Daniele Romanini

Politecnico di Milano – sede di Cremona
Corso di Laurea in Ingegneria Informatica
N. Matricola: 790664

Progetto Software

A.A. 2014-2015

Realizzazione del gioco "Briscola" 1 vs 1 in rete locale, tramite l'utilizzo di socket

Requisiti

- Sistema Operativo Unix o derivato (ad esempio Linux o Mac Os X)
- Compilatore C (gcc)
- Connessione a una rete locale
- Software di rete TCP/IP

*Si noti che la connessione a una rete locale è necessaria se si utilizzano dispositivi diversi per giocare. E' possibile anche utilizzare un unico dispositivo.
**Il progetto è stato sviluppato e testato utilizzando due dispositivi: uno con sistema operativo Mac Os X (versione 10.9.5) e l'altro con sistema operativo Linux Mint (versione 16).

L'applicazione sviluppata (che è un'applicazione distribuita) permette il gioco della briscola a due giocatori (avversari), i cui dispositivi sono connessi alla stessa rete locale, comunicando tramite terminale (shell).
I due giocatori sono stati implementati come "client"; pertanto essi dovranno lanciare, dopo la compilazione del relativo codice, l'eseguibile *player_briscola*.
N.B.: Prima di eseguire le applicazioni client, occorre compilare ed eseguire l'applicazione "server" (*server_briscola*). I client infatti si connetteranno al server, che funge da arbitro/mazziere e si occupa della comunicazione tra i due giocatori.
Il server può essere eseguito sia su uno dei due dispositivi utilizzati dai giocatori che su un terzo dispositivo.

Socket - introduzione

I singoli programmi che costituiscono un'applicazione distribuita possono utilizzare i servizi tipici del sistema operativo della macchina sulla quale sono eseguiti e i servizi forniti dal software di rete (TCP/IP nel nostro caso). Ciò avviene invocando l'esecuzione di particolari funzioni che sono fornite già pronte nell'ambito del sistema operativo.
Esiste una API (Application Program Interface, cioè un insieme di definizioni di funzioni utilizzabili dai programmi applicativi) standard per l'utilizzazione dei servizi TCP/IP, nota come *interfaccia SOCKET*.
Per avvisare il compilatore del fatto che il programma contiene riferimenti a queste funzioni è necessario richiedere all'inizio del codice sorgente l'inclusione delle seguenti librerie (contenenti la dichiarazione delle funzioni che verranno poi utilizzate):
<sys/types.h>
<sys/socket.h>
<netinet/in.h>
<arpa/inet.h>

Il software di rete TCP/IP permette a due processi residenti su macchine diverse di comunicare trasferendo dati tra di loro. Esso fornisce una comunicazione pari-a-pari (peer to peer) tra i processi.
La maggior parte delle applicazioni distribuite viene realizzata secondo il modello di cooperazione detto *Client/Server*.
Un processo Server è un processo che offre un servizio ad altri processi. Un Server accetta le richieste che arrivano attraverso la rete, esegue il servizio e fornisce eventualmente un risultato al richiedente.
Un processo Client è un processo che richiede dei servizi a un processo Server e attende normalmente una risposta.

Un Processo P che vuole comunicare con un altro processo Q deve essere in grado di identificare la controparte. Ciò avviene individuando prima di tutto la macchina sulla quale il processo Q è attivo e poi il processo Q stesso nell'ambito dei processi attivi su tale macchina.
L'identificazione della macchina avviene, nelle reti TCP/IP, tramite un indirizzo IP (numero di 32 bit assegnato alla macchina, che la identifica univocamente nell'ambito della rete).

Esempio: 131.175.16.91. Esso è la rappresentazione esterna (stringa) dell'indirizzo IP di una macchina. Esiste una funzione dell'interfaccia socket che converte un numero in formato esterno (stringa) in un indirizzo IP:

(long)inet_addr(indirizzo esterno);

Essa è utilizzata per acquisire da terminale un indirizzo IP nel formato esterno e convertirlo in un vero indirizzo a 32 bit.
Per identificare un processo nell'ambito di una macchina si utilizza un Port, cioè un numero, di solito compreso tra 1024 e 49151. Le porte potrebbero essere non disponibili o occupate da altri processi, e quindi potrebbe essere restituito un errore di connessione.

I due processi che stabiliscono una connessione tra loro sono detti punti terminali (della connessione).
Ogni punto terminale è identificato dall'indirizzo costituito dalla coppia <indirizzo_IP, Port>.
La connessione è quindi identificata dai punti terminali che la costituiscono.
Dopo aver stabilito una connessione i due processi possiedono un canale di comunicazione bidirezionale, orientato allo stream (cioè alla trasmissione di un flusso continuo di byte).

