Matrix Goose Game Project

Luca Pietrogrande

Matricola 1070408

Laboratorio di Ingegneria Informatica

Corso di Laurea in Ingegneria dell'Informazione
Universitá degli Studi Di Padova
30 giugno 2016

Indice

1 Introduzione e obbiettivi del progetto

1.1 Cos'é Matrix Goose Game?

Matrix Goose Game é una variante del classico gioco da tavolo, il Gioco dell'Oca, in cui due giocatori hanno la possibilitá di spostarsi in nove direzioni in un campo a due dimensioni.
Lo scopo del gioco consiste nello spostarsi dalla propria casella iniziale fino alla propria casella finale.
Il campo di gioco é rappresentato da una matrice di altezza x e larghezza y variabili, i due giocatori hanno come caselle iniziali e finali, rispettivamente, le coppie di celle con indici $[0,0]$ e $[x-1,y-1]$ e con indici $[0,y-1]$ $[x-1,0]$.
Le altre celle possono essere vuote o contenere un valore intero che varia tra 1 e n con $n \in [3-9]$ a seconda della grandezza del campo.
Ad ogni turno un giocatore lancia i dadi e si sposta di un numero di passi pari al risultato ottenuto nella direzione che preferisce, se la cella di arrivo non é vuota il giocatore puó scegliere nuovamente la direzione e spostarsi di un numero di passi pari al valore della casella appena raggiunta.
In quest'applicazione si é voluto sviluppare il gioco con un architettura Client-Server implementata in linguaggio C, potendo quindi gestire la comunicazione tra le due parti attraverso i socket.
In particolare i due giocatori sono un utente, il lato client, ed il computer, il lato server.

1.2 Principi di progetto

L'applicazione é stata implementata in linguaggio C, con un'architettura di tipo Client-Server, ed é composta da quattro file sorgente: *GameServer.c*, *GameClient.c*, *GameUtility.c* e *GameUtility.h*.
GameServer.c e *GameClient.c* se compilati forniscono i file eseguibili di Server e Client, mentre *GameUtility.c* e *GameUtility.h* sono una libreria dinamica ed il suo file di header. Nella progettazione dell'applicazione sono stati seguiti due principi di programmazione che di seguito vengono illustrati.

1.2.1 Computazione lato server

Seppure lo specifico contesto di un Gioco dell'Oca su matrice non richieda una particolare potenza di calcolo, in questo progetto si é deciso di privilegiare l'elaborazione di tutti i dati mediante il server.
Il server quindi é l'unica delle due parti che ha accesso alla libreria di utilitá e che esegue i calcoli necessari al gioco. Questo perché si suppone che in linea generale l'unitá dedicata al lato server abbia

una potenza di calcolo maggiore rispetto a quella utilizzata al lato client.
Il lato client si limita a ricevere gli input da parte dell'utente e a comunicarli al server, di seguito una volta ricevuta l'elaborazione da parte del server la comunica all'utente.

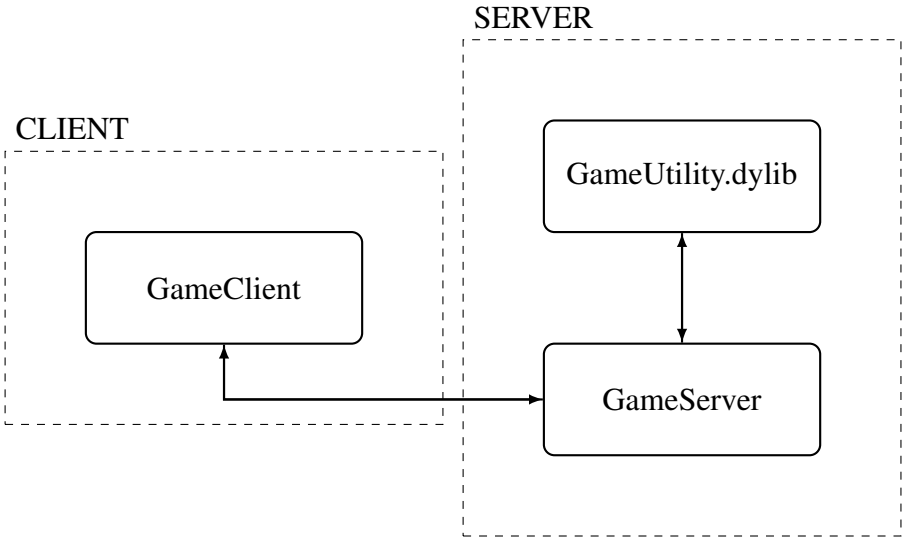

Figura 1.1: Schema a blocchi di interazione tra i file.

1.2.2 Limitazione della Comunicazione

Nell'ottica della programmazione di un'applicazione di tipo Client-Server, seppur applicando varie tecniche di programmazione difensiva, non si puó avere totale controllo sulle funzione incaricate alla comunicazione attraverso i socket, le quali possono dare errori dipendenti dallo stato della connessione e quindi non direttamene gestibili a livello di codice.
Da questa considerazione si é deciso di limitare la comunicazione tra Client e Server ai dati strettamente necessari ai fini del gioco a discapito, come si spiegherá in seguito, di qualche locazione di memoria.

