Documentazione Progetto LSO

Gennaro Iacuaniello

Anno accademico 2025/2026

Contents

1 Guida all'uso del sistema

Il sistema è formato da due moduli principali: server e client. Entrambi possono essere utilizzati sia autonomamente, sia tramite docker.

1.1 Setup del sistema

1.1.1 Compilazione autonoma

Per compilare il server singolarmente, spostarsi nella relativa cartella e digitare il comando:

```
gcc server.c authentication/authentication.c network/network.c session/session.c
match/match.c -I./authentication -I./network -I./session -I./match -o server.out
                                    -lpthread
```

In cui "-lpthread" serve poiché il server utilizza thread con la libreria pthread, mentre le varie "-I" servono ad indicare al compilatore path ulteriori in cui cercare gli header inclusi nel codice (in particolare, in questo caso, gli header creati per i vari moduli del server).

Fatto questo, basta eseguire il comando:

```
./server.out
```

e il server sarà pronto per ricevere connessioni.

Per compilare invece il client singolarmente, spostarsi nella relativa cartella e digitare il comando:

```
gcc client.c authentication/authentication.c network/network.c session/session.c
match/match.c -I./authentication -I./network -I./session -I./match -o client.out
```

In cui le varie "-I" servono ad indicare al compilatore path ulteriori in cui cercare gli header inclusi nel codice (in particolare, in questo caso, gli header creati per i vari moduli del client).

Fatto questo, bisogna eseguire il comando:

```
./client.out <server_ip/name> <port>
```

Sostituendo i parametri `<server_ip/name>` e `<port>` rispettivamente con l'indirizzo IP (o hostname) del server e la porta di comunicazione scelta.

1.1.2 Utilizzo con docker

Volendo invece usare docker, è stato preparato un file docker-compose, in cui, in particolare, viene creata una rete virtuale per il gioco, e viene creato un volume per i dati degli utenti (username e password hashata).

È sufficiente posizionarsi nella cartella generale del progetto, e digitare:

```
docker compose build
```

A questo punto, se si vuole eseguire il server, basta digitare:

```
docker compose up -d server
```

Dove "up" permette di "attivare" tutto ciò che serve (crea la rete, monta i volumi), e "-d" permette di avviare il server in modalità "detached" (ossia in background). Mentre, se si vuole eseguire il client, basta digitare:

```
docker compose run --rm client
```

Dove "–rm" serve a rimuovere il container appena si chiude il gioco, in modo da non avere container "orfani". Inoltre, grazie alle direttive:

```
stdin_open:  true

tty:  true
```

Il client viene avviato in un terminale interattivo.

Quando poi si è terminato, basta digitare:

```
docker compose down
```

Con questo comando si ferma il processo del server in background e se ne rimuove il container, si rimuove la rete virtuale creata per il gioco, e, se si aggiunge l'opzione "-v", viene anche rimosso il volume creato per i dati degli utenti.

1.2 Utilizzo del sistema

Per quanto riguarda il server, basta avviarlo ed esso resta in ascolto delle connessioni da parte dei vari client. Per quanto riguarda invece il client, appena avviato, se necessario, mostra un avviso sulle dimensioni del proprio terminale e come gestirle. Una volta premuto invio, vengono mostrate le informazioni sul server a cui ci si è connessi, e si viene poi reindirizzati al menù di autenticazione nel gioco.
Qui è possibile: effetuare il login, registrarsi, oppure uscire dal gioco, inserendo i numeri relativi alle rispettive opzioni. Una volta effettuato il login (o la registrazione) con username e password, si viene reindirizzati al menù principale.
Qui è possibile: creare una partita, unirsi ad una partita esistente (ed effettivamente disponibile, quindi non in corso e non piena), oppure uscire dal gioco, sempre inserendo i numeri relativi alle rispettive opzioni.
Se si decide di creare una nuova partita, viene chiesto di scegliere la dimensione della mappa con cui si decidere giocare, e si viene poi reindirizzati alla lobby appena creata come "host", avendo quindi le opzioni di iniziare la partita o di cambiare le dimensioni della mappa, oltre alle opzioni comuni a tutti di poter uscire dalla lobby o dall'intero gioco.
Se invece si decide di unirsi ad una partita esistente, viene mostrata una lista di tutte le partite disponibili (ossia non in corso e non piene). Qui è possibile inserire 0 (zero) se si desidera tornare indietro, oppure l'id relativo alla lobby in cui si desidera entrare. Se l'unione alla lobby va a buon fine, si entra nella partita come "guest", avendo quindi solo le opzioni di uscire dalla lobby o dall'intero gioco, oppure restare semplicemente in attesa dell'avvio della partita da parte dell' "host". Se invece l'unione alla lobby non dovesse andare a buon fine, si riceverebbe un messaggio di errore e si verrebbe poi reindirizzati al menù precedente.
Una volta iniziata la partita, è possibile muoversi utilizzando i tasti w, a, s, d; ogni volta che ci si muove, è possibile "vedere" in un quadrato 5x5 attorno a sè, mentre ogni 30 secondi si ricevono informazioni "globali" da parte del server (posizioni e celle colorate degli altri giocatori, ma non i muri scoperti, che sono "personali").
Alla fine della partita, vengono mostrati i risultati finali, e si viene poi riportati alla lobby.
Mentre si è in una lobby, in caso di disconnessione dell'host, si potrebbe anche essere "promossi" da guest a host.

