Compte rendu TP2/3 Réseau

Léo Boisson

Table des matières

1 Introduction

1.1 Historique

Les sockets sont un ensemble de fonctions de communications, proposés en 1980 pour le **Berkeley Software Distribution** (BSD), en open source, par l'université de Berkeley. Elles permettent a des applications de se connecter entre elles, via un principe client/serveur.
Aujourd'hui, les sockets sont disponibles dans quasiment tous les langages de programmations, et font offices de norme.
On distingue deux modes de communication avec les sockets :

- Le mode connecté, qui utilise le protocole TCP. Dans ce mode, une connexion durable est établie entre les deux processus, afin que l'adresse de destination ne soit pas nécessaire à chaque envoie de données.
- Le mode non-connecté, qui utilise le protocole UDP. Ce mode nécessite l'adresse de destination à chaque envoi, et il n'y a pas de confirmation du bon envoi des données. Ce mode est plus adapté à l'envoi de flux audio ou vidéo.

+

1.2 Objectif du TP

L'objectif de ce TP est d'aborder le développement de sockets, et de se familiariser avec les outils qui vont avec (les primitives). Pour cela, nous allons programmer des applications client/serveur basique, pour ensuite observer et analyser les échanges de données entre ces applications.

2 Application client/serveur echo

2.1 principe de fonctionnement

Nous réalisons une application de type client/serveur, afin de calculer le temps que mets une requête du client à parvenir au serveur, à être traitée, puis à revenir au client. Nous utilisons la primitive *gettimeofday*, qui renvoie l'heure en secondes et microsecondes (avec la zone temporelle passée en argument).
En utilisant cette fonction deux fois dans le code du client, une première fois lors de l'envoi au serveur, et une seconde lors de la réception depuis le serveur, il nous suffit de soustraire la première à la seconde pour obtenir le temps d'aller et retour d'une requête.

2.2 Code source du client

On constateras les *gettimeofday*, lignes 81 et 104.

```
1 #include <sys/time.h>
2 #include <sys/types.h>
3 #include <sys/socket.h>
4 #include <netinet/in.h>
5 #include <arpa/inet.h>
6 #include <netdb.h>
7 #include <stdio.h>
```

```c
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <time.h>

#define SUCCESS 0
#define ERROR 1
#define SERVER_PORT 1500
#define MAX_MSG 100

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

    int sd, rc;
    static char rcv_msg[MAX_MSG];

    struct sockaddr_in localAddr, servAddr;
    struct hostent * h;

    struct timeval tv_envoi, tv_retour;
    struct timezone tz;
    long long diff;

    //Verification du nombre d'arguments
    if(argc != 2)
    {
        printf("Usage : %s <server>\n", argv[0]);
        exit(ERROR);
    }

    //Recupere la structure de type hostent pour l'hote passe en param.
    h = gethostbyname(argv[1]);
    if(h == NULL)
    {
        printf("%s : Uncknow host '%s'", argv[0], argv[1]);
        exit(ERROR);
    }

    //Initialisation des champs de la structure srvAddr
    servAddr.sin_family = h->h_addrtype;
    memcpy((char *)&servAddr.sin_addr.s_addr, h->h_addr_list[0], h->h_length);
    servAddr.sin_port = htons(SERVER_PORT);

    //Creation de la socket
    sd = socket(AF_INET, SOCK_STREAM, 0);
    if(sd < 0)
    {
        perror("Cannot open socket : ");
        exit(ERROR);
    }

    //Bind
    localAddr.sin_family = AF_INET;
    localAddr.sin_addr.s_addr = htonl(INADDR_ANY);
    localAddr.sin_port = htons(0);

    rc = bind(sd, (struct sockaddr *) &localAddr, sizeof(localAddr));
    if(rc < 0)
    {
```

```
67          printf("%s : cannot bind port TCP %u\n", argv[0],
                SERVER_PORT);
68          perror("Error : ");
69          exit(ERROR);
70      }
71
72      //Connexion au serveur
73      rc = connect(sd, (struct sockaddr *) &servAddr, sizeof(
            servAddr));
74      if(rc < 0)
75      {
76          perror("Cannot connect : ");
77          exit(ERROR);
78      }
79
80      //Recuperation du temps pre-envoi du message "echo"
81      gettimeofday(&tv_envoi, &tz);
82
83      //Envoi du message "echo"
84      rc = send(sd, "echo", strlen("echo")+1, 0); // +1 pour le
            \0
85      if(rc < 0)
86      {
87          printf("Cannot send data : ");
88          close(sd);
89          exit(ERROR);
90      }
91
92      printf("%s : data send (%s)\n", argv[0], "echo");
93
94      //Reception du message retourne par le serveur
95      rc = recv(sd,rcv_msg,MAX_MSG,0);
96      if(rc < 0)
97      {
98          printf("Cannot receive data : ");
99          close(sd);s
100         exit(ERROR);
101     }
102
103     //Recuperation du temps post-reception du retour
104     gettimeofday(&tv_retour, &tz);
105
106     printf("%s : data received (%s)\n", argv[0], rcv_msg);
107
108      //Calcul de la difference entre les deux temps
109     diff=(tv_retour.tv_sec-tv_envoi.tv_sec) * 1000000L + (
            tv_retour.tv_usec-tv_envoi.tv_usec);
110
111     printf("Duree A/R : %llu usec\n",diff);
112
113     return(SUCCESS);
114 }
```

