UNIVERSITE DE STRASBOURG

UFR de mathématique et d'informatique

Projet Algorithmes des Réseaux
Licence 3 en Informatique

C-Tron

(Cloud Gaming Edition)

Réalisé par : **Baye Lahad MBACKE**

Année universitaire 2022-2023

Introduction

Le but de ce projet est de recréer ce jeu dans le terminal, à l'aide de la libraire ncurses. Une partie opposera deux joueurs, qui communiqueront via des sockets par l'intermédiaire d'un serveur qui fait tourner le jeu. L'architecture du jeu suivra un modèle s'approchant d'une forme de cloud gaming : le serveur est en charge des calculs (des positions des joueurs, des collisions, etc.), et les clients ne font qu'afficher le jeu à l'écran.

En premier lieu on va expliquer l'implémentation au cote client, ensuite l'implémentions au cote serveur et en fin on répondra les questions posées.

Au cote client

Au cote client, comme on l'a spécifié dans l'énonce, on a utilisé le protocole **TCP,** en premier lieu on a créé un socket en spécifiant qu'on utilise adresse **IPv4** et le protocole **TCP,** ensuite on a défini Après savoir configure le socket, on fait une demande de connexion vers le serveur **(**désigné par un point de communication adresse IP du serveur, numéro de port d'écoute du serveur) avec la fonction **connect ()**

Après l'établissement de la connexion, on envoie le nombre de joueur via la fonction **send ().**

On a utilisé **fd_set** pour créer des descripteur de fichier qui sont ensbl et tmp, ensuite on initialise les descripteur de fichier avec **FD_ZERO(),** on a ajouté le socket et l'entrée standard dans le descripteur en utilisant **FD_SET().**

On fait une boucle while, dans cette boucle on y défini la fonction select () permet de surveiller des ensembles de descripteurs.

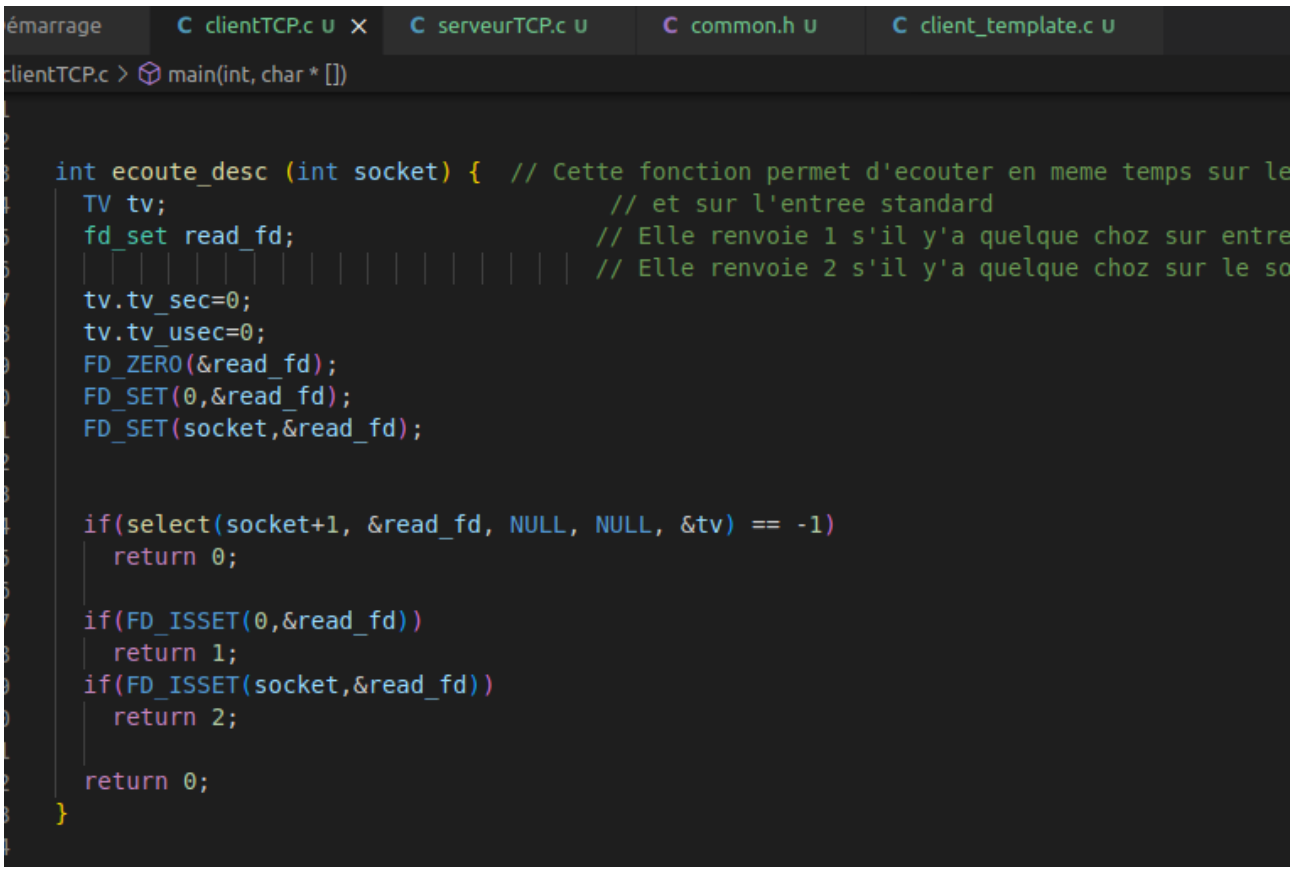
Apres savoir envoyer le nombre joueur, on vérifie ce qui se trouve dans le descripteur s'il est un socket ou pas en faisant FD_ISSET (socket_cli, &tmp) qui va retourner une valeur non nulle si le socket est présent 0 sinon.