Analizziamo di seguito alcuni aspetti generali per la gestione dei socket, utilizzati anche per lo sviluppo del progetto in questione.

-Passaggio dei parametri al main:

Quando viene lanciato in esecuzione un programma *main* con la classica intestazione:

*void main(int argc, char*argv[])*

il significato dei parametri è il seguente:

- *argc* contiene il numero dei parametri ricevuti
- *argv* è un vettore di puntatori a stringhe, ognuna delle quali è un parametro

Il parametro *argv[0]* contiene sempre il nome del programma lanciato in esecuzione.

Prima di tutto occorre compilare (se non è già stato fatto) e lanciare in esecuzione il server e, in seguito, i client. Per tutto questo utilizziamo il terminale (shell).

Linea di comando per la compilazione:

gcc nomefile.c –o nomeeseguibile

NOTA: Nel nostro caso, per la corretta compilazione occorre compilare, insieme a client e server, anche il file *deffbriscola.c*, contenente le definizioni delle funzioni utilizzate dal programma (si è creato un header file, "*funzionibriscola.h*", contenente l'intestazione delle funzioni utilizzate, opportunamente incluso nei file .c). Occorre quindi digitare le seguenti linee di comando, necessarie al linking dei due moduli:

gcc server_briscola.c deffbriscola.c –o server
gcc player_briscola.c deffbriscola.c –o player

Linea di comando per l'esecuzione del server:

./nomeeseguibileserver NUMEROPORTA

Ad esempio, se il numero della porta è 5000:

./nomeeseguibileserver 5000

5000 sarà quindi argv[1].

Linea di comando per l'esecuzione del server:

./nomeeseguibileclient INDIRIZZOIPHOST NUMEROPORTA

Ad esempio, se il numero della porta è 5000, e l'indirizzo IP è 131.175.16.91:

./nomeeseguibileclient 131.175.16.91 5000

5000 sarà quindi argv[2], mentre *131.175.16.91* sarà argv[1].

Il numero di Port deve essere lo stesso per Server e Client.
L'indirizzo IP deve essere quello della macchina dove è in esecuzione il Server (se si lavora sulla stessa macchina l'indirizzo IP da specificare è 127.0.0.1).

Il numero di porta sarà poi acquisito nel server e nel client tramite una variabile intera *port* e la funzione *atoi* (per convertire una stringa ad un intero), rispettivamente nel seguente modo:

```
porta=atoi(argv[1]);
porta=atoi(argv[2]);
```

NOTA: Se si effettuano due iterazioni di seguito del programma sullo stesso numero di porta, potrebbe essere restituito un errore di connessione dalla parte client poiché la porta "non si è ancora liberata" dall'iterazione precedente (o potrebbe essere occupata da altri processi). Basterà cambiare numero di porta, o aspettare qualche minuto (se la porta non è occupata da altri processi).
E' importante comunque che il numero di porta specificato da linea di comando sia lo stesso per i due client e per il server.

- Struttura indirizzo e inizializzazione

Sia dalla parte Client che dalla parte Server, è necessario definire variabili di tipo *struct sockadrr_in,* adatta a contenere un indirizzo di un punto terminale.
La struttura *sockadrr_in* è definita nel modo seguente:

```
struct sockaddr_in {
        short sin_family;
        u_short sin_port;
        struct in_addr sin_addr;
        char sin_zero[8];
}
```

Significato dei singoli campi:

- *sin_family*: indica la famiglia degli indirizzi; nel nostro contesto questo campo assumerà sempre il valore della costante predefinita AF_INET (Address Family InterNET)
- *sin_port*: questo campo indica il numero di Port; si tratta di un intero a 16 bit senza segno, quindi i suoi valori vanno da 0 a 64k.
- *sin_addr*: questo campo indica l'indirizzo IP; è una struct, nella quale si userà solamente il campo *s_addr* (di tipo u_long, cioè 32 bit senza segno)
- *sin_zero:* campo non utilizzato.

La struttura sockadrr_in è più complessa di quanto strettamente necessario per TCP/IP; ciò è dovuto al fatto che l'interfaccia Socket è stata progettata per funzionare anche su altri protocolli di rete, oltre a TCP/IP.