2 Architettura e progettazione dell'applicazione

L'architettura Client-Server utilizzata é piuttosto basilare, un Client gestisce gli input da parte dell'utente e un Server simula un avversario stampando ad output solo informazioni relative alla connessione.
L'esecuzione quindi si compone di varie fasi.

2.1 Connessione

L'esecuzione di *GameServer.exe* porta a lato server la creazione di un socket per la connessione con un predeterminato numero di porta e pone il Server in stato di ascolto. L'esecuzione di *GameClient.exe*, in generale su una qualsiasi macchina connessa alla rete, porta anche a lato client la creazione di un socket per la connessione. Il Client tenta di connettersi al Server attraverso l'indirizzo IP e il numero di porta specificati. Se il Server é in fase di ascolto e la sua coda non é piena si stabilisce la connessione. Se la coda é vuota il gioco avviene subito, altrimenti il Client deve aspettare che il Server si liberi.

2.2 Configurazione della partita

Ad inizio partita l'utente é chiamato ad inserire le dimensioni (altezza x e larghezza y) del campo e queste vengono comunicate al Server.
Ricevute le coordinate del campo, il Server, attraverso la libreria di utilitá *GameUtility.dylib*, riempie la matrice del campo con un numeri interi casuali tra 0 e un valore massimo dipendente dalle dimensioni scelte. Il campo viene poi convertito in un puntatore di tipo *char** includendo i caratteri ASCII necessari alla formattazione ottimale per la stampa.
Una volta ottenuta la matrice "stampabile" del campo, il Server la invia al Client che d'ora in poi potrá ricevere solamente le comunicazioni relative all'aggiornamento delle posizione delle pedine.

2.3 Gioco

L'inizio della mossa vede il Server generare casualmente il valore dei due dadi e comunicarli al Client. L'utente sceglie la direzione in cui spostarsi e questa viene inviata al Server.

Il Server elabora la mossa dell'utente. Se la casella di arrivo contiene un valore, vi é un'ulteriore scambio di comunicazioni tra le due parti per far conoscere la nuova direzione scelta dall'utente, questo avviene finché la casella d'arrivo non é vuota.
Successivamente il Server elabora la propria mossa ed invia al Client le posizioni aggiornate di entrambe le pedine.

2.4 Termine della partita

Dopo ogni mossa il Server controlla, prima dell'invio delle posizioni aggiornate, se uno dei due giocatori ha raggiunto la casella finale e in questo caso comunica al Client che la partita é finita.
L'utente ha la possibilitá di giocare ancora o di uscire, se decide di non giocare ulteriormente il successivo utente in coda puó iniziare la propria partita.

3 Sviluppo e Implementazione

In questa sezione di descriveranno nel dettaglio i frammenti di codice e le funzioni dedicati alle fasi precedentemente descritte. Ci si soffermerá sulle scelte di programmazione effettuate.

3.1 Connessione

3.1.1 Server-side

Per facilitá di lettura sono stati separati due frammenti contigui di codice presenti nel metodo main() del file *GameServer.c*.
Nel primo frammento si prepara la variabile *serverAddress* di tipo *struct sockaddr_in* per la creazione del socket desiderato. Ogni campo é descritto dai commenti in linea e le funzioni *htonl(int x)* e *htons(int x)* sono necessarie per una codifica dei numeri interi coerente con quella utilizzata a livello di rete.
La costante MAX_QUEUE rappresenta il numero massimo di Client in coda al Server ed é stata scelta pari a 2 per garantire un tempo di attesa ragionevole.

```
//server socket descriptor
int serverSocket;
. . .
//constant for the maximun number of possible connection
int const MAX_QUEUE = 2;
//server port number
int serverPort=1748;
//struct to describe the server address
struct sockaddr_in serverAddress;
//address type is set to internet
serverAddress.sin_family=AF_INET;
//server accept any address
serverAddress.sin_addr.s_addr=htonl(INADDR_ANY);
//server port is set to the specified number
serverAddress.sin_port=htons(serverPort);
```

Listing 3.1: *GameServer.c*, funzione *main()*

Nel secondo frammento vi é la creazione effettiva del socket con le specifiche desiderate, in caso di errore si richiama la funzione *closeWithError(char* s)* di cui si illusterá il funzionamento nella sezione 3.4.1.

```
//attempting to create the socket
if((serverSocket=socket(PF_INET,SOCK_STREAM,IPPROTO_TCP))<0)
{
perror("function socket() for socket creation failed");
exit(1);
}
//attempting to bind the specified address to socket
if(bind(serverSocket,(struct sockaddr *) &serverAddress, sizeof(
   serverAddress))<0){. . .}
//attempting to set server on listening
if(listen(serverSocket,MAX_QUEUE)<0){. . .}
. . .
//accepting a new connection
if((clientSocket=accept(serverSocket,(struct sockaddr *) &clientAddress, &
   clientLen))<0)
{
closeWithError("function accept() for accepting a new client failed");
}
else
{
printf("\nSuccesfully connected with a new Client\n");
}
```

