setColor(GRAPHICS_FG_COLOR_RED, GRAPHICS_BG_COLOR_GREEN);
1293             printf("%s", SYMBOL_PACKET);
1294             reset();
1295             pthread_mutex_unlock(&mutexCursor);
1296         }
1297     }
1298 }

```

```

1276     * - -
1277     /* Verifico se nella posizione verso sud sia presente una locazione */
1278 } else if(session->field[clientInfo->currRows+1][clientInfo->currCols] ==
1279     * SESSION_LOCATION_VALUE){
1280     for(int i=0; i<SESSION_LOCATIONS_STRLEN; i++){
1281         if(session->locations[i].currRows == clientInfo->currRows+1 && session-
1282             * locations[i].currCols == clientInfo->currCols){
1283                 location = i;
1284                 break;
1285             }
1286     }
1287     /* Controllo se la locazione è quella indicata dal packet, in caso di
1288        esito negativo invio al client un messaggio di segnalazione */
1289     if(session->packets[clientInfo->havePacket].location != location){
1290         sendMsg(clientInfo, "$Server: Attento! Hai sbagliato locazione,
1291             * riprova!");
1292     } else{
1293         /* In caso di esito positivo aggiorno il numero di pacchi consegnati
1294            dal client e invio un messaggio alla sessione di avvenuta consegna */
1295         clientInfo->packetsDelivered += 1;
1296         clientInfo->havePacket = -1;
1297         sendMsg(clientInfo, "$remove blink");
1298         sprintf(msg, "$delivered %s", clientInfo->username);
1299         dynamicBuffer = (char*)calloc(GRAPHICS_CHAT_WIDTH, sizeof(char));
1300         strcpy(dynamicBuffer, msg);
1301         pthread_create(&tidUpdatePlayersInfo, NULL, updatePlayersInfo,
1302             dynamicBuffer);
1303         sessionNumOfPackets -= 1;
1304         sendMsgToSession(NULL, msg);
1305         sprintf(msg, "$Server: %s ha consegnato un pacco, complimenti!",
1306             clientInfo->username);
1307         sendMsgToSession(NULL, msg);
1308         timestamp = time(NULL);
1309         pthread_mutex_lock(&mutexLogs);
1310         sprintf(logsBuffer, " > [Sessione %d] Pacco consegnato da %s - %s",
1311             idSession, clientInfo->username, ctime(&timestamp));
1312         if(write(logs, logsBuffer, strlen(logsBuffer)) == -1){
1313             if((msgChat = allocMsg("Errore durante la scrittura nel file di
1314             * logs.", GRAPHICS_FG_COLOR_RED, true)) != NULL){
1315                 pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
1316             }
1317         }
1318         pthread_mutex_unlock(&mutexLogs);
1319     }
1320     break;
1321     /* Se la posizione dove prendere è N */
1322 case 'N':
1323     /* Verifico che il client sia in possesso di un packet, in caso negativo
1324        invio un messaggio di segnalazione */
1325     if(clientInfo->havePacket == -1){
1326         sendMsg(clientInfo, "$Server: Devi prima raccogliere un pacco!");
1327         /* Se possiede un pacco verifico se nella posizione verso sud sia presente
1328            un muro, in caso di esito positivo invio al client un messaggio di
1329            segnalazione */
1330     } else if(clientInfo->currRows-1 == -1 || (session->field[clientInfo-
1331             >currRows-1][clientInfo->currCols] != SESSION_GRASS_VALUE
1332             && session->field[clientInfo->currRows-1][clientInfo->currCols] !=
```

```

        SESSION_LOCATION_VALUE)) {
1321     if(session->field[clientInfo->currRows-1][clientInfo->currCols] ==
1322         SESSION_WALL_VALUE){
1323         sendMsg(clientInfo, "$Server: Non vorrai mica distruggere un muro con
1324             un pacchetto?!");
1325         sprintf(msg, "$add wall %d %d", clientInfo->currRows-1, clientInfo-
1326             >currCols);
1327         sendMsg(clientInfo, msg);
1328         /* Se possiede un pacco verifico se nella posizione verso sud sia
1329             presente un player, in caso di esito positivo invio al client un
1330             messaggio di segnalazione */
1331     } else if(session->field[clientInfo->currRows-1][clientInfo->currCols] ==
1332         SESSION_PLAYER_VALUE){
1333         sendMsg(clientInfo, "$Server: Non puoi lanciare pacchetti contro gli
1334             altri giocatori!");
1335     } else{
1336         sendMsg(clientInfo, "$Server: Non puoi posizionare il pacco qui!");
1337     }
1338 } else{
1339     /* Verifico se nella posizione verso sud sia presente dell'erba , in caso
1340         di esito invio un messaggio al client delle nuove posizione per poter
1341         aggiornare la mappa */
1342     if(session->field[clientInfo->currRows-1][clientInfo->currCols] ==
1343         SESSION_GRASS_VALUE){
1344         pthread_mutex_lock(&(session->mutexSession));
1345         session->field[clientInfo->currRows-1][clientInfo->currCols] =
1346             SESSION_PACKET_VALUE;
1347         session->packets[clientInfo->havePacket].currRows = clientInfo-
1348             >currRows-1;
1349         session->packets[clientInfo->havePacket].currCols = clientInfo-
1350             >currCols;
1351         pthread_mutex_unlock(&(session->mutexSession));
1352         clientInfo->havePacket = -1;
1353         sendMsg(clientInfo, "$remove blink");
1354         sprintf(msg, "$add packet %d %d", clientInfo->currRows-1, clientInfo-
1355             >currCols);
1356         sendMsgToSession(NULL, msg);
1357         pthread_mutex_lock(&mutexCursor);
1358         moveto(field[clientInfo->currRows-1][clientInfo->currCols].posX,
1359             field[clientInfo->currRows-1][clientInfo->currCols].posY);
1360         setColor(GRAPHICS_FG_COLOR_RED, GRAPHICS_BG_COLOR_GREEN);
1361         printf("%s", SYMBOL_PACKET);
1362         reset();
1363         pthread_mutex_unlock(&mutexCursor);
1364         /* Verifico se nella posizione verso sud sia presente una locazione */
1365     } else if(session->field[clientInfo->currRows-1][clientInfo->currCols] ==
1366         SESSION_LOCATION_VALUE){
1367         for(int i=0; i<SESSION_LOCATIONS_STRLEN; i++){
1368             if(session->locations[i].currRows == clientInfo->currRows-1 &&
1369                 session->locations[i].currCols == clientInfo->currCols){
1370                 location = i;
1371                 break;
1372             }
1373         }
1374         /* Controllo se la locazione è quella indicata dal packet, in caso di
1375             esito negativo invio al client un messaggio di segnalazione */
1376         if(session->packets[clientInfo->havePacket].location != location){
1377             sendMsg(clientInfo, "$Server: Attento! Hai sbagliato locazione,

```

```

        *
        riprova!");
1360
1361     /* In caso di esito positivo aggiorno il numero di pacchi
        consegnati dal client e invio un messaggio alla sessione di
        avvenuta consegna */
1362     clientInfo->packetsDelivered += 1;
1363     clientInfo->havePacket = -1;
1364     sendMsg(clientInfo, "$remove blink");
1365     sprintf(msg, "$delivered %s", clientInfo->username);
1366     dynamicBuffer = (char*)calloc(GRAPHICS_CHAT_WIDTH, sizeof(char));
1367     strcpy(dynamicBuffer, msg);
1368     pthread_create(&tidUpdatePlayersInfo, NULL, updatePlayersInfo,
        dynamicBuffer);
1369     sessionNumOfPackets -= 1;
1370     sendMsgToSession(NULL, msg);
1371     sprintf(msg, "$Server: %s ha consegnato un pacco, complimenti!",
        clientInfo->username);
1372     sendMsgToSession(NULL, msg);
1373     timestamp = time(NULL);
1374     pthread_mutex_lock(&mutexLogs);
1375     sprintf(logsBuffer, " > [Sessione %d] Pacco consegnato da %s -
        %s", idSession, clientInfo->username, ctime(&timestamp));
1376     if(write(logs, logsBuffer, strlen(logsBuffer)) == -1){
1377         if((msgChat = allocMsg("Errore durante la scrittura nel file di
        logs.", GRAPHICS_FG_COLOR_RED, true)) != NULL){
1378             pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
1379         }
1380     }
1381     pthread_mutex_unlock(&mutexLogs);
1382 }
1383 }
1384 }
1385 break;
/* Se la posizione dove prendere è E */
1386 case 'E':
1387     /* Verifico che il client sia in possesso di un packet, in caso negativo invio
        un messaggio di segnalazione */
1388     if(clientInfo->havePacket == -1){
1389         sendMsg(clientInfo, "$Server: Devi prima raccogliere un pacco!");
1390         /* Se possiede un pacco verifico se nella posizione verso sud sia presente
            un muro, in caso di esito positivo invio al client un messaggio di
            segnalazione */
1391     } else if(clientInfo->currCols+1 == GRAPHICS_FIELD_WIDTH || (session-
        * >field[clientInfo->currRows][clientInfo->currCols+1] != SESSION_GRASS_VALUE
        * && session->field[clientInfo->currRows][clientInfo->currCols+1] != SESSION_LOCATION_VALUE)){
1392         if(session->field[clientInfo->currRows][clientInfo->currCols+1] ==
        * SESSION_WALL_VALUE){
1393             sendMsg(clientInfo, "$Server: Non vorrai mica distruggere un muro con un
        * pacchetto?!");
1394             sprintf(msg, "$add wall %d %d", clientInfo->currRows, clientInfo-
        * >currCols+1);
1395             sendMsg(clientInfo, msg);
1396             /* Se possiede un pacco verifico se nella posizione verso sud sia
                presente un player, in caso di esito positivo invio al client un
                messaggio di segnalazione */
1397         } else if(session->field[clientInfo->currRows][clientInfo->currCols+1] ==
        * SESSION_PLAYER_VALUE){

```

```

1400             sendMsg(clientInfo, "$Server: Non puoi lanciare pacchetti contro gli
1401             altri giocatori!");
1402         } else{
1403             sendMsg(clientInfo, "$Server: Non puoi posizionare il pacco qui!");
1404         }
1405     } else{
1406         /* Verifico se nella posizione verso sud sia presente dell'erba , in caso
1407         di esito invio un messaggio al client delle nuove posizione per poter
1408         aggiornare la mappa */
1409         if(session->field[clientInfo->currRows][clientInfo->currCols+1] ==
1410             SESSION_GRASS_VALUE){
1411             pthread_mutex_lock(&(session->mutexSession));
1412             session->field[clientInfo->currRows][clientInfo->currCols+1] =
1413                 SESSION_PACKET_VALUE;
1414             session->packets[clientInfo->havePacket].currRows = clientInfo->currRows;
1415             session->packets[clientInfo->havePacket].currCols = clientInfo-
1416                 >currCols+1;
1417             pthread_mutex_unlock(&(session->mutexSession));
1418             clientInfo->havePacket = -1;
1419             sendMsg(clientInfo, "$remove blink");
1420             sprintf(msg, "$add packet %d %d", clientInfo->currRows, clientInfo-
1421                 >currCols+1);
1422             pthread_mutex_lock(&mutexCursor);
1423             moveto(field[clientInfo->currRows][clientInfo->currCols+1].posX,
1424                 field[clientInfo->currRows][clientInfo->currCols+1].posY);
1425             setColor(GRAPHICS_FG_COLOR_RED, GRAPHICS_BG_COLOR_GREEN);
1426             printf("%s", SYMBOL_PACKET);
1427             reset();
1428             pthread_mutex_unlock(&mutexCursor);
1429             sendMsgToSession(NULL, msg);
1430             /* Verifico se nella posizione verso sud sia presente una Locazione */
1431         } else if(session->field[clientInfo->currRows][clientInfo->currCols+1] ==
1432             SESSION_LOCATION_VALUE){
1433             for(int i=0; i<SESSION_LOCATIONS_STRLEN; i++){
1434                 if(session->locations[i].currRows == clientInfo->currRows && session-
1435                     >locations[i].currCols == clientInfo->currCols+1){
1436                     location = i;
1437                     break;
1438                 }
1439             }
1440             /* Controllo se la locazione è quella indicata dal packet, in caso di
1441             esito negativo invio al client un messaggio di segnalazione */
1442             if(session->packets[clientInfo->havePacket].location != location){
1443                 sendMsg(clientInfo, "$Server: Attenzione! Hai sbagliato locazione,
1444                 riprova!");
1445             } else{
1446                 /* In caso di esito positivo aggiorno il numero di pacchi
1447                 consegnati dal client e invio un messaggio alla sessione di
1448                 avvenuta consegna */
1449                 clientInfo->packetsDelivered += 1;
1450                 clientInfo->havePacket = -1;
1451                 sendMsg(clientInfo, "$remove blink");
1452                 sprintf(msg, "$delivered %s", clientInfo->username);
1453                 dynamicBuffer = (char*)calloc(GRAPHICS_CHAT_WIDTH, sizeof(char));
1454                 strcpy(dynamicBuffer, msg);
1455                 pthread_create(&tidUpdatePlayersInfo, NULL, updatePlayersInfo,
1456                     dynamicBuffer);
1457                 sessionNumOfPackets -= 1;

```

```

1443             sendMsgToSession(NULL, msg);
1444             sprintf(msg, "$Server: %s ha consegnato un pacco, complimenti!",
1445                     clientInfo->username);
1446             sendMsgToSession(NULL, msg);
1447             timestamp = time(NULL);
1448             pthread_mutex_lock(&mutexLogs);
1449             sprintf(logsBuffer, " > [Sessione %d] Pacco consegnato da %s -
1450             %s", idSession, clientInfo->username, ctime(&timestamp));
1451             if(write(logs, logsBuffer, strlen(logsBuffer)) == -1){
1452                 if((msgChat = allocMsg("Errore durante la scrittura nel file di
1453                 logs.", GRAPHICS_FG_COLOR_RED, true)) != NULL){
1454                     pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
1455                 }
1456             }
1457         }
1458     break;
1459 /* Se la posizione dove prendere è 0 */
1460 case '0':
1461     /* Verifico che il client sia in possesso di un packet, in caso negativo invio
1462     un messaggio di segnalazione */
1463     if(clientInfo->havePacket == -1){
1464         sendMsg(clientInfo, "$Server: Devi prima raccogliere un pacco!");
1465         /* Se possiede un pacco verifico se nella posizione verso sud sia presente
1466         un muro, in caso di esito positivo invio al client un messaggio di
1467         segnalazione */
1468     } else if(clientInfo->currCols-1 == -1 || (session->field[clientInfo-
1469     >currRows][clientInfo->currCols-1] != SESSION_GRASS_VALUE
1470     && session->field[clientInfo->currRows][clientInfo->currCols-1] != SESSION_LOCATION_VALUE)){
1471         if(session->field[clientInfo->currRows][clientInfo->currCols-1] ==
1472             SESSION_WALL_VALUE){
1473             sendMsg(clientInfo, "$Server: Non vorrai mica distruggere un muro con un
1474             pacchetto?!");
1475             sprintf(msg, "$add wall %d %d", clientInfo->currRows, clientInfo-
1476             >currCols-1);
1477             sendMsg(clientInfo, msg);
1478             /* Se possiede un pacco verifico se nella posizione verso sud sia
1479             presente un player, in caso di esito positivo invio al client un
1480             messaggio di segnalazione */
1481         } else if(session->field[clientInfo->currRows][clientInfo->currCols-1] ==
1482             SESSION_PLAYER_VALUE){
1483             sendMsg(clientInfo, "$Server: Non puoi lanciare pacchetti contro gli
1484             altri giocatori!");
1485         } else{
1486             sendMsg(clientInfo, "$Server: Non puoi posizionare il pacco qui!");
1487         }
1488     } else{
1489         /* Verifico se nella posizione verso sud sia presente dell'erba , in caso
1490         di esito invio un messaggio al client delle nuove posizione per poter
1491         aggiornare la mappa */
1492         if(session->field[clientInfo->currRows][clientInfo->currCols-1] ==
1493             SESSION_GRASS_VALUE){
1494             pthread_mutex_lock(&(session->mutexSession));
1495             session->field[clientInfo->currRows][clientInfo->currCols-1] =
1496             SESSION_PACKET_VALUE;

```