Nell'applicazione è stata definita una funzione *addr_initialize*, necessaria per inizializzare una variabile di tipo *struct sockadrr_in.*

```
void adrr_initialize(struct sockaddr_in* indirizzo, int port, long IPaddr){
        indirizzo->sin_family = AF_INET;
```

```
        indirizzo->sin_port=htons((u_short) port);
        indirizzo->sin_addr.s_addr=IPaddr;
}
```

Port è il numero di porta (passato da terminale), mentre IPaddr è l'indirizzo IP (passato da terminale nel lato client, mentre per il lato server esso verrà inizializzato a una costante INADDR_ANY).

Dichiarazione delle variabili di tipo indirizzo e il passaggio dei parametri alla funzione *addr_initialize* nella nostra applicazione*:*

- Server:

```
/*definizione del tipo lunghezza di un indirizzo (nel programma è incluso nel file header*/
typedef uint32_t socklen_t;

porta=atoi(argv[1]);

/*dichiarazione delle funzioni indirizzo e lunghezza di essi. Si è definito un array di lunghezza 2, poiché si connetteranno 2 giocatori, cioè 2 client */
struct sockaddr_in server_addr, client_addr[2];
socklen_t client_len[2];
client_len[0]=sizeof(client_addr[0]);
client_len[1]=sizeof(client_addr[1]);

adrr_initialize (&server_addr, porta, INADDR_ANY);
```

- Client:

```
struct sockaddr_in server_addr;

porta=atoi(argv[2]);

adrr_initialize (&server_addr, porta, (long)inet_addr(argv[1]));
```

- Passaggi per stabilre una connessione – Server

1. Creare un socket tramite la funzione *socket()*.
2. Assegnare un indirizzo locale a un socket tramite la funzione *bind()*.
3. Ascoltare connessioni con la funzione *listen()*.
4. Accettare una connessione tramite la funzione *accept()*. Essa normalmente si blocca fino a quando un client non si connette al server.
5. Inviare e ricevere dati.

Vediamo nel dettaglio le varie funzioni sopra esposte:

1. ***socket()***

   int socket(int pf, int type, int protocol);

   pf: famiglia di protocolli (nel nostro caso AF_INET)

   type: Tipo di socket (nel nostro caso SOCK_STREAM, cioè orientato alla trasmissione bidirezionale di un flusso continuo di dati

   protocol: protocollo da usare con la socket per la PF e il type indicati. Posto a 0, come nel nostro caso, indica il protocollo di default per la coppia (pf, type).

   Il valore di ritorno è quello del nuovo descrittore del socket, oppure -1 in caso di errore.

2. ***bind()***

   *int bind (int sockfd, const struct sockaddr * addr, socklen_t addrlen);*

   sockfd: descrittore del socket

   **addr:* indirizzo da assegnare (puntatore a una struttura sockaddr)

   addrlen: lunghezza in byte dell'indirizzo.

   Il valore di ritorno è 0 in caso di successo, -1 altrimenti.

3. ***listen()***

   int listen(int sockfd, int backlog);

   sockfd: descrittore del socket

   backlog: massima lunghezza della coda di connessioni che possono essere tenute pendenti (di solita settata a 5). Una richiesta che arriva quando la coda è piena viene rifiutata e il client riceve un errore dalla *connect()*.

   addrlen: lunghezza in byte dell'indirizzo.

   La funzione restituisce 0 in caso di successo, -1 altrimenti.

4. ***accept()***

*int accept (int sockfd, struct sockaddr * addr, socklen_t* addrlen);*

sockfd: descrittore del socket

addr: puntatore a un buffer che riceve l'indirizzo dell'entità che fa richiesta di connessione. Nel nostro caso le variabili di tipo "indirizzo di sockaddr_in" saranno castate a sockaddr*: *(struct sockaddr*)&nome_struttura*

addrlen: puntatore alla variabile che contiene la lunghezza di *addr*.

Il valore di ritorno è un numero intero non negativo, che sarà il descrittore del socket accettato. In caso di errore restituisce -1.

La funzione accept() pone in stato di attesa una connessione sul socket specificato. Il programma resta quindi bloccato finchè non arriverà una richiesta di connessione da un client.

- Passaggi per stabilire una connessione – Client

1. Creare un socket tramite la funzione *socket().*
2. Connettere il socket all'indirizzo del server con la funzione *connect().*
3. Inviare e ricevere dati.

Vediamo nel dettaglio le varie funzioni sopra esposte:

1. ***socket()***

   Vedi sopra (Server)

2. ***connect()***

   *int connect(int sockfd, , const struct sockaddr * addr, socklen_t addrlen);*

sockfd: descrittore del socket (non ancora connesso)

**addr:* indirizzo del socket con cui dovrebbe essere stabilita la connessione assegnare (puntatore a una struttura *sockaddr*)

addrlen: lunghezza in byte dell'indirizzo *addr*.

