PROGETTO_ESAME_RETI_NOBLER_AGOUZOUL

Sofian Agouzoul, 0124002029
Matteo Nobler, 0124001853

03/Dicembre/2024

Starship

Part I
Traccia

Una navicella spaziale deve evitare i detriti provenienti da una tempesta di meteoriti. La navicella (client) entra nel settore spaziale dei meteoriti (server) connettendosi in UDP. La tempesta di meteoriti genera in modo casuale n pacchetti UDP ogni 2 secondi che rappresentano i detriti in una griglia MxM. La navicella riceve un alert nel caso in cui il detrito spaziale si trovi sulla sua stessa posizione e può spostarsi prima dell'impatto. Ogni spostamento della navicella viene notificato al server il quale genera nuovi detriti nella direzione della navicella stessa.

1 DETTAGLI IMPLEMENTATIVI

L'applicazione è stata codificata seguendo il paradigma client-server, con i due che adoperano il protocollo UDP per comunicare tra di loro.
Mediante l'utilizzo delle funzioni sendto() e recvfrom(), il client ed il server, comunicano tra loro scambiandosi pacchetti UDP ogni 2 secondi. Ricordiamo che queste funzioni sono l'equivalente delle funzioni read() e write().
In particolare, il server genera i meteoriti, ovvero i pacchetti UDP, e li invia al client che dovrà spostarsi all'interno della griglia di gioco per evitare eventuali collisioni. Di seguito si riportano le specifiche implementative del Server e del Client:

Part II
CORPO DEL SERVER

2 PROCEDURE

1. PROCEDURA INITIALIZE_SERVER()

```
// Inizializza il socket del server
int initialize_server(struct sockaddr_in *server_addr)
{
 int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
 if(sockfd < 0)
 {
  perror("Errore creazione socket");
  exit(EXIT_FAILURE);
 }
 memset(server_addr, 0, sizeof(*server_addr));
 server_addr->sin_family = AF_INET;
 server_addr->sin_port = htons(PORT);
 server_addr->sin_addr.s_addr = inet_addr("127.0.0.1");
 if(bind(sockfd, (const struct sockaddr *)server_addr, sizeof(*server_addr)) < 0)
 {
  perror("Errore binding socket");
  exit(EXIT_FAILURE);
 }
 return sockfd;
}
```

Questa funzione incapsula la fase creazione della socket adoperata per la comunicazione con un client. Banalmente crea il file-descriptor della socket, imposta correttamente Porta e Indirizzo IP su cui tale server risponde e richiama la bind() per assegnare tale combinazione [IP;Porta] al descrittore della socket.
Se tutte le operazioni vanno a buon fine, è ritornato il file-descriptor della socket.

2. PROCEDURA UPDATE_METEORITES()

```
// Aggiorna i meteoriti e verifica collisioni e impatti imminenti
void update_meteorites()
{
 for(int i = 0; i < meteorite_count; i++)
 {
  meteorites[i][0]++;                    // Sposta i meteoriti verso il basso
  // Rimuovi meteoriti fuori dalla griglia
  if(meteorites[i][0] >= GRID_SIZE)
  {
   meteorites[i][0] = meteorites[meteorite_count - 1][0];
   meteorites[i][1] = meteorites[meteorite_count - 1][1];
   meteorite_count--;
   i--;
  }
 }
}
```

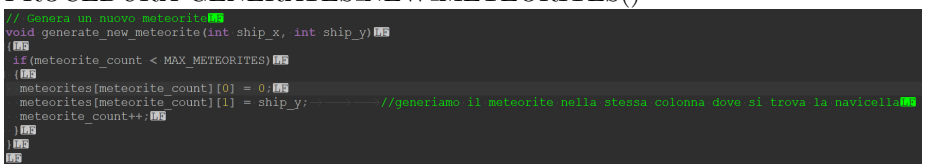
Questa funzione si occupa di spostare le meteore dall'alto verso il basso, con un andamento che potremmo definire "a cascata". Per fare ciò, semplicemente l'indice di riga di una meteora viene incrementato (perchè si passa da una riga più in alto, che ha un indice più piccolo, a una riga più in basso, che ha un indice più grande).
Incrementando l'indice di riga, ovviamente prima o poi la meteora raggiungerà la riga più bassa della matrice, e in tal caso essa viene rimossa, decrementando anche il numero di meteore presenti sul campo.

