Ingénierie des réseaux

Travaux pratiques : Le jeu du pendu

FRAPPER colin
N'GUESSAN Jean-Philippe

Sommaire

I - Introduction

Dans le cadre du module de l'ingénierie des réseaux il nous était demandé de réaliser une application au choix qui utilise la relation client serveur. Dans cette optique nous avons décidé de réaliser le jeu du pendu. Ce jeu consiste à trouver un mot en devinant les lettres qui le constitue. Nous avons décidé d'inclure plusieurs clients qui seront amenés à jouer ensemble. Les objectifs principaux de ce projet étaient de comprendre et manipuler l'architecture client/serveur, le fonctionnement des interfaces systèmes et les sockets.

Ce rapport présente l'application que nous avons développée. Nous aborderons dans un premier temps l'aspect fonctionnel de celle-ci, en détaillant les fonctionnalités présentes, notamment par rapport au cahier des charges proposé au départ.

Nous détaillerons ensuite les aspects techniques de l'application, notamment nos choix d'implémentation. Nous terminerons enfin par un bilan du projet, des difficultés rencontrés, des perfections possibles, mais également les apports de ce projet d'un point de vue plus personnel.

II - Présentation de notre application

Le pendu est un jeu qui consiste à trouver un mot en devinant quelles sont les lettres qui le compose. Ce jeu se joue naturellement à plusieurs selon un déroulement bien précis. Quand le dessin est terminé on voit un bonhomme qui se pend.

Le but de notre projet est de réaliser un jeu en réseau, le jeu du Pendu. Le serveur jouera le rôle de la personne qui proposera un mot au client (celui qui devra trouver le mot) quand il y a un seul client. Selon comment le projet avance il sera possible de jouer en multi joueurs contre le serveur et essayer de trouver le mot, et même de jouer l'un contre l'autre de façon à ce que le serveur achemine chaque message/réponse à chaque joueur.
Pour ce qui est de la partie 1 client utilisant un serveur, le serveur va se charger du mot aléatoire et le client pourra émettre ses propositions avec une possibilité de 10 coups. La seconde partie illustre un serveur et plusieurs utilisateurs, le serveur se charge une fois de plus de proposer le mot et les clients de trouver le mot avec une possibilité de 10 coups sachant que chaque client doit proposer au moins une lettre. Pour finir, la troisième partie illustre le faite qu'un client propose le mot et les autres clients proposes une lettre pour pouvoir terminer la partie.Cependant, celui qui propose la mot ne doit pas proposé de lettre pour la partie en cours, le serveur aura pour rôle de transférer la réponse de chaque client ici.

III - Les sockets et protocoles

Les sockets sont la partie essentielles du processus qui sert à communiquer entre un client et un serveur connecté sur le même serveur via une adresse IP et un port. Ils peuvent être de deux types différents pour deux protocoles différent, le protocole UPD et le protocole TCP.

A - Principe de fonctionnement

Comme dit précédemment, les sockets sont un outil de communication entre un client et serveur. Dans le cadre du protocole TCP, le client et le serveur sont liés par le numéro de port. Le serveur se met donc à l'écoute des clients sur ce serveur en particulier. Pour se faire
le client et le serveur vont devoir effectuer des actions différentes
Côté serveur il faut tout d'abord créer le socket et créer l'interface sur laquelle notre socket va écouter pour lier cette interface à notre socket. Ensuite, il faut que le serveur se mette à l'écoute des clients qui vont vouloir se connecter. Il peut alors refuser ou accepter ces connexions. On peut alors effectuer les actions désiré, jusqu'à ce que le serveur décide de refermer les sockets.
Côté client, le principe reste à peu près le même avec quelques différences. En effet, tout d'abord, le client doit se connecter au serveur grâce à son adresse IP. Une fois connecté le client peut alors communiquer avec le serveur et donc lui envoyer les données qu'il souhaite en les écrivant dans le socket. Ces données sont contraintes par une taille maximale que le serveur peut recevoir. Une fois la communication terminée il faut alors fermer le socket.

B - Fonctions & Structure en C

Pour réaliser les actions décrites précédemment, le langage C inclut diverses fonctions et structures permettant de faciliter ces sockets. Pour le projet nous avons utilisé ces quatres structures :
● **sockaddr_in** : Cette structure permet de définir le socket avec son adresse, son numéro de port, sa longueur...
● **sockaddr** : La structure sockaddr_in existe uniquement dans le but que la taille de la structure sockaddr_in soit la même que celle de la structure sockaddr.
● **hostent** : Cette structure permet de définir la machine host avec des informations comme son nom, ses alias, ect...
● **servent** : Cette structure permet de définir la machine serveur avec des informations comme son nom, ect...De plus le langage C nous fournit beaucoup de fonctions.
● **Socket** : Cette fonction permet tout simplement de créer des sockets de type int
● **connect** : Cette fonction permet de connecter notre socket à la structure sockadrr

IV - Implémentation

Pour ce projet, afin de réaliser la communication entre les clients et le serveur, nous avons été imposé de travailler avec les sockets. La programmation par socket est aujourd'hui encore très utilisée dans de multiples langages de programmation. Nous avons utilisé le langage C, dû au fait qu'il utilise les manipulations de bas niveau sur les sockets.

A chaque nouvelle connection d'un client, le serveur est placé en "écoute" de ce dernier et prêt à établir la connection entre ces deux entités.

A - Client

Du côté du client, afin de créer la connection nous avons une fonction **init_connection** qui prend en paramètre l'adresse ip fournie par le client, le but de cette fonction est de renvoyer la socket du serveur associé à l'adresse passée en paramètre.