Il valore di ritorno è 0 in caso di successo, -1 altrimenti.

Se il socket specificato non è stato associato ad un numero di port locale, la connect genera un numero di port e lo attribuisce al punto di terminazione locale del socket connesso. (Per stampare il valore del port, anche nel server, è necessario prima trasformarlo dal formato di rete a quello dell'host, tramite la funzione *ntohs*).

- Inviare e ricevere dati

Per inviare e ricevere dati (da entrambi i lati, poiché la connessione è bidirezionale) esistono varie funzioni. Nel nostro caso si sono utilizzati le funzioni send(), che permette a un processo di inviare dei dati, e recv(), che permette a un processo di riceverli.

- **send()** :

*int send(int sockfd, , const char*buf, int len, int flags);*

sockfd: descrittore del socket (non ancora connesso)
buf: indica l'inizio della zona di memoria da cui prelevare i dati da spedire.
len: numero di byte da spedire
flags: serve per alcune funzioni speciali. Nel nostro caso, si è utilizzato sempre 0.

Il valore di ritorno è il numero di byte effettivamente inviati, oppure un numero negativo in caso di errore.
La funzione copia i dati da inviare in un'area di memoria di sistema (buffer) e permette al programma di continuare l'esecuzione anche se i dati non sono ancora arrivati a destinazioni. Ha quindi un comportamento non bloccante

- **recv()** :

*int recv(int sockfd, , const char*buf, int len, int flags);*

sockfd: descrittore del socket (non ancora connesso)
buf: indica l'inizio della zona di memoria in cui scrivere i dati ricevuti
len: numero di byte da ricevere
flags: serve per alcune funzioni speciali. Nel nostro caso, si è utilizzato sempre 0.

Il valore di ritorno è il numero di byte effettivamente ricevuti, oppure un numero negativo in caso di errore.

La funzione ha un comportamento bloccante: blocca il programma finché non arrivano dei dati o non si verifica un evento particolare (chiusura connessione o errore).

- Chiudere la connessione

Quando si è conclusa l'interazione occorre interrompere la comunicazione e deallocare le risorse tramite la funzione close(), dove *sockfd* è il descrittore del socket.

int close(int sockfd);

Essa restituisce 0 in caso di successo, -1 in caso di errore.

Fonti:
- *G. Pelagatti – Informatica II. Sistema Operativo Linux e TCP/IP – Progetto Leonardo, Bologna – Febbraio 2008*
- *www.linuxhowtos.org*

Gioco della Briscola

- Implementazione e gestione delle carte, del mazzo e delle mani

1. **Le carte**

   Le carte sono state definite come delle strutture (di tipo *carta*) contenenti il seme e il tipo (cioè il valore).
   Prima di tutto è stato quindi necessario definire i tipi enumerativi *semecarta* e *tipocarta* (in quest'ultima i tipi sono stati messi in ordine di valore)

   typedef enum {DENARI, SPADE, BASTONI, COPPE} semecarta;

```
typedef enum {DUE, QUATTRO, CINQUE, SEI, SETTE, FANTE,
CAVALLO, RE, TRE, ASSO} tipocarta;

typedef struct {
  semecarta seme;
  tipocarta tipo;
}carta;
```

2. **Il mazzo e le mani**

   Il mazzo e le mani sono stati gestiti come liste singolarmente concatenate. Ogni elemento della lista contiene l'informazione, che è di tipo *carta*, e il puntatore al prossimo elemento.

```
typedef struct puntLista {
  carta cartacorrente;
  struct puntLista *prox;
} cardstruct;

typedef cardstruct *listaDiCarte;
```

3. **Funzioni definite e utilizzate per gestire le carte, il mazzo e le mani**

- Funzione per inizializzare una lista (mazzo o mano)

```
void inizializzalist (listaDiCarte *p) {
 *p=NULL;
}
```

- Funzione per inserire una carta in testa a una lista (utilizzata per riempire il mazzo oppure alla manoper aggiungere una nuova carta pescata)

```
void insInTesta (listaDiCarte *p, carta inscarta) {
  cardstruct *punt;
```

```
  punt=malloc(sizeof(cardstruct));
  punt->cartacorrente=inscarta;
  punt->prox=*p;
  *p=punt;
}
```

- Funzione per stampare una lista di carte (utilizzata nello specifico per mostrare all'utente la propria mano).

```
void stampaLista (listaDiCarte *p) {
  cardstruct *punt;
  punt=*p;
  int i=0;
  while(punt!=NULL){
    printf("  %d.: ", i+1);
    stampacarta (punt->cartacorrente);
    punt=punt->prox;
    i++;
  }
  printf("\n");
}
```