Listing 3.2: *GameServer.c*, funzione *main()*

3.1.2 Client-side

A lato client, dopo aver creato il socket correttamente, si cerca la connessione verso il Server attraverso l'indirizzo IP e la porta specificati tramite la struttura *serverAddress* di tipo *struct sockaddr_in*.
Si noti che il numero di porta é lo stesso utilizzato dal Server e che in questa versione dell'applicazione l'indirizzo IP scelto corrisponde all'indirizzo di Loopback, poiché Server e Client sono eseguiti sullo stesso computer.

```
//client socket descriptor
int clientSocket;
```

```
//server port number
int serverPort=1748;
//struct to describe the server address
struct sockaddr_in serverAddress;
//address type is set to internet
serverAddress.sin_family=AF_INET;
//client will connect to the loopback address (server on this machine)
serverAddress.sin_addr.s_addr=inet_addr("127.0.0.1");
//client will connect to the specified port
serverAddress.sin_port=htons(serverPort);
//attempting to create the socket
if((clientSocket=socket(PF_INET,SOCK_STREAM,IPPROTO_TCP))<0)
{
  DieWithError("function socket() for socket creation failed");
}
//attempting to connect to the Server
if(connect(clientSocket,(struct sockaddr *) &serverAddress, sizeof(
   serverAddress))<0)
{
  DieWithError("function connect() for connection to server failed");
}
printf("\nSuccesfully connected to the Game Server\n");
```

Listing 3.3: *GameClient.c*, funzione *main()*

3.2 Configurazione della partita

3.2.1 Server-side

Dopo aver allocato la memoria per contenere le informazioni necessarie al gioco, il Server tenta di ricevere le coordinate inserite dall'utente.
Se il metodo *recv()* va a buon fine, si procede con la creazione del campo. Dopo aver allocato la memoria necessaria a contenere il campo, in versione puntatore ad *int* e in versione puntatore a *char*, si invia la seconda al Client.
Inviata la versione *char** del campo, il Server dealloca la memoria utilizzata da questa in quanto le elaborazioni di gioco verrano eseguite solo sulla versione *int**. In caso di errori di connessione si rimanda alle sezione 3.4.1.

```
//field coordinates, wx[0] is the number of raws and wx[1] of columns
wx=malloc(2*sizeof(int));
//the results of the two dices
ds=malloc(2*sizeof(int));
//the move done by the server or by the client
move=malloc(3*sizeof(int));
//the positions of both players, server (0-1) and the client (2-3)
pawns=malloc(4*sizeof(int));
printf("\n\nMatch began\n\n");
//attempting to receive coordinates to build the field
if(recv(clientSocket,wx,2*sizeof(int),0)<=0)
{
closeWithError("function recv() for receiving coordinates failed");
}
else
{
  printf("\nNumber of raws: %d\nNumber of colums: %d\n", *(wx), *(wx+1));
}
totc=(*(wx)**(wx+1));
rs=(*(wx));
cs=(*(wx+1));
//setting intial positions
*(pawns)=0;
*(pawns+1)=cs-1;
*(pawns+2)=0;
*(pawns+3)=0;
m=1;
field=malloc(totc*sizeof(int));
stringField=malloc((2*totc+rs)*sizeof(char));
setField(field,totc,cs);
stringField=printField(field,rs,cs);
//attempting to send the field
if(send(clientSocket,stringField,((2*totc+rs)*sizeof(char)),0)<0)
{
  closeWithError("function send() for sending field failed");
}
  else
{
  printf("\nField has been sent to the client\n");
```

```
}
//setting the sent string with the field free
free(stringField);
```

Listing 3.4: *GameServer.c*, funzione *main()*

3.2.2 Client-side

A lato client, l'utente inserisce le coordinate finche non si soddisfano i vincoli richiesti. Queste vengono inviate al Server che risponde con il puntatore a *char* del campo di gioco costruito.
D'ora in poi il Client riceverá solo le posizioni aggiornate attraverso il puntatore *pawns* a tipo *int* e modificherá il *proprio* campo di gioco prima di stamparlo nuovamente.
Una possibile alternativa consisteva nell'inviare dopo ogni mossa un campo di gioco aggiornato al Client e stamparlo direttamente senza mantenerne una copia stabilmente in memoria, in questo modo tuttavia si sarebbe aumentata considerevolmente la mole di dati da trasmettere tra le due parti.
In caso di errori di connessione si rimanda alla sezione 3.4.1.

```
//the lower bound for major fields
int const SIZE_B=100;
//the minimum size of dimensions for minor fiels
int const MIN_D=4;
//the minimum size of dimensions for major fields
int const MIN_L=10;
. . .
//xy[0] is the number of number of raws, xy[1] is the number of colums
xy=malloc(2*sizeof(int));
//the results of the two dices
ds=malloc(2*sizeof(int));
//pointer to the client and server positions(server 0-1 and client 2-3)
pawns=malloc(4*sizeof(int));
///game field building with the user
do
{
  printf("\nLet's build the game field \nMinimum dimensions value for
   minor fields: \%d\nMinimum dimensions value for major fields: \%d\
   nMajor fields from a total boxes of \%d\nInsert the number of raws you
   want: ",MIN_D,MIN_L,SIZE_B);
      scanf("%d", xy);
        fflushscanf;
```

```
        printf("\nInsert the number of colums you want: ");
        scanf("%d", xy+1);
        fflushscanf;
}while(*xy<MIN_D || *(xy+1)<MIN_D || (((*xy<MIN_L)||(*(xy+1)<MIN_L))&&(*xy
   **(xy+1)>=SIZE_B)));
//attempting to send the coordinates to the Server
if(send(clientSocket,xy,2*sizeof(int),0)<0)
{
        DieWithError("function send() for sending coordinates failed");
}
printf("\nCoordinates have been communicated to the server\nYou might wait
    the server to be free\n");
//the number of cells of the field matrix
totc=(*(xy)**(xy+1));
//the number of raws of the field matrix
rs=(*(xy));
//the number of colums of the field matrix
cs=(*(xy+1));
//setting the coordinates free
free(xy);
//char pointer that contains the formatted field
printedField=malloc((2*totc+rs)*sizeof(char));
//attempting to receive the field string
if(recv(clientSocket,printedField,(2*totc+rs)*sizeof(char),0)<=0)
{
        DieWithError("function recv() for receiving the field failed");
}
```

