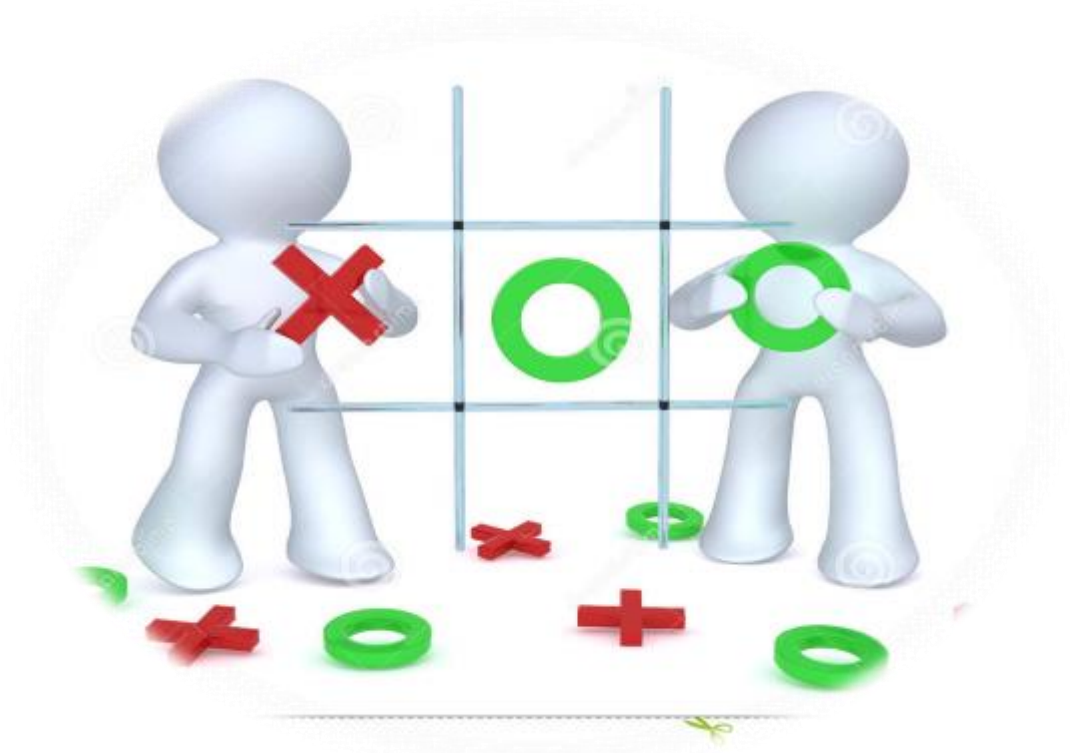
PROGETTO LABORATORIO DI SISTEMI OPERATIVI



UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
SCUOLA POLITECNICA E DELLE SCIENZE DI BASE
DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

CORSO DI LAUREA IN INFORMATICA
INSEGNAMENTO DI LABORATORIO DI SISTEMI OPERATIVI

Autori:
Di Luca Giuseppe N86002488
Di Martino Carmine N86002553
Gruppo N.20

Docente:
Faella Marco

DESCRIZIONE DEL PROGETTO

Descrizione sintetica

Il progetto consiste nella realizzazione di una applicazione che consenta a diversi client di giocare a tris contro altri giocatori.
Il server, unico per tutti i possibili client, gestisce simultaneamente le diverse partite fino ad un massimo di 15.
Ogni partita coinvolge due client e ogni client può essere coinvolto in una sola partita Alla volta.
Al termine di una partita, i due client coinvolti tornano disponibili e possono venire accoppiati con altri client disponibili, se ce ne sono in quel momento, per poi cominciare una nuova partita.
L'applicazione è scritta utilizzando il linguaggio C ed i processi comunicano attraverso socket TCP.

GUIDA ALLA COMPILAZIONE

Descrizione delle modalità di compilazione:

- **Client**
  Per compilare correttamente il file client.c è necessario eseguire da shell il comando gcc in questo modo:

```
gcc client.c -o Client.out
```

- **Server**
  Per compilare correttamente il file server.c è necessario eseguire da shell il comando gcc in questo modo:

```
gcc server.c -pthread -o Server.out
```

Al termine della compilazione saranno creati due file eseguibili, rispettivamente Client.out e Server.out.