/*E' stata implementata anche la funzione *stampaListaConPunti*, che permette di stampare l'elenco delle carte con il rispettivo valore.*/

- Funzione per stampare una specifica carta (la carta è passata come parametro)

```
void stampacarta(carta card){
  char* semiarray[] = {"DENARI", "SPADE", "BASTONI", "COPPE"};
    char* tipiarray[] = {"DUE", "QUATTRO", "CINQUE", "SEI", "SETTE",
"FANTE", "CAVALLO", "RE", "TRE", "ASSO"};
  printf("  %s di %s\n", tipiarray[card.tipo], semiarray[card.seme]);
}
```

- Funzione per eliminare la prima carta da una lista (utilizzata per pescare una carta, dal lato server, e poi inviarla al client. La prima volta che si

invoca questa funzione è per determinare la briscola, che è la prima carta che viene tolta dal mazzo). Ritorna la carta eliminata

```
carta eliminaCarta (listaDiCarte *p) {
  cardstruct *punt;
  carta cartadelete;
  if ((*p)!=NULL){
    punt=*p;
    cartadelete=punt->cartacorrente;
    *p=(*p)->prox;
    free (punt);
  }
  return cartadelete;
}
```

- Funzione per eliminare una specifica carta da una lista (utilizzata per eliminare dalla mano la carta da giocare scelta dall'utente)

```
void eliminaCartaSpecifica(listaDiCarte *l, carta cartapassata) {
  cardstruct *puntCorr, *puntPrec;
  puntPrec=NULL;
  puntCorr=*l;
  int var=0;
  while(puntCorr!=NULL && var==0) {
    if(cartapassata.seme==puntCorr->cartacorrente.seme &&
cartapassata.tipo==puntCorr->cartacorrente.tipo){
      var=1;
    } else{
      puntPrec=puntCorr;
      puntCorr=puntCorr->prox;
    }
  }
  if(puntCorr!=NULL) {
    if (puntCorr==*l)
      *l=puntCorr->prox; //è proprio *l che devo eliminare
    else
      puntPrec->prox=puntCorr->prox; //bypasso
    free(puntCorr);
```

```
  }
}
```

- Funzione per riempire il mazzo appena inizializzato e poi mescolarlo (per questo si chiama la funzione *mescolamazzoarray*)

```
void riempiemescola (listaDiCarte *p){
  carta mazzo[40];
  int cont1, cont2, i=0, cont;
  for(cont1=0; cont1<4; cont1++){
    for(cont2=0; cont2<10; cont2++){
      mazzo[i].seme=cont1;
      mazzo[i].tipo=cont2;
      i++;
    }
  }
   mescolamazzoarray(mazzo); //il mazzo è riempito, chiamo la funzione per mescolarlo
  for (cont=0; cont<40; cont++){
    insInTesta(p, mazzo[cont]);
  }
}



void mescolamazzoarray(carta *mazzo){
  carta cartaappoggio;
  int i=0, j=0, p;
 for (p=10; p!=0; p--){  /*è possibile utilizzare un numero arbitrario per rendere efficace il mescolamento*/
    for(i=40; i!=0; i--){
      srand(time(NULL));
      j=rand()%40;
      cartaappoggio=mazzo[i-1];
      mazzo[i-1]=mazzo[j];
      mazzo[j]=cartaappoggio;
    }
    for(i=40; i!=0; i--){
```

```
    srand(time(NULL));
    j=rand()%i;
    cartaappoggio=mazzo[i-1];
    mazzo[i-1]=mazzo[j];
    mazzo[j]=cartaappoggio;
   }
  }
}
```

- Funzione utilizzata nel lato client per permettere al giocatore di scegliere quale carta giocare. Viene chiamata una sottofunzione *scegliere*.

```
int choice(int cont) {
        int scelta;
        tcflush(STDIN_FILENO, TCIFLUSH);
    if (cont<18){
      printf("\nOra è il tuo turno. Scegli quale carta buttare, premendo
1, 2 o 3: ");
     scelta=scegliere();
     printf("\nAttendi...\n");
     while (scelta!=1 && scelta !=2 && scelta !=3){
        printf("\nErrore nell'effettuare la scelta. Rifai. Premi 1, 2 o 3:
");
      scelta=scegliere();
      printf("\nAttendi...\n");
     }
    }else if (cont==18){
      printf("\nOra è il tuo turno. Scegli quale carta buttare, premendo
1 o 2: ");
     scelta=scegliere();
     printf("\nAttendi...\n");
     while (scelta!=1 && scelta !=2){
      printf("\nErrore nell'effettuare la scelta. Rifai. Premi 1 o 2: ");
      scelta=scegliere();
      printf("\nAttendi...\n");
     }
    } else if (cont==19){
```

```
        printf("\nOra è il tuo turno. Scegli quale carta buttare, premendo
1: ");
       scelta=scegliere();
       printf("\nAttendi...\n");
       while (scelta!=1){
         printf("\nErrore nell'effettuare la scelta. Rifai. Premi 1: ");
         scelta=scegliere();
         printf("\nAttendi...\n");
       }
     }
return scelta;
}


int scegliere (){
  int scelta;
        while (scanf("%d", &scelta) != 1) {
     printf("\nInserisci un numero valido \n");
        while (getchar() != '\n');
  }
        return scelta;
}
```