3. PROCEDURA CHECK_COLLISION()

```
//Verifica se si sono verificate collisioni
void check_collision(int ship_x, int ship_y, int *collision_met_x, int *collision_met_y, int *collision)
{
 for(int i=0;i<meteorite_count;i++)
 {
  if((meteorites[i][0] == ship_x) && (meteorites[i][1] == ship_y))
  {
   *collision_met_x = meteorites[i][0];
   *collision_met_y = meteorites[i][1];
   *collision = 1;
   break;
  }
 }
}
```

Questa funzione si occupa di vedere se l'astronave ha colliso con un meteorite.
L'idea è che si vede ogni meteorite presente se ha colliso con l'astronave.
La collisione avviene se ascissa e ordinata dell'astronave (ship_x;ship_y) coincidono con ascissa e ordinata di un meteorite (meteorites[i][0];meteorites[i][1]).
Verificando ogni meteorite, se si trova quello che ha colliso, si memorizzano ascissa e ordinata di tale meteorite, si valorizza a 1 la variabile che indica l'avvenuta collisione e si esce dal ciclo di controllo (visto che si è trovato il meteorite che ha impattato non è necessario continuare a verificare).

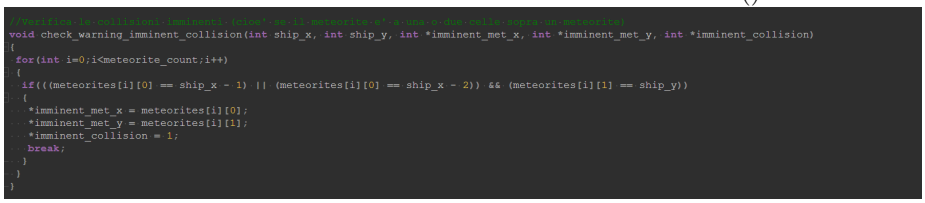
4. PROCEDURA GENERATES_NEW_METEORITES()



```
// Genera un nuovo meteorite
void generate_new_meteorite(int ship_x, int ship_y)
{
 if(meteorite_count < MAX_METEORITES)
 {
  meteorites[meteorite_count][0] = 0;
  meteorites[meteorite_count][1] = ship_y;          //generiamo il meteorite nella stessa colonna dove si trova la navicella
  meteorite_count++;
 }
}
```

Questa funzione si occupa di generare un nuovo meteorite se possibile, ossia se sul campo vi è un numero di meteoriti inferiore a un limite superiore impostato mediante macro.
Il meteorite viene generato nella riga più in alto della matrice del campo di meteore e nella stessa colonna dove si trova l'astronave.
Se si è riuscito a generare un nuovo meteorite, il contatore dei meteoriti presenti su campo viene incrementato.

5. PROCEDURA CHECK_WARNING_IMMINENT_COLLISION()



```
//Verifica la collisione imminente (else' se il meteorite e' a una o due celle sopra la navicella)
void check_warning_imminent_collision(int ship_x, int ship_y, int *imminent_met_x, int *imminent_met_y, int *imminent_collision)
{
 for(int i=0;i<meteorite_count;i++)
 {
  if(((meteorites[i][0] == ship_x - 1) || (meteorites[i][0] == ship_x - 2)) && (meteorites[i][1] == ship_y))
  {
   *imminent_met_x = meteorites[i][0];
   *imminent_met_y = meteorites[i][1];
   *imminent_collision = 1;
   break;
  }
 }
}
```