GUIDA ALL'USO

Descrizione delle modalità di esecuzione:

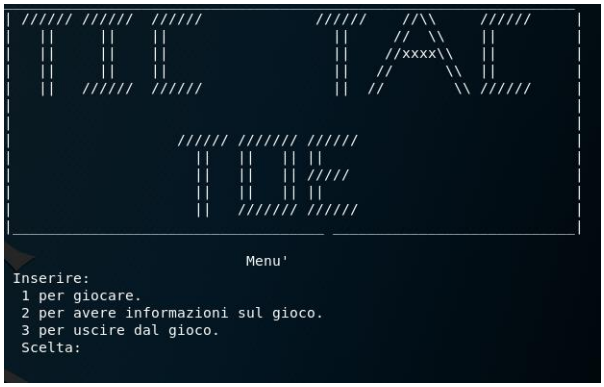
- **Client**

  Per eseguire correttamente il file eseguibile del client
  è necessario digitare da shell:

  ```
  ./Client.out  <IP>   <ServerPort>
  ```

  Dove il paramentro IP indica l'indirizzo del server e il parametro ServerPort indica la porta sul quale è in ascolto il server.

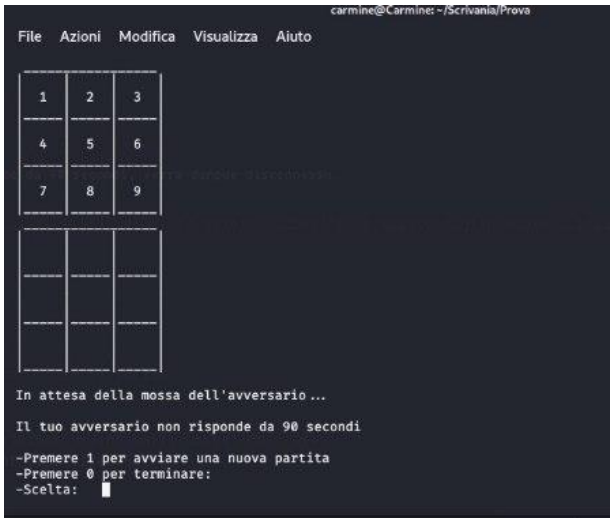
  Una volta eseguito l'applicativo del client apparirà il menù.



  L'utente potrà quindi scegliere di iniziare una nuova partita digitando 1. Subito dopo aver deciso di giocare gli verrà chiesto è il suo nome.

L’utente quindi inizierà una nuova partita se c’è già un altro client connesso pronto per giocare, altrimenti visualizzerà un messaggio di attesa, nel caso in cui invece sono avviate il numero massimo di partite l’utente sarà messo in attesa affinche una partita termini.

Una volta iniziata la partita l’utente potrà effettuare la mossa che desidera entro un limite di tempo di 90 secondi, pena la sua disconnessione.



Nel caso in cui un utente si disconnetta il suo avversario potrà decidere se giocare una nuova partita o chiudere il gioco (non c’è limite di tempo). Finita la partita (qualsiasi sia il risultato) ogni client avrà 15 secondi per decidere se rimare connesso per giocare altre partite oppure chiudere il gioco (pena la sua disconnessione).

L’utente potrà scegliere di avere informazioni del gioco digitando 2

Oppure potrà premere 3 per chiudere il gioco.