Listing 3.5: *GameClient.c*, funzione *main()*

3.3 Gioco

Rimandando all'appendice, per la lettura dei frammenti di codice che riguardano la gestione della comunicazione tra Server e Client ad ogni mossa, ci si sofferma ora sulle modalitá di elaborazione dei dati attraverso le funzioni della libreria dinamica *GameUtility.c*.

3.3.1 Inizializzazione del campo di gioco

L'inzializzazione di un campo di gioco casuale é affidata alla funzione *setField(int* settingField, int n, int cl)*. I parametri della funzione sono *settingField*, il puntatore *int** al campo da inizializzare, *n*, il numero di elementi della matrice del campo, e *cl*, il numero di elementi di una riga.
Si imposta, una volta per tutta l'esecuzione dell'applicazione, il seed della sequenza di numeri casuali. Si sceglie il valore massimo c che puó essere contenuto in ogni cella in funzione del valore di n. Si inizializzano tutte le celle della matrice con valori in $[0-n]$.
Si porta a 0 il 90% delle celle per garantire una maggiore giocabilitá. Infine si portano a 0 anche le celle di arrivo per i due giocatori per poter garantire la fine della partita.

```
//local variables which saves field parameters
int rs;
int cs;
int totc;
. . .
//@param settingField the matrix of the field to set up with random values
   .
//@param n the number of elements of the matrix.
//@param cl the number of elements in a raw
void setField(int* settingField,int n,int cl)
{
    //setting local variable with actual values of total cells
    totc=n;
    //setting the cells with random values between 0 and c depending on n
    srand(time(NULL));
    int c=(4*n+100-1)/100;
    if(c>9)
    {
      c=9;
    }
    for(int i=0;i<n;i++)
    {
      *(settingField+i)=rand()\%c;;
    }
    //setting the 90 percent of the cells to 0;
    srand(time(NULL));
    for(int i=0;i<n;i++)
    {
```

```
      if((rand()\%50)<45)
      {
        *(settingField+i)=0;
      }
    }
    *(settingField)=0;
    *(settingField+n-cl)=0;
}
```

Listing 3.6: *GameUtility.c*, funzione *setField(int* settingField, int n, int cl)*

3.3.2 Mossa dell'utente

La mossa dell'utente avviene specificando il numero di passi, che corrisponde al risultato ottenuto con il lancio di due dadi, la direzione scelta dall'utente e la posizione iniziale.
Il lancio dei due dadi é ottenuto con la doppia invocazione del metodo *dice()*. Successivamente la funzione *updatePosition(int* move, int quantity)* si occupa di aggiornare la posizione dell'utente.
La direzione specificata puó essere una qualsiasi delle 9 possibili ed in caso di attraversamento dei limiti del campo in una o entrambe le componenti, la pedina prosegue rientrando dal lato opposto.
L'aggiornamento della posizione avviene separatamente per le due componenti mediante l'utilizzo di 2 strutture condizionali *switch{ case ... }*.

```
//local variables which saves field parameters
int rs;
int cs;
int totc;
//the size threshold to consider a field as small or large
int const SIZE_THRESHOLD=100;
. . .
//@param move pointer to the specified move, move points to intial raw
   coordinate
//move+1 points to intial column coordinate
//move+2 points to the direction to walk towards
//@param quantity the number of cells to walk
void updatePosition(int* move,int quantity)
{
    int raw=*(move);
    int column=*(move+1);
    int direction=*(move+2);
```

```
    //update x
    switch (direction) {
      case 1:
      case 2:
      case 3:
        *(move)=(*(move)+quantity)\%(rs);
        break;
      case 7:
      case 8:
      case 9:
        *(move)=(rs+(*(move)-quantity))\%(rs);
        break;
    }
    //update y
    switch (direction) {
      case 3:
      case 6:
      case 9:
          *(move+1)=(*(move+1)+quantity)\%(cs);
        break;
      case 1:
      case 4:
      case 7:
          *(move+1)=(cs+(*(move+1)-quantity))\%(cs);
        break;
    }
}
```