Dans un premier temps, la fonction gethostbyname qui permet de récupérer, à partir de l'adresse IP serveur, la structure de socket du serveur

```
if ((hp = gethostbyname(address)) == NULL)
{
    perror("Problème avec l'hostname");
    exit(1);
}
```

Une fois la structure de socket du serveur récupérée, il nous faut définir le socket avec son adresse, son port, sa longueur

```
bcopy(hp->h_addr, &adresse_serv.sin_addr, hp->h_length);
adresse_serv.sin_family = hp->h_addrtype;
adresse_serv.sin_port = htons(PORT);

if ((sock_client = socket(hp->h_addrtype, SOCK_STREAM, 0)) < 0)
{
    perror("socket");
    exit(1);
}
```

Si tout ce passe bien, la socket est créée. Pour conclure la connection avec le serveur il nous faut la "connecter" au serveur, on utilise pour cela la fonction connect.

```
if (connect(sock_client, (SOCKADDR *) &adresse_serv, sizeof (adresse_serv)) < 0)
{
    perror("connect");
    close(sock_client);
    exit(EXIT_FAILURE);
}
```

Nous avons donc la connexion avec le serveur établie.

B - Serveur

Côté serveur, de même que pour le côté client, nous avons une fonction **init_connection** prenant une adresse déjà défini en paramètre. Le but de cette fonction est donc d'initialiser la socket du serveur.

```
int sock = socket(AF_INET, SOCK_STREAM, 0);
```

Note : SOCK_STREAM utilisé pour le protocole TCP/IP contrairement au protocole UDP où l'on aurait utilisé SOCK_DGRAM.
La socket est alors créée, et le descripteur associé sauvegardé.
Il s'agit maintenant de paramétrer le socket :

```
sin.sin_addr.s_addr = inet_addr(adresse);
sin.sin_port = htons(PORT);
sin.sin_family = AF_INET;
```

```
if(bind(sock,(SOCKADDR *) &sin, sizeof sin) == SOCKET_ERROR)
{
   perror("bind()");
   exit(errno);
}

if(listen(sock, MAX_CLIENTS) == SOCKET_ERROR)
{
   perror("listen()");
   exit(errno);
}
```

La fonction bind permet de lier un socket avec une structure sockaddr et ensuite la fonction listen définie la taille de la file de connexions en attente pour notre socket sock.
Ensuite dans une seconde fonction **application**, une socket d'écoute sera donc ouvert à chaque nouvelle connection d'un client.
Pour la partie multi-client, nous avons choisi d'utiliser les select plutot que les threads, la raison est que select est bien meilleur par rapport aux threads au niveau des performances et nous voulions utiliser une solution moins "classique" que les threads, de façon à apprendre une nouvelle manière de programmer un serveur multi-clients.

V - Fonctionnalités implémentées

Au niveau de nos fonctionnalités, nous avons plusieurs “choix” de jeu.

A - Le menu

A chaque nouvelle connection d’un client, le serveur envoie le menu à ce dernier afin qu’il sache ce qu’il peut faire dans ce jeu.

B - La liste

Le client peut également la liste des joueurs connectés de façon à ce qu’il sache s’il peut jouer contre quelqu’un ou encore si personne n’est connecté qu’il joue tout seul contre le serveur

C - Le chat

Nous avons pensé qu’il était naturellement possible de pouvoir discuter avec les personnes connectées, nous avons donc implémentés un chat. Il a été intéressant de créer un chat de façon à ce que les joueurs puissent communiquer ensemble pour savoir s’ils veulent jouer l’un contre l’autre, ou encore jouer chacun de leur côté, ou encore tous ensemble.

D - Jouer contre le serveur

Ce choix de jeu nous a semblé évident dans le sens où si le client est tout seul à être connecté, il faut quand même lui permettre de jouer. Le client peut alors choisir de jouer contre le serveur avec un certain niveau de difficulté (facile, moyen, difficile).

E - Jouer contre un autre client

Dans l’esprit de vouloir “challenger” un ami, nous avons implémenté ce choix, le client peut avoir jouer “contre” un autre client avec la commande /play_with [pseudo], le serveur demande alors un mot au client adverse qu’il transmettra à son adversaire.

F - Jouer tous ensemble

Ce choix de jeu représente le pendu “multi clients”, on peut également choisir la difficulté du mot à trouver, et chaque client devront alors tour à tour trouver une lettre du mot

proposé. Le jeu s'arrête lorsque les clients ont gagné contre le serveur ou lorsqu'ils ont perdu.

VI- Diagramme de séquence et jeux d'essais

Afin de mieux représenter les actions du client, nous allons réaliser quelques diagrammes de séquences ainsi que le test case associé.

A - Le menu

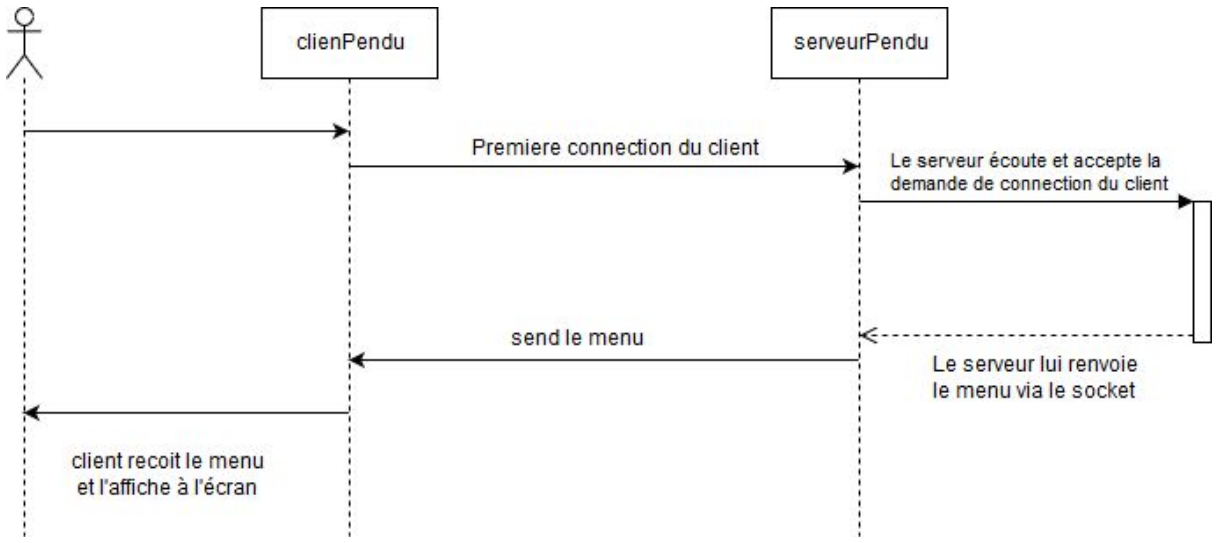

Image 1 : Diagramme de séquence représentant l'envoie du menu lors de la connection d'un nouveau client