On récupère ce que le serveur nous envoie en utilisant **recv ()**

Le serveur envoie le plateau du jeu en initiant la partie si le client envoie 2 comme nombre de joueur ou bien, le serveur envoie un message « wait » pour lui dire d'attendre l'autre joueur si le client envoie 1 comme nombre de joueur.

On a créé une fonction appelé **ecoute_desc(int)**qui prend en paramètre le socket et retourne 1 si c'est l'entrée standard qui est dans le descripteur, retourne 2 si c'est le socket qui dans le descripteur et retourne 0 s'il y'a erreur.

On a utilisé cette fonction pour envoyer les inputs du client vers le serveur et aussi on l'utilise pour afficher l'état actuel du plateau de jeu que le serveur envoie au client



Au cote serveur

Au cote serveur on a fait initialement comme le client en créant un socket avec le protocole TCP et adresse IPv4. Apres savoir configurer le socket, on a défini un point de communication sockaddr_in (@IP, port d'écoute), on a utilisé la fonction **bind()** pour lier les deux. Ensuite on a placé le socket en mode ouverture passive avec la fonction **listen ().**

On a répété la même chose de ce qu'on a fait avec le descripteur de fichier au cote client.

On a créé une boucle while dans lequel on écoute le socket client pour récupérer le nombre de joueur, on quitte la boucle que si le nombre de joueur est 2.

Si on le client envoie 2 comme nombre de joueur, le serveur lui envoie le plateau di jeu et la partie démarre, si le client envoie 1 comme nombre de joueur, le serveur lui envoie un message 'wait' pour lui dire d'attendre l'autre joueur et le socket du 1er comme le 2eme joueur est stocke dans un tableau. Ce dernier sera parcouru pour envoyer les différents clients les informations actuels du plateau de jeu. Si le nombre de joueur est égale à 2 on quitte la boucle car ce dernier permettait seulement d'accepter les clients en utilisant **accepte ()** et initier la partie du jeu.

On a utilisé une deuxième boucle pour faire l'écoute des descripteurs avec le **select ().**

Pour mettre le socket des différents clients sur le descripteur, on fait le parcours du tableau et on les ajoute dans le descripteur de fichier

```
while (1)
{
    max = socket_serv; // on considere ici que socket_serv est le maximum
  for ( int i = 0 ; i <nb_player.nb_players; i++)
    {
      // descripteur socket client
      sd = client[i];
      //si le socket est nb_player.nb_players  on l'ajoute dans le descripteur de socket
      if(sd > 0)
        FD_SET(sd,&ensbl);
      //on cherche le plus grand descripteur
      if(sd > max)
        max = sd;
    }
```

A chaque instant on doit vérifier s'il y'a quelque chose dans le descripteur notamment les entrées inputs du client envoyé vers le serveur

```
select(max+1,&ensbl,NULL,NULL,&tv);
for (int i = 0; i < nb_player.nb_players; i++)
{
    sd = client[i]; //on recupere le descripteur du client ici
  if (FD_ISSET(sd,&ensbl)) //On verifie le  descripteur s'il contient kelk choz
  {
      if (cli_input.id==1)  // Joueur 1
        {
          if (cli_input.input == 'z') { //Haut
          Rinfo.board[x][y] = 50;
            y = y -1;
            oldX = -1;
            oldY = y;

          if (collision(x,y,x1,y1,Rinfo)==1)
            Rinfo.winner=2;
```

A chaque changement de direction le serveur calcule les nouvelles coordonnées et mis à jour le plateau du jeu et l'envoie au client.