```

1482 session->packets[clientInfo->havePacket].currRows = clientInfo->currRows;
1483 session->packets[clientInfo->havePacket].currCols = clientInfo->currCols-
1484 +
1485 pthread_mutex_unlock(&(session->mutexSession));
1486 clientInfo->havePacket = -1;
1487 sendMsg(clientInfo, "$remove blink");
1488 sprintf(msg, "$add packet %d %d", clientInfo->currRows, clientInfo-
1489 +
1490 >currCols-1);
1491 sendMsgToSession(NULL, msg);
1492 pthread_mutex_lock(&mutexCursor);
1493 moveto(field[clientInfo->currRows][clientInfo->currCols-1].posX,
1494 +
1495 field[clientInfo->currRows][clientInfo->currCols-1].posY);
1496 setColor(GRAPHICS_FG_COLOR_RED, GRAPHICS_BG_COLOR_GREEN);
1497 printf("%s", SYMBOL_PACKET);
1498 reset();
1499 pthread_mutex_unlock(&mutexCursor);
1500 /* Verifico se nella posizione verso sud sia presente una Locazione */
1501 } else if(session->field[clientInfo->currRows][clientInfo->currCols-1] ==
1502 +
1503 SESSION_LOCATION_VALUE){
1504 for(int i=0; i<SESSION_LOCATIONS_STRLEN; i++){
1505 if(session->locations[i].currRows == clientInfo->currRows && session-
1506 +
1507 >locations[i].currCols == clientInfo->currCols-1){
1508 location = i;
1509 break;
1510 }
1511 }
1512 /* Controllo se la Locazione è quella indicata dal packet, in caso di
1513 esito negativo invio al client un messaggio di segnalazione */
1514 if(session->packets[clientInfo->havePacket].location != location){
1515 sendMsg(clientInfo, "$Server: Attenzione! Hai sbagliato locazione,
1516 +
1517 riprova!");
1518 } else{
1519 /* In caso di esito positivo aggiorno il numero di pacchi
1520 consegnati dal client e invio un messaggio alla sessione di
1521 avvenuta consegna */
1522 clientInfo->packetsDelivered += 1;
1523 clientInfo->havePacket = -1;
1524 sendMsg(clientInfo, "$remove blink");
1525 sprintf(msg, "$delivered %s", clientInfo->username);
1526 dynamicBuffer = (char*)calloc(GRAPHICS_CHAT_WIDTH, sizeof(char));
1527 strcpy(dynamicBuffer, msg);
1528 pthread_create(&tidUpdatePlayersInfo, NULL, updatePlayersInfo,
1529 +
1530 dynamicBuffer);
1531 sessionNumOfPackets -= 1;
1532 sendMsgToSession(NULL, msg);
1533 sprintf(msg, "$Server: %s ha consegnato un pacco, complimenti!",
1534 +
1535 clientInfo->username);
1536 sendMsgToSession(NULL, msg);
1537 timestamp = time(NULL);
1538 pthread_mutex_lock(&mutexLogs);
1539 sprintf(logsBuffer, " > [Sessione %d] Pacco consegnato da %s -
1540 +
1541 %s", idSession, clientInfo->username, ctime(&timestamp));
1542 if(write(logs, logsBuffer, strlen(logsBuffer)) == -1){
1543 if((msgChat = allocMsg("Errore durante la scrittura nel file di
1544 +
1545 logs.", GRAPHICS_FG_COLOR_RED, true)) != NULL){
1546 pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
1547 }
1548 }
1549 pthread_mutex_unlock(&mutexLogs);

```

```

1527                 pLnread_mutex_unlock(&mutexLogs);
1528             }
1529         }
1530     }
1531     break;
1532 }
1533 }
1534 }
1535
1536
1537 //____ FUNCTION FOR MANAGEMENT OPERATION CLIENT
*
1538
1539
1540 /**
1541 * @param void.
1542 *
1543 * @return: 1 in caso di errore, 0 in caso successo.
1544 *
1545 * La funzione connection effettua la connessione al server
1546 * mediante le specifiche fornite dall'utente.
1547 *
1548 */
1549 int connection(void){
1550
1551     int bytesReaded;
1552     char buffer[BUFFER_STRLEN];
1553     int port;
1554     bool error;
1555     struct sockaddr_in serverAddress;
1556
1557     resizeTerminal(GRAPHICS_SCREEN_HEIGHT, GRAPHICS_SCREEN_WIDTH);
1558     setDefaultBackground(GRAPHICS_BG_COLOR_WHITE);
1559     setDefaultForeground(GRAPHICS_FG_COLOR_BLACK);
1560     reset();
1561     memset(&serverAddress, '0', sizeof(serverAddress));
1562     serverAddress.sin_family = AF_INET;
1563     serverAddress.sin_addr.s_addr = htonl(INADDR_ANY);
1564     clean();
1565     printf("\n » Inserire la porta del server: ");
1566     fflush(stdout);
1567
1568     do{
1569         setCursor(true);
1570         error = false;
1571         if((bytesReaded = read(STDIN_FILENO, buffer, BUFFER_STRLEN)) == -1){
1572             setColor(GRAPHICS_FG_COLOR_RED, 0);
1573             perror("\n    <!> Errore read");
1574             reset();
1575             return 1;
1576         }
1577         buffer[bytesReaded] = '\0';
1578         /* Verifico se la porta fornita è valida */
1579         for(int i=0; i<strlen(buffer)-1; i++){
1580             if(buffer[i] < '0' || buffer[i] > '9'){
1581                 error = true;
1582                 break;
1583             }
1584         }

```

```

1585     }
1586     if(error){
1587         setColor(GRAPHICS_FG_COLOR_RED, 0);
1588         printf("\n    <!> Porta non valida, caratteri non ammessi - riprovare: ");
1589         reset();
1590         fflush(stdout);
1591     } else{
1592         if((port = atoi(buffer)) == 0 || !(port >= 0 && port <= 65535)){
1593             setColor(GRAPHICS_FG_COLOR_RED, 0);
1594             printf("\n    <!> Porta non valida, out of range - riprovare: ");
1595             reset();
1596             fflush(stdout);
1597             error = true;
1598         } else{
1599             listenerPort = port;
1600             setColor(GRAPHICS_FG_COLOR_GREEN, 0);
1601             setCursor(false);
1602             printf("\n    » Porta %d inserita con successo.\n", port);
1603             reset();
1604             fflush(stdout);
1605             serverAddress.sin_port = htons(atoi(buffer));
1606         }
1607
1608         if(!error){
1609             listenerSocket = socket(PF_INET, SOCK_STREAM, 0);
1610             /* Assegno un'indirizzo alla socket del server */
1611             if(bind(listenerSocket, (struct sockaddr*)&serverAddress,
1612                     sizeof(serverAddress)) == -1){
1613                 setColor(GRAPHICS_FG_COLOR_RED, 0);
1614                 setCursor(false);
1615                 perror("\n <!> Errore bind");
1616                 puts("");
1617                 reset();
1618                 sleep(1);
1619                 error = true;
1620             }
1621
1622             /* Rimango in ascolto di richieste di connessione da parte di client */
1623             if(listen(listenerSocket, LISTENER_QUEUE_STRLEN) == -1){
1624                 setColor(GRAPHICS_FG_COLOR_RED, 0);
1625                 setCursor(false);
1626                 perror("\n <!> Errore listen");
1627                 puts("");
1628                 reset();
1629                 sleep(1);
1630             }
1631         }
1632     }
1633     }while(error != false);

1634     printf("\n    » Inserire la difficoltà [0 <-> 10 => (0 = Random | 1 = No wall <-> 10 =
1635     * Estrema)]: ");
1636     fflush(stdout);
1637     do{
1638         setCursor(true);
1639         error = false;
1640         if((bytesReaded = read(STDIN_FILENO, buffer, BUFFER_STRLEN)) == -1){
1641             setColor(GRAPHICS_FG_COLOR_RED, 0):

```

```

1640     perror("\n      <!> Errore read");
1641     reset();
1642     return 1;
1643 }
1644
1645     buffer[bytesReaded] = '\0';
1646
1647     for(int i=0; i<strlen(buffer)-1; i++){
1648         if(buffer[i] < '0' || buffer[i] > '9'){
1649             error = true;
1650             break;
1651         }
1652     }
1653 }
1654
1655     if(error){
1656         setColor(GRAPHICS_FG_COLOR_RED, 0);
1657         printf("\n      <!> Difficoltà non valida, caratteri non ammessi - riprovare: ");
1658         reset();
1659         fflush(stdout);
1660     } else{
1661         if(atoi(buffer) < 0 || atoi(buffer) > 10){
1662             setColor(GRAPHICS_FG_COLOR_RED, 0);
1663             printf("\n      <!> Difficoltà non valida, out of range - riprovare: ");
1664             reset();
1665             fflush(stdout);
1666             error = true;
1667         } else{
1668             difficulty = atoi(buffer);
1669             setColor(GRAPHICS_FG_COLOR_GREEN, 0);
1670             setCursor(false);
1671             if(difficulty != 0){
1672                 printf("\n      » Difficoltà %d inserita con successo.\n", difficulty);
1673             } else{
1674                 printf("\n      » Difficoltà randomica inserita con successo.\n");
1675             }
1676             reset();
1677             fflush(stdout);
1678             sleep(1);
1679         }
1680     }
1681
1682     }while(error != false);
1683     return 0;
1684 }
1685 }
1686
1687
1688 /**
1689 * @param arg : NULL.
1690 *
1691 * @return: puntatore a void.
1692 *
1693 * La funzione connectionRequestsManagement gestisce le operazione di
1694 * connessione alla socket del server da parte dei client.
1695 *
1696 */
1697 void* connectionRequestsManagement(void* arg){
1698

```

```

1699 int connectSocket;
1700 pthread_t tidListenerClient;
1701 struct sockaddr_in clientAddress;
1702 char clientAddressIPv4[INET_ADDRSTRLEN];
1703 int clientAddressSize = sizeof(clientAddress);
1704 LpClientInfo clientInfo;
1705 char msg[GRAPHICS_CHAT_WIDTH];
1706 LpMsg msgChat;
1707 pthread_t tidUpdateChat;
1708 char logsBuffer[BUFFER_STRLEN];
1709
1710 while(true){
1711     /* Accetto le richieste di connessione da parte dei client */
1712     if((connectSocket = accept(listenerSocket, (struct sockaddr*)&clientAddress,
1713         * &clientAddressSize)) == -1){
1714         if((msgChat = allocMsg(" <!> Impossibile accettare una richiesta di
1715         * connessione.", GRAPHICS_FG_COLOR_RED, true))) {
1716             pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
1717         }
1718     } else{
1719         inet_ntop(AF_INET, &clientAddress, clientAddressIPv4, INET_ADDRSTRLEN);
1720         clientInfo = NULL;
1721     }
1722     /* Creo una struttura ClientInfo che conterrà tutte le informazioni riguardo al
1723     * client */
1724     if((clientInfo = newClientInfo(connectSocket, clientAddressIPv4, "")) == NULL){
1725         sprintf(msg, " <!> Impossibile allocare memoria, %s disconnesso.",
1726             * clientAddressIPv4);
1727         if((msgChat = allocMsg(msg, GRAPHICS_FG_COLOR_RED, true))){
1728             pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
1729         }
1730     } else{
1731         /* Aggiorno le strutture che contengono informazioni sullo status attuale */
1732         pthread_mutex_lock(&mutexClientInfo);
1733         insertClientInfo(&listClientInfo, clientInfo);
1734         pthread_mutex_unlock(&mutexClientInfo);
1735         /* Creo un thread che gestirà l'accesso del client al Server */
1736         pthread_create(&tidListenerClient, NULL, listenerClient, clientInfo);
1737         clientInfo->tidHandler = tidListenerClient;
1738         sprintf(msg, "Nuova connessione accettata: [Client: %s] - %s",
1739             * clientAddressIPv4, ctime(&(clientInfo->timestamp)));
1740         totalConnections += 1;
1741         pthread_mutex_lock(&mutexCursor);
1742         moveto(status[4].posX, status[4].posY);
1743         printf(" \u25cf Total CN: %7.2d", totalConnections);
1744         pthread_mutex_unlock(&mutexCursor);
1745         connectedClients += 1;
1746         pthread_mutex_lock(&mutexCursor);
1747         moveto(status[2].posX, status[2].posY);
1748         printf(" \u25cf Connected: %6.2d", connectedClients);
1749         pthread_mutex_unlock(&mutexCursor);
1750         msg[strlen(msg)-1] = '\0';
1751         if((msgChat = allocMsg(msg, GRAPHICS_FG_COLOR_GREEN, false))){
1752             pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
1753         }
1754
1755         pthread_mutex_lock(&mutexLogs);
1756         sprintf(logsBuffer, " > Nuova connessione accettata: [Client: %s] - %s",

```

```

*
    clientAddressIPv4, ctime(&(clientInfo->timestamp)));
1752 if(write(logs, logsBuffer, strlen(logsBuffer)) == -1){
1753     if((msgChat = allocMsg("Errore durante la scrittura nel file di logs.",
1754     GRAPHICS_FG_COLOR_RED, true)) != NULL){
1755         pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
1756     }
1757     pthread_mutex_unlock(&mutexLogs);
1758 }
1759 }
1760 memset(msg, '\0', GRAPHICS_CHAT_WIDTH);
1761 }
1762 }
1763
1764 /**
1765 * @param void.
1766 *
1767 * @return: return 1 in caso di errore e 0 altrimenti.
1768 *
1769 * La funzione initFile apre i file di logs, database
1770 * e di playground.
1771 *
1772 */
1773
1774 int initFile(void){
1775
1776 /* Apro il file di Log */
1777 if((logs = open("../files/logs.txt", O_RDWR | O_CREAT | O_TRUNC, S_IRWXU)) == -1){
1778     return 1;
1779 }
1780
1781 /* Apro il file di database */
1782 if((database = open("../files/database.txt", O_RDWR | O_CREAT | O_APPEND, S_IRWXU))
1783 == -1){
1784     return 1;
1785 }
1786
1787 /* Apro il file generatore di playground */
1788 if((fieldGenerator = open("../files/field.txt", O_RDONLY)) == -1){
1789     return 1;
1790 }
1791
1792 return 0;
1793 }
1794
1795
1796 /**
1797 * @param msg: Messaggio da scrivere sulla chat.
1798 * @param colorPair: Colore con cui stampare il messaggio.
1799 * @param error: Indica se il messaggio Ã" di errore.
1800 *
1801 * @return: una struttura di tipo LpMsg.
1802 *
1803 * La funzione allocMsg alloca un nodo Msg per il thread
1804 * updateChat.
1805 *
1806 */

```