Listing 3.7: *GameUtility.c*, funzione *updatePosition(int* move, int quantity)*

```
//the size threshold to consider a field as small or large
int const SIZE_THRESHOLD=100;
. . .
//@return a integer 1-6 for large fields, 1-4 for small fields.
int dice()
{
    if(totc>SIZE_THRESHOLD)
    {
      return rand()\%6+1;
    }
```

```
    return rand()\%4+1;
}
```

Listing 3.8: *GameUtility.c*, funzione *int dice()*

3.3.3 Mossa del computer

La mossa del computer avviene, come per l'utente, attraverso i metodi *dice()* e *updatePosition()*.
La direzione scelta dal computer é il valore di ritorno della funzione *bestChoice(int x,int y,int distance)* di cui si illustra ora il funzionamento. La funzione valuta la direzione migliore in funzione della posizione di partenza e della quantitá di passi da eseguire.
Inizialmente valuta se si trova in una posizione giá allineata con la casella di arrivo, in tal caso restituisce il valore della direzione che porta la pedina piú vicina a tale casella. Altrimenti valuta con che possibili direzioni scelte ci si puó allineare alla casella di arrivo e restituisce il valore della direzione corrispondente.
Se non é possibile allinearsi in alcun modo alla casella finale la funzione restituisce un valore di direzione casuale.

```
//local variables which saves field parameters
int rs;
int cs;
. . .
//@param x points to intial raw coordinate.
//@param y points to intial column coordinate.
//@param distance the number of cells to walk.
//@return the number of the best way to walk towards
int bestChoice(int x,int y,int distance)
{
  int b;
  //if already diagonally aligned
  if(y==(rs-1-x))
  {
    return 1;
  }
  //if already orizzontaly aligned
  if(x==(rs-1))
  {
    if((cs+(y-distance))\%(cs)<(y+distance)\%(cs))
    {
```

```
      return 4;
    }
    return 6;
  }
  //if already vertically aligned
  if(y==0)
  {
    if((rs+(x-distance))\%(rs)<(x+distance)\%(rs))
    {
      return 8;
    }
    return 2;
  }
  //trying to orizzontaly or diagonnally align to goal
  if((rs+(x-distance))\%(rs)==(rs) || rs-1-(rs+(x-distance))\%(rs)==y)
  {
    return 8;
  }
  if((x+distance)\%(rs)==(rs) || rs-1-(x+distance)\%(rs)==(y))
  {
    return 2;
  }
  //trying to vertically or diagonally align to goal
  if((cs+(y-distance))\%(cs)==0 || (cs+(y-distance))\%(cs)==rs-1-x)
  {
    return 4;
  }
  if((y+distance)\%(cs)==0 || (y+distance)\%(cs)==rs-1-x)
  {
    return 6;
  }
  //giving a casual direction
  do
  {
    b=rand()\%9+1;
  }while(b==5);
  return b;
}
```

Listing 3.9: *GameUtility.c*, funzione *int bestChoice(int x, int y, int distance)*

3.4 Termine della partita

La struttura logica condizionale di entrambi i sorgenti é costituita da due cicli annidati del tipo *while(a=='Y') { ... while(!quit) { ... } ... }* come mostrato in figura.

```
while(a=='Y')
{
  //Configurazione della partita
  . . .

  while(!quit)
  {
  //Gioco
  . . .

  //Controllo vittoria
  }
}
```

Listing 3.10: *GameClient.c* e *GameServer.c*, funzione *main()*, struttura condizionale

Le due variabili flag sono *a* di tipo *char* e *quit* di tipo *int*.
La variabile *a* assume i valori 'Y' o 'N' e rappresenta la risposta dell'utente relativa allo svolgimento di un ulteriore partita, la variabile *quit* assume valori 0 se ancora non c'é un vincitore, 1 se il Server é il vincitore e 2 se il Client é il vincitore.

La fase di controllo vittoria é elaborata dalla funzione *int winner(int* ps)* nella libreria di utilitá. Il Server la richiama e successivamente comunica il risultato al Client.
La funzione *int winner(int* ps)* ha come parametro un puntatore alle 4 coordinate delle due pedine e restituisce un valore di ritorno diverso a seconda del vincitore. Si noti come in caso di vittoria di entrambi i giocatori la partita continui senza la vittoria di alcun giocatore.

```
//local variables which saves field parameters
int rs;
int cs;
int totc;
//@param ps 4 integers pointer to the pawns
//ps points to server raw coordinate
```

```
//ps+1 points to server column coordinate
//ps+2 points to client raw coordinate
//ps+3 points to client column coordinate.
//@return 1 server is the winner, -1 client is the winner, 0 there is no
   winner.
int winner(int* ps)
{
  //flags to indicate winners
  int s=0;
  int c=0;
  //if server is winner
  if((*(ps)==(rs-1)) && (*(ps+1)==0))
  {
    s=1;
  }
  //if client is winner
  if((*(ps+2)==(rs-1)) && (*(ps+3)==(cs-1)))
  {
    c=1;
  }
  return 0-c+s;
}
```