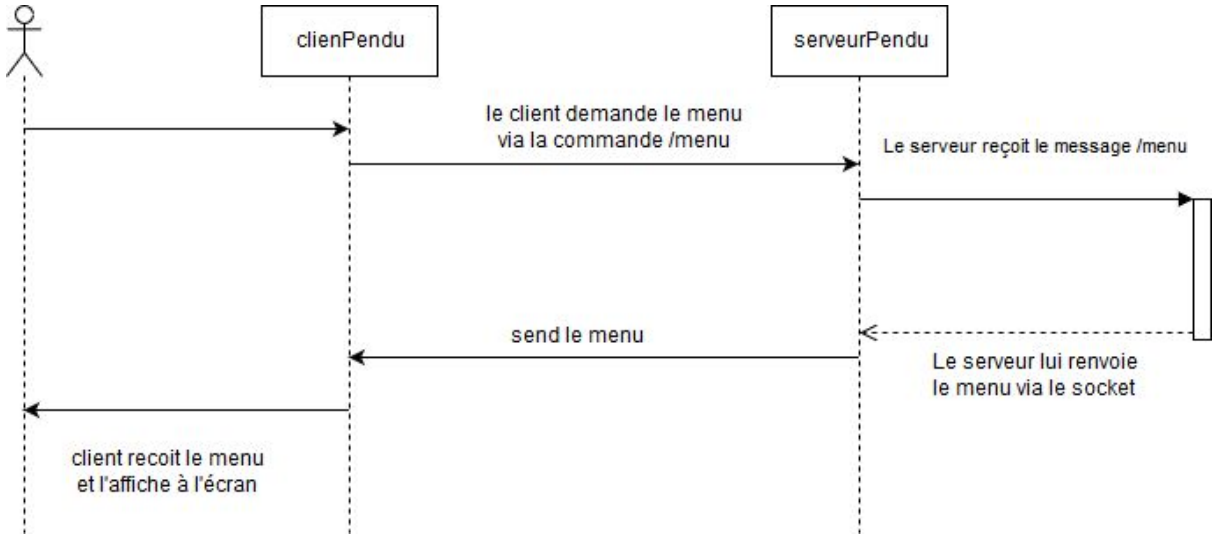

Image 2 : Diagramme de séquence représentant l'envoie du menu lors de la demande d'un client via la commande /menu

B - La liste

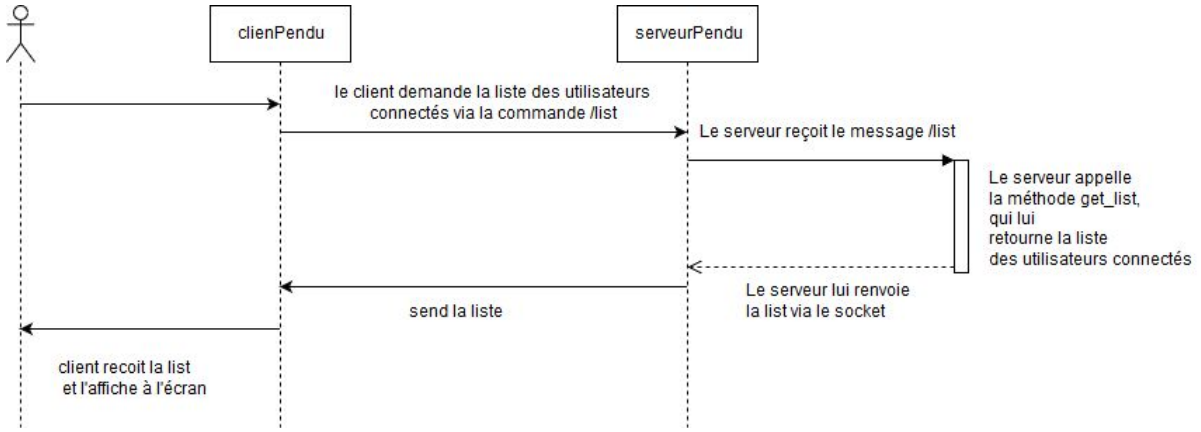

Image 3 : Diagramme de séquence représentant l'envoie de la liste des utilisateurs connectés au client qui la demande.