```

1807 LpMsg allocMsg(char* msg, int color, bool error){
1808     LpMsg msgChat;
1809     if((msgChat = (Msg*)malloc(sizeof(Msg))) != NULL){
1810         strcpy(msgChat->msg, msg);
1811         msgChat->color = color;
1812         msgChat->error = error;
1813     }
1814     return msgChat;
1815 }
1816 }
1817
1818
1819 /**
1820 * @param clientInfo: puntatore della struttura che contiene informazioni
1821 * del client.
1822 * @param outcomingMsg: Messaggio da scrivere da scrivere al client.
1823 *
1824 * @return: void.
1825 *
1826 * La funzione sendMsg inoltra il messaggio catturato
1827 * dal listenerTextField al client.
1828 *
1829 */
1830 void sendMsg(LpClientInfo clientInfo, char* outcomingMsg){
1831
1832     char buffer[OUTCOMING_MSG_STRLEN];
1833     char msg[GRAPHICS_CHAT_WIDTH];
1834     LpMsg msgChat;
1835     pthread_t tidUpdateChat;
1836
1837     memset(buffer, '\0', OUTCOMING_MSG_STRLEN);
1838     strcpy(buffer, outcomingMsg);
1839     pthread_mutex_lock(&(clientInfo->mutexSocket));
1840     write(clientInfo->clientSocket, buffer, OUTCOMING_MSG_STRLEN);
1841     pthread_mutex_unlock(&(clientInfo->mutexSocket));
1842 }
1843
1844
1845 /**
1846 * @param clientInfo: puntatore della struttura che contiene informazioni
1847 * del client.
1848 * @param outcomingMsg: Messaggio da scrivere da scrivere al client.
1849 *
1850 * @return: void.
1851 *
1852 * La funzione sendMsgToSession inoltra il messaggio catturato
1853 * dal listenerTextField a tutti i client presenti nel game
1854 * della sessione.
1855 *
1856 */
1857 void sendMsgToSession(LpClientInfo clientInfo, char* outcomingMsg){
1858     pthread_mutex_lock(&mutexClientInfo);
1859     for(int i=0; i<SESSION_PLAYERS_STRLEN; i++){
1860         if(session->clients[i] != NULL && session->clients[i] != clientInfo){
1861             sendMsg(session->clients[i], outcomingMsg);
1862         }
1863     }
1864     pthread_mutex_unlock(&mutexClientInfo);

```

```

1865
1866    }
1867
1868
1869 /**
1870 * @param clientInfo: puntatore della struttura che contiene informazioni
1871 * del client.
1872 * @param outgoingMsg: Messaggio da scrivere da scrivere al client..
1873 *
1874 * @return: void.
1875 *
1876 * La funzione sendMsgToAll manda il messaggio a tutti i client presenti
1877 * in sessione.
1878 *
1879 */
1880 void sendMsgToAll(LpClientInfo clientInfo, char* outgoingMsg){
1881     pthread_mutex_lock(&mutexClientInfo);
1882     LpClientInfo tmp = listClientInfo;
1883     while(tmp != NULL){
1884         if((tmp->status == CLTINF_LOGGED || tmp->status == CLTINF_PLAYING) && tmp != clientInfo){
1885             sendMsg(tmp, outgoingMsg);
1886         }
1887         tmp = tmp->nextClientInfo;
1888     }
1889     pthread_mutex_unlock(&mutexClientInfo);
1890 }
1891
1892
1893 /**
1894 * @param toShuffle: array di caratteri da randomizzare.
1895 * @param dim : dimensione dell'array toShuffle.
1896 *
1897 * @return: void.
1898 *
1899 * La funzione shuffleLocations si occupa di randomizzare
1900 * le posizioni delle locazioni.
1901 *
1902 */
1903 void shuffleLocations(char toShuffle[], int dim){
1904     int index = dim;
1905     while(index > 1){
1906         int indexShuffle = rand()%index;
1907         index -= 1;
1908         char tmp = toShuffle[indexShuffle];
1909         toShuffle[indexShuffle] = toShuffle[index];
1910         toShuffle[index] = tmp;
1911     }
1912 }
1913
1914
1915 /**
1916 * @param toShuffle: array di caratteri da randomizzare.
1917 * @param dim : dimensione dell'array toShuffle.
1918 *
1919 * @return: void.
1920 *
1921 * La funzione shufflePackets si occupa di randomizzare

```

```

1922     * Le posizioni dei packets.
1923     *
1924     */
1925 void shufflePackets(Packet toShuffle[], int dim){
1926     int index = dim;
1927     while(index > 1){
1928         int indexShuffle = rand()%index;
1929         index -= 1;
1930         Packet tmp = toShuffle[indexShuffle];
1931         toShuffle[indexShuffle] = toShuffle[index];
1932         toShuffle[index] = tmp;
1933     }
1934 }
1935
1936
1937 /**
1938     * @param toShuffle: array di caratteri da randomizzare.
1939     * @param dim : dimensione dell'array toShuffle.
1940     *
1941     * @return: void.
1942     *
1943     * La funzione shuffleSpawn si occupa di randomizzare
1944     * le posizioni degli spawn dei player.
1945     *
1946 */
1947 void shuffleSpawn(Spawn toShuffle[], int dim){
1948     int index = dim;
1949     while(index > 1){
1950         int indexShuffle = rand()%index;
1951         index -= 1;
1952         Spawn tmp = toShuffle[indexShuffle];
1953         toShuffle[indexShuffle] = toShuffle[index];
1954         toShuffle[index] = tmp;
1955     }
1956 }
1957
1958
1959 /**
1960     * @param clientinfo: Nodo ClientInfo con informazioni sul client.
1961     *
1962     * @return: true in caso di errore, false altrimenti.
1963     *
1964     * La funzione Login gestisce il Login da parte
1965     * del client al Server.
1966     *
1967 */
1968 bool login(LpClientInfo clientInfo{
1969
1970     bool passed = false;                                /* Flag che indica il
1971     * superamento di una fase */
1972     char incomingMsg[INCOMING_MSG_STRLEN];           /* Buffer per messaggi in
1973     * entrata */
1974     char record[GRAPHICS_CHAT_WIDTH];                 /* Buffer per la Lettura
1975     * di un record del DB */
1976     char username[CLTINF_USERNAME_STRLEN];            /* Buffer per username del
1977     * client */
1978     char password[CLTINF_PASSWORD_STRLEN];            /* Buffer per password del
1979     * client */
1980
1981     . . .

```

```

1975     char incomingRecord[GRAPHICS_CHAT_WIDTH];           /* Username + Password
   * inseriti dall'utente   */
1976     int recordIndex;                                /* Indice di posizione del
   * buffer record      */
1977     char character[1];                            /* Array per la lettura dei
   * dati dal DB      */
1978     int bytesReaded;                           /* Numero di bytes letti dalla
   * read      */
1979     char msg[GRAPHICS_CHAT_WIDTH];
1980     LpMsg msgChat;
1981     pthread_t tidUpdateChat;
1982     bool alreadyLogged;
1983     LpClientInfoToJoin clientInfoToJoin;
1984     char logsBuffer[BUFFER_STRLEN];
1985
1986     memset(incomingMsg, '\0', INCOMING_MSG_STRLEN);
1987
1988     do{
1989         sendMsg(clientInfo, "$Server: Inserisci l'username.");
1990         memset(incomingMsg, '\0', INCOMING_MSG_STRLEN);
1991
1992         do{
1993             passed = true;
1994             /* Leggo l'username inserito dal client */
1995             if((bytesReaded = read(clientInfo->clientSocket, incomingMsg,
1996               INCOMING_MSG_STRLEN)) <= 0){
1997                 return false;
1998             }
1999             incomingMsg[bytesReaded] = '\0';
2000
2001             if(!strcmp(incomingMsg, "@exit")){
2002                 return true;
2003             } else{
2004                 if(strlen(incomingMsg) > CLTINF_USERNAME_STRLEN){
2005                     sendMsg(clientInfo, "$Server: Attenzione, username troppo lungo - [massimo
2006                       10 caratteri], riprovare.");
2007                     passed = false;
2008                 } else{
2009                     for(int i=0; i<strlen(incomingMsg); i++){
2010                         if(incomingMsg[i] == ' '){
2011                             sendMsg(clientInfo, "$Server: Attenzione, gli spazi non sono
2012                               consentiti, riprovare.");
2013                             passed = false;
2014                             break;
2015                         }
2016                     }
2017                 }
2018                 memset(incomingMsg, '\0', INCOMING_MSG_STRLEN);
2019             }
2020         }while(passed != true);
2021
2022         sendMsg(clientInfo, "$Server: Inserisci la password. ");
2023         memset(incomingMsg, '\0', INCOMING_MSG_STRLEN);
2024
2025         do{
2026             passed = true;

```

```

2020     passed = true;
2021     /* Leggo La password inserita dal client */
2022     if((bytesReaded = read(clientInfo->clientSocket, incomingMsg,
2023     * INCOMING_MSG_STRLEN)) <= 0){
2024         return false;
2025     }
2026     incomingMsg[bytesReaded] = '\0';
2027
2028     /* Verifico se L'utente ha inserito il comando di uscita */
2029     if(!strcmp(incomingMsg, "@exit")){
2030         return true;
2031     } else{
2032         /* Controllo se La password supera La Lunghezza di 10 caratteri stabiliti */
2033         if(strlen(incomingMsg) > CLTINF_PASSWORD_STRLEN){
2034             sendMsg(clientInfo, "$Server: Attenzione, password troppo lunga - [massimo
2035             * 10 caratteri], riprovare.");
2036             passed = false;
2037         } else{
2038             /* Controllo che nella password non siano presenti caratteri di spazio */
2039             for(int i=0; i<strlen(incomingMsg); i++){
2040                 if(incomingMsg[i] == ' '){
2041                     sendMsg(clientInfo, "$Server: Attenzione, gli spazi non sono
2042                     * consentiti, riprovare.");
2043                     passed = false;
2044                     break;
2045                 }
2046             }
2047             if(passed != false){
2048                 strcpy(password, incomingMsg);
2049             }
2050             memset(incomingMsg, '\0', INCOMING_MSG_STRLEN);
2051         }
2052     }
2053     }while(passed != true);
2054
2055     memset(incomingRecord, '\0', CLTINF_USERNAME_STRLEN*2);
2056     sprintf(incomingRecord, "%s %s", username, password);
2057     pthread_mutex_lock(&mutexDatabase);
2058     lseek(database, 0, SEEK_SET);
2059     memset(record, '\0', GRAPHICS_CHAT_WIDTH);
2060     recordIndex = 0;
2061
2062     /* Ciclo sul file database per verificare che lo username non esista già */
2063     do{
2064         /* La lettura avviene un carattere alla volta fintanto non viene trovato un
2065         * newline */
2066         alreadyLogged = false;
2067         if((bytesReaded = read(database, character, 1)) > 0){
2068             /* Il carattere è diverso da un newline? */
2069             if(character[0] != '\n'){
2070                 record[recordIndex++] = character[0];
2071             } else{
2072                 if(!strcmp(record, incomingRecord)){
2073                     pthread_mutex_lock(&mutexClientInfo);
2074                     LpClientInfo tmp = listClientInfo;
2075                     while(tmp != NULL && alreadyLogged == false){
2076                         if((tmp->status == CLTINF_LOGGED || tmp->status == CLTINF_PLAYING) &&
2077                             !strcmp(tmp->username, username) && tmp != clientInfo){
2078                             sendMsg(clientInfo, "$Server: L'utente specificato è già connesso al
2079

```

```

2079     semuringClientInfo, &server, L'utente specificato è già connesso al
2080     Server.");
2081     sleep(1);
2082     alreadyLogged = true;
2083 }
2084 tmp = tmp->nextClientInfo;
2085 }
2086 pthread_mutex_unlock(&mutexClientInfo);

2087 if(alreadyLogged == false){
2088     clientInfo->status = CLTINF_LOGGED;
2089     strcpy(clientInfo->username, username);
2090     pthread_mutex_lock(&mutexClientInfo);
2091     clientInfoToJoin = newClientInfoToJoin(clientInfo);
2092     if(clientInfoToJoin != NULL){
2093         enqueueClientInfoToJoin(&headClientInfoToJoinQueue,
2094         &tailClientInfoToJoinQueue, clientInfoToJoin);
2095         waitingClients += 1;
2096         pthread_mutex_lock(&mutexCursor);
2097         moveto(status[3].posX, status[3].posY);
2098         printf(" \u25cf Waiting: %8.2d", waitingClients);
2099         pthread_mutex_unlock(&mutexCursor);
2100     } else{
2101         return false;
2102     }
2103     pthread_mutex_unlock(&mutexClientInfo);
2104     sprintf(msg, "Login del client %s avvenuto con successo - %s",
2105     *clientInfo->clientAddressIPv4, ctime(&(clientInfo->timestamp)));
2106     if((msgChat = allocMsg(msg, GRAPHICS_FG_COLOR_GREEN, false))){
2107         pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
2108     }

2109     pthread_mutex_lock(&mutexLogs);
2110     sprintf(logsBuffer, " > Login effettuato con successo: [Client: %s -
2111     *%s] - %s", clientInfo->clientAddressIPv4, clientInfo->username,
2112     ctime(&(clientInfo->timestamp)));
2113     if(write(logs, logsBuffer, strlen(logsBuffer)) == -1){
2114         if((msgChat = allocMsg("Errore durante la scrittura nel file di
2115         *logs.", GRAPHICS_FG_COLOR_RED, true)) != NULL){
2116             pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
2117         }
2118     }
2119     pthread_mutex_unlock(&mutexLogs);
2120     sendMsg(clientInfo, "$Server: Login effettuato con successo.");
2121     sleep(1);
2122     sendMsg(clientInfo, "$Server: In attesa che si liberi una sessione...");
2123     sleep(1);
2124 }
2125 break;
2126 } else{
2127     /* Ripulisce le strutture di appoggio */
2128     memset(record, '\0', GRAPHICS_CHAT_WIDTH);
2129     recordIndex = 0;
2130 }
2131 }
2132 }
2133 }while(bytesReaded != 0);
2134 pthread_mutex_unlock(&mutexDatabase);
2135

```