2 Comunicazione tra client e server

La comunicazione tra client e server avviene grazie all'utilizzo di socket TCP e le loro primitive send e recv. In particolare, il protocollo di comunicazione tra client e server si basa sullo scambio di messaggi, corredati quando necessario da codici per la loro interpretazione (prestando attenzione alla loro conversione in network byte order e viceversa).
Per esempio, nel caso di login e registrazione, se l'operazione va a buon fine viene restituito il codice 0, altrimenti 1 (o anche 2 per la registrazione) per indicare un errore. Ancora, mentre si è in una lobby, viene inviato il codice 0 all'host, 1 ai guest, e poi 2 per indicare che sta iniziando la partita. Similmente, durante una partita, codice 0 indica che sono informazioni riguardanti la mappa locale, 1 indica mappa globale, 2 indica fine della partita.
Per la gestione dell'invio dei messaggi, sono state implementate diverse funzioni di "send_all" e "recv_all" a seconda della varie situazioni in cui un client può trovarsi. Infatti se dovesse esserci un errore mentre si è in una lobby o in una partita in corso, viene gestito il processo di disconnessione di un player in modo che per gli altri giocatori tutto possa continuare normalmente.
Inoltre, sia mentre si è in una lobby sia mentre si è in una partita, il server ha bisogno di inviare dati ai client a intervalli di tempo regolari. Per questo motivo, nel server ci sono vari timer che controllano quando inviare questi dati, e nel client è stata usata la funzione select, per leggere sia eventuali dati in arrivo dal server sia dati che devono essere inviati dall'utente al server (opzioni nel caso della lobby, mosse nel caso della partita).

3 Dettagli implementativi

3.1 Gestione server multi-thread

Nel server, il thread principale accetta le connessioni con accept, e crea un thread dedicato per gestire l'intera sessione del client specifico:

```
int main(int argc, char* argv[]){
      signal(SIGPIPE, SIG_IGN);
      ...
      int listen_socket;
      struct sockaddr_in server_addr;

      listen_socket = socket(AF_INET, SOCK_STREAM, 0);
      if (listen_socket < 0) { ...}
      int on = 1;
      setsockopt(listen_socket, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
      memset(&server_addr, 0, sizeof(server_addr));
      server_addr.sin_family      = AF_INET;
      server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
      server_addr.sin_port        = htons(PORT);
      if (bind(listen_socket, (struct sockaddr*)&server_addr, sizeof(server_addr)) <
          0) {...}
      if (listen(listen_socket, 1) < 0) {...}
      printf("Server in ascolto su porta %d...\n", PORT);
      while(1){
            int connection_socket = accept(listen_socket, NULL, NULL);
            if (connection_socket < 0) {...}
            int* socket_for_thread =(int*) malloc(sizeof(int));
            if (!socket_for_thread) {...}
            pthread_t handle_thread;
            *socket_for_thread = connection_socket;
            int err = pthread_create(&handle_thread, NULL, handle_client, (void*)
                socket_for_thread);
            if (err != 0) {...}
            pthread_detach(handle_thread);
      }
      close(listen_socket);
      return 0;
}
```

Listing 1: Gestione creazione thread per ogni client che si connette

Notare inoltre l'uso di SO_REUSEADDR e il detach del thread subito dopo la creazione.