Listing 3.11: *GameUtility.c*, funzione *int winner(int* ps)*

Il Server richiama la funzione *int winner(int* ps)*, verifica se c'é un vincitore ed in caso di vittoria imposta la coordinata *x* del vincitore a -1. Successivamente le coordinate aggiornate vengono comunque comunicate al Client.
La variabile *win* é utilizzata come flag al posto di *quit*, la variabile utilizzata nel Client, per rimanere nell'esecuzione del ciclo *while()* piú interno.
Successivamente se la partita é terminata si riceve la risposta dell'utente per proseguire o meno con un'altra partita. Si noti come tutta la struttura condizionale del Server sia inclusa in un ciclo infinito che consente una volta chiusa la connessione con un client di gestire la successiva.

```
while(1)
{
  //Creazione socket e connessione ad un Client
  . . .

  while(c=='Y')
```

```
{
  //Configurazione della partita
  . . .
  while(!win)
  {
    //Gioco
    . . .
    win=winner(pawns);
    //preparing notice of the winner in order to send it to the client
    if(win>0)
    {
      *(pawns)=-1;
    }
    if(win<0)
    {
      *(pawns+2)=-1;
    }
    //sending the updated position
    if(send(clientSocket,pawns,(4*sizeof(int)),0)<0)
    {
      closeWithError("function send() for sending positions failed");
    }
  }
  //receiving the answer from the client
  if(recv(clientSocket,&c,sizeof(char),0)<=0)
  {
    closeWithError("function recv() for receiving answer failed");
    c='N';
  }
  //reinitiating the win flag
  win=0;
  //setting the field free only if it had been previously allocated
  if(m==1)
  {
    free(field);
    m=0;
  }
  //setting all pointers free
  free(ds);
```

```
    free(move);
    free(pawns);
    printf("\n\n-- Match ended --\n\n");
  }
  //reinitiate the answer flag
  c='Y';
  printf("\n\nClient disconnected\n\n");
  //closing the client socket
  close(clientSocket);
}
```

Listing 3.12: *GameServer.c*, funzione *main()*

Il Client controlla se c'é un vincitore tramite le coordinate x delle posizioni appena ricevute e in caso di una coordinata pari a -1 setta la variabile flag *quit* per uscire dal ciclo, ad un valore pari a 1 se il vincitore é il computer e ad un valore pari a 2 se il vincitore é l'utente.
Terminata la partita chiede all'utente se vuole giocarne un'altra e comunica la decisione al Server. Se l'utente decide di non giocare ulteriormente, il Client chiude la propria connessione e termina l'esecuzione.

```
while(c=='Y')
{
  //Configurazione della partita
  . . .

  while(!win)
  {
    //Gioco
    . . .

    //if the server is the winner
    if(*(pawns)==-1)
            {
              *(pawns)=rs-1;
              quit=1;
          }
    //if the client is the winner
          if(*(pawns+2)==-1)
          {
              *(pawns+2)=rs-1;
```

```
            quit=2;
          }
        //printing positions on the field string
          if((*pawns)!=*(pawns+2)||(*pawns+1)!=(*pawns+3))
        {
            *(printedField+(*(pawns+2))*(2*cs+1)+2*(*(pawns+3)))='P';
            *(printedField+(*(pawns))*(2*cs+1)+2*(*(pawns+1)))='C';
        }
          else
          {
            *(printedField+(*(pawns+2))*(2*cs+1)+2*(*(pawns+3)))='B';
          }
          printf("\n\%s\n",printedField);
      }
    if(quit==1)
      {
        printf("\nYou lost, let's play another match");
      }
      if(quit==2)
      {
        printf("\nYou won, nice play");
      }
do
      {
        printf("\nDo you want to play another match? (Y/N) : ");
          scanf("\%c",&a);
          fflushscanf;
      }while(!(a=='Y'||a=='N'));
      //sending the client answer to the server
      if(send(clientSocket,&a,sizeof(char),0)<0)
      {
          DieWithError("function send() for sending client answer failed
 ");
      }
      //reinitiating the quit flag
      quit=0;
      //setting all pointers free
      free(printedField);
      free(ds);
```

```
        free(pawns);
}
/setting socket to closed
close(clientSocket);
```

Listing 3.13: *GameClient.c*, funzione *main()*

3.4.1 Comunicazione fallita e chiusura forzata

Spesso ci si puó trovare di fronte ad una situazione in cui un Client decide di interrompere la partita forzatamente. Questo puó avvenire ad esempio per la pressione della sequenza di tasti *ctrl+v*, per la chiusura del terminale o per la perdita di connessione.
In questo caso il Server decide di chiudere la connessione con il Client e di rimanere in ascolto per altre connessioni.
Le funzioni di comunicazione utilizzate *recv()* e *send()* hanno comportamenti di default differenti di fronte alla mancanza di connessione con il Client cercato.