Les variables **oldX**, **oldY** contient les anciennes coordonnées du joueur 1 c'est à dire les coordonnes après un input du joueur 1 #

Les variables **oldX1**, **oldY1** contient les anciennes coordonnées du joueur 2 c'est à dire les coordonnes après un input du joueur 2 @

On a créé ces variables pour permettre le jeu (les lightcycles) de continuer (même direction) si le client ne fait aucun inputs (pas de changement de direction).

Le changement des directions se fait selon le joueur si c'est le **joueur 1**, le client envoie comme **id=1** et les inputs ne peuvent être que (z, q, s, d) dans ce cas c'est seulement **x** et **y** qui peuvent changer.

Après chaque changement on donne les dernières coordonnées au tronc du lightcycles en faisant Rinfo.board[x][y] = 50, on a mis à jour les coordonnées puis on donne la nouvelle coordonnée a la tête du lightcycles Rinfo.board[x][y] = 1

Si c'est le **joueur 2**, le client envoie comme **id=2** et les inputs ne peuvent être que (i, j, k, l) dans ce cas c'est seulement **x1** et **y1** qui peuvent changer.

Après chaque changement on donne les dernières coordonnées au tronc du lightcycles en faisant

Rinfo.board[x1] [y1] = 51, on a mis à jour les coordonnées puis on donne les nouvelles coordonnées à la tête du lightcycles Rinfo.board[x1] [y1] = 2.

Apres chaque modification de coordonnes on vérifie s'il y'a pas de collision si ce n'est pas le cas, le serveur envoie les données du plateau actuels. Si c'est le cas on cherche le joueur qui a commis la collision.

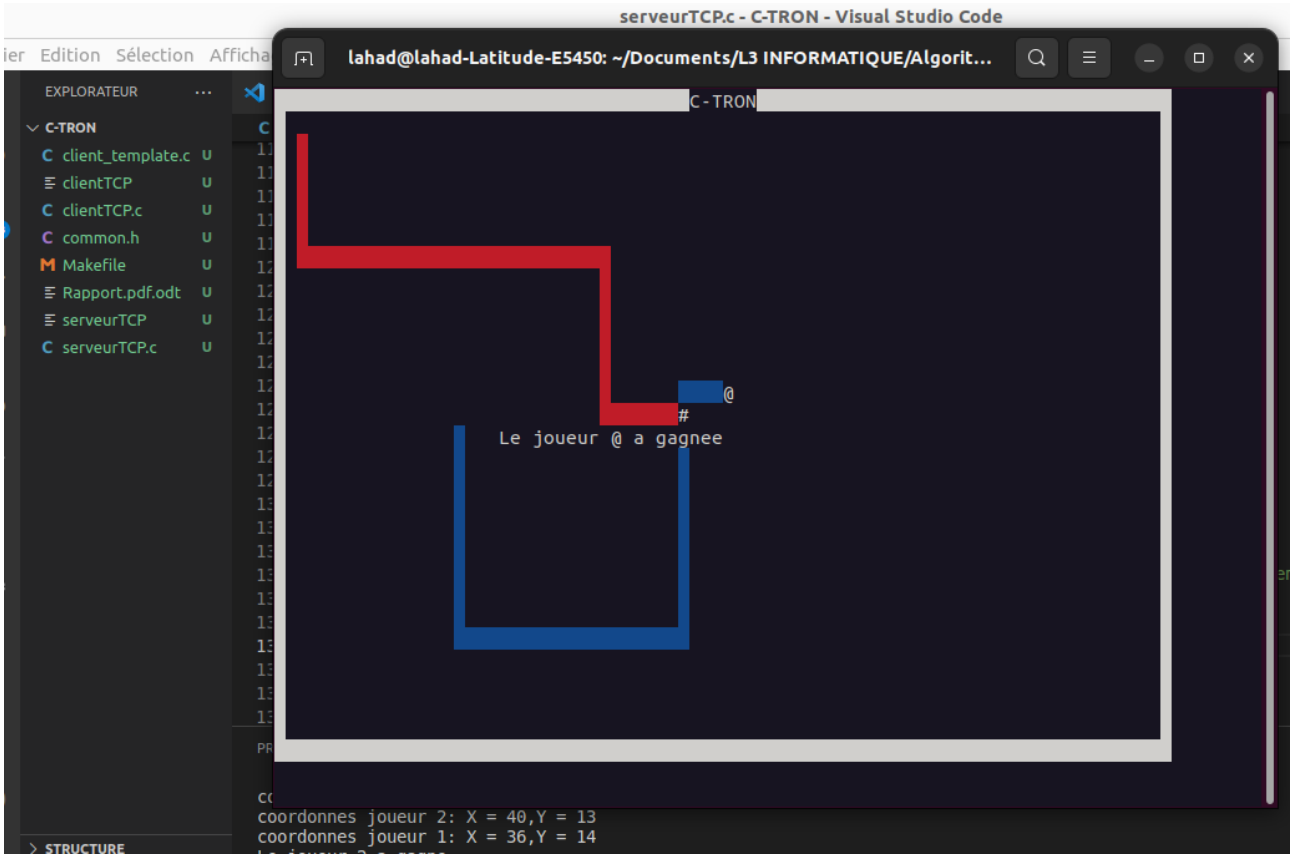
Si la fonction **collision ()** qui prend en paramètre les coordonnées du joueur 1, joueur 2 et le plateau du jeu. Si la fonction retourne 1 c'est le joueur 1, qui a commis la collision, si elle retourne 2 c'est le joueur 2 qui a fait collision, si elle retourne 3 leurs têtes se cognent donc c'est match nul.

```
int collision(int x,int y,int x1,int y1, struct display_info board)
{
    if ( x<0 || x == XMAX-1) {   // on verifie si le joueur 1 cogne le mur
      return 1;
    }
    else if (y<0 || y == YMAX -1) { // on verifie si le joueur 1 cogne le mur
      return 1;
    }

    if (x1<0 || x1 == XMAX -1) {  // on verifie si le joueur 2 cogne le mur
      return 2;
    }
    else if (y1<0 || y1 == YMAX-1) {  // on verifie si le joueur 2 cogne le mur
      return 2;
    }
    if (board.board[x][y]==51) // on verifie si le joueur 1 a heurte le joueur 2
        return 1;
    if (board.board[x1][y1]==50) //on verifie si le joueur 2 a heurte le joueur 1
        return 2;
    if (board.board[x][y]==2 || board.board[x1][y1]==1)//on verifie si la tete des 2 joueur se
            return 3;

  return 0;
}
```