C - Le chat

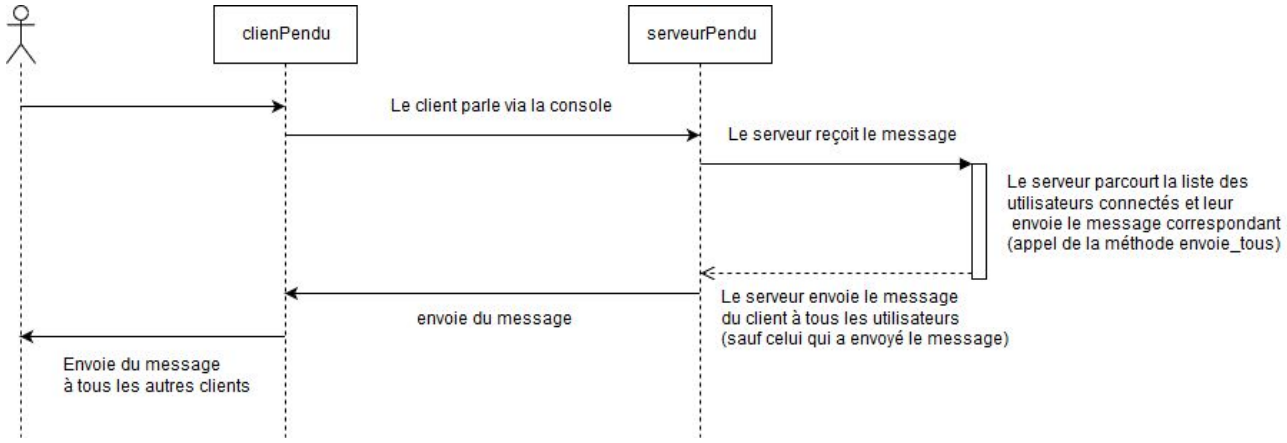

Image 4 : Diagramme de séquence représentant le chat entre les clients connectés

D - Jouer contre le serveur

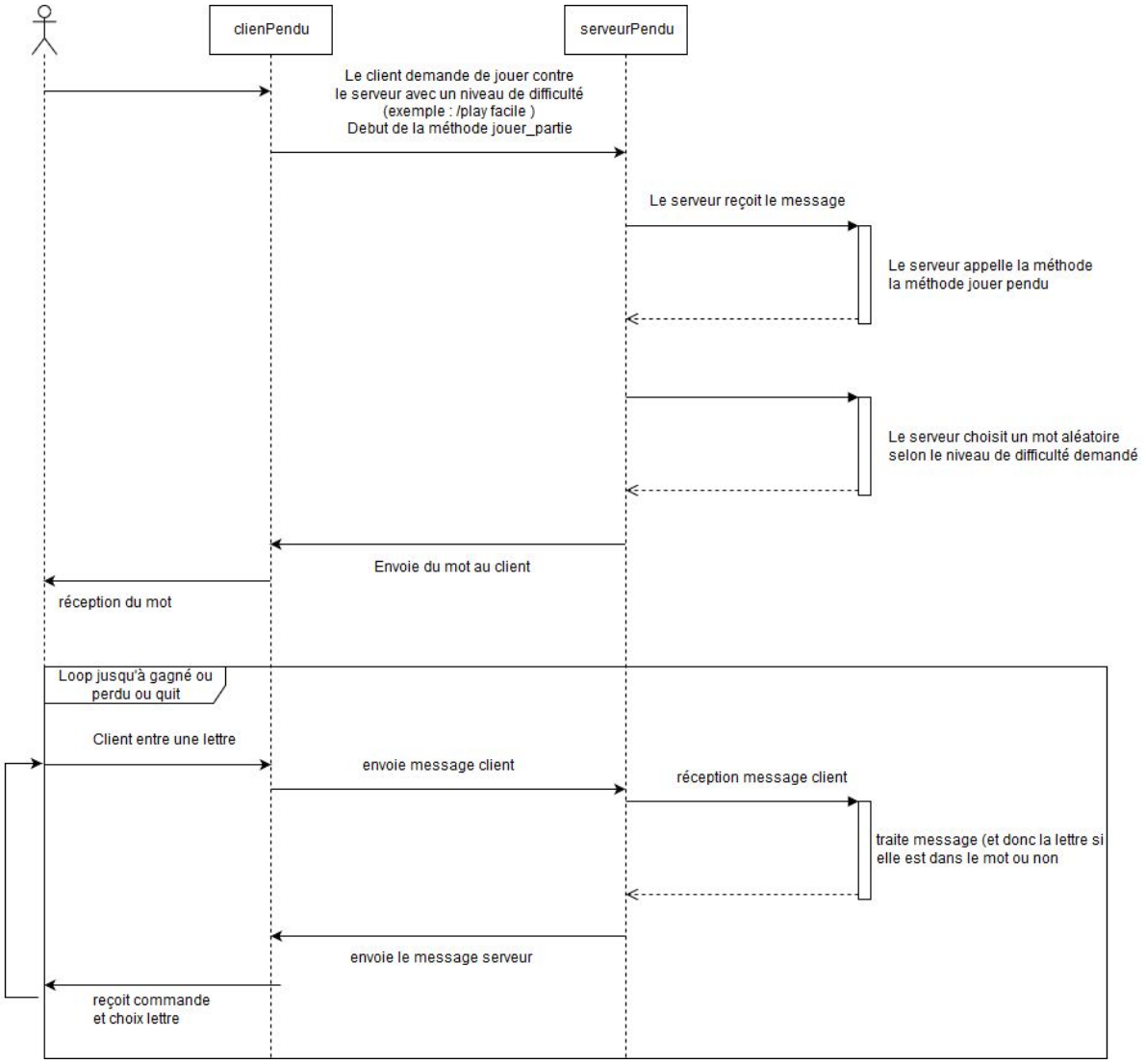

Image 5 : Diagramme de séquence représentant le circuit du jeu du pendu classique contre le serveur