- **Server**

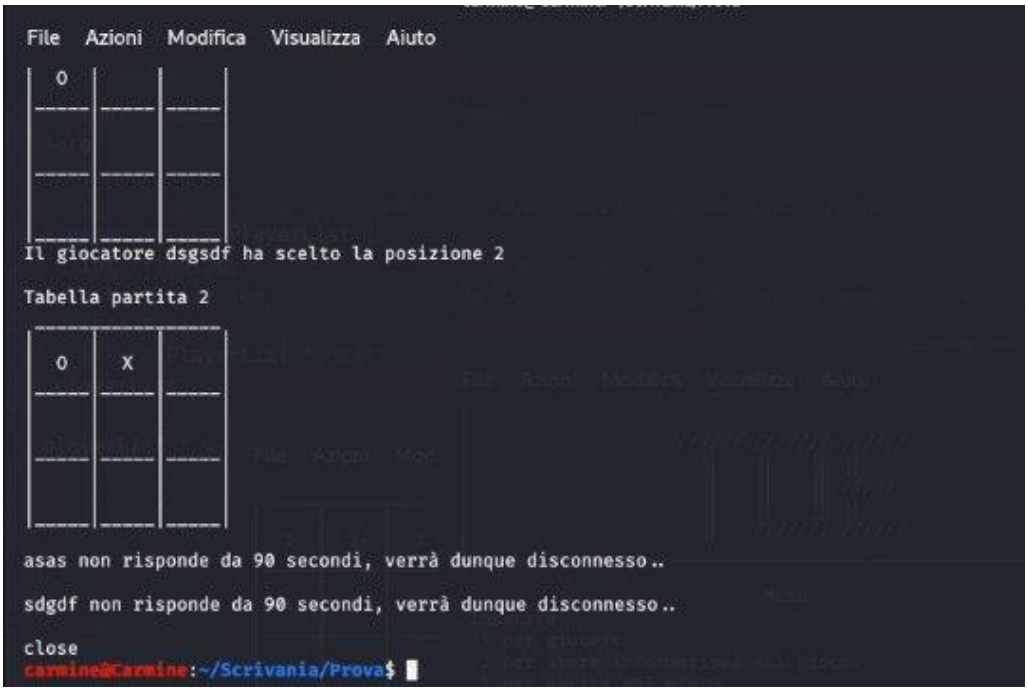
Per eseguire correttamente il file eseguibile del server è necessario digitare da shell:

```
./Server.out  <ServerPort>
```

Dove ServerPort indica su quale porta si mettere in ascolto il server.

Dopo aver digitato i comandi come descritto il server verrà avviato con successo e sarà dunque pronto a gestire le connessioni e quindi le diverse partite simultaneamente fino ad un massimo di 15.

Nel caso si ecceda in numero di partite il server rimarrà in attesa (passiva) affinchè qualche utente si disconnetta per poi riprendere a connettere altri eventuali client.
Il server prevede anche la sua disconnessione attraverso la stringa “close”, digitando quindi “close” in qualsiasi momento il server verrà disconnesso.



PROTOCOLLO DI COMUNICAZIONE TRA CLIENT E SERVER

La comunicazione tra client e server avviene attraverso le chiamate di sistema `read` e `write`.

Schema generale:

a. Ogni client subito dopo aver stabilito la connessione con il server, invia il suo nome al server.

b. Il server, provvede a selezionare due client e ad ognuno di esso invia l'id (se è un nuovo utente attraverso un intero, altrimenti se è un utente che ha già effettuato almeno una partita attraverso un carattere, 'A' se è il primo a connettersi o 'P' se è il secondo).
c. Il server invia il messaggio 'A' al primo client connesso (che lo mette in attesa di un secondo client), quando si connette anche il secondo client, il server invia il messaggio 'S' ad entrambi il quale specifica che la partita è iniziata.

d. Iniziata la partita il primo client riceve il carattere 'T' (inviatogli dal server attraverso la funzione `getPlayerMove`) il quale specifica al client che deve effettuare un mossa (qualora la mossa del client risulta invalida, il server provvede ad inviare il carattere 'I' il quale specifica che essendo la mossa non valida, deve rieffettuarla scegliendo una casella libera, se invece la mossa risulta valida il server provvede ad inviare il carattere 'U' attraverso la funzione `sendUpdate` ad entrambi i client per far si che aggiornino le tabelle di gioco). Mentre il secondo client riceve il carattere 'W' il quale specifica che deve attendere la mossa dell'avversario.

e. Dalla quinta mossa in poi (che è il numero minimo di giocate totali per vincere) il server controlla attraverso la funzione `checkPlayboard` se c'è un vincitore, in tal caso gli invia il carattere 'V' ed il carattere 'L' all'avversario che specificano rispettivamente chi ha vinto e chi perso e che dunque la partita è terminata. Nel caso invece si è arrivati alla nona giocata senza che ci sia un vincitore, il server provvede ad inviare al client il carattere 'D' che specifica appunto che la partita è terminata in pareggio.