Questa funzione si occupa di inviare un warning per avvisare il client della troppa vicinanza con un meteorite che si trova sulla sua stessa traiettoria, di fatto avvisandolo di doversi spostare per evitare l'impatto.
Essenzialmente la procedura verifica se il meteorite è sulla stessa colonna della matrice (meteorites[i][1] == ship_y) e a una o due celle di distanza sopra a dove si trova l'astronave ((meteorites[i][0] == ship_x - 1) or (meteorites[i][0] == ship_x - 2)), e se questa condizione si verifica, delle apposite variabili vengono settate per indicare al server di avvisare il client dell'impatto imminente.
Più precisamente si imposta imminent_collision a 1 per indicare che c'è il rischio di impatto imminente, e si valorizzano imminent_met_x e imminent_met_y per specificare quale è il meteorite con cui si sta rischiando di impattarsi.

6. PROCEDURA SEND_METEORITES()

```c
// Invia i meteoriti al client
void send_meteorites(int sockfd, struct sockaddr_in *client_addr, socklen_t len)
{
 for(int i = 0; i < meteorite_count; i++)
 {
  int meteorite_data[2] = {meteorites[i][0], meteorites[i][1]};
  printf("<SERVER>:: Invio meteorite: (%d, %d)\n", meteorite_data[0], meteorite_data[1]);
  sendto(sockfd, meteorite_data, sizeof(meteorite_data), 0, (struct sockaddr *)client_addr, len);
 }
 //Segnale di fine dati
 int end_signal = -1;
 printf("<SERVER>:: Invio segnale di fine dati.\n");
 sendto(sockfd, &end_signal, sizeof(end_signal), 0, (struct sockaddr *)client_addr, len);
}
```

Questa funzione richiama al suo interno in ciclo la funzione sendto() per inviare i vari meteoriti generati al client.
Conoscendo il numero 'meteorite_count' di meteoriti presenti sul campo (che era correttamente valorizzato e controllato da generate_new_meteorites()), il ciclo di invio meteoriti viene ripetuto per 'meteorite_count' volte, e, terminato il ciclo, un ulteriore sendto() invierà un valore speciale (che è semplicemente un -1) per indicare che tutti i meteoriti che dovevano essere inviati sono stati inviati.

7. PROCEDURA RECEIVE_SHIP_POSITION()

```c
//Riceve la nuova posizione della navicella
int receive_ship_position(int sockfd, struct sockaddr_in *client_addr, socklen_t *len, int *ship_x, int *ship_y)
{
 char buffer[1024];
 int n = recvfrom(sockfd, buffer, sizeof(buffer), 0, (struct sockaddr *)client_addr, len);
 if(n < 0)
 {
  perror("<SERVER>:: Errore ricezione dati");
  return -1;
 }
 buffer[n] = '\0';
 int new_x, new_y;
 if(sscanf(buffer, "Posizione: %d, %d", &new_x, &new_y) != 2)
 {
  fprintf(stderr, "<SERVER>:: Errore parsing posizione navicella\n");
  return -1;
 }
 *ship_x = new_x;
 *ship_y = new_y;
 return 0;
}
```

Questa funzione richiama al suo interno una recvfrom() per ricevere in un buffer la posizione della navicella client.
Il buffer è formattato in una maniera preimpostata, in maniera tale che con una sscanf() siano correttamente leggibili e valorizzabili le variabili intere new_x e new_y contenenti ascissa e ordinata di dove si trova la navicella dopo uno spostamento. A partire da queste variabili, basta fare due assegnazioni per valorizzare correttamente le coordinate (ship_x;ship_y) della navicella.