2.3 Code source du serveur

Le serveur ne fait que réceptionner le message du client, un "echo", et le renvoi immédiatement.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
```

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>


#define SUCCESS 0
#define ERROR 1
#define SERVER_PORT 1500
#define MAX_MSG 100

#define END_LINE 0x0



//Declaration de la fonction read_line
int read_line (int newSd,char * line_to_return);

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

    int sd,newSd, r;
    socklen_t cliLen;

    struct sockaddr_in cliAddr, servAddr;
    char line[MAX_MSG];

    //Creation de la socket
    sd = socket(AF_INET,SOCK_STREAM,0);
    if(sd<0)
    {
        perror("Cannot open socket");
        exit(ERROR);
    }

    //Initialisation des champs de la structure servAddr
    servAddr.sin_family = AF_INET;
    servAddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servAddr.sin_port = htons (SERVER_PORT);

    //Bind
    if(bind(sd,(struct sockaddr *) &servAddr, sizeof(servAddr)
        )<0)
    {
        perror("Cannot bind port");
        exit(ERROR);
    }

    //Listen
    listen(sd,5);

    while (1)
    {
        printf("%s : waiting for data on port TCP %u \n",argv
            [0],SERVER_PORT);
        cliLen = sizeof(cliAddr);
        newSd = accept(sd,(struct sockaddr *) &cliAddr,&cliLen
            );
        if(newSd<0)
```

```
63          {
64               perror("Cannot accept connection");
65               exit(ERROR);
66          }
67
68          memset(line,0x0,MAX_MSG);
69
70          /* Reception de la requete */
71          while(read_line(newSd,line)!= ERROR)
72          {
73               printf("%s: received from %s:TCP%d : %s\n",argv
                     [0],inet_ntoa(cliAddr.sin_addr),ntohs(cliAddr.
                     sin_port),line);
74
75               //Envoi de la reponse
76               r = send(newSd, line, strlen(line)+1, 0); // +1
                     pour le \0
77               if(r < 0)
78               {
79                    printf("Cannot send data :");
80                    close(newSd);
81                    exit(ERROR);
82               }
83               memset(line,0x0,MAX_MSG);
84          }
85
86     }
87
88     return(SUCCESS);
89 }
90
91 int read_line (int newSd,char * line_to_return)
92 {
93     static int rcv_ptr=0;
94     static char rcv_msg[MAX_MSG];
95     static int n;
96     int offset;
97
98     offset=0;
99
100    while(1)
101    {
102         if(rcv_ptr==0)
103         {
104              /* read data from socket */
105              memset(rcv_msg,0x0,MAX_MSG);/* init buffer */
106              n = recv(newSd,rcv_msg,MAX_MSG,0); /* wait for
                     data */
107              if(n < 0)
108              {
109                   perror("Cannot receive data ");
110                   return ERROR;
111              }
112              else if(n==0)
113              {
114                   printf("Connection closed by client \n");
115                   close(newSd);
116                   return ERROR;
117              }
118         }
119
120         /* if new data read on socket */
```

```
121             /* OR */
122             /* if another lise is still in buffer */
123             /* copy ligne into 'line_to_return'*/
124             while(*(rcv_msg+rcv_ptr)!=END_LINE && rcv_ptr<n)
125             {
126                 memcpy(line_to_return+offset,rcv_msg+rcv_ptr,1);
127                 offset++;
128                 rcv_ptr++;
129             }
130
131             /* end of line + end of buffer => return line */
132             if(rcv_ptr == n-1)
133             {
134                 /* set last by to END_LINE */
135                 *(line_to_return+offset)=END_LINE;
136                 rcv_ptr=0;
137                 return ++offset;
138             }
139
140
141             /* end of line but still some data in buffer => return
                    line */
142             if(rcv_ptr < n-1)
143             {
144                 /* set last by to END_LINE */
145                 *(line_to_return+offset)=END_LINE;
146                 rcv_ptr++;
147                 return ++offset;
148             }
149
150             /* end of buffer but line is not ended => */
151             /* wait for more data to arrive on socket */
152             if(rcv_ptr == n)
153             {
154                 rcv_ptr = 0;
155             }
156         }
157 }
```