Pour envoyer les mis à jour du plateau, on vérifie d'abord si les deux clients sont sur le même terminal (on l'a matérialisé nbj==2) dans ce cas on envoie les infos sur le socket client qui héberge le terminal. Si (nbj ! =2) dans ce cas les joueurs sont sur des terminale différents, alors on parcourt le tableau qui contient les sockets clients et on fait un broadcaste sur les différentes terminales

```c
if (nbj==2)
{
   if(send(sd,&Rinfo,sizeof(display_info),0)==-1)
      printf("L'envoie echoue du mis a jour \n");
   if (Rinfo.winner==1)
      exit(0);
   if (Rinfo.winner==2)
       exit(0);
   if (Rinfo.winner==3)
       exit(0);
}
else
{
  for (int s = 0; s < nb_player.nb_players; s++)
   {
     if(send(client[s],&Rinfo,sizeof(display_info),0)==-1)
         exit(0);
     if (Rinfo.winner==1)
         exit(0);
     if (Rinfo.winner==2)
         exit(0);
     if (Rinfo.winner==3)
         exit(0);
   }
}
```

Exemple de collision



Réponses aux Questions

1. Discutez des avantages et inconvénients de ce genre d'architecture. En particulier, examinez l'impact des caractéristiques physiques de la connexion (délai, gigue, taux de pertes...).

Chaque fois qu'un joueur cliquait pour effectuer une action dans un jeu (par exemple, pour changer de direction), l'événement de clic est envoyé au serveur, le serveur traite l'information et renvoie aux clients ceci se fait avec beaucoup de latence (délai significatif) le jeu n'est très fluide ceci est un grand inconvénient. On peut constater aussi que des fois ça va vite des fois ça ralenti ce qui veut dure que le gigue est considérables ici.

2. L'utilisation du protocole TCP est-elle souhaitable pour les jeux en ligne ? Qu'en est-il du Cloud Gaming ? Idéalement, quel(s) protocoles utiliseriez-vous dans le cas présent ?

L'utilisation du protocole TCP est-elle souhaitable pour les jeux en ligne ?

Non, le protocole TCP n'est pas souhaitable pour les jeux en ligne, surtout quand il s'agit un jeu en ligne de multijoueur. Le problème en est que si nous le protocole CP, dès qu'un paquet est perdu, on arrête et attend que la donnée soit renvoyée, donc le protocole TCP pour n'est pas souhaitable pour les jeux en ligne en temps réel

Qu'en est-il du Cloud Gaming ?

Non le protocole TCP n'est pas souhaitable pour Cloud Gamin car les jeux basés sur le cloud ont des contraintes temporelles, ça doit se faire en temps réels et de faible latence pour offrir une expérience de jeu rapide et fluide avec le moins de gigue possible, alors que TCP ne propose pas ça donc ce dernier n'est pas souhaitable.

Idéalement, quel(s) protocoles utiliseriez-vous dans le cas présent ?

Idéalement j'aurai du utilise le protocole UDP pour minimiser le temps de latence, et que aussi UDP est plus le protocole le plus adapte pour des applications qui ont des contraintes en temps réels.