/*sono stati inseriti controlli affinchè il numero inserito sia valido (1,2,3 in caso di mano piena) e che venga inserito un intero e non una stringa.
La funzione *tcflush (*inclusa nella libreria *termios.h*) serve per pulire il buffer di input nel caso in cui un giocatore, mentre non era il suo turno, abbia digitato qualcosa sulla shell. Grazie a questa funzione, tutto ciò verrebbe ignorato.*/

- Funzione utilizzata nel lato server per stabilire il vincitore della singola mano.

```
int vincita (carta prima, carta seconda, semecarta semebriscola){
    if (prima.seme == seconda.seme){
      if(prima.tipo>seconda.tipo)
        return 0;
      else return 1;
    } else {                                  //se i semi delle carte sono diversi
      if (prima.seme==semebriscola)
        return 0;
```

```
    if (seconda.seme==semebriscola)
      return 1;
   }
   return 0;
}
```

/*la funzione prende come parametri la carta giocata dal primo giocatore (colui che nell'ambito di quella manche ha giocato per primo) e dal secondo, e il seme della carta rappresentante la briscola. Dopo aver fatto gli opportuni confronti, la funzione ritorna 0 se il vincitore della manche (cioè colui che prende le carte) è il primo (inteso sempre come colui che ha giocato per primo quella singola manche) o il secondo*/

- Per rendere il codice più leggibile, sono state inoltre definite alcune funzioni per ricevere e inviare specifici tipi di dati. Esse al loro interno chiamano le classiche send e recv e effettuano gli opportuni controlli.
Esse si trovano nel file *deffbriscola.c*, mentre i prototipi di esse (riportati qui sotto) sono reperibili nel file *funzionibriscola.h*

```
void inviaintero(int , int* );
void recintero(int , int* );
void inviacarattere(int , char* );
void reccarattere(int , char* );
void inviacarta(int , carta* );
void reccarta(int , carta* );
void inviastringa(int , char* , int );
void recstringa(int , char* , int );
```

- Esecuzione del codice

Vediamo di seguito cosa viene visualizzato da terminale.

Lato server:
Appena lanciato in esecuzione verrà visualizzato:

```
In attesa di connessione: Player 1
```

Una volta che entrambi i giocatori si sono connessi verrà visualizzato:

```
In attesa di connessione: Player 1
   Ho accettato una connesione: Player 1
In attesa di connessione: Player 2
   Ho accettato una connesione: Player 2
```

Poi verrà pescata la briscola e verranno inviate le carte ai giocatori (le carte inviate non vengono stampate perché il server potrebbe essere in esecuzione sulla stessa macchina di uno dei due giocatori, e visualizzare le carte dell'avversario non sarebbe corretto ai fini del gioco).

```
La briscola è:   QUATTRO di SPADE
Ora manderai LA BRISCOLA ai giocatori
 Ora manderai le CARTE ai giocatori
```

Dopo che la partita è iniziata, si aspetterà la ricezione delle single carte giocate dai giocatori, dopodichè si visualizzeranno un riepilogo della mano, i punti totalizzati durante la mano, e il vincitore (colui che acquisirà le carte).

```
Riepilogo:
  Carta ricevuta dal PRIMO:   TRE di COPPE

  Carta ricevuta dal SECONDO:   DUE di DENARI

Ha vinto il PRIMO  (Player 1)  con 10 punti. La mano successiva inizierà
ancora lui
```

Si invieranno quindi le nuove carte ai giocatori.

Ho mandato le nuove carte ai giocatori.

Alla fine di ogni partita viene visualizzato il risultato e il vincitore.

---------- FINE PARTITA!! ----------

La partita è stata vinta dal PRIMO giocatore (colui che nell'ultima mano ha giocato per PRIMO), ovvero il Player 2, con 68 punti !!!

