1 Introduction

Le but de ce projet est la réalisation d'une application Client-Serveur mettant en place un service de jeu "Morpion". Les notions de IPC (Inter Process Communication) et System Calls sont explorés.

2 Description d'architecture

Le programme est écrit en C, en utilisant tous les appels systèmes qui permettent de communiquer avec l'API (Application Programming Interface) des sockets. Un grand nombre de systèmes d'exploitation offrent cette interface.

Ces sockets font possible la communication inter processus aussi bien sur une même machine qu'à travers un réseau. Sachant où trouver un socket les processus peuvent écrire ou lire des données. Une adresse socket est la combinaison d'une adresse IP (Internet Protocol) et un numéro de port. Cette notion de port permet de distinguer les différents interlocuteurs dans une même machine, et séparer leur services offert. Par example, plusieurs logiciels serveur et clients peuvent être créé et la communication entre eux se fait sans interférence.

Un socket est aussi défini par le protocole de transport des octets (segments de paquets) dans le réseau. Pour ce projet le protocole choisi est TCP (Transmission Control Protocol), qui est fiable et en mode connecté (avec validation et vérification d'erreurs). Le réseau utilisera le protocole IPv6.

Le serveur devra offrir la possibilité de connexion à un nombre non-limité de clients. Pour respecter cette contrainte, le serveur créera un processus fils (fork) et un nouveau socket pour la transmission des données entre lui et son nouveau client. Dans un modèle client-serveur, ce derniers envoyent normalement les rêquetes, et le serveur devra dans la mesure du possible, traiter les données pour distribuer la charge de travail.

2.1 Gestion d'erreurs et exceptions

Toute erreur qui a lieu au moment de la connexion résulte en `exit()` du programme. L'appel système `perror()` est utilisé pour afficher un message d'erreur sur la sortie d'erreur standard (`stderr`) décrivant la dernière erreur rencontrée durant un appel système. La librarie `errno.h` décrit aussi un grand nombre des macros et codes d'erreurs qui en font partie de la gestion d'erreurs de la fonction mentionée.

3 System calls

- `getaddrinfo()`: L'usage de cette fonction est optionel, vu que client et serveur sont dans la mâchine et on connaît le protocole, le type de socket et la version IP à utiliser. Néanmoins, ça peut servir pour étendre les fonctionnalités du programme.
- `socket()`: Crée un point final de communication et renvoie un descripteur de fichier. Les sockets peuvent avoir de différents types, qui correspondent à la famille de protocoles à utiliser.
- `bind()`: Fournie un nom (adresse) à un socket, à l'aide des structures de données précédemment completées (comme `sockaddr`). Avant cet appel, le socket créé avec `socket()` n'a pas de nom formel.
- `listen()`: Marque le socket référencé comme un socket passive, c'est-à-dire comme un socket qui sera utilisé pour accepter les demandes de connexions entrantes. Un de ses paramètre définit la longeur maximale de la file des connexions en attente.
- `accept()`: Pour les sockets en mode connecté (`SOCK_STREAM` par example, qui utilise le protocole TCP), cet appel extrait la première connexion de la file d'attente et renvoie un nouveau "file descriptor" qui fait référence à un nouveau socket. Ce descripteur de fichier peut être utilisé pour transmettre information entre les processus, tandis que le socket originel est laissé intact et continue à l'écoute.

- `connect()`: Connecte un socket (défini toujours par un descripteur de fichier) à une adresse (IP + Port). Si le socket est en mode connecté, alors cet appel essaye de connecter avec un autre socket à l'adresse indiquée.
- `send()`: Permet de transmettre un message à destination d'une autre socket. L'appel système `write()` est le même sans avoir la possibilité d'ajouter des *flags*.
- `recv()`: Utilisé pour recevoir des messages depuis un socket, c'est-à-dire, lire des données dans le socket reférence par un descripteur de fichier.

3.1 Structures de données

Deux structures de donnees doivent être remplies (à la main ou avec les méthodes auxilaires).

```
struct addrinfo {
    int              ai_flags;
    int              ai_family;    // AF_INET, AF_INET6, AF_UNSPEC
    int              ai_socktype;  // SOCK_STREAM (TCP), SOCK_DGRAM (UDP)
    int              ai_protocol;  // 0 -> any
    size_t           ai_addrlen;   // size of ai_addr in bytes
    struct sockaddr *ai_addr;      // struct sockaddr_in (IPv4) or _in6
    char            *ai_canonname;
    struct addrinfo *ai_next;      // pointer next node
};
```

Dans notre cas et pour ce projet, cette structure de donées contient information superflue, et on s'interesse principalement à `*ai_addr` qui est de type `struct sockaddr_in6`. Contient toute l'information pour créer un socket.

```
struct sockaddr_storage {
    sa_family_t  ss_family;     // address family

    char      __ss_pad1[_SS_PAD1SIZE];
    int64_t   __ss_align;
    char      __ss_pad2[_SS_PAD2SIZE];
};
```

`sockaddr_storage` contient une adresse réseau qu'on peut voir à l'aide de la function `inet_ntop()`. (network to presentation)