```

2132     if(clientInfo->status != CLTINF_PLAYING && clientInfo->status != CLTINF_LOGGED &&
2133     * alreadyLogged != true){
2134         sendMsg(clientInfo, "$Server: Username o password errati, riprovare.");
2135         sleep(1);
2136     }
2137 }while(clientInfo->status != CLTINF_LOGGED && clientInfo->status != CLTINF_PLAYING);
2138
2139 return false;
2140
2141 }
2142
2143
2144 /**
2145 * @param clientinfo: Nodo ClientInfo con informazioni sul client.
2146 *
2147 * @return: 1 in caso di errore, 0 altrimenti.
2148 *
2149 * La funzione signin gestisce il signin da parte
2150 * del client al Server. Verifica inoltre che l'username del client
2151 * sia univoco.
2152 *
2153 */
2154 bool signin(LpClientInfo clientInfo){
2155
2156     bool passed = false;                                /* Flag che indica il superamento di
2157     * una fase */
2158     char incomingMsg[INCOMING_MSG_STRLEN];           /* Buffer per messaggi in
2159     * entrata */
2160     char record[GRAPHICS_CHAT_WIDTH];                /* Buffer per la lettura di un record
2161     * del DB */
2162     char username[CLTINF_USERNAME_STRLEN];           /* Buffer per username del
2163     * client */
2164     char password[CLTINF_PASSWORD_STRLEN];           /* Buffer per password del
2165     * client */
2166     int recordIndex;                                 /* Indice di posizione del buffer
2167     * record */
2168     char character[1];                               /* Array per la lettura dei dati dal
2169     * DB */
2170     char* usernameDB;                             /* Username estratto dal
2171     * DB */
2172     int bytesReaded;                            /* Numero di bytes letti dalla
2173     * read */
2174     char msg[GRAPHICS_CHAT_WIDTH];
2175     LpMsg msgChat;
2176     pthread_t tidUpdateChat;
2177     time_t timestamp;
2178     char logsBuffer[BUFFER_STRLEN];
2179
2180     memset(incomingMsg, '\0', INCOMING_MSG_STRLEN);
2181     sendMsg(clientInfo, "$Server: Inserisci l'username - massimo 10 caratteri.");
2182
2183     /* Ciclo finchè il client non inserisce un username che rispetti le condizioni di
2184     * lunghezza, e che non sia già esistente */
2185     do{
2186         passed = true;
2187         /* Leggo l'username inserito dal client */
2188         if((bytesReaded = read(clientInfo->clientSocket, incomingMsg,

```

```

        ```)
INCOMING_MSG_STRLEN)) <= 0){
 return false;
}
incomingMsg[bytesReaded] = '\0';

/* Verifico se l'utente ha inserito il comando di uscita */
if(!strcmp(incomingMsg, "@exit")){
 return true;
} else{
 /* Controllo se lo username supera la lunghezza di 10 caratteri stabiliti */
 if(strlen(incomingMsg) > CLTINF_USERNAME_STRLEN){
 sendMsg(clientInfo, "$Server: Attenzione, username troppo lungo - [massimo
 10 caratteri], riprovare.");
 passed = false;
 } else{
 /* Controllo se nell'username siano presenti caratteri di spazio */
 for(int i=0; i<strlen(incomingMsg); i++){
 if(incomingMsg[i] == ' '){
 sendMsg(clientInfo, "$Server: Attenzione, gli spazi non sono consentiti,
 riprovare.");
 passed = false;
 break;
 }
 }
 /* Nel caso in cui l'username fosse ancora valido proseguo con le verifiche
 */
 if(passed != false){
 pthread_mutex_lock(&mutexDatabase);
 lseek(database, 0, SEEK_SET);
 memset(record, '\0', GRAPHICS_CHAT_WIDTH);
 recordIndex = 0;

 /* Ciclo sul file database per verificare che lo username non esista già */
 do{
 /* La lettura avviene un carattere alla volta fintanto non viene trovato
 un newline oppure \0 */
 if((bytesReaded = read(database, character, 1)) > 0){
 /* Il carattere è diverso da un newline oppure un fine stringa? */
 if(character[0] != '\n'){
 record[recordIndex++] = character[0];
 } else{
 record[recordIndex] = '\0';
 usernameDB = strtok(record, " ");
 /* Controllo se l'username esista già nel database, se sì, stampo
 un errore e riclico il do */
 if(!strcmp(usernameDB, incomingMsg)){
 sendMsg(clientInfo, "$Server: Attenzione, username non
 disponibile, riprovare.");
 passed = false;
 break;
 } else{
 /* Ripulisco le strutture di appoggio */
 memset(record, '\0', GRAPHICS_CHAT_WIDTH);
 recordIndex = 0;
 }
 }
 }
 }while(bytesReaded != 0);
 }
 }
}

```

```

2230
2231 /* Verifico L'eventuale presenza di errori */
2232 if(bytesReaded == -1){
2233 sprintf(msg, " <!> Impossibile leggere dati dal Database.");
2234 if((msgChat = allocMsg(msg, GRAPHICS_FG_COLOR_RED, true))){
2235 pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
2236 }
2237 sleep(2);
2238 /* La terminazione è necessariamente bruta a causa dell'impossibilità di
2239 catturare */
2240 /* L'exit status del thread chiamante signin (numero di utenti non
2241 deterministico). */
2242 /* Chiaramente una soluzione sarebbe potuta essere una lista di tid,
2243 tuttavia */
2244 /* sarebbe stata un'implementazione "inutilmente" costosa, considerando
2245 che in certi */
2246 /* casi la terminazione è
2247 obbligatoria. */
2248 pthread_kill(pthread_self(), SIGTERM);
2249 } else if(passed != false){
2250 strcpy(username, incomingMsg);
2251 }
2252 pthread_mutex_unlock(&mutexDatabase);
2253 }
2254 memset(incomingMsg, '\0', INCOMING_MSG_STRLEN);
2255 }
2256 }while(passed != true);

2257 sendMsg(clientInfo, "$Server: Inserisci la password - massimo 10 caratteri.");
2258 memset(incomingMsg, '\0', INCOMING_MSG_STRLEN);

2259 do{
2260 passed = true;
2261 /* Leggo La password inserita dal client */
2262 if((bytesReaded = read(clientInfo->clientSocket, incomingMsg,
2263 * INCOMING_MSG_STRLEN)) <= 0){
2264 return false;
2265 }
2266 incomingMsg[bytesReaded] = '\0';

2267 /* Verifico se l'utente ha inserito il comando di uscita */
2268 if(!strcmp(incomingMsg, "@exit")){
2269 return true;
2270 } else{
2271 /* Controllo se la password supera la Lunghezza di 10 caratteri stabiliti */
2272 if(strlen(incomingMsg) > CLTINF_PASSWORD_STRLEN){
2273 sendMsg(clientInfo, "$Server: Attenzione, password troppo lunga - [massimo
2274 10 caratteri], riprovare.");
2275 passed = false;
2276 } else{
2277 /* Controllo che nella password non siano presenti caratteri di spazio */
2278 for(int i=0; i<strlen(incomingMsg); i++){
2279 if(incomingMsg[i] == ' '){
2280 sendMsg(clientInfo, "$Server: Attenzione, gli spazi non sono
2281 consentiti, riprovare.");
2282 passed = false;
2283 break;

```

```

2280 }
2281 }
2282 if(passed != false){
2283 strcpy(password, incomingMsg);
2284 }
2285 }
2286 memset(incomingMsg, '\0', INCOMING_MSG_STRLEN);
2287 }
2288 }while(passed != true);

2289
2290 memset(record, '\0', GRAPHICS_CHAT_WIDTH);
2291 /* Concateno username e password in un solo buffer */
2292 sprintf(record, "%s %s\n", username, password);
2293 pthread_mutex_lock(&mutexDatabase);

2294
2295 /* Scrivo i dati di registrazione del client nel database */
2296 if(write(database, record, strlen(record)) == -1){
2297 sprintf(msg, " <!> Impossibile scrivere dati nel Database.");
2298 if((msgChat = allocMsg(msg, GRAPHICS_FG_COLOR_RED, true))){
2299 pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
2300 }
2301 sleep(2);
2302 /* La terminazione è necessariamente bruta a causa dell'impossibilità di
2303 catturare */
2304 /* L'exit status del thread chiamante signin (numero di utenti non
2305 deterministico) */
2306 /* Chiaramente una soluzione sarebbe potuta essere una lista di tid,
2307 tuttavia */
2308 /* sarebbe stata un'implementazione "inutilmente" costosa, considerando che in
2309 certi */
2310 /* casi la terminazione è
2311 obbligatoria. */
2312 pthread_kill(pthread_self(), SIGTERM);
2313 }
2314 pthread_mutex_unlock(&mutexDatabase);
2315 /* Registrazione effettuata con successo */
2316 timestamp = time(NULL);
2317 pthread_mutex_lock(&mutexLogs);
2318 sprintf(logsBuffer, " > Registrazione del client %s avvenuta con successo - %s",
2319 *clientInfo->clientAddressIPv4, ctime(&(clientInfo->timestamp)));
2320 if(write(logs, logsBuffer, strlen(logsBuffer)) == -1){
2321 if((msgChat = allocMsg("Errore durante la scrittura nel file di logs.",
2322 *GRAPHICS_FG_COLOR_RED, true)) != NULL){
2323 pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
2324 }
2325 }
2326 pthread_mutex_unlock(&mutexLogs);
2327 sprintf(msg, "Registrazione del client %s avvenuta con successo.", clientInfo-
2328 *>clientAddressIPv4);
2329 if((msgChat = allocMsg(msg, GRAPHICS_FG_COLOR_GREEN, false))){
2330 pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
2331 }
2332 sendMsg(clientInfo, "$Server: Registrazione avvenuta con successo, ora puoi
2333 effettuare il login.");
2334 clientInfo->status = CLTINF_REGISTERED;
2335
2336 return false;
2337
2338

```

```

2329 }
2330
2331
2332 /**
2333 * @param clientInfo : arg -> puntatore della struttura che contiene informazioni
2334 * del client connesso alla socket del server.
2335 *
2336 * @return: puntatore a void.
2337 *
2338 * La funzione listenerClient si occupa di gestire le azioni di ascolto
2339 * dei messaggi da parte del client, e l'invio dei comandi utilizzabili
2340 * al client.
2341 *
2342 */
2343 void* listenerClient(void* arg){
2344
2345 LpClientInfo clientInfo = (LpClientInfo)arg;
2346 char incomingMsg[INCOMING_MSG_STRLEN]; /* Buffer messaggio in
2347 * entrata */
2348 int bytesReaded; /* Numero di bytes letti dalla
2349 * read */
2350 LpMsg msgChat;
2351 char msg[GRAPHICS_CHAT_WIDTH];
2352 pthread_t tidUpdateChat;
2353 bool exited = false;
2354
2355 /* Invio il messaggio di benvenuto */
2356 sendMsg(clientInfo, "$Server: Benvenuto/a! Digita @cmd per conoscere i comandi.");
2357 memset(incomingMsg, '\0', INCOMING_MSG_STRLEN);
2358
2359 while(exited != true && (bytesReaded = read(clientInfo->clientSocket, incomingMsg,
2360 INCOMING_MSG_STRLEN)) > 0){
2361 incomingMsg[bytesReaded] = '\0';
2362
2363 switch (clientInfo->status) {
2364 case CLTINF_GUEST:
2365 if(!strcmp(incomingMsg, "@cmd")){
2366 sendMsg(clientInfo, "$Server: Comandi disponibili -> [@cmd - @login -
2367 * @signin - @exit]");
2368 } else if(!strcmp(incomingMsg, "@signin")){
2369 exited = signin(clientInfo);
2370 if(exited){
2371 sendMsg(clientInfo, "$Server: Comandi disponibili -> [@cmd - @login -
2372 * @signin - @exit]");
2373 exited = false;
2374 }
2375 }
2376 } else if(!strcmp(incomingMsg, "@exit")){
2377 exited = true;
2378 } else if(incomingMsg[0] == '@'){
2379 sendMsg(clientInfo, "$Server: Comando non disponibile, riprova!");
2380 } else{

```

```

2381 sendMsg(clientInfo, "$Server: Per utilizzare la chat devi prima effettuare
* il login!");
2382 }
2383 break;
2384
2385 case CLTINF_REGISTERED:
2386 if(incomingMsg[0] == '@'){
2387 if(!strcmp(incomingMsg, "@cmd")){
2388 sendMsg(clientInfo, "$Server: Comandi disponibili -> [@cmd - @login -
* @exit]");
2389 } else if(!strcmp(incomingMsg, "@login")){
2390 exited = login(clientInfo);
2391 if(exited){
2392 sendMsg(clientInfo, "$Server: Comandi disponibili -> [@cmd - @login -
* @exit]");
2393 exited = false;
2394 }
2395 } else if(!strcmp(incomingMsg, "@exit")){
2396 exited = true;
2397 } else if(incomingMsg[0] == '@'){
2398 sendMsg(clientInfo, "$Server: Comando non disponibile, riprova!");
2399 } else{
2400 sendMsg(clientInfo, "$Server: Per utilizzare la chat devi prima
* effettuare il login!");
2401 }
2402 }
2403 break;
2404
2405 case CLTINF_LOGGED:
2406 if(incomingMsg[0] == '@'){
2407 if(!strcmp(incomingMsg, "@cmd")){
2408 sendMsg(clientInfo, "$Server: Comandi disponibili -> [@cmd - @time -
* @exit]");
2409 } else if(!strcmp(incomingMsg, "@exit")){
2410 exited = true;
2411 } else if(!strcmp(incomingMsg, "@time")){
2412 if(session->joinable){
2413 int queueLen;
2414 pthread_mutex_lock(&mutexTimer);
2415 int totalSeconds = alarm(0);
2416 alarm(totalSeconds);
2417 pthread_mutex_unlock(&mutexTimer);
2418 LpClientInfoToJoin tmp = headClientInfoToJoinQueue;
2419 while(tmp != NULL){
2420 queueLen += 1;
2421 tmp = tmp->nextClientInfoToJoin;
2422 }
2423 totalSeconds += (queueLen/5)*TIME;
2424 int minutes = totalSeconds / 60;
2425 int seconds = totalSeconds % 60;
2426 if(minutes == 0){
2427 sprintf(msg, "$Server: Giocherai all'incirca tra: %.2d secondi.",
2428 seconds);
2429 } else{
2430 sprintf(msg, "$Server: Giocherai all'incirca tra: %.2d:%.2d
* minuti.", minutes, seconds);
2431 }
2432 sendMsg(clientInfo, msg);
2433 }

```