E - Jouer contre un autre client

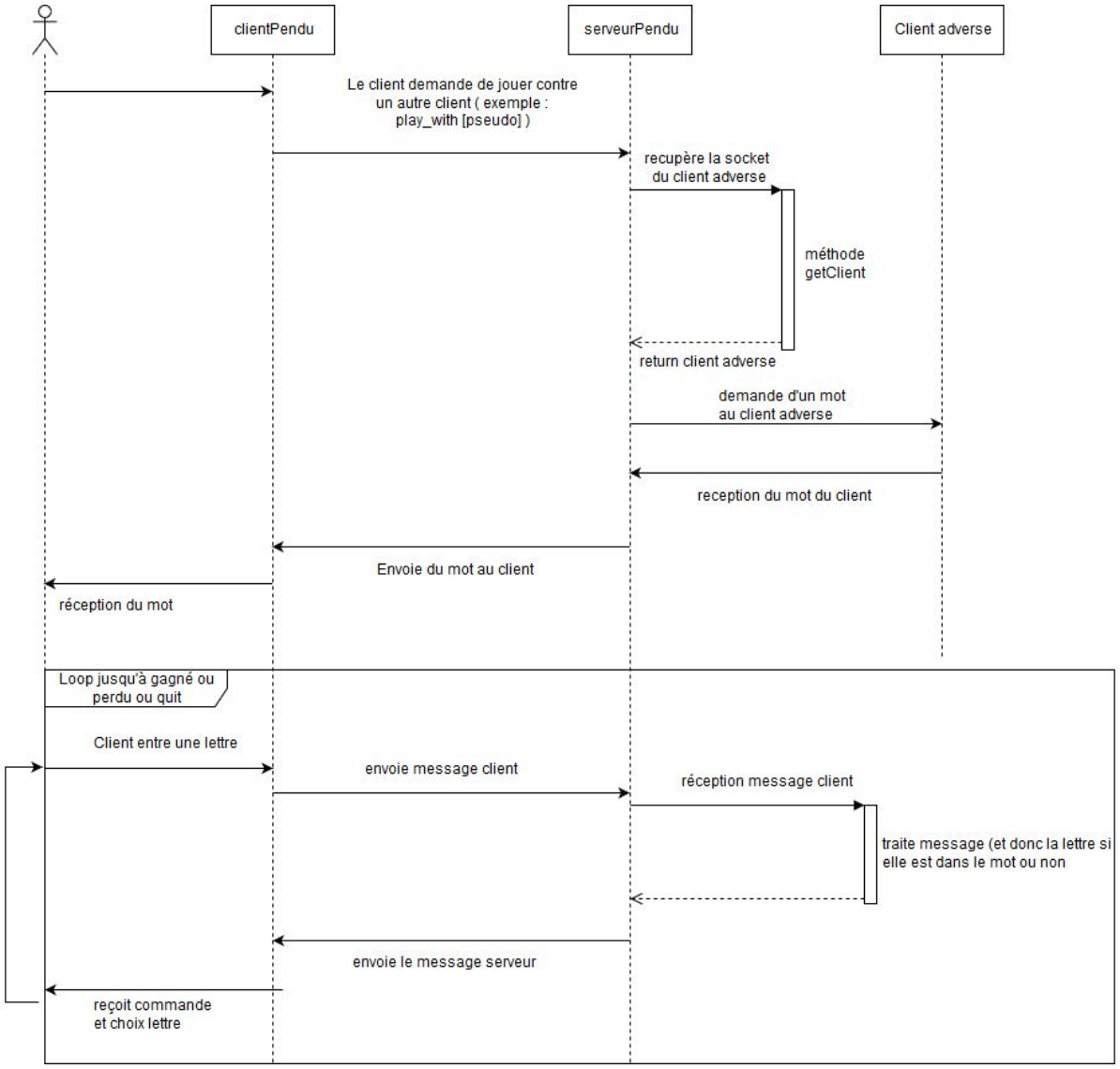

Image 6 : Diagramme de séquence représentant le circuit du jeu du pendu classique contre un autre client

F - Jouer tous ensemble

Cette fonctionnalité permet de faire jouer tous les clients tour à tour.