8. PROCEDURA PRINT_GRID()

```c
// Stampa la griglia di gioco
void print_grid(int ship_x, int ship_y)
{
 char grid[GRID_SIZE][GRID_SIZE];
 memset(grid, '.', sizeof(grid));
 for(int i = 0; i < meteorite_count; i++)
 {
  if(meteorites[i][0] < GRID_SIZE && meteorites[i][1] < GRID_SIZE)
  {
   grid[meteorites[i][0]][meteorites[i][1]] = '*';
  }
 }
 if(ship_x >= 0 && ship_x < GRID_SIZE && ship_y >= 0 && ship_y < GRID_SIZE)
 {
  grid[ship_x][ship_y] = 'S';
 }
 printf("\n");
 for(int x = 0; x < GRID_SIZE; x++)
 {
  for(int y = 0; y < GRID_SIZE; y++)
  {
   printf("%c ", grid[x][y]);
  }
  printf("\n");
 }
}
```

Questa funzione si occupa di utilizzare le variabili ship_x, ship_y e il set di coordinate meteorites[][] dei meteoriti per stampare graficamente una matrice con dei simboli indicanti la posizione spaziale di meteoriti e navicella sul campo.
In pratica questa funzione si occupa di mostrare un resoconto grafico della partita, stampando una matrice di caratteri rappresentante il campo di gioco, così che l'utente abbia modo di capire quali sono mosse più ottimali per proseguire senza andarsi a schiantare su un meteorite e perdere.
Per i simboli usati usati per la grafica si è deciso come segue:
'.' : indica uno spazio vuoto libero
'*' : indica un meteorite
'S' : indica la navicella

9. MAIN_SERVER_1

```c
int main(int argc,char **argv)
{
 struct sockaddr_in server_addr, client_addr;
 socklen_t len = sizeof(client_addr);
 // Inizializza il server
 int sockfd = initialize_server(&server_addr);
 srand(time(NULL));
 int ship_x,ship_y;
 int collision = 0;
 int imminent_collision = 0;
 int imminent_met_x, imminent_met_y;
 int collision_met_x, collision_met_y;
 int quitting = 0;
 printf("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n");
 printf("Ricorda:\nship_x = RIGA SULLA MATRICE\nship_y = COLONNA SULLA MATRICE\n");
 printf("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n");
  printf("<SERVER>:: In attesa che la navicella client sia attiva...\n");
 receive_ship_position(sockfd, &client_addr, &len, &ship_x, &ship_y);
 printf("<SERVER>:: Posizione iniziale della nave ricevuta, che comincino i giochi!\n");
 while(1)
 {
  quitting = 0;
  imminent_collision = 0;

  /*Genera nuovi meteoriti*/
  generate_new_meteorite(ship_x, ship_y);

  /*Stampa la griglia aggiornata*/
  print_grid(ship_x, ship_y);

  /*Invia meteoriti e avvisi*/
  send_meteorites(sockfd, &client_addr, len);

  /*Controlla collisioni imminenti perche' ci sono meteoriti vicini alla navicella*/
  check_warning_imminent_collision(ship_x, ship_y, &imminent_met_x, &imminent_met_y, &imminent_collision);
  if(imminent_collision)
  {
   //se vi e' un impatto imminente, va inviato il messaggio di warning
   send_warning(sockfd, &client_addr, len, imminent_met_x, imminent_met_y);
  }
```

Il server inizia impostando correttamente il suo insieme (ID;Porta) che andrà a usare e la socket stessa adoperata per la comunicazione. Chiamando poi receive_ship_position(), si bloccherà in attesa di ricevere la posizione iniziale della navicella client. Ricevuta tale posizione, si entra nel loop principale del gioco. Il protocollo di livello applicazione usato dal server per la comunicazione con il client è composto dai seguenti passaggi, ripetuti in loop:

