



SAPIENZA
UNIVERSITÀ DI ROMA

Report Progetto
Sistemi Operativi
VideoGame

Studente: Giovanni Giampaolo

Anno Accademico 2017/2018

Matricola: 1463046

*Prof. Giorgio **Grisetti***

*Tutor: Irvin **Aloise***

Sommario	2
Scelte di Progetto e Specifiche di Avvio	3
Specifiche sull'implementazione	5
• <code>so_game_client.c</code>	5
• <code>so_game_server.c</code>	7
• <code>gamer</code>	10
• <code>function.c</code>	10
• <code>Varie</code>	10

Sommario

Il progetto proposto, da sviluppare in linguaggio C, utilizzando, costrutti ed una base progettuale fornita dai docenti ha l'obiettivo di creare un "videogame" appoggiandosi alla libreria grafica OpenGL e ponendo attenzione alla parte progettuale ed in particolare seguendo queste linee guida:

- Implementare un lato Server del videogame:
 - Il server opera in TCP e UDP:
 - Possibilità di registrare un Client quando il Server è online;
 - "Deregistrare" un client;
 - Inviare la mappa, quando il Client la richiede.
 - Parte UDP
 - Il Server riceve aggiornamenti periodici dal Client nella forma *<timestamp, translational acceleration, rotational acceleration>*
Ogni "epoca" viene generato il messaggio dal Client e viene spedito uno status update;
 - Il Server manda ad ogni Client connesso la posizione degli "agenti" accanto ad esso.
- Lato Client, il Client ha la possibilità di fare quanto segue:
 - Si connette al Server (connessione TCP);
 - Richiede la mappa e riceve un ID assegnatogli dal Server (TCP);
 - Riceve aggiornamenti di stato dal Server;
 - Periodicamente deve:
 - Aggiornare il "viewer" fornito;
 - Leggere i comandi da tastiera;
 - Mandare i pacchetti di "controllo" al Server.

Scelte di Progetto e specifiche di avvio

Per orientarmi con il lavoro ho cominciato analizzando i termini di consegna stabiliti dal progetto, i file ed i costrutti forniti di base. Inizialmente ho avuto delle difficoltà nell'utilizzare la libreria OpenGL ma sono riuscito ad oltrepassare questo ostacolo.

Muovendo i primi passi nell'implementazione il primo grande ostacolo è stato comprendere come creare e come far comunicare in modo efficiente Client e Server. Innanzitutto ho creato un file chiamato "**client_server_test**" e alcune funzioni "amichevoli" **SEND** e **RECEIVE** in Client Server TCP. Per testare lo scambio di dati, evitando di fare confusione e comprendendo bene la base della comunicazione, ho creato questo file.c.

Ho basato la struttura generale sul seguente tipo di comunicazione tra Client e Server: un **Server multithread non bloccante** resta in attesa della connessione, comunicazione, di un Client, avvenuta la connessione, il Server assegna un **ID** (codice univoco identificativo per un Client che si connette a tale Server).

Per lanciare il videogame bisogna aprire una shell per il Server e le relative shell dedicate ai Client che vogliamo entrino in gioco.

Si avvia prima la shell **Server** con il seguente comando:

```
./so_game_server images/test.pgm test/maze.ppm
```

Per avviare il Server bisogna lanciare prima il file poi fornire le seguenti informazioni:

1. <elevation_image>
2. <texture_image>

Bisogna quindi fornire le due immagini presenti nella cartella "image".

Per connettere un **Client**, come specificato sopra, bisogna aprire una shell per ognuno di essi e lanciare il comando:

```
./so_game_client images/arrow-right.ppm 127.0.0.1
```

ovvero lanciare il file Client fornendo informazioni riguardo la texture del giocatore e l'indirizzo del Server a cui vogliamo connetterci, infatti dovrò fornire info del tipo :

1. <player_texture>
2. <server_address>

Per ogni nuovo Client che vado a connettere, effettuo la medesima procedura e, come anticipato sopra, viene fatta distinzione tra i vari Client con l'apposito **ID**.