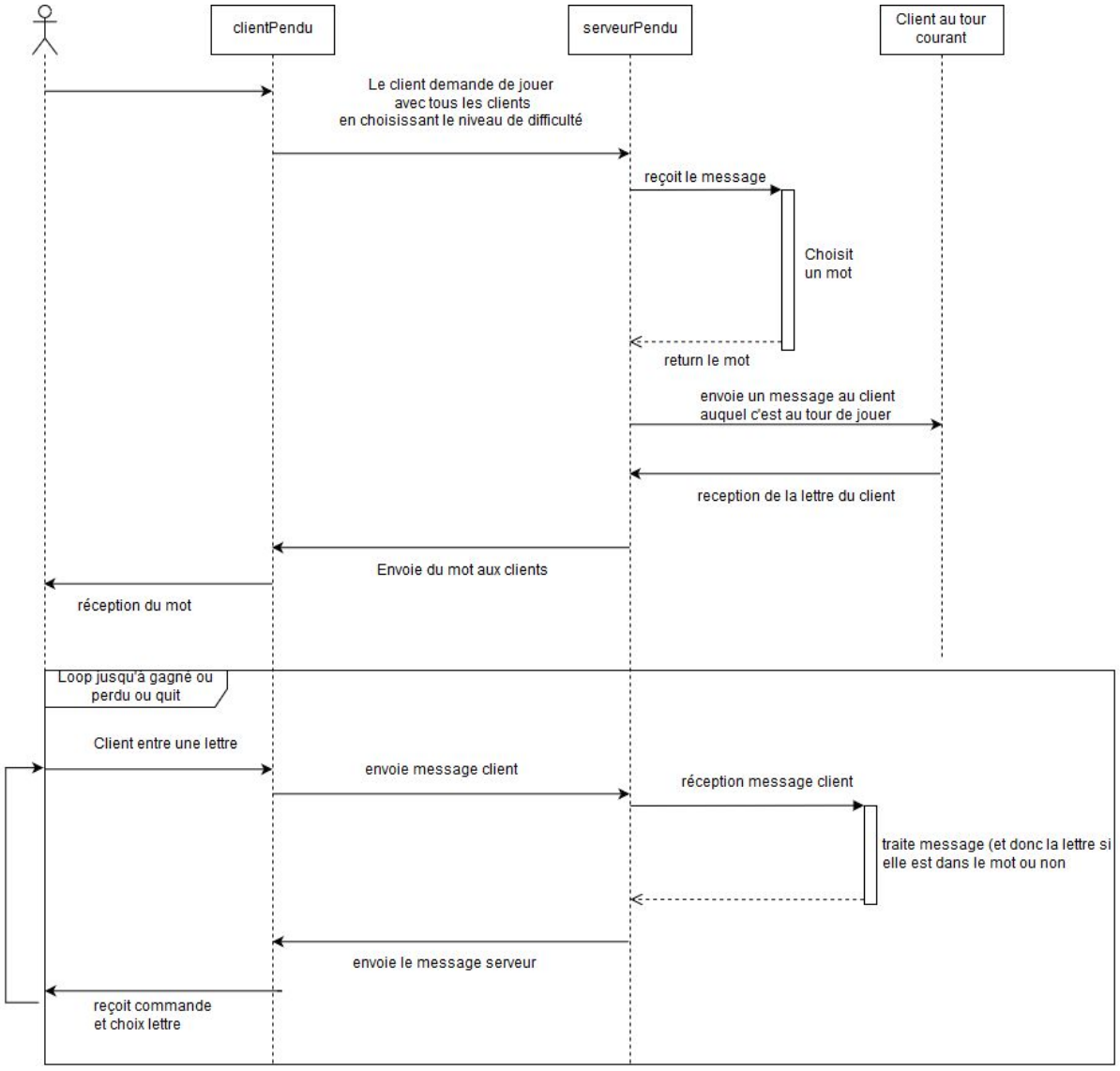

Image 7 : Diagramme de séquence représentant le circuit du jeu du pendu (tous les clients contre le serveur, jeu tour à tour)

Les Jeux d'essais :

1. Test scenario (jeu du pendu classique contre le serveur)

Le client est tout seul à être connecté mais veut tout de même jouer au pendu, il lance donc une partie contre le serveur

Action	**Résultat attendu**	**Résultat actuel**
1. Le client écrit dans la console **/play**	Le serveur lui renvoie un mot de difficulté moyenne	pass

2. Le client écrit dans la console **/play difficile**	Le serveur lui renvoie un mot de difficulté extrême.	pass
3. Le client trouve le mot du pendu.	Un message est envoyé au client comme quoi il a gagné	pass
4. Le client ne trouve pas le mot du pendu	Un pendu est affiché en plus d'un message comme quoi il n'a pas réussi à trouver le mot	pass
5. Le client quitte la partie par un /quit	Partie abandonné, le serveur est prévenu	pass (quelques problèmes, manque d'implémentation de la commande /quit il ne peut que quitter la partie à l'aide de Ctrl + C)

2. Test scenario (jeu du pendu classique contre le serveur)

Le client veut jouer contre un ami et lui (ou une personne inconnue), il récupère le pseudo de ces derniers à l'aide de /list, il lance ensuite la partie avec un /play_with [pseudo]

Action	**Résultat attendu**	**Résultat actuel**
1. Le client écrit dans la console **/play_with Colin**	Le serveur lui renvoie le mot que Colin a choisi	pass
2. Le client trouve le mot de Colin	Un message est envoyé au client comme quoi il a gagné, ainsi qu'à Colin comme quoi son adversaire a trouvé son mot	pass
3. Le client trouve le mot de Colin	Un message est envoyé au client comme quoi il a perdu, ainsi qu'à Colin comme quoi son adversaire n'a pas trouvé son mot	pass
5. Le client quitte la partie par un /quit	Partie abandonné, le serveur est prévenu ainsi que l'adversaire comme quoi il est déconnecté.	pass (quelques problèmes, manque d'implémentation de la commande /quit il ne peut que quitter la partie à l'aide de Ctrl + C)

3. Test scenario (jeu du pendu classique avec tous les clients contre le serveur)

Les clients discutent dans le chat et ont décidés qu'ils voulaient tous jouer ensemble tour à tour, ils se fixent une difficulté et lance une partie

Action	**Résultat attendu**	**Résultat actuel**
1. Un client écrit dans la console **/play_all difficile**	Le serveur envoie le mot caché à tous les clients, celui qui a initialisé la partie commence	pass
2. Les clients trouvent le mot	Un message est envoyé aux clients comme quoi ils ont gagnés.	pass
3. Les clients ne trouvent pas le mot	Un message est envoyé aux clients comme quoi ils n'ont pas gagnés, le pendu se dresse pour tous les clients.	pass
5. Un des clients quitte la partie par un /quit	Partie continue de jouer sans le client déconnecté, le serveur est prévenu ainsi que l'adversaire comme quoi il est déconnecté.	FAIL(quelques problèmes, manque d'implémentation de la commande /quit il ne peut que quitter la partie à l'aide de Ctrl + C)

VII - Fonctionnalités futures et échec

Nous pensons qu'après avoir réalisé il est important de prendre du recul et d'identifier ce qu'on aurait pu mieux faire afin de s'améliorer et de savoir pourquoi nous n'avions pas pu réaliser certaines fonctionnalités.

Le premier "problème" de notre jeu est qu'un seul client peut jouer à la fois contre le serveur, en effet il nous faut attendre que la boucle du jeu du pendu du serveur soit finit pour qu'il envoie le mot à un deuxième client ce qui est dommage, ou alors il faut utiliser la méthode où l'on peut jouer tous ensemble.

Le second problème est un problème d'affichage, en effet voir les résultats du jeu ou encore le menu sur la console n'est jamais très plaisant, nous avons donc identifié dans la phase de conception qu'il existait une bibliothèque GTK qui permettait de réaliser une jolie interface

graphique, malheureusement nous n'avons pas réussi par manque de temps (ce n'est en rien une excuse valable).

Un autre problème qui est des plus important et qu'on aurait aimé régler à temps, lorsqu'un client demande de jouer avec un autre client, nous aurions dû implémenter une fonction/méthode permettant de demander l'autorisation du client adverse avant de lancer la partie.

Comme nous l'avions dit dans la phase de conception, nous avions imaginé qu'il nous aurait été possible d'implémenter un timer pour augmenter la difficulté à trouver le mot.