La funzione *recv()* restituisce come valore di ritorno un intero che rappresenta la lunghezza del messaggio ricevuto o un valore negativo in caso di errore.
In caso di mancata connessione restituisce un valore nullo. Di conseguenza per gestire l'errore di trasmissione e la chiusura forzata del Client con cui si é connessi é sufficiente controllare che il valore di ritorno di *recv()* sia ≤ 0. Di seguito vi é il frammento di una gestione di errore per *recv()*.

```
//receiving the direction chosen by the client
if(recv(clientSocket,move+2,sizeof(int),0)<=0)
{
  closeWithError("function recv() for receiving direction failed");
}
else
{
  printf("\nDirection has been received\n");
}
```

Listing 3.14: *GameServer.c*, funzione *main()*

La funzione *send()* restituisce come valore di ritorno un intero che é minore di 0 se ci sono stati errori di trasmissione. Per gestire tali errori quindi é necessario controllare se il valore di ritorno é negativo. Di seguito vi é un frammento di codice di esempio.

```
//sending the dices results to client
if(send(clientSocket,ds,(2*sizeof(int)),0)<0)
{
  closeWithError("function send() for sending dices failed");
}
else
{
  printf("\nDices have been sent to the client\n");
}
```

Listing 3.15: *GameServer.c*, funzione *int main()*

Tuttavia se il Client perde la connessione, quando il Server tenta di inviare un messaggio tramite la funzione *send()* si genera un segnale SIG_PIPE.
Il comportamento di default del Server di fronte a tale segnale é quello di terminare l'esecuzione. Considerato che non é accettabile che la perdita di connessione da parte di un client porti il Server a terminare l'esecuzione, é necessario catturare il segnale di SIG_PIPE ed impedire che il Server termini. Di seguito si riporta il codice con cui si definisce l'handler per la gestione del segnale e la funzione richiamata da esso.

```
//handling the client shutdown
struct sigaction client_behaviour;
client_behaviour.sa_handler=setting_handler;
sigemptyset(&client_behaviour.sa_mask);
client_behaviour.sa_flags=0;

sigaction(SIGPIPE,&client_behaviour,NULL);
```

Listing 3.16: *GameServer.c*, funzione *main()*

```
//@param sign the signal to be handle
void setting_handler(int sign)
{
  c='N';
  win=1;
  printf("\n\nClient has closed connection\n\n");
}
```

Listing 3.17: *GameServer.c*, funzione *setting_handler(int sign)*

Di seguito anche l'handler per la gestione della chiusura forzata del Server tramite la sequenza di tasti *ctrl+v* e la funzione da esso richiamata, il quale cattura il segnale SIGINT e prima di chiudere l'applicazione chiude anche le connessioni attive.

```
//handling the server shutdown
struct sigaction server_behaviour;
server_behaviour.sa_handler=closing_handler;
sigemptyset(&server_behaviour.sa_mask);
server_behaviour.sa_flags=0;

sigaction(SIGINT,&server_behaviour,NULL);
```

Listing 3.18: *GameServer.c*, funzione *main()*

```
//@param sign the signal to be handle
closing_handler(int sign)
{
  if(close(clientSocket)==0)
  {
    printf("\n\nClient socket has been correctly closed\n\n");
  }
  if(close(serverSocket)==0)
  {
    printf("\n\nServer socket has been correctly closed\n\n");
  }
  exit(0);
}
```

Listing 3.19: *GameServer.c*, funzione *closing_handler(int sign)*

Infine si riporta il codice delle funzioni *closeWithError(char* message)* e *DieWithError(char* message)*, che gestiscono il fallimento nella trasmissione per i metodi *recv()* e *send()*, rispettivamente nel Server e nel Client.

```
//@param message the string to show on standard output.
void closeWithError(char* message)
{
  perror(message);
  c='N';
  win=1;
```

```
}
```

Listing 3.20: *GameServer.c*, funzione *closeWithError(char* message)*

```
//@param message the string to show on standard output.
void DieWithError(char* message)
{
  perror(message);
  exit(1);
}
```

Listing 3.21: *GameClient.c*, funzione *int DieWithError(char* message)*

4 Valutazione e Collaudo

Questa prima versione dell'Applicazione consente l'utilizzo di Client e Server sulla stessa macchina poiché, per fini didattici, é stato specificato per la connessione al Server l'indirizzo IP di Loopback "127.0.0.1".
Per eseguire l'Applicazione é necessario, dopo aver compilato con le specifiche contenute nel file *CMakeLists.txt*, far partire l'eseguibile *GameServer.exe*. Successivamente si lancia l'eseguibile *GameClient.exe* e si puó iniziare a giocare.
Nel caso si lanci un'altra istanza di *GameClient.exe* il secondo Client viene messo in coda ed attende che il primo abbia terminato le proprie partite.
Come scritto in precendenza, la coda del Server accetta al massimo due client, questa scelta permette di assicurare ad un qualsiasi Client in coda di essere il prossimo ad essere servito.
Infine, una nota a proposito della giocabilitá sottolinea come i numeri corrispondenti alle direzioni possibili siano coerenti con la disposizone dei numeri nel tastierino numerico presente nelle tastiere standard.