4 Listing

```
1
#include <stdlib.h>
3 #include <stdio.h>
#include <unistd.h>
5 #include <errno.h>
#include <string.h>
7 #include <netdb.h>
#include <sys/types.h>
9 #include <netinet/in.h>
#include <sys/socket.h>
11 #include <arpa/inet.h>

13 #define PORT "5555"
#define MAXDATASIZE 200
15 #define SIZE 3
#define GAME_START 1
17
// get sockaddr, IPv6:
19 void *get_in_addr(struct sockaddr *sa)
{
21   return &(((struct sockaddr_in6*)sa)->sin6_addr);
}
23


25 void get_picture(char morpion[SIZE][SIZE]) {
  printf("\n--␣Game␣Board␣--␣\n\n");
27   printf("+---+---+---+\n");
  printf("|␣%c␣|␣%c␣|␣%c␣|\n", morpion[0][0], morpion[0][1], morpion
    [0][2]);
29   printf("+---+---+---+\n");
  printf("|␣%c␣|␣%c␣|␣%c␣|\n", morpion[1][0], morpion[1][1], morpion
    [1][2]);
31   printf("+---+---+---+\n");
  printf("|␣%c␣|␣%c␣|␣%c␣|\n", morpion[2][0], morpion[2][1], morpion
    [2][2]);
33   printf("+---+---+---+\n\n");
}
35
void get_invite(int sockfd) {
37   char buf[70];
  recv(sockfd, buf, sizeof buf, 0);
39   printf(buf);
  int answer;
41   scanf("%i", &answer);
  send(sockfd, &answer, sizeof(int), 0);
43 }


45
void play(int sockfd) {
47   char buf[SIZE][SIZE];
```

```
  while (recv(sockfd, buf, MAXDATASIZE-1, 0) != 0) {
    system("clear");            /* Only works for UNIX, not portable */
    get_picture(buf);
    printf("Case à choisir (1-9): ");
    int to_send;
    scanf("%i", &to_send);
    send(sockfd, &to_send, 4, 0);
  }
}

int main(int argc, char *argv[]) {
  int sockfd;
  struct addrinfo goodies, *realinfo, *p;
  int ecode;
  char s[INET6_ADDRSTRLEN];
  if (argc != 2) {
    fprintf(stderr,"usage: client hostname\n");
    exit(1);
  }

  memset(&goodies, 0, sizeof goodies);
  goodies.ai_family = AF_UNSPEC;
  goodies.ai_socktype = SOCK_STREAM;
  if ((ecode = getaddrinfo(argv[1], PORT, &goodies, &realinfo)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(ecode));
    return 1;
  }
  if ((sockfd = socket(realinfo->ai_family, realinfo->ai_socktype,
                       realinfo->ai_protocol)) == -1) {
    perror("client: Failed to created socket");
    exit(EXIT_FAILURE);
  }
  if (connect(sockfd, realinfo->ai_addr, realinfo->ai_addrlen) == -1) {
    close(sockfd);
    perror("client: connect");
    exit(EXIT_FAILURE);
  }

  /* Network to presentation of address */
  inet_ntop(realinfo->ai_family, get_in_addr((struct sockaddr *)realinfo
   ->ai_addr),
            s, sizeof s);

  printf("Connecting to %s\n", s);
  freeaddrinfo(realinfo); // all done with this structure

  get_invite(sockfd);
  play(sockfd);
  close(sockfd);
  return 0;
}
```

client.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <signal.h>

#define PORT "5555"
#define BACKLOG 5               /* Pending connections the queue will
   hold */
#define SIZE 3
#define PLAYER 1
#define MACHINE 0

const int GAME_START = 1;

struct addrinfo *getGoodies(void) {
  struct addrinfo *goodies;
  struct addrinfo suggestions;            /* Main data structure */
  memset(&suggestions, 0, sizeof(suggestions));
  suggestions.ai_family = AF_INET6;      /* IPv4 not IPv6 */
  suggestions.ai_protocol = IPPROTO_TCP; /* TCP no raw or UDP */
  suggestions.ai_socktype = SOCK_STREAM; /* TCP connection oriented */

  int ecode;
  if((ecode = getaddrinfo(NULL, PORT, &suggestions, &goodies)) != 0) {
    printf("Failed getting address information: %s\n" , gai_strerror(ecode
   ));
  } /* status should be 0, not -1 */

  return goodies;
}

int prepareSocket(struct addrinfo* goodies) {
  int sockfd;
  if ((sockfd = socket(goodies -> ai_family,
                       goodies -> ai_socktype,
                       goodies -> ai_protocol)) == -1) {
    perror("server - Failed creating socket"); /* and print last error
   encountered */
    exit(EXIT_FAILURE);
  }

  int reuse_addr_to_true = 1;
  if (setsockopt(sockfd, SOL_SOCKET,
                 SO_REUSEADDR, &reuse_addr_to_true, sizeof(int)) == -1) {
    perror("server - Failed setting reuse address option for the socket");
```