```

2432 } else{
2433 sendMsg(clientInfo, "$Server: Comando attualmente non disponibile.");
2434 }
2435 } else{
2436 sendMsg(clientInfo, "$Server: Comando non disponibile, riprova!");
2437 }
2438 } else{
2439 sprintf(msg, "%s: %s", clientInfo->username, incomingMsg);
2440 sendMsgToAll(clientInfo, msg);
2441 }
2442 break;
2443
2444 case CLTINF_PLAYING:
2445 if(incomingMsg[0] == '@'){
2446 if(!strcmp(incomingMsg, "@exit")){
2447 exited = true;
2448 } else if(!strcmp(incomingMsg, "@S") || !strcmp(incomingMsg, "@N") ||
2449 * !strcmp(incomingMsg, "@O") || !strcmp(incomingMsg, "@E")){
2450 movePlayer(clientInfo, incomingMsg);
2451 } else if(!strcmp(incomingMsg, "@PS") || !strcmp(incomingMsg, "@PN") ||
2452 * !strcmp(incomingMsg, "@PO") || !strcmp(incomingMsg, "@PE") ||
2453 !strcmp(incomingMsg, "@DS") || !strcmp(incomingMsg, "@DN") ||
2454 !strcmp(incomingMsg, "@DO") || !strcmp(incomingMsg, "@DE")){
2455 actionPlayer(clientInfo, incomingMsg);
2456 } else if(!strcmp(incomingMsg, "@cmd")){
2457 sendMsg(clientInfo, "$Server: Comandi disponibili -> [@S - @N - @E - @O
2458 * - @P(S/N/E/O) - @D(S/N/E/O)]");
2459 }
2460 } else if(incomingMsg[0] == '%'){
2461 int index = 0;
2462 for(index=0; index<strlen(incomingMsg)-1; index++){
2463 incomingMsg[index] = incomingMsg[index+1];
2464 }
2465 incomingMsg[index] = '\0';
2466 sprintf(msg, "%s: %s", clientInfo->username, incomingMsg);
2467 sendMsgToSession(clientInfo, msg);
2468 } else{
2469 sprintf(msg, "%s: %s", clientInfo->username, incomingMsg);
2470 sendMsgToAll(clientInfo, msg);
2471 }
2472 break;
2473 }
2474
2475 memset(incomingMsg, '\0', INCOMING_MSG_STRLEN);
2476
2477 if(exited){
2478 sendMsg(clientInfo, "$Server: Torna presto! Disconnessione in corso...");
2479 }
2480
2481 disconnectionManagement(clientInfo);
2482
2483 }
2484
2485 /**
2486 * @param clientinfo: puntatore della struttura che contiene informazioni del client.
2487 *
2488 * @return: puntatore a void.
2489 *
2490 * La funzione disconnectionManagement controlla le operazioni di accodamento

```

```

2480 * La funzione disconnectionManagement gestisce le operazioni di aggiornamento
2481 * mappa alla disconnessione di tale client, e alla comunicazione agli altri client
2482 * della disconnessione di esso.
2483 *
2484 */
2485 void disconnectionManagement(LpClientInfo clientInfo){
2486
2487 pthread_t tidUpdateChat;
2488 char msg[GRAPHICS_CHAT_WIDTH];
2489 LpMsg msgChat;
2490 char* dynamicBuffer;
2491 pthread_t tidUpdatePlayersInfo;
2492 time_t timestamp = time(NULL);
2493 char logsBuffer[BUFFER_STRLEN];
2494
2495 pthread_mutex_lock(&(session->mutexSession));
2496 /* Verifico lo stato del client */
2497 if(clientInfo->status == CLTINF_PLAYING){
2498 /* Se è in possesso di un packet allora*/
2499 if(clientInfo->havePacket != -1){
2500 /* Inserisco nella posizione del player disconnesso il packet e invio un
2501 * messaggio di segnalazione a tutti i client in sessione di gioco */
2502 sprintf(msg, "$add packet %d %d", clientInfo->currRows, clientInfo->currCols);
2503 sendMsgToSession(NULL, msg);
2504 sessionNumOfPackets += 1;
2505 session->field[clientInfo->currRows][clientInfo->currCols] =
2506 SESSION_PACKET_VALUE;
2507 session->packets[clientInfo->havePacket].currRows = clientInfo->currRows;
2508 session->packets[clientInfo->havePacket].currCols = clientInfo->currCols;
2509 pthread_mutex_lock(&mutexCursor);
2510 moveto(field[clientInfo->currRows][clientInfo->currCols].posX, field[clientInfo-
2511 * >currRows][clientInfo->currCols].posY);
2512 setColor(GRAPHICS_FG_COLOR_RED, GRAPHICS_BG_COLOR_GREEN);
2513 printf("%s", SYMBOL_PACKET);
2514 reset();
2515 pthread_mutex_unlock(&mutexCursor);
2516 } else{
2517 /* Altrimenti nella posizione del player disconnesso inserisco l'erba e invio
2518 * un messaggio di segnalazione a tutti i client in sessione di gioco */
2519 session->field[clientInfo->currRows][clientInfo->currCols] =
2520 SESSION_GRASS_VALUE;
2521 sprintf(msg, "$update %d %d", clientInfo->currRows, clientInfo->currCols);
2522 sendMsgToSession(NULL, msg);
2523 pthread_mutex_lock(&mutexCursor);
2524 moveto(field[clientInfo->currRows][clientInfo->currCols].posX,
2525 * field[clientInfo->currRows][clientInfo->currCols].posY);
2526 setColor(GRAPHICS_FG_COLOR_GREEN, GRAPHICS_BG_COLOR_GREEN);
2527 printf("%s", SYMBOL_GRASS);
2528 reset();
2529 pthread_mutex_unlock(&mutexCursor);
2530 }
2531 /* Elimino il client dalla coda della sessione */
2532 deleteClientFromSession(session, clientInfo);
2533 /* Invio un messaggio di segnalazione della disconnessione del client a tutti i
2534 * client in sessione */
2535 sprintf(msg, "$disconnected %s", clientInfo->username);
2536 dynamicBuffer = (char*)calloc(GRAPHICS_CHAT_WIDTH, sizeof(char));
2537 strcpy(dynamicBuffer, msg);
2538 pthread_create(&tidUpdatePlayersInfo, NULL, updatePlayersInfo, dynamicBuffer);

```

```

2537 pctrl_cau_create(&ctrlUpdateLayerStatus, NULL, updateLayerStatus, dynamicBuffer),
2538 sendMsgToSession(NULL, msg);
2539 pthread_cond_signal(&condMatchmaker);
2540 }
2541 pthread_mutex_unlock(&(session->mutexSession));
2542 pthread_mutex_lock(&mutexLogs);
2543
2544 if(clientInfo->status == CLTINF_LOGGED || clientInfo->status == CLTINF_PLAYING){
2545 sprintf(logsBuffer, " > Il client %s - %s si è disconnesso dal Server - %s",
2546 * clientInfo->username, clientInfo->clientAddressIPv4, ctime(×tamp));
2547 } else{
2548 sprintf(logsBuffer, " > Il client %s si è disconnesso dal Server - %s",
2549 * clientInfo->clientAddressIPv4, ctime(×tamp));
2550 }
2551
2552 if(write(logs, logsBuffer, strlen(logsBuffer)) == -1){
2553 if((msgChat = allocMsg("Errore durante la scrittura nel file di logs.",
2554 * GRAPHICS_FG_COLOR_RED, true)) != NULL){
2555 pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
2556 }
2557 }
2558
2559 pthread_mutex_unlock(&mutexLogs);
2560 /* Stampo nella chat del server il nome e l'orario della disconnectione del client
2561 */
2562 sprintf(msg, "Il client %s si è disconnesso dal Server - %s", clientInfo-
2563 * >clientAddressIPv4, ctime(×tamp));
2564 if((msgChat = allocMsg(msg, GRAPHICS_FG_COLOR_RED, false))){
2565 pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
2566 }
2567
2568 /* Incremento il numero di disconnectione nel server e aggiorno la grafica */
2569 totalDisconnections += 1;
2570 pthread_mutex_lock(&mutexCursor);
2571 moveto(status[5].posX, status[5].posY);
2572 printf(" \u25cf Total DC: %7.2d", totalDisconnections);
2573 pthread_mutex_unlock(&mutexCursor);
2574 connectedClients -= 1;
2575 pthread_mutex_lock(&mutexCursor);
2576 moveto(status[2].posX, status[2].posY);
2577 printf(" \u25cf Connected: %6.2d", connectedClients);
2578 pthread_mutex_unlock(&mutexCursor);
2579 pthread_mutex_lock(&mutexClientInfo);
2580
2581 /* Verifico se lo stato del client era di tipo LOGGED*/
2582 if(clientInfo->status == CLTINF_LOGGED){
2583 /* In esito positivo Lo elimino dalla coda Join */
2584 deleteClientInfoToJoin(&headClientInfoToJoinQueue, &tailClientInfoToJoinQueue,
2585 * clientInfo);
2586 waitingClients -= 1;
2587 pthread_mutex_lock(&mutexCursor);
2588 moveto(status[3].posX, status[3].posY);
2589 printf(" \u25cf Waiting: %8.2d", waitingClients);
2590 pthread_mutex_unlock(&mutexCursor);
2591 }
2592
2593 /* Chiudo la socket di comunicazione */
2594 close(clientInfo->clientSocket);
2595 /* Elimino dalla lista dei client . il client disconnesso */

```

```

2587 , <--> clientInfo.listClientInfo, <--> clientInfo);
2588 deleteClientInfo(&listClientInfo, &clientInfo);
2589 pthread_mutex_unlock(&mutexClientInfo);
2590 }
2591
2592 /**
2593 * @param void.
2594 *
2595 * @return: puntatore a void.
2596 *
2597 * La funzione initSession si occupa di creare un nuova sessione, e di rigenerare la
2598 * mappa
2599 * a ogni nuova sessione creata.
2600 *
2601 */
2602 int initSession(void){
2603
2604 int currLocation = 0;
2605 LpMsg msgChat;
2606 pthread_t tidUpdateChat;
2607 char msg[GRAPHICS_CHAT_WIDTH];
2608 char logsBuffer[BUFFER_STRLEN];
2609
2610 sleep(1);
2611
2612 /* Verifico che la sessione sia diversa da NULL , in caso di esito positivo eseguo
2613 * una free della sessione */
2614 if(session != NULL){
2615 free(session);
2616 }
2617
2618 /* Creo una nuova sessione */
2619 session = newSession();
2620 idSession += 1;
2621 pthread_mutex_lock(&mutexCursor);
2622 moveto(status[1].posX, status[1].posY);
2623 printf(" \u25cf ID Sessione: %.2d", idSession);
2624 pthread_mutex_unlock(&mutexCursor);
2625 pthread_mutex_lock(&mutexCursor);
2626
2627 /* Pulisco la grafica della matrice */
2628 for(int i=0; i<GRAPHICS_FIELD_HEIGHT; i++){
2629 for(int j=0; j<GRAPHICS_FIELD_WIDTH; j++){
2630 moveto(field[i][j]. posX, field[i][j]. posY);
2631 setColor(GRAPHICS_FG_COLOR_GREEN, GRAPHICS_BG_COLOR_GREEN);
2632 printf(" ");
2633 reset();
2634 }
2635 }
2636 }
2637
2638
2639 pthread_mutex_unlock(&mutexCursor);
2640 pthread_mutex_lock(&mutexLogs);
2641 sprintf(logsBuffer, " > Nuova sessione generata -> ID Sessione: %d\n", idSession);
2642 if(write(logs, logsBuffer, strlen(logsBuffer)) == -1){
2643 if((msgChat = allocMsg("Errore durante la scrittura nel file di logs.",
2644 * GRAPHICS_FG_COLOR_RED, true)) != NULL){
2645 pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);

```

```

2645 }
2646 }
2647 pthread_mutex_unlock(&mutexLogs);
2648
2649 if(session != NULL){
2650
2651 /* Inizializzo i valori della matrice */
2652 if(initField()){
2653 return 1;
2654 }
2655 /* Stampo al terminale la matrice */
2656 for(int i=0; i<GRAPHICS_FIELD_HEIGHT; i++){
2657 for(int j=0; j<GRAPHICS_FIELD_WIDTH; j++){
2658 pthread_mutex_lock(&mutexCursor);
2659 moveto(field[i][j].posX, field[i][j].posY);
2660 switch (session->field[i][j]){
2661 case 0:
2662 setColor(GRAPHICS_FG_COLOR_GREEN, GRAPHICS_BG_COLOR_GREEN);
2663 printf("%s", SYMBOL_GRASS);
2664 reset();
2665 break;
2666 case 1:
2667 setColor(GRAPHICS_FG_COLOR_BLUE, GRAPHICS_BG_COLOR_BLACK);
2668 printf("%s", SYMBOL_WALL);
2669 reset();
2670 break;
2671 case 2:
2672 setColor(GRAPHICS_FG_COLOR_WHITE, GRAPHICS_BG_COLOR_MAGENTA);
2673 printf("%c", session->locations[currLocation].name);
2674 reset();
2675 currLocation += 1;
2676 break;
2677 case 3:
2678 setColor(GRAPHICS_FG_COLOR_RED, GRAPHICS_BG_COLOR_GREEN);
2679 printf("%s", SYMBOL_PACKET);
2680 reset();
2681 break;
2682 case 4:
2683 setColor(GRAPHICS_FG_COLOR_GREEN, GRAPHICS_BG_COLOR_GREEN);
2684 printf("%s", SYMBOL_GRASS);
2685 reset();
2686 break;
2687 }
2688
2689 fflush(stdout);
2690 pthread_mutex_unlock(&mutexCursor);
2691 }
2692 }
2693 currLocation = 0;
2694 pthread_mutex_lock(&mutexLogs);
2695 sprintf(logsBuffer, " > Mappa per la sessione %d, generata: \n", idSession);
2696 if(write(logs, logsBuffer, strlen(logsBuffer)) == -1){
2697 if((msgChat = allocMsg("Errore durante la scrittura nel file di logs.",
2698 * GRAPHICS_FG_COLOR_RED, true)) != NULL){
2699 pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
2700 }
2701 }
2702 sprintf(logsBuffer, "\n");

```

```

2702 write(logs, logsBuffer, strlen(logsBuffer));
2703 /* Scrivo nel file di Log la matrice generata dalla sessione */
2704 for(int i=0; i<GRAPHICS_FIELD_HEIGHT; i++){
2705 for(int j=0; j<GRAPHICS_FIELD_WIDTH; j++){
2706 if(j==0){
2707 sprintf(logsBuffer, " ");
2708 write(logs, logsBuffer, strlen(logsBuffer));
2709 }
2710 switch (session->field[i][j]){
2711 case 0:
2712 sprintf(logsBuffer, "%s", SYMBOL_GRASS);
2713 break;
2714 case 1:
2715 sprintf(logsBuffer, "%s", SYMBOL_WALL);
2716 break;
2717 case 2:
2718 sprintf(logsBuffer, "%c", session->locations[currLocation].name);
2719 currLocation += 1;
2720 break;
2721 case 3:
2722 sprintf(logsBuffer, "%s", SYMBOL_PACKET);
2723 break;
2724 case 4:
2725 sprintf(logsBuffer, "%s", SYMBOL_GRASS);
2726 break;
2727 }
2728 write(logs, logsBuffer, strlen(logsBuffer));
2729 }
2730 sprintf(logsBuffer, "\n");
2731 write(logs, logsBuffer, strlen(logsBuffer));
2732 }
2733 sprintf(logsBuffer, "\n");
2734 write(logs, logsBuffer, strlen(logsBuffer));
2735 pthread_mutex_unlock(&mutexLogs);
2736
2737 if((msgChat = allocMsg("Mappa per la sessione corrente generata!",
2738 * GRAPHICS_FG_COLOR_BLUE, false)) != NULL){
2739 pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
2740 }
2741 } else{
2742 return 1;
2743 }
2744
2745 return 0;
2746 }
2747
2748 /**
2749 * @param arg: NULL.
2750 *
2751 * @return: puntatore a void.
2752 *
2753 * La funzione sessionManagement si occupa delle operazioni di assegnazioni degli
2754 * esiti
2755 * di gioco e della disconnessione dei client a fine del tempo della sessione.
2756 *
2757 */

```