5 Conclusioni e lavoro futuro

Il linguaggio C, in questa applicazione, ha fornito al programmatore gli strumenti utili per poter implementare piú aspetti della programmazione.
Principalmente si é potuta gestire la comunicazione tra due unitá di rete distinte attraverso i socket.
Inoltre le funzioni per l'allocazione dinamica della memoria hanno consentito di ottimizzare lo spazio occupato in esecuzione, liberando le celle di memoria con dati non piú utili.
Infine la notazione dei puntatori é stata utilizzata per agevolare le operazioni di calcolo delle mosse da parte del Server.

Il naturale ulteriore sviluppo dell'applicazione va verso una gestione multithreading delle partite, con il Server che gioca contemporaneamente con piú client.

6 Appendice

Di seguito si riportano i frammenti di codice relativi alla comunicazione tra Client e Server ad ogni mossa.

6.1 Server-side

```
//rolling the dice for the client move
*(ds)=dice();
*(ds+1)=dice();
//sending the dices results to client
if(send(clientSocket,ds,(2*sizeof(int)),0)<0)
{
  closeWithError("function send() for sending dices failed");
 }
 else
{
  printf("\nDices have been sent to the client\n");
}
//receiving the direction chosen by the client
if(recv(clientSocket,move+2,sizeof(int),0)<=0)
{
  closeWithError("function recv() for receiving direction failed");
}
else
{
  printf("\nMove has been received\n");
}
//updating the client position
*(move)=*(pawns+2);
*(move+1)=*(pawns+3);
updatePosition(move,*(ds)+*(ds+1));
//value of the new position for another move
add=*(field+(*move)*cs+*(move+1));
if(send(clientSocket,&add,sizeof(int),0)<0)
```

```
{
  closeWithError("function send() for sending add failed");
}
while(add>0)
{
  //receiving the direction chosen by the client
  if(recv(clientSocket,move+2,sizeof(int),0)<=0)
  {
    closeWithError("function recv() for receiving direction failed");
  }
  updatePosition(move,add);
  add=*(field+(*move)*cs+*(move+1));
  if(send(clientSocket,&add,sizeof(int),0)<0)
  {
    closeWithError("function send() for sending add failed");
  }
}
//saving the updated positions
*(pawns+2)=*(move);
*(pawns+3)=*(move+1);
//roll the dice for the server move
*(ds)=dice();
*(ds+1)=dice();
//updating the server position
path=bestChoice(*(pawns),*(pawns+1),*(ds)+*(ds+1));
*(move)=*(pawns);
*(move+1)=*(pawns+1);
*(move+2)=path;
updatePosition(move,*(ds)+*(ds+1));
//value of the new position for another move
add=*(field+(*move)*cs+*(move+1));
while(add>0)
{
  path=bestChoice(*(move),*(move+1),add);
  *(move+2)=path;
  updatePosition(move,add);
  add=*(field+(*move)*cs+*(move+1));
}
//saving the updated positions
```

```
*(pawns)=*(move);
*(pawns+1)=*(move+1);
```

Listing 6.1: *GameServer.c*, funzione *main()*

6.2 Client-side

```
//cleaning positions on the field string
*(printedField+(*(pawns+2))*(2*cs+1)+2*(*(pawns+3)))=' ';
*(printedField+(*(pawns))*(2*cs+1)+2*(*(pawns+1)))=' ';
//receiving the dices results from server
if(recv(clientSocket,ds,2*sizeof(int),0)<=0)
{
  DieWithError("function recv() for receiving the dices failed");
}
printf("\nRoll of the dice gave \%d + \%d = \%d\n",*(ds),*(ds+1),*(ds)+*(
   ds+1));
do
{
  printf("\nChoose your move typing a number :\n7|8|9\n4| |6\n1|2|3\n");
  scanf("\%d",&choice);
  if(choice<0||choice>9)
  {
    printf("\nYou have to type an integer number beetwen 1-9\n");
  }
}while(choice<0||choice>9);
fflushscanf;
//sending path choice to the server
if(send(clientSocket,&choice,sizeof(int),0)<0)
{
  DieWithError("function send() for sending choice failed");
}
//receiving the value of the new cell after the move
if(recv(clientSocket,&add,sizeof(int),0)<=0)
{
  DieWithError("function recv() for receiving the dices failed");
}
while(add!=0)
{
```

```
  printf("\nYou can walk for \%d more steps\n",add);
  printf("\nChoose your move typing a number :\n7|8|9\n4| |6\n1|2|3\n");
  scanf("\%d",&choice);
  fflushscanf;
  //sending path choice to the server
  if(send(clientSocket,&choice,sizeof(int),0)<0)
  {
    DieWithError("function send() for sending choice failed");
  }
  if(recv(clientSocket,&add,sizeof(int),0)<0)
  {
    DieWithError("function recv() for receiving the dices failed");
  }
}
//receiving the updated positions from server
if(recv(clientSocket,pawns,4*sizeof(int),0)<=0)
{
  DieWithError("function recv() for receiving the positions failed");
}
```

Listing 6.2: *GameClient.c*, funzione *main()*