Avvenuta l'accettazione di tale connessione il Client invierà la sua **immagine** in modo tale che il Server potrà creare l'oggetto **giocatore**, apposito per il Client che si è connesso in quel momento. Successivamente il Server invia, al Client connesso, la **Mappa**.

Specifiche sull'implementazione

In questa sezione, spiego le scelte e le strutture adottate durante l'implementazione:

- **so_game_client.c**

Le strutture dati per gestire i thread in questo file sono:

UpdaterArgs

- volatile int run;
- Vehicle* vehicle;
- World* world;

listenArgs

- volatile int run;
- World* world;

notificationArgs

- volatile int run;
- World* world;

Run è la variabile che gestisce il ciclo in ogni thread; in ogni struttura è presente la variabile puntatore a world e in UpdaterArgs è presente il puntatore a vehicle, il quale serve a mandare gli update del vehicle al Server.

Run diviene "1" appena i thread vengono lanciati e diventa "0" quando il Client preme "ESC" e si disconnette dal gioco.

void* updater_thread(void* args_)

*Il thread tramite connessione UDP invia pacchetti in modo continuo al Server con le **informazioni** sul proprio vehicle quali:*

ID, rotational_force, translational_force.

void* world_listener(void* args_)

Questo thread tramite connessione UDP, riceve, in modalità Multicast, aggiornamenti da parte del Server sulla posizione di tutti i vehicle in gioco, quali:

ID, x, y, theta.

*Una volta presi i dati si **aggiorna** ogni vehicle connesso al momento.*

```
void* notification_listener(void* args_)
```

*Questo blocco di codice mi permette di restare in ascolto per ricevere notifiche dal Server, in modalità Multicast, per un nuovo giocatore connesso o per un giocatore che viene rimosso dal gioco. Riceve un **pacchetto** con un ID e, dopo averlo letto, verifica l'azione da compiere, ovvero, se aggiungere o rimuovere un giocatore con un certo ID. Nello specifico se il primo char del buffer è una "a", leggo la restante parte con cui trovo l'ID e vado ad aggiungere un nuovo giocatore a world, altrimenti se il primo char del buffer è "r" vado a cercare il giocatore, tramite l'ID, e lo rimuovo.*

```
int main(int argc, char** argv**)
```

*Inizialmente si instaura la connessione TCP con il Server, ricevo un ID che diventa l'ID del giocatore, carico la sua texture, che ho specificato dalla Shell al lancio del Client, e con due **RECV** ricevo Elevation e Texture della mappa. Quindi creo il world, aggiungo il mio vehicle, con con l'ID assegnato, e prima di lanciare il world lancio tutti i thread che gestiscono la comunicazione UDP.*

- **so_game_server.c**

Le strutture dati per gestire i thread in questo file sono:

UpdaterArgs

- volatile int run;
- World* world;

multicastArgs

- volatile int run;
- World* world;

notificationArgs

- volatile int run;

handler_Args

- int socket_desc;
- struct sockaddr_in *client_addr;
- World* world;

***Run** è la variabile che gestisce il ciclo in ogni thread, ed internamente a tutte le strutture è presente la variabile puntatore a world, meno che per notificationArgs. **handler_Args** è la struttura dati del thread principale che gestisce la connessione TCP con il main del Client. Il socket_desc è il descrittore della **Socket** che gestisce la connessione con il Client e il client_addr è la struttura che contiene l'indirizzo del Client.*

void* updater thread(void* args)

Questo thread tramite connessione UDP, riceve da tutti i giocatori connessi pacchetti che contengono informazioni sul vehicle, quali:

ID, translational_force, rotational_force.

*Con queste informazioni aggiorna il **vehicle**, di quell'ID, e fa un update del world.*

void* multi thread(void*args)

*Questo thread, tramite connessione UDP, crea ed invia un pacchetto in Multicast con tutte le **informazioni** dei giocatori che in quel momento sono in gioco, quali:*

ID, x, y, theta.


```
void* notify(void*args)
```

Questo thread, ogni volta che un giocatore si connette, viene aggiunto al world ed invia un pacchetto a tutti i giocatori presenti. Questo pacchetto contiene "action" (a || r) e l'ID del giocatore, ed invia con connessione UDP in Multicast.