1 Genera nuovi meteoriti nella riga più in altro della matrice che fa da campo di gioco

2 Stampa suddetta griglia per avere un resoconto grafico della situazione

3 Invia I meteoriti generati al client per rendergli nota la loro posizione

4 Vedi se il client è troppo vicino a uno dei meteoriti e nel caso invia un messaggio di warning

10. MAIN_SERVER_PT2

```
else
{
 /*altrimenti si invia la lunghezza 0 per dire che il client non si deve bloccare in attesa di bytes componenti un messaggio di warning
 int message_lenght = 0;
 sendto(sockfd, &message_lenght, sizeof(int), 0, (struct sockaddr *)&client_addr, len);
}

/*Se il client ha richiesto di uscire dalla partita, terminare il server*/
recvfrom(sockfd, &quitting, sizeof(int), 0, (struct sockaddr *)&client_addr, &len);
if(quitting == 1)
{
 printf("<SERVER>:: Il client si e' disconnesso... mi disconnetto anche io!\n");
 break;
}

/*Riceve la nuova posizione della navicella*/
if(receive_ship_position(sockfd, &client_addr, &len, &ship_x, &ship_y) < 0)
{
 continue;
}

/*Controlla se vi e' stata una collisione*/
check_collision(ship_x, ship_y, &collision_met_x, &collision_met_y, &collision);

/*Invia l'esito del check della collisione al client, cosi' anche il client si disconnette se ha colliso*/
sendto(sockfd, &collision, sizeof(int), 0, (struct sockaddr *)&client_addr, len);
if(collision)
{
 printf("<SERVER>:: Collisione rilevata con il meteorite in posizione (%d, %d)! La navicella è stata distrutta.\n", collision_met_x, collision_met_y);
 break;
}

/*Aggiorna i meteoriti*/
update_meteorites();

/*Controlla se vi e' stata una collisione anche in questo caso per evitare sovrascritture della posizione dei meteorite*/
check_collision(ship_x, ship_y, &collision_met_x, &collision_met_y, &collision);

/*Invia l'esito del check della collisione al client, cosi' anche il client si disconnette se ha colliso*/
sendto(sockfd, &collision, sizeof(int), 0, (struct sockaddr *)&client_addr, len);
```

5 Se non è vicino a un meteorite, nessun messaggio di warning va inviato, ergo il server invierà un messaggio vuoto per evitare che il client resti bloccato su una recvfrom()

6 Attende che il client invii un comando; innanzitutto vede se il comando è una richiesta di uscire dal gioco, ergo resta bloccato su una prima recvrom()

7 Se il client non ha richiesto di uscire, il server proseguirà e riceverà con un apposita funzione la nuova posizione della navicella

8 In seguito al movimento, il server controlla eventuali collisioni

9 Adoperando una funzione check_collision(), viene valorizzata un apposita variabile per indicare se vi è stata o meno una collisione. Se questa collisione è avvenuta, viene inviato al client il messaggio che ha perso, indicando anche le coordinate del meteorite su cui si è schiantato.

10 Se la collisione non è avvenuta, la partita può proseguire. I meteoriti possono spostarsi verso il basso.

11 Poichè però si deve vedere anche se in seguito allo spostamento dei meteoriti il client ha colliso (questa cosa può avvenire se navicella e meteorite erano sulla stessa colonna e vi era un solo spazio vuoto di distanza tra i due), è nuovamente richiamata check_collision(), con l'invio dell'esito del check.

11. MAIN_SERVER_PT3

```
if(collision)
{
 printf("<SERVER>:: Collisione rilevata con il meteorite in posizione (%d, %d)! La navicella è stata distrutta.\n", collision_met_x, collision_met_y);
 break;
}

 sleep(2);
}
close(sockfd);
return 0;
```

In questo ultimo blocco del main, se viene settata la variabile collision ad 1, il server stamperà un messaggio per cui la navicella si collide con un meteorite, scollegandosi e terminando cosi la partita