```

2758 void* sessionManagement(void* arg){
2759
2760 pthread_t tidMatchmaker;
2761 pthread_t tidUpdateChat;
2762 pthread_t tidTimeProvider;
2763 pthread_t tidUpdatePlayersInfo;
2764 LpClientInfoToJoin clientInfoToJoin;
2765 char msg[GRAPHICS_CHAT_WIDTH];
2766 int maxPackets = 0;
2767 int client = -1;
2768 int occurrence = 1;
2769 LpMsg msgChat;
2770 char logsBuffer[BUFFER_STRLEN];
2771 char* dynamicBuffer;
2772
2773 session->joinable = true;
2774 pthread_create(&tidMatchmaker, NULL, matchmaker, NULL);
2775 pthread_mutex_lock(&(session->mutexSession));
2776
2777 /* Il thread dorme finchè non entra un client */
2778 while(session->numOfPlayers < 1){
2779 pthread_cond_wait(&condSession, &(session->mutexSession));
2780 }
2781
2782 /* Appena entra un client in sessione avvia il thread che gestisce le operazioni di
2783 * durata di tempo della sessione */
2784 pthread_mutex_unlock(&(session->mutexSession));
2785 pthread_create(&tidTimeProvider, NULL, timeProvider, NULL);
2786 pthread_join(tidTimeProvider, NULL);
2787 session->joinable = false;
2788 pthread_cond_signal(&condMatchmaker);
2789 pthread_join(tidMatchmaker, NULL);
2790 pthread_mutex_lock(&mutexClientInfo);
2791 /* A fine del tempo verifica chi è il vincitore */
2792 for(int i=0; i<SESSION_PLAYERS_STRLEN; i++){
2793 if(session->clients[i] != NULL){
2794 if(session->clients[i]->packetsDelivered > maxPackets){
2795 maxPackets = session->clients[i]->packetsDelivered;
2796 occurrence = 1;
2797 client = i;
2798 } else if(session->clients[i]->packetsDelivered == maxPackets){
2799 occurrence += 1;
2800 }
2801 }
2802 }
2803 pthread_mutex_unlock(&mutexClientInfo);
2804 /* Se c'è un vincitore invia un messaggio di segnalazione a tutti i client nella
2805 * sessione di gioco */
2806 if(maxPackets > 0 && occurrence == 1){
2807 sprintf(msg, "$Server: Abbiamo un vincitore: %s!", session->clients[client]-
2808 >username);
2809 sendMsgToSession(NULL, msg);
2810 pthread_mutex_lock(&mutexLogs);
2811 sprintf(logsBuffer, " > [Sessione %d] Vincitore: %s -> Numero di pacchetti
2812 consegnati: %.d.\n", idSession, session->clients[client]->username, session-
2813 >clients[client]->packetsDelivered);
2814 if(write(logs, logsBuffer, strlen(logsBuffer)) == -1){
2815 if((msgChat = allocMsg("Errore durante la scrittura nel file di logs.",

```

```

 * GRAPHICS_FG_COLOR_RED, true)) != NULL){
2811 pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
2812 }
2813 }
2814 pthread_mutex_unlock(&mutexLogs);
2815 /* Altrimenti nel caso non sia stato raccolto un packet invia un messaggio di
 * segnalazione a tutti i client nella sessione di gioco */
2816 } else if(maxPackets == 0){
2817 sprintf(msg, "$Server: Che delusione, nessuno ha raccolto un pacco!");
2818 sendMsgToSession(NULL, msg);
2819 /* Altrimenti nel caso non ci sia un vincitore invia un messaggio di segnalazione
 a tutti i client nella sessione di gioco */
2820 } else if(maxPackets > 0 && occurrence > 1){
2821 sprintf(msg, "$Server: Non abbiamo nessun vincitore, partita equilibrata!");
2822 sendMsgToSession(NULL, msg);
2823 }
2824 sleep(3);
2825 sprintf(msg, "$Server: Disconnessione dalla sessione corrente in corso...");
2826 sendMsgToSession(NULL, msg);
2827 pthread_mutex_lock(&(session->mutexSession));
2828 /* Invia un messaggio di reset a tutti i client , e li disconnette dalla sessione
 rimettendoli nella coda di Join */
2829 for(int i=0; i<SESSION_PLAYERS_STRLEN; i++){
2830 if(session->clients[i] != NULL){
2831 session->clients[i]->havePacket = -1;
2832 sendMsgToSession(NULL, "$reset");
2833 sprintf(msg, "$disconnected %s", session->clients[i]->username);
2834 sendMsgToSession(NULL, msg);
2835 dynamicBuffer = (char*)calloc(GRAPHICS_CHAT_WIDTH, sizeof(char));
2836 strcpy(dynamicBuffer, msg);
2837 pthread_create(&tidUpdatePlayersInfo, NULL, updatePlayersInfo, dynamicBuffer);
2838 session->clients[i]->status = CLTINF_LOGGED;
2839 clientInfoToJoin = newClientInfoToJoin(session->clients[i]);
2840
2841 if(clientInfoToJoin != NULL){
2842 pthread_mutex_lock(&mutexClientInfo);
2843 enqueueClientInfoToJoin(&headClientInfoToJoinQueue,
2844 &tailClientInfoToJoinQueue, clientInfoToJoin);
2845 pthread_mutex_unlock(&mutexClientInfo);
2846 waitingClients += 1;
2847 pthread_mutex_lock(&mutexCursor);
2848 moveto(status[3].posX, status[3].posY);
2849 printf(" \u25cf Waiting: %8.2d", waitingClients);
2850 pthread_mutex_unlock(&mutexCursor);
2851 } else{
2852 close(session->clients[i]->clientSocket);
2853 }
2854 }
2855 /* Invia un messaggio a tutti i client connessi al server di una nuova rigenerazione
 di sessione */
2856 pthread_mutex_unlock(&(session->mutexSession));
2857 sprintf(msg, "$Server: Generazione di una nuova sessione in corso...");
2858 sendMsgToAll(NULL, msg);
2859 sleep(1);
2860 sprintf(msg, "$Server: In attesa che si liberi una sessione...");
2861 sendMsgToAll(NULL, msg);
2862 sleep(2);

```

```

2863
2864 }
2865
2866
2867 /**
2868 * @param arg : NULL.
2869 *
2870 * @return: puntatore a void.
2871 *
2872 * La funzione matchmaker si occupa di gestire delle operazioni
2873 * di inserimento dei client joinabili nella sessione.
2874 *
2875 */
2876 void* matchmaker(void* arg){
2877
2878 LpClientInfo clientToJoin;
2879 char msg[GRAPHICS_CHAT_WIDTH];
2880 bool joined = false;
2881 char* dynamicBuffer;
2882 pthread_t tidUpdatePlayersInfo;
2883
2884 while(true){
2885 pthread_mutex_lock(&session->mutexSession);
2886
2887 /* Il thread rimane in attesa appena si è raggiunto il numero massimo di player
2888 in sessione */
2889 while(session->numOfPlayers > SESSION_PLAYERS_STRLEN-1 && session->joinable !=
2890 false){
2891 pthread_cond_wait(&condMatchmaker, &(session->mutexSession));
2892 }
2893
2894 /* Se la sessione è joinable allora */
2895 if(session->joinable){
2896 pthread_mutex_lock(&mutexClientInfo);
2897 /* Verifica se sia presente un player nella coda Join , in caso di esito
2898 positivo lo inserisce in sessione */
2899 if(headClientInfoToJoinQueue != NULL){
2900 clientToJoin = dequeueClientInfoToJoin(&headClientInfoToJoinQueue,
2901 &tailClientInfoToJoinQueue);
2902 waitingClients -= 1;
2903 pthread_mutex_lock(&mutexCursor);
2904 moveTo(status[3].posX, status[3].posY);
2905 printf(" \u25cf Waiting: %8.2d", waitingClients);
2906 pthread_mutex_unlock(&mutexCursor);
2907 clientToJoin->status = CLTINF_PLAYING;
2908 clientToJoin->packetsDelivered = 0;
2909 sendMsg(clientToJoin, "$remove blink");
2910 sendMsg(clientToJoin, "$reset");
2911 sleep(2);
2912 sendMsg(clientToJoin, "$Server: Che il gioco abbia inizio! Buon
2913 divertimento!");
2914 insertClientInSession(session, clientToJoin);
2915 joined = true;
2916 pthread_cond_signal(&condSession);
2917 }
2918 pthread_mutex_unlock(&mutexClientInfo);
2919 /* Informa al nuovo client nella sessione tutti gli utenti attualmente in
2920 sessione */
2921 . . .
2922 }

```

```

2915 if(joined){
2916 for(int i=0; i<SESSION_PLAYERS_STRLEN; i++){
2917 if(session->clients[i] != NULL && session->clients[i] != clientToJoin){
2918 sprintf(msg, "$connected %s %d", session->clients[i]->username, session-
2919 *->clients[i]->packetsDelivered);
2920 sendMsg(clientToJoin, msg);
2921 }
2922 /* Invia un messaggio a tutti gli utenti in sessione dell'aggiunta del nuovo
2923 * client alla sessione */
2924 sprintf(msg, "$connected %s %d", clientToJoin->username, clientToJoin-
2925 *->packetsDelivered);
2926 dynamicBuffer = (char*)calloc(GRAPHICS_CHAT_WIDTH, sizeof(char));
2927 strcpy(dynamicBuffer, msg);
2928 pthread_create(&tidUpdatePlayersInfo, NULL, updatePlayersInfo, dynamicBuffer);
2929 sendBasicInformation(clientToJoin);
2930 sendMsgToSession(NULL, msg);
2931 sprintf(msg, "$Server: Tremate, %s si è unito alla sessione!", clientToJoin-
2932 *->username);
2933 sendMsgToSession(clientToJoin, msg);
2934 joined = false;
2935 }
2936 pthread_mutex_unlock(&session->mutexSession);
2937 } else{
2938 pthread_mutex_unlock(&session->mutexSession);
2939 pthread_exit(NULL);
2940 }
2941
2942 /**
2943 * @param signal: segnale catturato
2944 *
2945 * @return void.
2946 *
2947 * La funzione signalHandler cattura i segnali in questione
2948 * e ne detta il comportamento al loro verificarsi.
2949 *
2950 */
2951 void signalHandler(int signal){

2952
2953 char msg[GRAPHICS_CHAT_WIDTH];
2954 LpMsg msgChat;
2955 pthread_t tidUpdateChat;
2956 LpClientInfo tmp;
2957
2958 switch(signal){
2959 case SIGINT:
2960 sprintf(msg, "Disconnessione del server in corso!");
2961 if((msgChat = allocMsg(msg, GRAPHICS_FG_COLOR_RED, true))){
2962 pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
2963 }
2964 close(listenerSocket);
2965 close(logs);
2966 close(database);
2967 close(fieldGenerator);
2968 tmp = listClientInfo;
2969 free(listClientInfo);
2970 exit(0);
2971 }
2972 }

```

```

2959 write(tmp != NULL){
2960 close(tmp->clientSocket);
2961 tmp = tmp->nextClientInfo;
2962 }
2963 sleep(2);
2964 clean();
2965 setCursor(true);
2966 raise(SIGTERM);
2967 break;
2968 case SIGSTOP:
2969 sprintf(msg, "Disconnessione del server in corso!");
2970 if((msgChat = allocMsg(msg, GRAPHICS_FG_COLOR_RED, true))){
2971 pthread_create(&tidUpdateChat, NULL, updateChat, msgChat);
2972 }
2973 close(listenerSocket);
2974 close(logs);
2975 close(database);
2976 close(fieldGenerator);
2977 tmp = listClientInfo;
2978 while(tmp != NULL){
2979 close(tmp->clientSocket);
2980 tmp = tmp->nextClientInfo;
2981 }
2982 sleep(2);
2983 clean();
2984 setCursor(true);
2985 raise(SIGTERM);
2986 break;
2987 }
2988 }
2989 */
2990 /****** END SERVER
2991 *****/

```

```
1 |
2 | **** INIT GRAPHICS HEADER
3 | **** /
4 |
5 |
6 |
7 |
8 | /* Altezza e Larghezza chat */
9 | #define GRAPHICS_CHAT_HEIGHT 9
10 | #define GRAPHICS_CHAT_WIDTH 82
11 |
12 | /* Altezza del box status del server */
13 | #define GRAPHICS_STATUS_HEIGHT 7
14 |
15 | /* Altezza e Larghezza screen */
16 | #define GRAPHICS_SCREEN_HEIGHT 37
17 | #define GRAPHICS_SCREEN_WIDTH 97
18 |
19 | /* Altezza e larghezza playground */
20 | #define GRAPHICS_FIELD_HEIGHT 15
21 | #define GRAPHICS_FIELD_WIDTH 57
22 |
23 | /* Altezza dell'info-box dei players */
24 | #define GRAPHICS_PLAYERSINFO_HEIGHT 5
25 |
26 | /* Codici stile testo */
27 | #define GRAPHICS_TEXT_DEFAULT 0
28 | #define GRAPHICS_TEXT_BOLD 1
29 | #define GRAPHICS_TEXT_UNDERSCORE 4
30 |
31 | /* Codici colore background */
32 | #define GRAPHICS_FG_COLOR_BLACK 30
33 | #define GRAPHICS_FG_COLOR_RED 31
34 | #define GRAPHICS_FG_COLOR_GREEN 32
35 | #define GRAPHICS_FG_COLOR_YELLOW 33
36 | #define GRAPHICS_FG_COLOR_BLUE 34
37 | #define GRAPHICS_FG_COLOR_MAGENTA 35
38 | #define GRAPHICS_FG_COLOR_CYAN 36
39 | #define GRAPHICS_FG_COLOR_WHITE 37
40 |
41 |
42 | /* Codici colore background */
43 | #define GRAPHICS_BG_COLOR_BLACK 40
44 | #define GRAPHICS_BG_COLOR_RED 41
45 | #define GRAPHICS_BG_COLOR_GREEN 42
46 | #define GRAPHICS_BG_COLOR_YELLOW 43
47 | #define GRAPHICS_BG_COLOR_BLUE 44
48 | #define GRAPHICS_BG_COLOR_MAGENTA 45
49 | #define GRAPHICS_BG_COLOR_CYAN 46
50 | #define GRAPHICS_BG_COLOR_WHITE 47
51 |
52 |
53 | /* Lunghezza nome utente */
54 | #define GRAPHICS_USERNAME_STRLEN 10
55 |
56 | /* Pin di aggancio per la scansione dell'hud */
57 | #define GRAPHICS_PIN_FIELD 'x'
```

```

58 #define GRAPHICS_PIN_CHAT '!'
59 #define GRAPHICS_PIN_TIMER '#'
60 #define GRAPHICS_PIN_COUNTER '@'
61 #define GRAPHICS_PIN_PLAYERS '+'
62 #define GRAPHICS_PIN_STATUS '?'
63 #define GRAPHICS_PIN_START '>'
64
65 #define GRAPHICS_KEY_ENTER 10
66 #define GRAPHICS_KEY_BACKSPACE 127
67
68 #define GRAPHICS_TEXTFIELD_STRLEN 70
69
70
71 extern int defaultColorBackground;
72 extern int defaultColorForeground;
73
74
75
76 /**
77 * @attr posX: Posizione X nel terminale.
78 * @attr posY: Posizione Y nel terminale.
79 *
80 */
81 struct Coord{
82 int posX;
83 int posY;
84 };
85 typedef struct Coord Coord;
86
87
88 /**
89 * @attr posX: Posizione X nel terminale.
90 * @attr posY: Posizione Y nel terminale.
91 *
92 */
93 struct PlayersCounter{
94 Coord coords;
95 int value;
96 };
97 typedef struct PlayersCounter PlayersCounter;
98
99
100 /**
101 * @attr coords: Posizione della componente nel terminale.
102 * @attr isEmpty: Flag che indica se la box è piena oppure vuota.
103 * @attr color: Colore di stampa del messaggio.
104 * @attr msg: Messaggio contenuto nel box.
105 *
106 */
107 struct MsgBox{
108 Coord coords;
109 bool isEmpty;
110 int color;
111 char msg[GRAPHICS_CHAT_WIDTH];
112 };
113 typedef struct MsgBox MsgBox;
114
115

```