Nous avions également pensé à une dernière fonctionnalité où le client proposerai également la position de la lettre recherché dans le même but de rajouter une difficulté.
Une fonctionnalité qui permet à l'utilisateur de sauvegarder sa partie et à chaque fois qu'il gagne passe à un niveau supérieur sachant qu'au préalable il à commencé au niveau le plus bas .

VIII - Conclusion

Ce projet, réalisé dans le cadre de notre module de réseau, a constitué avant tout un moyen de nous initier à la mise en place d'architecture client-serveur. Ayant eu très peu l'occasion de manipuler ce genre d'architecture au cours de notre cursus, ce fut un moyen de s'y familiariser. Nous sommes satisfaits de la réalisation de ce projet puisque nous avons réussi notre objectif qui était de réaliser un pendu entre plusieurs clients.
Le choix technique du langage C nous a dans un premier temps relativement surpris, de par sa complexité. Nous avons toutefois bien compris que l'utilisation de ce langage dans le projet visait à nous exercer dans les manipulations de bas niveau. Ces manipulations de bas niveau nous ont permis justement de mieux comprendre le mécanisme des sockets, et leur mise en place.Pour conclure, ce projet nous as permis de mettre en pratique nos acquis en langage C et de manipuler les Sockets.

IX - Annexes

Partie serveur :

```
void init()
{
  #ifdef WIN32
```

```
        WSADATA wsa;
        int err = WSAStartup(MAKEWORD(2, 2), &wsa);
        if(err < 0)
        {
        puts("WSAStartup failed !");
        exit(EXIT_FAILURE);
        }
  #endif
}

void end()
{
  #ifdef WIN32
        WSACleanup();
  #endif
}

int init_connection(char* adresse)
{
  int sock = socket(AF_INET, SOCK_STREAM, 0);

  SOCKADDR_IN sin ;

  if(sock == INVALID_SOCKET)
  {
        perror("socket()");
        exit(errno);
  }

  sin.sin_addr.s_addr = inet_addr(adresse);
  sin.sin_port = htons(PORT);
  sin.sin_family = AF_INET;

  // Condition pour éviter le message classique "Address already in use"
  if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &(int){ 1 }, sizeof(int)) < 0)
  {
        perror("setsockopt(SO_REUSEADDR) failed");
  }


  if(bind(sock,(SOCKADDR *) &sin, sizeof sin) == SOCKET_ERROR)
  {
        perror("bind()");
        exit(errno);
  }
```

```
  if(listen(sock, MAX_CLIENTS) == SOCKET_ERROR)
  {
        perror("listen()");
        exit(errno);
  }
  return sock;
}

void application()
{
  char* ip = "127.0.0.1";

  int sock = init_connection(ip);
  char buffer[BUF_SIZE];
  // L'index du client actuel
  int actuel = 0;

  int max = sock;
  // Un tableau de clients
  Client clients[MAX_CLIENTS];

  fd_set rdfs;

  while(1)
  {
        int i = 0;
        FD_ZERO(&rdfs);

        // ajout file descriptor STDIN_FILENO
        FD_SET(STDIN_FILENO, &rdfs);

        // ajout de la connection au socket
        FD_SET(sock, &rdfs);

        // ajout de socket pour chaque client
        for(i = 0; i < actuel; i++)
        {
        FD_SET(clients[i].sock, &rdfs);
        }

        if(select(max + 1, &rdfs, NULL, NULL, NULL) == -1)
        {
        perror("select()");
        exit(errno);
        }
```

```
/* Si quelqu'un éssaye de taper quelque chose sur le clavier */
if(FD_ISSET(STDIN_FILENO, &rdfs))
{
/* On arrete le serveur */
break;
}
else if(FD_ISSET(sock, &rdfs)) // premiere connection du client
{

/* Nouveau client */
SOCKADDR_IN csin ;
size_t sinsize = sizeof csin;
int csock = accept(sock, (SOCKADDR *)&csin, &sinsize);
if(csock == SOCKET_ERROR)
{
perror("accept()");
continue;
}

// A l'isse de la connection du client il nous faut récupérer son nom
if(read_client(csock, buffer) == -1)
{
printf("Error\n");
/* disconnected */
continue;
}

// Calcul du nouveau maximum du fd
max = csock > max ? csock : max;

FD_SET(csock, &rdfs);

Client c = { csock };
strncpy(c.name, buffer, BUF_SIZE - 1);
clients[actuel] = c;
actuel++;
// envoie du menu à l'issue de la connection du client
char* data = menu();
write_client(c.sock, data);
}
else // Le serveur écoute le client
{
int i = 0;
for(i = 0; i < actuel; i++)
{
if(FD_ISSET(clients[i].sock, &rdfs)) // si le client est en train de parler
```

```
        {
        Client client = clients[i];
        int c = read_client(clients[i].sock, buffer);
        if(c == 0) // client déconnecté
        {
                closesocket(clients[i].sock);
                remove_client(clients, i, &actuel);
                // Envoie du message aux autres clients pour prévenir que le client s'est
déconnecté
                strncpy(buffer, client.name, BUF_SIZE - 1);
                strncat(buffer, " s'est déconnecté !", BUF_SIZE - strlen(buffer) - 1);
                envoie_tous(clients, client, actuel, buffer, 1);
        }
        else
        {
                if (strstr(buffer, "/list")) // le client demande la liste des personnes connectées
                {
                system("clear");
                write_client(clients[i].sock, get_list(clients,client, actuel));
                }
                else if(strstr(buffer, "/menu")) // le client demande le menu
                {
                system("clear");
                char* data = menu();
                write_client(clients[i].sock, data);
                }
                else if(strstr(buffer, "/play_with")) // le client cherche à jouer contre un client
connecté
                {
                system("clear");
                // On récupère le client demandé
                Client client_adverse = getClient(&buffer[11], clients, actuel);

                char status ;
                char demande[BUF_SIZE] = "Veuillez choisir un mot : " ;
                // Envoie d'un message afin que le client adverse choisisse un mot
                write_client(client_adverse.sock, demande);
                read_client(client_adverse.sock, buffer);
                printf("Le mot du client adverse : %s\n", buffer);
                // procédure de jeu avec un client adverse
                play_against(client_adverse.sock, clients[i].sock, buffer);
                }
                else if(strstr(buffer, "/play_all")) // Tous les clients jouent tour à tour
                {
                system("clear");
```

```
                    // procédure de jeu avec tous les clients (le &buffer[10] représente le niveau
de difficulté du jeu)
                    play_all(clients, client, actuel, &buffer[10]);
                    }
                    else if (strstr(buffer, "/play")) // Joue contre le serveur
                    {
                    system("clear");
                    // procédure de jeu avec tous les clients (le &buffer[6] représente le niveau de
difficulté du jeu)
                    jouerPendu(clients[i].sock, &buffer[6]);
                    }
                    else
                    {
                    // Chat classique, on envoie le message à tous les clients
                    envoie_tous(clients, client, actuel, buffer, 0);
                    }
          }
          }
          }
          }
   }
   // serveur déconnecté, on delete tout les clients
   clear_clients(clients, actuel);
   // close la connection
   end_connection(sock);
}

int main(int argc, char **argv)
{

   init();

   application();

   end();

   return EXIT_SUCCESS;
}
```