Lato client:

Dopo aver lanciato l'eseguibile, ci si connetterà al server. Si dovrà attendere l'eventuale connessione del secondo giocatore.

Sei connesso. Attendi...

Dopodiché si riceverà la carta che rappresenterà la briscola per l'intera partita e il numero identificativo del player (Player 1 o Player 2. Se ci si è connessi per prima si sarà il Player 1).

Ora riceverai LA BRISCOLA...
La BRISCOLA è: QUATTRO di SPADE

Tu sei il Player 2
Il tuo avversario inizierà la partita

Dopodichè inizierà la partita e verranno visualizzate le carte della propria mano. Verrà stampata anche la briscola e occorrerà effettuare una scelta premendo i numeri corrispondenti alle carte. Nel caso non si cominciasse per primi, occorrerà attendere il proprio turno.

Mano n. 1 - Player 2
Ecco la tua mano:
1.: DUE di DENARI
2.: TRE di DENARI

3.: SEI di COPPE

Ricordati che la BRISCOLA è: QUATTRO di SPADE

Non è il tuo turno. Attendi il tuo...

La carta che l'avversario ha giocato è: TRE di COPPE

Ora è il tuo turno. Scegli quale carta buttare, premendo 1, 2 o 3: *1*

Una volta che entrambi i giocatori hanno effettuato una scelta, si visualizzerà il riepilogo della manche con l'esito e i punti totalizzati.

Esito manche: HAI PERSO!!
RIEPILOGO MANCHE:
-Carta da te giocata: DUE di DENARI
-Carta giocata dall'avversario: TRE di COPPE

Hai giocato per SECONDO.
Punti totalizzati nella manche: 0

Verrà quindi pescata una nuova carta (se possibile) e comincerà una nuova mano.

Nuova carta pescata: QUATTRO di COPPE

Verrà infine visualizzato l'esito della partita con il totale dei punti realizzati. Verrà inoltre stampato l'elenco delle carte acquisite durante la partita con i punti corrispondenti.

---------- FINE PARTITA!! ----------

Punti totalizzati nella partita: 68

ESITO PARTITA: HAI VINTO!! Con 68 punti!!!

- Progetto

Il progetto (sia per la parte client che per la parte server) è stato diviso in tre parti:
- Pre-partita
- Ciclo-partita
- Post-partita

1. Pre-partita - Server

Dopo le dichiarazioni e inizializzazioni delle varie variabili, il server chiama le apposite funzioni per connettersi ai due client.
I descrittori dei client sono salvati in un apposito array di due celle (*arraysd[]).* Il primo giocatore ad essersi connesso sarà colui che inizierà la partita e sarà identificato per tutta la partita con il nome "Player 1", mentre il secondo con il nome "Player 2".
In *arraysd[0]* sarà salvato sempre il descrittore del socket che inizierà la mano successiva, mentre, di conseguenza, il secondo giocatore della mano sarà quello con il descrittore in *arraysd[1].*
Per permettere la distinzione tra *Player 1* e *Player 2*, inizialmente, si salvano i descrittori dei nuovi socket in due variabili intere (*new_sd1* e *new_sd2*), che permetteranno di distinguere i giocatori confrontando il loro valore con quello in *arraysd[0]* e *arraysd[1].*

Una volta connessi entrambi i giocatori, si invia ai client un semaforo, per comunicare loro che entrambi i giocatori sono connessi.
Si procede quindi a inviare loro il numero identificativo che li contraddistinguerà per l'intera partita (Player 1 o Player 2).

Si pesca quindi la briscola dal mazzo, la si salva in un'opportuna variabile e la si invia ai giocatori.

Si inviano poi le prime tre carte della mano ai rispettivi giocatori. La partita può ora cominciare.

2. **Pre-partita – Client**

   Dopo le varie dichiarazioni e inizializzazioni il client si connette al server tramite la funzione *connect()*.
   Quando anche l'altro giocatore sarà connesso, riceverà un "*semaforo*" (proprio per indicare che entrambi i giocatori sono connessi" e il numero del Player (se ci si è connessi per primi sarà Player 1, in caso contrario Player 2). Player 1 inizierà la partita.

   Dopodichè si riceveranno dal client le prime 3 carte della mano. Si inizializzerà a 3 una variabile che tiene traccia del numero di carte pescate (*ncartepescate*). La partita potrà quindi iniziare.