FUNZIONAMENTO DETTAGLIATO

Inizialmente il server ascolta le connessioni in arrivo, che vengono accettate da un thread parallelo la cui funzione di avvio acceptNewConnections oltre ad accettare nuove connessioni salvandole in una lista di attesa, provvede anche a creare un fork il cui compito sarà quello di chiudere il server nel caso in cui viene inserita la parola "close" da STDIN.
Le partite sono formate da due giocatori, quindi il server tramite la funzione ConnectTwoClients associa due client (che possono essere nuovi giocatori che si sono appena connessi, oppure giocatori che hanno terminato una partita e deciso di giocarne un'altra) prendendo le loro informazioni dalla lista di attesa (gestita secondo le convenzioni FIFO e "protetta" da un mutex per evitare le race condition) e salvandole in un'apposita struttura.

Essendo che il server gestisce simultaneamente più partite, viene creato un nuovo thread detached per ogni partita la cui funzione di avvio runGame ha come argomento la struttura generata dalla funzione ConnectTwoClients avente le informazioni dei due client che si sfidano. La funzione di avvio provvede dunque a gestire la partita attraverso il protocollo di comunicazione descritto nel paragrafo precedente.
Durante la comunicazione, per evitare lunghe attese, si è deciso di dare un tempo limite di 90 secondi per effettuare una mossa, nel caso in cui questo tempo viene sforato da parte di un client, quest'ultimo viene disconnesso ed invece il suo avversario ha la possibilità di scegliere se continuare a giocare o disconnettersi a sua volta.
Se invece un client decide di disconnettersi prima che il tempo limite viene sforato, oppure si disconnette in maniera anomala, il server accorgendosi della disconnessione del client, invia un messaggio al client "avversario" contente il carattere 'E' , in questo modo l'avversario può scegliere di continuare a giocare o disconnettersi.
Al termine della partita entrambi I client possono scegliere di effettuare una nuova partita oppure abbandonare il gioco.

DETTAGLI IMPLEMENTATIVI INTERESSANTI

Eliminazione di eventuali lunghe attese

Essendo la funzione scanf bloccante, un client avrebbe potuto inserire la mossa in un tempo indefinito, per evitare che ciò accadesse abbiamo dunque deciso di dargli un tempo limite di 90 secondi sfruttando la funzione alarm ed il segnale da essa generato.

```c
int move;
signal (SIGALRM, hand2);

/* Viene salvato in via precauzionale la socket per comunicare con il server
* nel caso in cui il client non risponde in tempo, verrà inviato un messaggio al server per disconnetterlo. */
management = serverSd;

while (1) { /*Chiedere fintanto che non si riceve una mossa valida */

    printf ("\n %s effettua la tua mossa scegliendo la posizione come indicato nella tabella di riferimento:   ", name);
    /* Il client ha 90 secondi di tempo per effettuare una mossa. */
    alarm(90);

    if (scanf ("%d", &move) != 1) {
        while (getchar() != '\n'); /* pulisce il buffer nel caso sia stata inserita una stringa. */
    }

    /* Se la mossa è stata effettuta in tempo, il segnale viene disattivato. */
    alarm(0);
```

In questo modo se l'utente non inserisce nei 90 secondi una mossa viene generato un segnale SIGALRM catturato dall'handler che provvede ad avvertire il server che il client ha sforato il limite di tempo e quindi deve essere disconnesso. Se invece il client risponde in tempo, alarm(0) provvede a cancellare la generazione del segnale.

```c
/* Gestisce il segnale alarm, impostato per far in modo che un utente non impieghi
* troppo tempo per inserire una mossa. */
void hand2 () {
  printf("\nHai sforato il limite di tempo per effettuare una mossa. La partita è termita!\n");
  writeToServer (management, -3);
  exit(-1);
}
```

A questo punto, se il client viene disconnesso per via del tempo, il problema viene trasferito all'altro client perché bloccato sulla system call read, per risolvere del tutto abbiamo quindi deciso di usare la struttura timeval `struct timeval timeout2 = {90, 0,};` e la funzione setsockopt per settare le opzioni della socket e fare in modo che Le operazioni di lettura siano bloccanti soltanto per 90 secondi.

```c
if (setsockopt (serverSd, SOL_SOCKET, SO_RCVTIMEO, (char *)&timeout2, sizeof(timeout2)) < 0) {
    perror("setsockopt failed\n");
    exit (-1);
}
```