Partie client :

```
void init()
{
  #ifdef WIN32 // windows
        WSADATA wsa;
        int err = WSAStartup(MAKEWORD(2, 2), &wsa);
        if(err < 0)
        {
        puts("WSAStartup failed !");
        exit(EXIT_FAILURE);
        }
  #endif
}

void end()
{
        #ifdef WIN32 // windows
        WSACleanup();
        #endif
}

void application(char *address, char *name)
{
  int sock = init_connection(address);
  char buffer[BUF_SIZE];

  fd_set rdfs;

  // Une fois connecté, on envoie notre nom au serveur.
  write_server(sock, name);

  while(1)
  {
        FD_ZERO(&rdfs);

        // ajout file descriptor STDIN_FILENO
        FD_SET(STDIN_FILENO, &rdfs);

        // ajout de la connection au socket
        FD_SET(sock, &rdfs);

        if(select(sock + 1, &rdfs, NULL, NULL, NULL) == -1)
        {
        perror("select()");
        exit(errno);
        }
```

```
        /* Si le client éssaye de taper quelque chose sur le clavier */
        if(FD_ISSET(STDIN_FILENO, &rdfs))
        {
        // On récupère ce que le client a écrit
        fgets(buffer, BUF_SIZE - 1, stdin);
        char *p = NULL;
        p = strstr(buffer, "\n");
        if(p != NULL)
        {
        *p = 0;
        }
        else
        {
        // clear le buffer
        buffer[BUF_SIZE - 1] = 0;
        }
        // On envoie le message au serveur
        write_server(sock, buffer);

        if(strstr(buffer, "/play_with")) // Demande de jouer contre un client
        {
        system("clear");
        jouer_partie(sock);
        }
        else if(strstr(buffer, "/play_all")) // Demande de jouer avec tous les clients contre le
serveur
        {
        system("clear");
        jouer_all(sock);
        }
        else if(strstr(buffer, "/play")) // Demande de jouer contre le serveur
        {
        system("clear");
        jouer_partie(sock);
        }
        else if(strstr(buffer, "/list"))
        {
        system("clear");
        }
        else if(strstr(buffer, "/menu"))
        {
        system("clear");
        }
        }
        // Le serveur ou les autres clients nous envoient un message
        else if(FD_ISSET(sock, &rdfs))
```

```
        {
        int n = read_server(sock, buffer);
        // Le serveur ne répond plus
        if(n == 0)
        {
        printf("Server disconnected !\n");
        close(sock);
        break;
        }
        else if(strstr(buffer, "PLAY_TOUS")) // Si c'est la demande de jouer tous ensemble
        {
        //system("clear");
        jouer_all(sock); // On joue avec tous le monde
        }
        else
        {
        puts(buffer); // On affiche le message, chat classique
        }
        }
  }
  end_connection(sock);
}

int init_connection(char *address)
{
  int sock_client;
  HOSTENT* hp;
  SOCKADDR_IN adresse_serv;

  if ((hp = gethostbyname(address)) == NULL)
  {
        perror("Problème avec l'hostname");
        exit(1);
  }

        bcopy(hp->h_addr, &adresse_serv.sin_addr, hp->h_length);
        adresse_serv.sin_family = hp->h_addrtype;
        adresse_serv.sin_port = htons(PORT);

        if ((sock_client = socket(hp->h_addrtype, SOCK_STREAM, 0)) < 0)
        {
        perror("socket");
        exit(1);
        }

        if (connect(sock_client, (SOCKADDR *) &adresse_serv, sizeof (adresse_serv)) < 0)
```

```
        {
        perror("connect");
        close(sock_client);
        exit(EXIT_FAILURE);
        }
        fprintf(stderr, "Connexion avec le serveur établie\n");

        return sock_client;
}

int main(int argc, char **argv)
{
  if(argc < 2) // Indiquer l'adresse utiliser et son pseudo
  {
        printf("Usage : %s [address] [pseudo]\n", argv[0]);
        return EXIT_FAILURE;
  }

  init();

  application(argv[1], argv[2]);

  end();

  return EXIT_SUCCESS;
}
```

Manuel d’utilisation

Coté serveur :

gcc serveurPendu.c -o serveur
./serveur

Coté client :
gcc clientPendu.c -o client
./client 127.0.0.1 [Nom du joueur]

Toujours coté client, si vous voulez jouer contre le serveur :
/play [niveau]

Une fois fini, connecté un nouveau client
./client 127.0.0.1 [Nom du joueur 2]

/play_with Nom du joueur

Une fois fini, jouer tous ensemble contre le serveur tour à tour :

/play_all [difficulté]