Part III
CORPO DEL CLIENT

3 PROCEDURE

1. PROCEDURA RECV_WARNING()

```c
void recv_warning(int sockfd, struct sockaddr_in *server_addr,socklen_t len)
{
 int message_lenght;
 //la prima recvfrom() dira' quanto lungo e' il messaggio di warning
 recvfrom(sockfd, &message_lenght, sizeof(int), 0, NULL, NULL);
 //sapendo ora quanti dati aspettarsi, se ci si deve aspettare piu' di 0 bytes di dati,
 //la seconda recvfrom() recuperera' il messaggio di warning vero e proprio
 if(message_lenght > 0)
 {
  char warning_message[message_lenght];
  recvfrom(sockfd, warning_message, message_lenght, 0, NULL, NULL);
  warning_message[message_lenght] = '\0';
  printf("<CLIENT>:: %s\n",warning_message);
 }
}
```

Questa funzione si occupa di recuperare eventuali messaggi di warning che il server ha inviato al client per avvisarlo della presenza di un meteorite troppo vicino a lui. Essenzialmente la funzione vede se il server ha inviato un messaggio di warning usando una prima recvfrom() per recuperare la lunghezza del messaggio inviato.
Se è recuperata una lunghezza superiore agli 0 caratteri, allora il messaggio di warning esiste, e quindi la funzione procederà invocando una seconda recvfrom() per recuperare il messaggio di warning vero e proprio.
La prima recvfrom() dice anche quanto è lungo il messaggio di warning, ergo da anche informazione alla seconda recvfrom() su quanti bytes aspettarsi di ricevere come messaggio di warning.

2. PROCEDURA CONTROL_GAME()

```c
void control_game(int *ship_x, int *ship_y,int *quit_request)
{
 char input[2];
 printf("Inserisci comando (w: sopra, a: sinistra, s: sotto, d: destra, q: esci):\n");
 char command;
 /*
 per formattare bene il comando (che e' inviato tramite stringhe), creo una stringa buffer di input che contiene TUTTO il comando
 (compreso lo '\n'), e subito dopo, leggo solo il PRIMO carattere del buffer, che e' il nostro effettivo comando
 questo garantisce che UN SOLO COMANDO e' letto e dunque interpretato.
 */
 scanf("%s",input);
 command = input[0];
 if((command == 'w') && (*ship_x > 0))
 {
  (*ship_x)--;
 }
 else if((command == 'a') && (*ship_y > 0))
 {
  (*ship_y)--;
 }
 else if((command == 's') && (*ship_x < GRID_SIZE - 1))
 {
  (*ship_x)++;
 }
 else if((command == 'd') && (*ship_y < GRID_SIZE - 1))
 {
  (*ship_y)++;
 }
 else if(command == 'q')
 {
  *quit_request = 1;
 }
}
```

Questa funzione incapsula i controlli di gioco che il client può usare per muovere la navicella. Il giocatore da terminale potrà premere uno dei quattro tasti W,A,S o D per indicare in quale direzione la navicella si deve muovere (in ordine: Alto, Sinistra, Sotto, Destra).
Controllando con degli if() a catena il tasto premuto, la navicella si sposterà di una casella nella direzione desiderata incrementando o decrementando la sua ascissa o la sua ordinata di un'unità, a patto che in quella direzione vi sia spazio per muoversi (se, ad esempio, la navicella prova a spostarsi a destra, ma essa sta già al bordo destro della mappa, non è possibile spostarsi ulteriormente a destra).
Inoltre è possibile premere il tasto 'q' per far si che il client notifichi al server della sua intenzione di volersi disconnettere dalla partita.

3. MAIN_CLIENT_PT1

```
int main(int argc,char **argv)
{
 int sockfd;
 struct sockaddr_in server_addr;
 socklen_t len = sizeof(server_addr);
 char buffer[1024];
 /*Creazione del socket*/
 if((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
 {
  perror("Errore creazione socket");
  exit(EXIT_FAILURE);
 }
 memset(&server_addr, 0, sizeof(server_addr));
 server_addr.sin_family = AF_INET;
 server_addr.sin_port = htons(PORT);
 server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
 int ship_x = GRID_SIZE - 1, ship_y = GRID_SIZE / 2;  //Posizione iniziale della navicella
 int quitting = 0;                                    ///opzione per uscire dal gioco
 int collision = 0;
 /*Invio la posizione INIZIALE della navicella...*/
 printf("<CLIENT NAVICELLA>:: posizione (ship_x, ship_y) iniziale: %d, %d\n", ship_x, ship_y);
 snprintf(buffer, sizeof(buffer), "Posizione: %d, %d", ship_x, ship_y);
 sendto(sockfd, buffer, strlen(buffer), 0, (const struct sockaddr *)&server_addr, len);
 /*...e subito dopo, inizia il loop del gioco*/
 printf("<CLIENT NAVICELLA>:: Entro nel loop di gioco...\n");
```