Dunque se la system call read fallisce restituendo il codice di errore EWOULDBLOCK a tal punto l'avversario è disconnesso e dunque viene chiamata la funzione anotherGame che permette al client ancora connesso di scegliere se giocare ancora o disconnettersi.

```
if (read (serverSd, &msg, sizeof(char)) != sizeof(char)) {
    if (errno == EWOULDBLOCK) {
        /* flag = 1 quando si entra in questa funzione a partita inoltrata. */
        if (flag == 1 && firstTimeRecycled == 0) {
            printf("\nIl tuo avversario non risponde da 90 secondi\n");
            ret = anotherGame (serverSd, 0);
            if (ret == 0) {
                exit (-1);
            }
        }
```

Disconnessione server

Abbiamo previsto anche la possibilità di disconnettere il server attraverso l'inserimento della parola close, per fare ciò abbiamo creato un fork nel thread che accetta le connessioni, dunque mentre il padre accetta le connessioni e le inserisce in lista, il figlio è in attesa passiva su una scanf, fin quando non viene inserita la parola close che "uccide" il processo figlio mandando quindi il segnale SIGCHLD al padre, che viene catturato attraverso un handler abrt che provvede alla disconnessione del padre.

```
if ( (pid = fork()) < 0) {
    perror ("Error. fork");
    exit (-1);
}
else if (pid == 0) {

    do {
        scanf("%s", closeServer);
    } while (strcmp(closeServer, "close") != 0);

    if (close (socketDes) != 0) {
        perror ("ERROR. Chiusura socket");
        exit (-1);
    }

    free (closeServer);

    exit (0);

}
else {
```

```
void abrt () {
  if(wait(NULL)<0){
    perror("wait error\n");
    exit(-1);
  }
  printf("\nIl server è stato arrestato...\n");
  exit (-1);
}
```

Eliminazione di tutte le attese attive

Grazie all'ausilio delle condition variable che consentono di attendere passivamente il verificarsi di una condizione su una risorsa condivisa, abbiamo potuto eliminare le attese attive.

```c
#define MAXPENDING 10

/* Variabile globale che tiene traccia del numero degli utenti che stanno giocando. */
int   COUNTPLAYER = 0;
/* Condition variable associata a COUNTPLAYER. */
pthread_cond_t COUNTPLAYER_FLAG = PTHREAD_COND_INITIALIZER;

/* Condition variable associata a waitingList. */
pthread_cond_t  LIST_FLAG = PTHREAD_COND_INITIALIZER;

/* Mutex associato alla variabile globale COUNTPLAYER. */
pthread_mutex_t MUTEX_COUNT = PTHREAD_MUTEX_INITIALIZER;

/* Mutex associato alla struttura globale playerList. */
pthread_mutex_t MUTEX_LIST = PTHREAD_MUTEX_INITIALIZER;
```

Il primo punto dove poteva verificarsi l'attesa attiva è nella funzione ConnectTwoClients, perchè nel caso in cui la lista che contiene gli utenti connessi in attesa di giocare fosse stata vuota avremmo dovuto far uso di un ciclo infinito per verificare continuamente la presenza di almeno un'elemento all'interno di essa , grazie alle condition variable e al mutex ad esso associato abbiamo potuto evitare il polling e le eventuali race condition.

```c
pthread_mutex_lock (&MUTEX_LIST);
  while (waitingList == NULL) {

      if (pthread_cond_wait (&LIST_FLAG, &MUTEX_LIST) != 0) {
          perror ("Error pthread_cond_wait");
          exit (-1);
      }

  }
  argument->clientSd[playerTurn] = waitingList->clientSd;
  if (playerTurn) {
      strcpy (argument->name2, waitingList->name);
  }
  else {
      strcpy (argument->name1, waitingList->name);
  }

  recycled = waitingList->recycled;
  deleteWaitingList ();

pthread_mutex_unlock (&MUTEX_LIST);
```

Perché nel caso di lista vuota viene chiamata la funzione pthread_cond_wait che mette il thread in attesa passiva rilasciando il mutex ed aspettando che il thread venga risvegliato per riacquisire il mutex. La necessità di inserire tale funzione in un while nasce perché una volta risvegliato il thread e riacquisito il mutex bisogna controllare se effettivamente la lista non è vuota per non cadere in errore.