```

116 /**
117 * @attr msgBox[]: Array di box messaggi di cui è composta la chat.
118 * @attr nextMsgPosition: Posizione prevista per il prossimo messaggio in arrivo.
119 *
120 */
121 struct Chat{
122 MsgBox msgBox[GRAPHICS_CHAT_HEIGHT];
123 int currMsgPosition;
124 };
125 typedef struct Chat Chat;
126
127 /**
128 * @attr coords: Posizione della componente nel terminale.
129 * @attr packetsDelivered: Numero di pacchetti consegnati.
130 * @attr username: Username del player.
131 *
132 */
133 struct PlayerBox{
134 Coord coords;
135 int packetsDelivered;
136 char username[GRAPHICS_USERNAME_STRLEN];
137 };
138 typedef struct PlayerBox PlayerBox;
139
140
141 /**
142 * @attr playerBox[]: Array di box player di cui è composta l'info-box dei players
143 * connessi alla sessione.
144 * @attr currPlayerPosition: Posizione del "cursore" nell'info-box dei players
145 * connessi alla sessione.
146 *
147 */
148 struct PlayersInfo{
149 PlayerBox playerBox[GRAPHICS_PLAYERSINFO_HEIGHT];
150 int currPlayerPosition;
151 };
152 typedef struct PlayersInfo PlayersInfo;
153
154 /**
155 * @attr msg: Messaggio da stampare sulla chat.
156 * @attr color: Colore del testo.
157 * @attr error: Indica se il messaggio è una notifica di errore.
158 *
159 */
160 struct Msg{
161 char msg[GRAPHICS_CHAT_WIDTH];
162 int color;
163 bool error;
164 };
165 typedef struct Msg Msg;
166 typedef struct Msg* LpMsg;
167
168 /**
169 * @param void.
170 * @return typed: Carattere premuto dall'utente.
171 */

```

```
172 *
173 * La funzione getch cattura il carattere premuto dall'utente e non
174 * ne effettua l'echo sul terminale.
175 */
176 int getch(void);
177
178
179 /**
180 * @param posX: Posizione X nel terminale (colonna).
181 * @param posY: Posizione Y nel terminale (riga).
182 * @return void.
183 *
184 * La funzione moveto sposta il cursore alla posizione indicata.
185 */
186 void moveto(int, int);
187
188
189 /**
190 * @param setted: Valore booleano -> true: visibile | false: invisibile.
191 * @return void.
192 *
193 * La funzione setCursor imposta il cursore a visibile o invisibile
194 * a seconda del valore del parametro setted.
195 */
196 void setCursor(bool);
197
198
199 /**
200 * @param void.
201 * @return void.
202 *
203 * La funzione beep emette il suono predefinito dal terminale.
204 */
205 void beep(void);
206
207
208 /**
209 * @param void.
210 * @return void.
211 *
212 * La funzione clean pulisce il terminale.
213 */
214 void clean(void);
215
216
217 /**
218 * @param foreground: Colore da impostare per il foreground.
219 * @param background: Colore da impostare per il background.
220 * @return void.
221 *
222 * La funzione setColor imposta il colore del foreground e quello
223 * del background del terminale. Se il colore non è riconosciuto
224 * allora viene impostato quello di default.
225 */
226 void setColor(int, int);
227
228
229 /**
230 * @param void
```

```
228 * @param color.
229 * @return void.
230 *
231 * La funzione reset imposta il colore del foreground e quello
232 * del background a default.
233 */
234 void reset(void);
235
236
237
238
239 /**
240 * @param color: Colore di default per il foreground.
241 * @return void.
242 *
243 * La funzione setDefaultForeground imposta il colore di default
244 * del testo (foreground) del terminale.
245 */
246 void setDefaultForeground(int);
247
248
249 /**
250 * @param color: Colore di default per il background.
251 * @return void.
252 *
253 * La funzione setDefaultBackground imposta il colore di default
254 * dello sfondo (background) del terminale.
255 */
256 void setDefaultBackground(int);
257
258
259 /**
260 * @param rows: Numero di righe della finestra.
261 * @param cols: Numero di colonne della finestra.
262 * @return void.
263 *
264 * La funzione resizeTerminal effettua il ridimensionamento
265 * del terminale.
266 */
267 void resizeTerminal(int, int);
268
269
270 /**
271 * @param style: Stile del testo.
272 * @return void.
273 *
274 * La funzione setTextStyle imposta lo stile del testo.
275 */
276 void setTextStyle(int);
277
278 **** END GRAPHICS HEADER ****
279
```

```
1 **** INIT GRAPHICS
2 ****
3
4 #include <stdio.h>
5 #include <termios.h>
6 #include <unistd.h>
7 #include <pthread.h>
8 #include "graphics.h"
9
10
11
12 int defaultColorForeground = GRAPHICS_FG_COLOR_BLACK;
13 int defaultColorBackground = GRAPHICS_BG_COLOR_WHITE;
14 pthread_t tidNoechoThread = 0;
15
16
17
18 /**
19 * @param color: Colore di default per il foreground.
20 * @return void.
21 *
22 * La funzione setDefaultForeground imposta il colore di default
23 * del testo (foreground) del terminale.
24 */
25 void setDefaultForeground(int color){
26 defaultColorForeground = color;
27 }
28
29
30 /**
31 * @param color: Colore di default per il background.
32 * @return void.
33 *
34 * La funzione setDefaultBackground imposta il colore di default
35 * dello sfondo (background) del terminale.
36 */
37 void setDefaultBackground(int color){
38 defaultColorBackground = color;
39 }
40
41
42 /**
43 * @param foreground: Colore da impostare per il foreground.
44 * @param background: Colore da impostare per il background.
45 * @return void.
46 *
47 * La funzione setColor imposta il colore del foreground e quello
48 * del background del terminale. Se il colore non è riconosciuto
49 * allora viene impostato quello di default.
50 */
51 void setColor(int foreground, int background){
52 if(foreground >= GRAPHICS_FG_COLOR_BLACK && foreground <= GRAPHICS_FG_COLOR_WHITE){
53 printf("\033[%dm", foreground);
54 } else{
55 printf("\033[%dm", defaultColorForeground);
56 }
57 if(background >= GRAPHICS_BG_COLOR_BLACK && background <= GRAPHICS_BG_COLOR_WHITE){
```

```

58 printf("\033[%dm", background);
59 } else{
60 printf("\033[%dm", defaultColorBackground);
61 }
62 fflush(stdout);
63 }
64
65
66 /**
67 * @param void.
68 * @return void.
69 *
70 * La funzione reset imposta il colore del foreground e quello
71 * del background a default.
72 */
73 void reset(void){
74 setColor(defaultColorForeground, defaultColorBackground);
75 }
76
77
78 /**
79 * @param void.
80 * @return typed: Carattere premuto dall'utente.
81 *
82 * La funzione getch cattura il carattere premuto dall'utente e non
83 * ne effettua l'echo sul terminale.
84 */
85 int getch(void){
86 struct termios oldattr, newattr;
87 int typed;
88 tcgetattr(STDIN_FILENO, &oldattr);
89 newattr = oldattr;
90 newattr.c_lflag &= ~(ICANON | ECHO);
91 tcsetattr(STDIN_FILENO, TCSANOW, &newattr);
92 typed = getchar();
93 tcsetattr(STDIN_FILENO, TCSANOW, &oldattr);
94 return typed;
95 }
96
97
98 /**
99 * @param posX: Posizione X nel terminale (colonna).
100 * @param posY: Posizione Y nel terminale (riga).
101 * @return void.
102 *
103 * La funzione moveto sposta il cursore alla posizione indicata.
104 */
105 void moveto(int posX, int posY){
106 printf("\033[%d;%dH", posY, posX);
107 fflush(stdout);
108 }
109
110
111 /**
112 * @param setted: Valore booleano -> true: visibile | false: invisibile.
113 * @return void.
114 *
115 * La funzione setCursor imposta il cursore a visibile o invisibile

```

```

116 * a seconda del valore del parametro setted.
117 */
118 void setCursor(bool setted){
119 if(setted){
120 printf("\e[?25h");
121 } else{
122 printf("\e[?25l");
123 }
124 fflush(stdout);
125 }
126
127
128 /**
129 * @param void.
130 * @return void.
131 *
132 * La funzione beep emette il suono predefinito dal terminale.
133 */
134 void beep(void){
135 printf("\a");
136 fflush(stdout);
137 }
138
139
140 /**
141 * @param void.
142 * @return void.
143 *
144 * La funzione clean pulisce il terminale.
145 */
146 void clean(void){
147 printf("\033[2J");
148 moveto(1,1);
149 fflush(stdout);
150 }
151
152
153 /**
154 * @param rows: Numero di righe della finestra.
155 * @param cols: Numero di colonne della finestra.
156 * @return void.
157 *
158 * La funzione resizeTerminal effettua il ridimensionamento
159 * del terminale.
160 */
161 void resizeTerminal(int rows, int cols){
162 printf("\e[8;%d;%dt", rows, cols);
163 fflush(stdout);
164 }
165
166
167 /**
168 * @param setted: Valore booleano -> true: underscore | false: no underscore.
169 * @return void.
170 *
171 * La funzione setUnderscore imposta l'underscore del testo a true oppure
172 * a false.
173 */

```

```
174 void setTextStyle(int style){
175 if(style == GRAPHICS_TEXT_DEFAULT || style == GRAPHICS_TEXT_BOLD
176 || style == GRAPHICS_TEXT_UNDERSCORE){
177 printf("\033[%dm", style);
178 } else{
179 printf("\033[%dm", GRAPHICS_TEXT_DEFAULT);
180 }
181 }
182
183 /****** END GRAPHICS
184 *****/
```

```

1 **** INIT SESSION HEADER ****
2
3
4 #include "../cltinf/clientinfo.h"
5 #include "../graphics/graphics.h"
6 #include <stdbool.h>
7 #include <pthread.h>
8
9
10
11 #define SESSION_PLAYERS_STRLEN 5
12 #define SESSION_PACKETS_STRLEN 21
13 #define SESSION_LOCATIONS_STRLEN 5
14 #define SESSION_SPAWN_STRLEN 8
15
16 #define SESSION_GRASS_VALUE 0
17 #define SESSION_WALL_VALUE 1
18 #define SESSION_LOCATION_VALUE 2
19 #define SESSION_PACKET_VALUE 3
20 #define SESSION_PLAYER_VALUE 4
21
22
23
24 /**
25 * @attribute currRows: Riga corrente.
26 * @attribute currCols: Colonna corrente.
27 */
28
29 struct Spawn{
30 int currRows;
31 int currCols;
32 };
33 typedef struct Spawn Spawn;
34
35
36 /**
37 * @attribute currRows: Riga corrente.
38 * @attribute currCols: Colonna corrente.
39 * @attribute name: Nome della locazione.
40 */
41
42 struct Location{
43 int currRows;
44 int currCols;
45 char name;
46 };
47 typedef struct Location Location;
48
49
50
51 /**
52 * @attribute currRows: Riga corrente.
53 * @attribute currCols: Colonna corrente.
54 * @attribute location: Locazione a cui trasportare il pacchetto.
55 * @attribute delivered: Flag che indica se il pacchetto è stato consegnato.
56 */
57
```

```

58 struct Packet{
59 int currRows;
60 int currCols;
61 int location;
62 bool delivered;
63 };
64 typedef struct Packet Packet;
65
66
67
68 /**
69 * @attribute clients: Clients connessi alla sessione.
70 * @attribute field: Campo di gioco.
71 * @attribute packets: Pacchetti della sessione.
72 * @attribute locations: Locazioni della sessione.
73 * @attribute spawn: Spawns della sessione.
74 * @attribute joinable: Flag che indica se la sessione è accessibile.
75 * @attribute numOfPlayers: Numero di giocatori correnti.
76 * @attribute numOfPackets: Numero di pacchetti correnti.
77 * @attribute mutexSession: mutex di sessione.
78 */
79
80 struct Session{
81 LpClientInfo clients[SESSION_PLAYERS_STRLEN];
82 int field[GRAPHICS_FIELD_HEIGHT][GRAPHICS_FIELD_WIDTH];
83 Packet packets[SESSION_PACKETS_STRLEN];
84 Location locations[SESSION_LOCATIONS_STRLEN];
85 Spawn spawn[SESSION_SPAWN_STRLEN];
86 bool joinable;
87 int numOfPlayers;
88 int numOfPackets;
89 pthread_mutex_t mutexSession;
90 };
91 typedef struct Session Session;
92 typedef Session* LpSession;
93
94
95
96 /**
97 * @param void.
98 *
99 * @return LpSession: Puntatore alla sessione allocata.
100 *
101 * La funzione newSession alloca una nuova sessione e la inizializza.
102 */
103
104 LpSession newSession();
105
106
107
108 /**
109 * @param session: Puntatore alla sessione.
110 * @param clientInfo: Puntatore al ClientInfo da eliminare dalla sessione.
111 *
112 * @return void.
113 *
114 * La funzione deleteClientFromSession elimina dalla lista dei clients della sessione
115 * il client specificato dal puntatore clientInfo.

```

```
116 */
117
118 void deleteClientFromSession(LpSession, LpClientInfo);
119
120
121
122 /**
123 * @param session: Puntatore alla sessione.
124 * @param clientInfo: Puntatore al ClientInfo da inserire dalla sessione.
125 *
126 * @return void.
127 *
128 * La funzione insertClientInSession inserisce nella lista dei clients della sessione
129 * il client specificato dal puntatore clientInfo.
130 */
131
132 void insertClientInSession(LpSession, LpClientInfo);
133
134 /***** END SESSION HEADER
135 *****/
```