La funzione main() del programma client inizia valorizzando con le apposite funzioni tutte le variabili necessarie per specificare correttamente il server con cui si comunicherà mediante protocollo UDP.
Vi è quindi la fase di creazione del file-descriptor sockfd della socket UDP con l'utilizzo della funzione socket() e specificando SOCK_DGRAM come opzione per il protocollo di livello trasporto che si andrà ad utilizzare.
Vi è poi anche l'inizializzazione della struttura di tipo struct sockaddr_in per indicare correttamente l'insieme (IP;Porta) su cui è attivo il server. Come ultima operazione di questa prima parte del main() vi è una prima sendto(), usata per inviare la posizione iniziale della navicella sul campo di gioco, così da permettere al server di sapere dove la navicella si trova inizialmente.

4. MAIN_CLIENT_PT2

```
while(1)
{
 /*Ricezione dei meteoriti*/
 while(1)
 {
  int meteorite_data[2];
  recvfrom(sockfd, meteorite_data, sizeof(meteorite_data), 0, NULL, NULL);
  if(meteorite_data[0] == -1)
  {
   break; ---> // Segnale di fine dati
  }
 }

 /*Vedi se arriva un messaggio di warning per meteoriti troppo vicini*/
 recv_warning(sockfd, &server_addr, len);

 /*Controllo comandi*/
 control_game(&ship_x,&ship_y,&quitting);

 /*Dire al server se ci si vuole scollegare o no*/
 sendto(sockfd, &quitting, sizeof(quitting), 0, (const struct sockaddr *)&server_addr, len);
 if(quitting == 1)
 {
  printf("<CLIENT NAVICELLA>:: Me ne esco!\n");
  break;
 }

 /*Invio della posizione della navicella*/
 printf("<CLIENT NAVICELLA>:: posizione (ship_x, ship_y): %d, %d\n", ship_x, ship_y);
 snprintf(buffer, sizeof(buffer), "Posizione: %d, %d", ship_x, ship_y);
 sendto(sockfd, buffer, strlen(buffer), 0, (const struct sockaddr *)&server_addr, len);

 /*Vedi se il server ti ha comunicato di esserti schiantato*/
 recvfrom(sockfd, &collision, sizeof(int), 0, NULL, NULL);
 if(collision)
 {
  printf("<CLIENT NAVICELLA>:: Mi sono schiantato! Ho perso!\n");
  break;
 }

 sleep(2);
}
close(sockfd);
```

Il main() poi continua, entrando nel vero loop infinito di gioco, che permette dunque a client e server di scambiare tra di loro informazioni e, di fatto, giocare una partita. Il protocollo di livello applicazione usato dal client per la comunicazione con il server è composto dai seguenti passaggi, ripetuti in loop:

1 Mettiti in ricezione delle coordinate dei vari meteoriti.

2 Mettiti in ricezione di eventuali messaggi di warning.

3 Dai all'utente possibilità di muovere la navicella o di uscire dalla partita.

4 Comunica al server se ci si vuole disconnettere o meno.

5 Formatta e invia correttamente le nuove coordinate a cui si trova ora la navicella dopo uno spostamento.

6 Mettiti in ricezione dell'esito del movimento per vedere se si è colliso o meno con un meteorite.

7 Mettiti in pausa per 2 secondi e reitera i passaggi di cui sopra.

Part IV
MANUALE ED ISTRUZIONI SU COMPILAZIONE ED ESECUZIONE

MAKEFILE

```
all: NavicellaClient MeteoritiServer
NavicellaClient: client_navicella.o
	gcc -o NavicellaClient client_navicella.o
MeteoritiServer: server_meteoriti.o
	gcc -o MeteoritiServer server_meteoriti.o
client_navicella.o: client_navicella.c
	gcc -c client_navicella.c
server_meteoriti.o: server_meteoriti.c
	gcc -c server_meteoriti.c
clean:
	rm -f *.o
```