Ad ogni inserimento in lista segue dunque una pthread_cond_broadcast che risveglia i thread in attesa su una condition variable.

```
pthread_mutex_lock (&MUTEX_LIST);
  insertWaitingList (name, clientSd, 0);
  if (pthread_cond_broadcast (&LIST_FLAG) != 0) {
      perror ("Error pthread_cond_broadcast");
      exit (-1);
  }
pthread_mutex_unlock (&MUTEX_LIST);
```

Il secondo punto dove poteva verificarsi l'attesa attiva è nella funzione acceptNewConnections perchè nel caso in cui si raggiunge il numero massimo di partite, senza far uso delle condition variable, il server sarebbe rimasto in attesa attiva fintanto che non si sarebbe disconnesso uno dei client.

```
/* Accetta nuove connessioni evitando l'attesa attiva nel caso si è raggiunto il numero massimo di partite. */
pthread_mutex_lock (&MUTEX_COUNT);
  while (COUNTPLAYER >= 30) {
      printf("\nRaggiunto il numero massimo di partite.\n");
      if (pthread_cond_wait (&COUNTPLAYER_FLAG, &MUTEX_COUNT) != 0) {
          perror ("Error pthread_cond_wait");
          exit (-1);
      }
  }
pthread_mutex_unlock (&MUTEX_COUNT);
```

Grazie alle condition variable abbiamo potuto evitare il polling.

```
pthread_mutex_lock (&MUTEX_COUNT);
  COUNTPLAYER --;
  if (pthread_cond_broadcast (&COUNTPLAYER_FLAG) != 0) {
      perror ("Error pthread_cond_broadcast");
      exit (-1);
  }
pthread_mutex_unlock (&MUTEX_COUNT);
```

Ad ogni disconnessione di qualche client segue dunque una pthread_cond_broadcast che risveglia i thread in attesa su una condition variable.

Grazie all'attesa passiva nel caso in cui si è raggiunto il numero massimo di client "gestibili" da parte del server, abbiamo potuto anche far i modo che il client che si connette quando il server è pieno legga a video un messaggio di attesa

```
while (1) {
    if (read (serverSd, &msg, sizeof(int)) != sizeof(int)) {
        if (errno == EWOULDBLOCK) {

            printf("\nAttualmente sono avviate il numero massimo di partite\n");
            printf("Attendere pochi istanti affinchè un'altra partita termini.\n");

            /* Imposta un nuovo timeout che servirà per le successive read. */
            if (setsockopt (serverSd, SOL_SOCKET, SO_RCVTIMEO, (char *)&timeout2, sizeof(timeout2)) < 0) {
                perror("setsockopt failed\n");
                exit (-1);
            }
            flag = 0;
        }
        else {
            perror("ERROR. Lettura dalla socket del server.");
            printf("\nIl server è stato arrestato.\n");
            exit (-1);
        }
    }
    else {
        /* Se legge correttamente bisogna uscire dal while. */
        break;
    }
}

if (flag) {
    if (setsockopt (serverSd, SOL_SOCKET, SO_RCVTIMEO, (char *)&timeout2, sizeof(timeout2)) < 0) {
        perror("setsockopt failed\n");
        exit (-1);
    }
```

Essendo che, appena dopo poco che il client stabilisce la connessione con il server gli viene inviato il suo id, abbiamo pensato di impostare inizialmente un timer in ricezione di tre secondi avvalendoci della struttura timeval e settando le opzione attraverso la funzione setsockopt (come spiegato nel paragrafo "eliminazione di tutte le attese attive") quindi se il client appena connesso non riceve l'id in tre secondi, allora sicuramente il server è in attesa passiva dunque la read dopo i 3 secondi fallisce con l'errore impostato a EWOULDBLOCK che abbiamo sfruttato per la stampa e dopo abbiamo settato il timer a 90 secondi per le successive read.