```
    exit(EXIT_FAILURE);
  }

  if (bind(sockfd, goodies -> ai_addr, goodies -> ai_addrlen) == -1) {
    close(sockfd);
    perror("server - Failed to assign addr to socket file descriptor");
    exit(EXIT_FAILURE);
  }

  freeaddrinfo(goodies);        /* No need for this anymore */

  if (listen(sockfd, BACKLOG)) {
    perror("server - Could not mark the socket referenced by sockfd as
     passive");
    exit(EXIT_FAILURE);
  }

  return sockfd;

}

int libre(char morpion[SIZE][SIZE], int x, int y) {
  if (morpion[x][y] == 'X' || morpion[x][y] == '0') {
    return 0;
  }
  return 1;
}

int mark_case(char morpion[SIZE][SIZE], int choice, int player) {
  int x;
  int y;
  if (!choice%SIZE) {
    x = (choice/SIZE) - 1; y = SIZE - 1;
  } else {
    x = choice/SIZE; y = (choice%SIZE) - 1;
  }

  if (libre(morpion, x, y)) {
    if (player) {
      morpion[x][y] = 'X';
    } else {
      morpion[x][y] = '0';
    }
    return 1;
  }
  return 0;
}

void choose_case(char morpion[SIZE][SIZE]) {
  int chosen = 0;
  while (!chosen) {
    int choice = rand() % 9 + 1;
    printf("choice machine: %i\n", choice);
```

```
    if (mark_case(morpion, choice, MACHINE)) chosen = 1;
  }
}

/* TODO: End game condition not working */

/* int game_won(char morpion[SIZE][SIZE]) { */
/*   if((morpion[0][0] == morpion[1][1] && */
/*       morpion[0][0] == morpion[2][2]) || */
/*      (morpion[0][2] == morpion[1][1] && */
/*       morpion[0][2] == morpion[2][0])) */
/*     return 1; */
/*   else */
/*     for(int row = 0; row <= 2; row ++) */
/*       if((morpion[row][0] == morpion[row][1] && */
/*        morpion[row][0] == morpion[row][2])|| */
/*       (morpion[0][row] == morpion[1][row] && */
/*        morpion[0][row] == morpion[2][row])) */
/*      return 1; */
/*   return 0; */
/* } */


void play_morpion(int new_fd) {
  int times;
  //int winner = 0;
  char morpion[SIZE][SIZE] = {{'1', '2', '3'},
                              {'4', '5', '6'},
                              {'7', '8', '9'}};
  int choice;
  for (times = 0; times < SIZE*SIZE; ++times) {

    /* if (game_won(morpion)) */
    /*   done = 1; winner = PLAYER; out(winner); */
    choose_case(morpion);
    send(new_fd, morpion, sizeof morpion, 0);
    /* if (game_won(morpion)) */
    /*   done = 1; winner = MACHINE; out(winner); */
    recv(new_fd, &choice, sizeof choice, 0);

    if (choice > 0 && choice < 10) {
      mark_case(morpion, choice, PLAYER);
    }
  }
}

int invite(const int new_fd) {
  char invite[70] = "\nEnvoyer le chiffre 1 pour jouer, le chiffre 2 pour
   se déconnecter: ";
  int answer = 0;
  send(new_fd, invite, sizeof invite, 0);
  int say;
  recv(new_fd, &say, sizeof say, 0);
```

```
  if (say == 1) {
    answer = say;
  }
  return answer;
}


int main(void) {

  struct addrinfo *goodies = getGoodies();
  int sockfd = prepareSocket(goodies); /* Listener */
  struct sockaddr_in6 guest;
  socklen_t sin_size;

  printf("Server waiting for connection...\n");

  while (1) {
    int new_fd;                          /* New file descriptor, where
   stuff happens */
    sin_size = sizeof guest;
    if ((new_fd = accept(sockfd, (struct sockaddr *)&guest, &sin_size))
   == -1) {
      perror("server - Error extracting connection request");
      exit(EXIT_FAILURE);
    }

    char from[INET6_ADDRSTRLEN];
    inet_ntop(AF_INET6, &(guest.sin6_addr), from, INET6_ADDRSTRLEN);
    printf("server - Got connection from: %s\n", from);

    if (!fork()) {
      close(sockfd);            /* Close listener, dont need it */
      if (invite(new_fd)) play_morpion(new_fd);
      close(new_fd);
      exit(EXIT_SUCCESS);
    }
    close(new_fd);              /* Already forked */

  }


  return 0;
}
```

server.c

```
CC = gcc
FLAGS = -Wall -Wextra
FILE = rapport
DIR = tex_files
OPTIONS = -xelatex -output-directory=$(DIR)
PDF_V = evince

compile: client.o server.o
client.o: client.c
	$(CC) $< -o $@

server.o: server.c
	$(CC) $< -o $@

# Run

connect: client.o
	./$< localhost

serve: server.o
	./$<

# LaTeX gen

tex: auto view

mac_tex: auto mac_view

auto: $(FILE).tex
	latexmk $(OPTIONS) $<

view: $(DIR)/$(FILE).pdf
	$(PDF_V) $< &

mac_view: $(DIR)/$(FILE).pdf
	open $< &
```

Makefile