3. **Ciclo-partita – Server**

   L'intera partita è gestita come un ciclo while, che continua fino a quando la variabile *finebriscola* non raggiunge il valore 4. Questa variabile è inizialmente settata a 0, e si inizia a incrementare solamente quando viene pescata la carta rappresentante la briscola da uno dei due giocatori. A quel punto, infatti, non ci saranno più carte da pescare, e rimarranno da giocare esattamente 3 manche.

   La mano inizia stampando il numero identificativo di ogni mano (la prima sarà la numero 1) e la briscola.
   Si invia un nuovo semaforo a ciascuno dei due giocatori per rendere noto che la manche è iniziata. I semafori (che sono caratteri) inviati sono due diversi: 'v' e 'r'. Se è ricevuto il semaforo 'v', allora significa che bisogna effettuare una scelta in quanto si è primi di mano. Se si è ricevuto 'r', bisogna attendere che l'avversario abbia giocato la propria carta per primo.

   Si ricevono quindi le carte da ogni giocatore. Non appena una carta è ricevuta, essa è inviata all'avversario, per rendergli nota la scelta della controparte.

A questo punto viene calcolato il punteggio totalizzato nella mano e viene poi invocata la funzione *vincita* per stabilire chi ha vinto. A quel punto si inviano i punteggi totalizzati ai rispettivi giocatori (se si è persa la mano ovviamente il punteggio inviato sarà sempre 0), assieme ad un messaggio di "vittoria" o "perdita" della manche.

Se il vincitore della manche è colui il cui descrittore è salvato in arraysd[0], non si effettua nessun cambiamento. In caso contrario i due descrittori vengono cambiati di posizione per permettere l'inversione dell'ordine.

Infine, se il mazzo non è terminato, vengono inviate la prima e la seconda carta del mazzo ai client (togliendole dal mazzo). Nel caso il mazzo sia terminato, viene incrementata la variabile *finebriscola* e viene inviata la briscola o, nel caso in cui anche questa fosse stata pescata, non viene inviato nulla.
Viene infine incrementato il contatore *contmani* (utilizzato per tenere traccia del numero della mano). Si procede quindi a una eventuale nuova iterazione del ciclo.

4. Ciclo-partita – Client

L'intera partita è gestita come un ciclo *for,* che continuerà fino a quando il numero di manche giocate (*contmani*, che parte dal valore 0, ed è incrementato di un'unità a ogni iterazione del ciclo) è diverso da 20.

Dopo aver stampato le tre carte costituenti la mano del giocatore, si riceverà dal server un carattere (memorizzato poi nella variabile *onetwo*) per indicare se si è primi (*onetwo=='v'*) o secondi (*onetwo=='r'*) di mano. Se si deve giocare per primi occorre effettuare una scelta (tramite la funzione *choice*). Dopo aver individuato la carta scelta nella "lista" della mano, essa verrà inviata al client che provvederà a recapitarla all'avversario. Si riceverà quindi la carta giocata dall'avversario.
Se si è secondi, prima di effettuare la scelta occorrerà attendere la scelta dell'avversario (che verrà comunicata).

Si riceveranno quindi i punti totalizzati durante la mano e verrà comunicato se si è vincitori o meno della manche. In caso positivo i punti potranno essere maggiori di 0.

Si provvede a eliminare dalla "lista" mano la carta scelta nella mano appena giocata e a ricevere una nuova carta, che verrà inserita in testa alla lista di carte costituenti la mano. Verrà quindi incrementata di 1 la variabile *ncartepescate*. Tutto ciò verrà svolto se e solo se è ancora possibile ricevere carte (cioè se *ncartepescate* è minore di 20, poiché il mazzo è costituito da 40 carte, e 20 verranno pescate da un singolo giocatore). Il ciclo potrà quindi riprendere.

5. **Post-partita**

   Alla fine della partita il server riceve dai client i punti rispettivamente totalizzati (salvati poi in un array di interi di dimensione 2 *pilapunti[]).* Essi vengono confrontati e viene decretato il vincitore.
   Viene quindi inviato un messaggio (hai vinto / hai perso) ai rispettivi client. Si noti che è anche possibile che la partita finisca in parità (60-60). In tal caso viene inviato a entrambi il messaggio "hai pareggiato".
   Verranno chiusi i descrittori dei socket utilizzati con la funzione *close().*
   Dopodiché il programma terminerà.

6. **Post-partita - Client**

   Una volta usciti dal ciclo-partita, verrà inviato al client il totale dei punti totalizzati durante l'intera partita e verrà ricevuto e stampato un messaggio di vincita/perdita/pareggio.
   Verranno chiusi i descrittori dei socket utilizzati con la funzione *close().*
   Il programma, quindi, terminerà.