3.2 Gestione SIGPIPE

Il server ignora il SIGPIPE, la cui gestione avviene tramite l'errno all'interno delle funzioni che si occupano della comunicazione di rete, come nel seguente esempio:

```
ssize_t send_all(int socket_for_thread, const void* buf, size_t n){

      size_t sent = 0;
      const char* ptr = (const char*)buf;

      while(sent < n){

            ssize_t w = send(socket_for_thread, ptr + sent, (size_t)(n - sent), 0);

            if (w < 0) {

                  if (errno == EINTR)
                        continue;

                  if (errno == EPIPE) {
                        perror("Client morto\n");
                        close(socket_for_thread);
                        pthread_exit(0);
                  }

                  perror("send");
                  close(socket_for_thread);
                  pthread_exit(0);
            }

            sent += (size_t)w;
      }

      return (ssize_t)sent;

}
```

Listing 2: Esempio gestione SIGPIPE

3.3 Gestione autenticazione utenti

Per la gestione dei dati degli utenti (username e password), è stato utilizzato un file binario "users.dat". L'accesso concorrente a questo file è stato gestito con una variabile di condizione. Inoltre, per motivi di sicurezza, la password non viene salvata in chiaro nel file, ma sottoposta prima ad hashing.

```
static pthread_mutex_t users_file_mutex = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t users_file_cond_var = PTHREAD_COND_INITIALIZER;
static int users_file_free = 1;

unsigned int registration(char* username, char* password){

      pthread_mutex_lock(&users_file_mutex);
      while(users_file_free == 0)
            pthread_cond_wait(&users_file_cond_var, &users_file_mutex);

      users_file_free = 0;
      FILE* file_users = fopen("data/users.dat", "rb+");

      if(file_users != NULL){
            User current_user;
            while(fread(&current_user, sizeof(User), 1, file_users) == 1){

                  if( strcmp(username, current_user.username) == 0 ){
                              fclose(file_users);
                              users_file_free = 1;
                              pthread_cond_signal(&users_file_cond_var);
                              pthread_mutex_unlock(&users_file_mutex);
                              return 1;
                        }
            }

            strcpy(current_user.username, username);
            unsigned long hashed_password = hash((unsigned char*)password);
            current_user.hashed_password = hashed_password;

            fwrite(&current_user, sizeof(User), 1, file_users);

            fclose(file_users);
            users_file_free = 1;
            pthread_cond_signal(&users_file_cond_var);
            pthread_mutex_unlock(&users_file_mutex);
            return 0;

      }else{
            perror("fopen");
            users_file_free = 1;
            pthread_cond_signal(&users_file_cond_var);
            pthread_mutex_unlock(&users_file_mutex);
            return 2;
      }
}
```

Listing 3: Piccole parti di codice della gestione autenticazione utenti

3.4 Gestione dell'arrivo di messaggi ad intervalli temporali dal server

Esempi di codice del client per la gestione dell'arrivo di messaggi ad intervalli temporali dal server:

```
void handle_being_in_lobby(int socket_fd){
    int stdin_open = 1;
    fd_set rset;
    ...
    while (1) {
        FD_ZERO(&rset);
        if (stdin_open)
            FD_SET(STDIN_FILENO, &rset);
        FD_SET(socket_fd, &rset);
        int maxfd = (STDIN_FILENO > socket_fd ? STDIN_FILENO : socket_fd) + 1;
        int nready = select(maxfd, &rset, NULL, NULL, NULL);

        if (nready < 0) {
            if (errno == EINTR)
                continue;
            perror("select");
            close(socket_fd);
            exit(1);
            break;
        }

        //Socket ready: read from server
        if (FD_ISSET(socket_fd, &rset)) {

            ssize_t r = recv_all(socket_fd, &status_received, sizeof(status_received
                ), 0);

            if (r == sizeof(status_received)){
                status_code = ntohl(status_received);
            }else{
                status_code = 1;    //if an error occured, makes sure that at least
                    does't have "host privileges"
            }

            if(status_code == 2){
                ...
            }

            r = recv_string(socket_fd, recvline, sizeof(recvline), 0);
            if (r == 0) {                   //Server closed
                if (stdin_open)
                    fprintf(stderr, "Server terminated prematurely\n");
                close(socket_fd);
                exit(1);
                break;
            }
            //CLEAR_SCREEN and CURSOR_HOME to put the cursor at the top left corner
                of the screen
            printf("\033[2J\033[H");
            printf("%s", recvline);

        }

        // stdin pronto: leggi e invia al server
        if (stdin_open && FD_ISSET(STDIN_FILENO, &rset)) {
```

```
            char input_buffer[64];
            if (fgets(input_buffer, sizeof(input_buffer), stdin) == NULL) {
                //EOF on stdin
                stdin_open = 0;
                option = UINT_MAX;     //Garbage value to send and so receive "
                    Option not valid"
                if (shutdown(socket_fd, SHUT_WR) < 0) {
                    perror("shutdown");
                    printf("\033[?1049l"); // EXIT_ALT_SCREEN
                    fflush(stdout);
                    exit(1);
                    break;
                }
            }else {
                //Tries to extract a uint32_t from the line read
                if (sscanf(input_buffer, "%u", &option) != 1) {

                    option = UINT_MAX;      //Garbage value to send and so receive "
                        Option not valid"

                }
            }

            option_to_send = htonl(option);

            //No need to clear the buffer since used fgets

            send_all(socket_fd, &option_to_send, sizeof(option_to_send));

            if(status_code == 0){
                ....
            }else if(status_code == 1){
                ...

            }

        }

    }


}
```