Sopra è riportato il Makefile usato per la compilazione. Digitando banalmente il comando 'make' sul terminale, partirà in automatico la compilaizone dei due sorgenti.

SIMULAZIONE DI GIOCO

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
Ricorda:
ship_x = RIGA SULLA MATRICE
ship_y = COLONNA SULLA MATRICE
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
<SERVER>:: In attesa che la navicella client sia attiva...
<SERVER>:: Posizione iniziale della nave ricevuta, che comincino i giochi!

. . . . . * . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . S . . . .
<SERVER>:: Invio meteorite: (0, 5)
<SERVER>:: Invio segnale di fine dati.
```

La situazione iniziale è che il client navicella parte nella riga più bassa della matrice a una posizione centrale, e un meteorite viene generato nella sua stessa colonna al lato opposto.

```
<CLIENT NAVICELLA>:: posizione (ship_x, ship_y) iniziale: 9, 5
<CLIENT NAVICELLA>:: Entro nel loop di gioco...
Inserisci comando (w: sopra, a: sinistra, s: sotto, d: destra, q:
 esci):
```

Lato client la posizione della navicella viene notificata e quindi si parte con il loop principale. Si può osservare che il programma client si bloccherà in attesa della pressione di uno dei tasti di movimento o del tasto di uscita.

```
. . . . . * . .
. . . . . * . .
. . . . . * . .
. . . . . * . .
. . . . * . S .
. . . . * . . .
. . . . * . . .
. . . . * . . .
. . . . . . . .
. . . . . . . .
<SERVER>:: Invio meteorite: (7, 5)
<SERVER>:: Invio meteorite: (6, 5)
<SERVER>:: Invio meteorite: (5, 5)
<SERVER>:: Invio meteorite: (4, 5)
<SERVER>:: Invio meteorite: (3, 6)
<SERVER>:: Invio meteorite: (2, 7)
<SERVER>:: Invio meteorite: (1, 7)
<SERVER>:: Invio meteorite: (0, 7)
<SERVER>:: Invio segnale di fine dati.
--------------------------------------------------------------------------
----------------
imminent_met_x = 2
imminent_met_y = 7
<SERVER>:: Invio avviso di impatto imminente.
--------------------------------------------------------------------------
```

Il server visualizza graficamente lo stato in cui meteoriti e navicella si ritrovano in un dato istante temporale.
Se inoltre, la navicella è solo a una o due celle di distanza sotto un meteorite, verrà inviato un messaggio di Warning per segnalare di un impatto imminente.

```
<CLIENT>:: WARNING:: Impatto Imminente con il meteorite a coordin
ate (2, 7)!!! Spostarsi!!!
```

Dopo un numero di cicli vediamo la situazione in cui la navicella si trova in una condizione di impatto imminente con un meteorite che si trova sulla sua stessa colonna.
L'alert viene inviato dal server qualora la navicella si trovi ad una cella di distanza dal meteorite o subito sotto di essa.

```
<SERVER>:: Collisione rilevata con il meteorite in posizione (3, 7)! La navice
lla è stata distrutta.
```

Riprendendo la situazione nella figura precedente, se muoviamo la navicella verso l'alto, questa impatterà il meteorite ,che tramite la funzione Update_meteorites(), si sposta verso il basso. Di conseguenza, il server rileva la collisione ed invia un avviso di collisione al client, il quale si disconnetterà.