Il semaforo, **lock**, impedisce che il ciclo while venga eseguito infinitamente, infatti, viene incrementato di 1 nel thread che gestisce la connessione con Client appena il giocatore si connette.

```
while(args->run){
    //notify
    //add/remove gamer
    sem_wait(&lock);
    sleep(1);
    if (action == ADD) size = SERIALIZE_GAMERS(Gamers, buf);

    else if (action == RMV) size = sprintf(buf, "%c%d", RMV, action_id);

    ret = SENDTO(udp_socket_desc, buf, size, &udp_server_addr);
}
```

```
void* connection handler(void*args)
```

Parte più importante in cui gestisco la connessione end to end del Client-Server in TCP.

```
ret = SEND(socket_desc, img_pack_buf, pack_len);
recv_bytes = RECV(socket_desc, buf, buf_len-1);
```

In questo caso si usa la connessione TCP perchè ho bisogno di sincronizzare la comunicazione tra Server e Client. Quando un Client si connette e la connessione viene accettata, il Server assegna un ID, invia il messaggio di benvenuto e comunica l'user ID. Successivamente viene creato ed aggiunto a world tale vehicle. Il Server così è pronto ad inviare il pacchetto map contenente la map texture e la map elevation. Nella sezione critica, ho ritenuto opportuno inserire il **semaforo** synchro perchè potrebbe verificarsi il caso in cui due thread contemporaneamente aggiungono o rimuovono un giocatore

contando un numero non corretto di essi.

```
//start critical section  
    sem_wait(&synchro);  
    gamer_counter = NUM_PLAYERS(Gamers);  
  
//unlock notify thread --> ADD  
    action = ADD;  
    action_id = user_id;  
    sem_post(&lock);  
    sem_post(&synchro);  
//end critical section
```

Nella parte successiva alla sezione critica resto in attesa che un giocatore prema "ESC" per uscire dal gioco, e così inizia la seconda sezione critica per rimuovere lo stesso dal world.

```
//start critical section  
    sem_wait(&synchro);  
//fase di detach  
    World_detachVehicle(args->world, vehicle);  
//unlock notifier  
    action_id = user_id;  
    action = RMV;  
    sem_post(&lock);  
    gamer_counter = NUM_PLAYERS(Gamers);  
    sem_post(&synchro);  
//end critical section
```

In questa seconda sezione critica gestisco l'eliminazione di un giocatore.

```
int main(int argc, char **argv)
```

Nel main viene impostata la connessione TCP per i Client dopodichè si caricano le immagini specificate nella shell, costruisco il world e prima di lanciarlo faccio partire i thread che gestiscono le connessioni UDP.

Dopodichè resto in attesa della connessione TCP da parte del Client.

```
while(1){  
    client_desc = accept(socket_desc, (struct sockaddr*) client_addr,  
    (socklen_t*) &sockaddr_len);
```

(ACCEPT) La funzione è bloccante, resto quindi in attesa finché non si connette un Client. Arriva il client che fa **CONNECT** e si sblocca. Con l'operazione di connessione lancio il thread **connection_handler** che gestirà la connessione con quel giocatore. Infine il ciclo ricomincia in attesa di un'altra connessione.

- **gamer**

L'oggetto **gamer** mi è stato utile per poter gestire tutte le entità connesse durante il gioco. Nello specifico nel momento in cui un nuovo giocatore si connette, memorizzo un oggetto **gamer** avente quell'ID in modo tale da non correre il rischio di creare oggetti aventi un ID già occupato. Tale struttura è anche utile per contare i giocatori in gioco in un determinato momento, e più in generale, questo file, contiene di funzioni di supporto.

- **function.c**

In questo file ho gestito:

- **SEND**
- **RECV**
- **SENDTO**
- **RECVFROM**
- **RANDOM_INTEGER** (Per la scelta casuale di un ID)
- **Generiche funzioni di supporto per gli errori** (Valori di ritorno di **SYS**)

Ho creato delle funzioni "smart" per rendere il codice più leggibile.

- **Varie**

*Il file **help.h** contiene tutte le **#define** utilizzate dalle varie funzioni.*