```

1 **** INIT SESSION
2 ****/
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <string.h>
7 #include <sys/types.h>
8 #include "session.h"
9
10
11
12 /**
13 * @param void.
14 *
15 * @return LpSession: Puntatore alla sessione allocata.
16 *
17 * La funzione newSession alloca una nuova sessione e la inizializza.
18 */
19
20 LpSession newSession(){
21 LpSession newSession = (LpSession)malloc(sizeof(Session));
22 if(newSession != NULL){
23 for(int i=0; i<SESSION_PLAYERS_STRLEN; i++){
24 newSession->clients[i] = NULL;
25 }
26 for(int i=0; i<SESSION_PACKETS_STRLEN; i++){
27 newSession->packets[i].delivered = false;
28 }
29 for(int i=0; i<GRAPHICS_FIELD_HEIGHT; i++){
30 for(int j=0; j<GRAPHICS_FIELD_WIDTH; j++){
31 newSession->field[i][j] = SESSION_GRASS_VALUE;
32 }
33 }
34 newSession->numOfPlayers = 0;
35 pthread_mutex_init(&(newSession->mutexSession), NULL);
36 }
37 return newSession;
38 }
39
40
41
42 /**
43 * @param session: Puntatore alla sessione.
44 * @param clientInfo: Puntatore al ClientInfo da eliminare dalla sessione.
45 *
46 * @return void.
47 *
48 * La funzione deleteClientFromSession elimina dalla lista dei clients della sessione
49 * il client specificato dal puntatore clientInfo.
50 */
51
52 void deleteClientFromSession(LpSession session, LpClientInfo clientInfo){
53 for(int i=0; i<SESSION_PLAYERS_STRLEN; i++){
54 if(session->clients[i] == clientInfo){
55 session->numOfPlayers -= 1;
56 session->clients[i] = NULL;
57 break;
58 }
59 }
60 }

```

```
58 }
59 }
60 }
61
62
63
64 /**
65 * @param session: Puntatore alla sessione.
66 * @param clientInfo: Puntatore al ClientInfo da inserire dalla sessione.
67 *
68 * @return void.
69 *
70 * La funzione insertClientInSession inserisce nella lista dei clients della sessione
71 * il client specificato dal puntatore clientInfo.
72 */
73
74 void insertClientInSession(LpSession session, LpClientInfo clientInfo){
75 for(int i=0; i<SESSION_PLAYERS_STRLEN; i++){
76 if(session->clients[i] == NULL){
77 session->numOfPlayers += 1;
78 session->clients[i] = clientInfo;
79 break;
80 }
81 }
82 }
83
84 **** END SESSION
85 ****/
```

```

1 **** INIT CLTINF HEADER
2 ****/
3
4 #include <netinet/in.h>
5 #include <time.h>
6 #include <stdbool.h>
7 #include <pthread.h>
8
9 #define CLTINF_USERNAME_STRLEN 10
10 #define CLTINF_PASSWORD_STRLEN 10
11
12 #define CLTINF_GUEST 1
13 #define CLTINF_REGISTERED 2
14 #define CLTINF_LOGGED 3
15 #define CLTINF_PLAYING 4
16
17
18
19 /**
20 * @attribute tidHandler: TID del thread principale che gestisce un determinato
21 * client.
22 * @attribute clientSocket: Socket descriptor della socket associata ad un
23 * determinato client.
24 * @attribute clientAddressIPv4: IPv4 del client.
25 * @attribute username: Username del client.
26 * @attribute timestamp: Ora della connessione
27 * @attribute stato: LOGGED, REGISTERED, GUEST.
28 * @attribute prevClientInfo: Puntatore al nodo clientInfo precedente.
29 * @attribute nextClientInfo: Puntatore al nodo clientInfo successivo.
30 * @attribute mutexSocket: Mutex associato alla socket.
31 */
32
33 struct ClientInfo{
34 pthread_t tidHandler;
35 int clientSocket;
36 char clientAddressIPv4[INET_ADDRSTRLEN];
37 char username[CLTINF_USERNAME_STRLEN];
38 time_t timestamp;
39 int packetsDelivered;
40 int status;
41 int currRows;
42 int currCols;
43 int havePacket;
44 pthread_mutex_t mutexSocket;
45 struct ClientInfo* prevClientInfo;
46 struct ClientInfo* nextClientInfo;
47 };
48
49
50
51 /**
52 * @attribute clientInfo: Puntatore alla struttura clientInfo.
53 * @attribute prevClientInfoToJoin: Puntatore al nodo ClientInfoToJoin precedente.
54 * @attribute nextClientInfoToJoin: Puntatore al nodo ClientInfoToJoin successivo.
55 *

```

```

56 */
57
58 struct ClientInfoToJoin{
59 LpClientInfo clientInfo;
60 struct ClientInfoToJoin* nextClientInfoToJoin;
61 };
62 typedef struct ClientInfoToJoin ClientInfoToJoin;
63 typedef ClientInfoToJoin* LpClientInfoToJoin;
64
65
66
67 /**
68 * @param clientInfo: Puntatore alla struttura clientInfo associata.
69 *
70 * @return newClientInfoToJoin: Puntatore al nodo newClientInfoToJoin allocato.
71 *
72 * La funzione newClientInfoToJoin alloca un nuovo nodo ClientInfoToJoin con gli
73 * attributi passati
74 * in ingresso.
75 */
76
77
78
79
80 /**
81 * @param headClientInfoToJoin: Puntatore alla testa della coda di ClientInfoToJoin.
82 *
83 * @return bool: true (vuota) | false (non vuota).
84 *
85 * La funzione isEmptyClientInfoToJoin verifica se la coda passata in ingresso
86 * è vuota oppure no.
87 */
88
89 bool isEmptyClientInfoToJoin(LpClientInfoToJoin);
90
91
92
93 /**
94 * @param headClientInfoToJoin: Doppio puntatore alla testa della coda di
95 * ClientInfoToJoin.
96 * @param tailClientInfoToJoin: Doppio puntatore alla coda della coda di
97 * ClientInfoToJoin.
98 * @param newClientInfoToJoin: Puntatore al nodo ClientInfoToJoin da accodare.
99 *
100 * @return void.
101 *
102 * La funzione enqueueClientInfoToJoin accoda il nodo ClientInfoToJoin passato in
103 * ingresso.
104 */
105
106
107 void enqueueClientInfoToJoin(LpClientInfoToJoin*, LpClientInfoToJoin*,
108 LpClientInfoToJoin);
109
110
111 /**
112 * @param headClientInfoToJoin: Doppio puntatore alla testa della coda di

```



```
159 */
160
161 void insertClientInfo(LpClientInfo*, LpClientInfo);
162
163
164
165 /**
166 * @param targetedClientInfo: Doppio puntatore al nodo clientInfo da eliminare.
167 *
168 * @return void.
169 *
170 * La funzione deleteClientInfo elimina il nodo specificato da targetedClientInfo
171 * dalla
172 * Lista in cui risiede lo stesso.
173 */
174
175
176 /***** END CLTINF HEADER
177 *****/
```

```

1 **** INIT CLTINF
2 ****
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <string.h>
7 #include <sys/types.h>
8 #include "clientinfo.h"
9
10
11
12 /**
13 * @param clientInfo: Puntatore alla struttura clientInfo associata.
14 *
15 * @return newClientInfoToJoin: Puntatore al nodo newClientInfoToJoin allocato.
16 *
17 * La funzione newClientInfoToJoin alloca un nuovo nodo ClientInfoToJoin con gli
18 * attributi passati
19 * in ingresso.
20 */
21
22 LpClientInfoToJoin newClientInfoToJoin(LpClientInfo clientInfo){
23 LpClientInfoToJoin newClientInfoToJoin =
24 (LpClientInfoToJoin)malloc(sizeof(ClientInfoToJoin));
25 if(newClientInfoToJoin != NULL){
26 newClientInfoToJoin->clientInfo = clientInfo;
27 newClientInfoToJoin->nextClientInfoToJoin = NULL;
28 }
29 return newClientInfoToJoin;
30 }
31
32
33 /**
34 * @param headClientInfoToJoin: Puntatore alla testa della coda di ClientInfoToJoin.
35 *
36 * @return bool: true (vuota) | false (non vuota).
37 *
38 * La funzione isEmptyClientInfoToJoin verifica se la coda passata in ingresso
39 * è vuota oppure no.
40 */
41
42 bool isEmptyClientInfoToJoin(LpClientInfoToJoin headClientInfoToJoin){
43 if(headClientInfoToJoin != NULL){
44 return false;
45 }
46 return true;
47 }
48
49
50 /**
51 * @param headClientInfoToJoin: Doppio puntatore alla testa della coda di
52 * ClientInfoToJoin.
53 * @param tailClientInfoToJoin: Doppio puntatore alla coda della coda di
54 * ClientInfoToJoin.
55 * @param newClientInfoToJoin: Puntatore al nodo ClientInfoToJoin da accodare.

```

```

54 *
55 * @return void.
56 *
57 * La funzione enqueueClientInfoToJoin accoda il nodo ClientInfoToJoin passato in
58 * ingresso.
59 */
60
61 void enqueueClientInfoToJoin(LpClientInfoToJoin* headClientInfoToJoin,
62 LpClientInfoToJoin* tailClientInfoToJoin, LpClientInfoToJoin newClientInfoToJoin){
63 if(newClientInfoToJoin != NULL){
64 if(isEmptyClientInfoToJoin(*headClientInfoToJoin)){
65 *headClientInfoToJoin = newClientInfoToJoin;
66 } else{
67 (*tailClientInfoToJoin)->nextClientInfoToJoin = newClientInfoToJoin;
68 }
69 }
70
71
72 /**
73 * @param headClientInfoToJoin: Doppio puntatore alla testa della coda di
74 * ClientInfoToJoin.
75 * @param tailClientInfoToJoin: Doppio puntatore alla coda della coda di
76 * ClientInfoToJoin.
77 *
78 * @return LpClientInfo: Puntatore alla struttura ClientInfo associata al nodo
79 * eliminato.
80 *
81 * La funzione dequeueClientInfoToJoin deaccoda la testa della coda.
82 */
83
84 LpClientInfo dequeueClientInfoToJoin(LpClientInfoToJoin* headClientInfoToJoin,
85 LpClientInfoToJoin* tailClientInfoToJoin){
86 LpClientInfo tmp = NULL;
87 if(!isEmptyClientInfoToJoin(*headClientInfoToJoin)){
88 tmp = (*headClientInfoToJoin)->clientInfo;
89 LpClientInfoToJoin tmpClientInfoToJoin = *headClientInfoToJoin;
90 *headClientInfoToJoin = (*headClientInfoToJoin)->nextClientInfoToJoin;
91 if(isEmptyClientInfoToJoin(*headClientInfoToJoin)){
92 *tailClientInfoToJoin = NULL;
93 }
94 free(tmpClientInfoToJoin);
95 }
96 return tmp;
97 }
98
99 /**
100 * @param headClientInfoToJoin: Doppio puntatore alla testa della coda di
101 * ClientInfoToJoin.
102 * @param tailClientInfoToJoin: Doppio puntatore alla coda della coda di
103 * ClientInfoToJoin.
104 * @param targetedClientInfo: Doppio puntatore al nodo clientInfo da eliminare.
105 *
106 * @return void.

```

```

104 *
105 * La funzione deleteClientInfoToJoin elimina il nodo specificato da
106 * targetedClientInfo dalla
107 * coda in cui risiede lo stesso.
108 */
109
110 void deleteClientInfoToJoin(LpClientInfoToJoin* headClientInfoToJoin,
111 LpClientInfoToJoin* tailClientInfoToJoin, LpClientInfo targetedClientInfo){
112 LpClientInfoToJoin tmp = *headClientInfoToJoin;
113 LpClientInfoToJoin prevClientInfoToJoin = NULL;
114 while(tmp != NULL && tmp->clientInfo != targetedClientInfo){
115 prevClientInfoToJoin = tmp;
116 tmp = tmp->nextClientInfoToJoin;
117 }
118 if(tmp != NULL){
119 if(tmp == *headClientInfoToJoin && tmp == *tailClientInfoToJoin){
120 *headClientInfoToJoin = NULL;
121 *tailClientInfoToJoin = NULL;
122 } else{
123 if(tmp == *headClientInfoToJoin){
124 *headClientInfoToJoin = tmp->nextClientInfoToJoin;
125 }
126 if(tmp == *tailClientInfoToJoin){
127 *tailClientInfoToJoin = prevClientInfoToJoin;
128 }
129 if(prevClientInfoToJoin != NULL){
130 prevClientInfoToJoin->nextClientInfoToJoin = tmp->nextClientInfoToJoin;
131 }
132 free(tmp);
133 }
134 }
135
136
137 /**
138 * @param tidHandler: TID del thread principale che gestisce un determinato client.
139 * @param clientSocket: Socket descriptor della socket associata ad un determinato
140 * client.
141 * @param clientAddressIPv4: IPv4 del client.
142 * @param username: Username del client.
143 *
144 * @return newClientInfo: Puntatore al nodo clientInfo allocato.
145 *
146 * La funzione newClientInfo alloca un nuovo nodo clientInfo con gli attributi
147 * passati
148 * in ingresso.
149 */
150
151 LpClientInfo newClientInfo(int clientSocket, char clientAddressIPv4[], char
152 username[]){
153 LpClientInfo newClientInfo = (LpClientInfo)malloc(sizeof(ClientInfo));
154 if(newClientInfo != NULL){
155 newClientInfo->clientSocket = clientSocket;
156 strcpy(newClientInfo->clientAddressIPv4, clientAddressIPv4);
157 strcpy(newClientInfo->username, username);
158 newClientInfo->timestamp = time(NULL);
159 newClientInfo->status = CLTINF_GUEST;
160 ...
161 }

```

```

157 newClientInfo->havePacket = -1;
158 newClientInfo->prevClientInfo = NULL;
159 newClientInfo->nextClientInfo = NULL;
160 pthread_mutex_init(&(newClientInfo->mutexSocket), NULL);
161 }
162 return newClientInfo;
163 }
164
165
166
167 /**
168 * @param ListClientInfo: Doppio puntatore alla lista doppiamente concatenata di
169 * clientInfo.
170 * @param newClientInfo: Nodo clientInfo da inserire.
171 *
172 * @return void.
173 *
174 * La funzione insertClientInfo inserisce un nuovo nodo clientInfo in testa alla
175 * lista
176 * ListClientInfo.
177 */
178
179 void insertClientInfo(LpClientInfo* listClientInfo, LpClientInfo newClientInfo){
180 if(newClientInfo != NULL){
181 newClientInfo->nextClientInfo = *listClientInfo;
182 if(*listClientInfo != NULL){
183 newClientInfo->prevClientInfo = (*listClientInfo)->prevClientInfo;
184 (*listClientInfo)->prevClientInfo = newClientInfo;
185 if(newClientInfo->prevClientInfo != NULL){
186 newClientInfo->prevClientInfo->nextClientInfo = newClientInfo;
187 }
188 }
189 }
190
191
192
193 /**
194 * @param targetedClientInfo: Doppio puntatore al nodo clientInfo da eliminare.
195 *
196 * @return void.
197 *
198 * La funzione deleteClientInfo elimina il nodo specificato da targetedClientInfo
199 * dalla
200 * lista in cui risiede lo stesso.
201 */
202
203 void deleteClientInfo(LpClientInfo* listClientInfo, LpClientInfo* targetedClientInfo){
204 if(targetedClientInfo != NULL){
205 LpClientInfo tmp = *targetedClientInfo;
206 *targetedClientInfo = (*targetedClientInfo)->nextClientInfo;
207 if(tmp->prevClientInfo != NULL){
208 tmp->prevClientInfo->nextClientInfo = *targetedClientInfo;
209 }
210 if(*targetedClientInfo != NULL){
211 (*targetedClientInfo)->prevClientInfo = tmp->prevClientInfo;
212 }
213 }
214 }
```

```
212 if(listClientInfo == tmp){
213 *listClientInfo = *targetedClientInfo;
214 }
215 free(tmp);
216 }
217 }
218
219 /****** END CLTINF
220 *****/
```