Listing 4: Gestione messaggi ad intervalli temporali dal server nella lobby

3.5 Gestione dell'accesso concorrente ai dati di una partita

Per massimizzare la concorrenza e gestire in modo efficiente le risorse quando ci sono molte lobby attive, ogni lobby ha una propria variabile di condizione per gestire l'accesso ai dati relativi ad essa, mentre l'accesso alla lista di tutte le lobby è bloccato solo quando si deve aggiungere/rimuovere una lobby.

```
Message_with_local_information* get_message_with_local_information(Match_list_node*
    match_node, int id_in_match){

      ...

      pthread_mutex_lock(&match_node->match->match_mutex);

      while(match_node->match->match_free == 0)
            pthread_cond_wait(&match_node->match->match_cond_var, &match_node->match
                ->match_mutex);

      match_node->match->match_free = 0;

      // ... Calcolo dei dati locali da inviare ad un player ...

      match_node->match->match_free = 1;
      pthread_cond_signal(&match_node->match->match_cond_var);
      pthread_mutex_unlock(&match_node->match->match_mutex);

      message_with_local_information->receiver_id = htonl((uint32_t)id_in_match);

      return message_with_local_information;

}
```

Listing 5: Esempio gestione dati di una partita

3.6 Interfaccia Utente da Terminale

Per rendere l'esperienza di gioco fluida, sono state sfruttate alcune proprietà del terminale. Con specifiche sequenze di escape ANSI, è possibile utilizzare un "alternate screen buffer", dove si svolge l'intera sessione di gioco. Esso viene di volta in volta sovrascritto, in modo da evitare che una partita comporti moltissime stampe sequenziali all'interno del terminale di partenza dell'utente. Inoltre, per distinguere visivamente i muri, le celle vuote e i vari giocatori, sono state definite delle macro contenenti i codici ANSI per i colori di background (es. `BG_BLUE`, `BG_RED`) e foreground (es. `FG_WHITE_BOLD`), e sono stati utilizzati i caratteri UTF-8 ⋆ e ♢ per individuare rispettivamente il proprio personaggio e gli avversari.

```
void print_map(uint32_t players_positions[MAX_PLAYERS_MATCH][2], uint32_t my_id,
    char** map, uint32_t size){
    ...
    //CLEAR_SCREEN and CURSOR_HOME to put the cursor at the top left corner of the
        screen
    printf("\033[2J\033[H");
    ...
    for (i = 0; i < size; i++) {...
        for (j = 0; j < size; j++) {
            //Check players positions
            int player_on_cell = -1;
            for (p = 0; p < MAX_PLAYERS_MATCH; p++) {
                if (players_positions[p][0] == i && players_positions[p][1] == j) {
                    player_on_cell = p; break; //Player founded
                }
            }
            //If there is a player, print its symbol and its color
            if (player_on_cell != -1) {
                if (player_on_cell == my_id) {
                    //This player
                    printf("%s%s⋆ %s", FG_WHITE_BOLD, bg_colors[player_on_cell],
                        RESET);
                } else {
                    //Enemies
                    printf("%s%s♢ %s", FG_BLACK_BOLD, bg_colors[player_on_cell],
                        RESET);
                }
                continue; //Go to next cell
            }
            //When there is no player
            switch (map[i][j]) {
                case 'e':
                    printf("%s  %s", BG_BLACK, RESET);
                    break;
                case 'w':
                    printf("%s  %s", BG_WHITE, RESET);
                    break;
                ...
            }
        }
        ...
    }
    ...
}
```

Listing 6: Esempio di utilizzo dei codici ANSI per la stampa a schermo della mappa