2.4 Analyse des trames réseau

Les programmes affichent ceci dans le terminal :

FIGURE 1 – Pour le serveur :

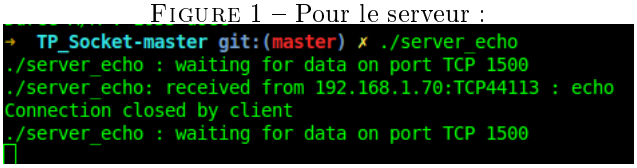
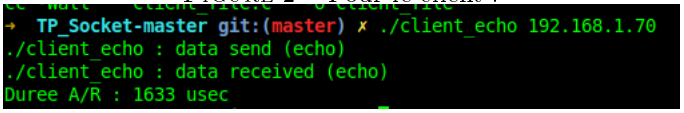
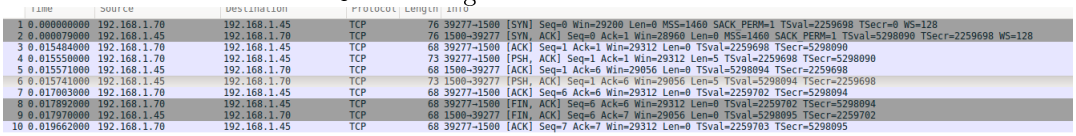


FIGURE 2 – Pour le client :



Ensuite, nous regardons les trames échangées avec wireshark, en utilisant un filtre sur le port. Nous obtenons ceci :

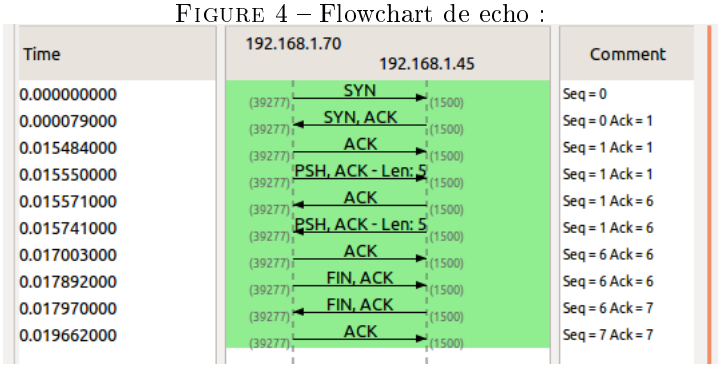
FIGURE 3 – Affichage dans wireshark :



	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	192.168.1.70	192.168.1.45	TCP	76	39277→1500 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=2259698 TSecr=0 WS=128
2	0.000079000	192.168.1.45	192.168.1.70	TCP	76	1500→39277 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=5298090 TSecr=2259698 WS=128
3	0.015484000	192.168.1.70	192.168.1.45	TCP	68	39277→1500 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=2259698 TSecr=5298090
4	0.015550000	192.168.1.70	192.168.1.45	TCP	73	39277→1500 [PSH, ACK] Seq=1 Ack=1 Win=29312 Len=5 TSval=2259698 TSecr=5298090
5	0.015571000	192.168.1.45	192.168.1.70	TCP	68	1500→39277 [ACK] Seq=1 Ack=6 Win=29056 Len=0 TSval=5298094 TSecr=2259698
6	0.015741000	192.168.1.45	192.168.1.70	TCP	73	1500→39277 [PSH, ACK] Seq=1 Ack=6 Win=29056 Len=5 TSval=5298094 TSecr=2259698
7	0.017003000	192.168.1.70	192.168.1.45	TCP	68	39277→1500 [ACK] Seq=6 Ack=6 Win=29312 Len=0 TSval=2259702 TSecr=5298094
8	0.017892000	192.168.1.70	192.168.1.45	TCP	68	39277→1500 [FIN, ACK] Seq=6 Ack=6 Win=29312 Len=0 TSval=2259702 TSecr=5298094
9	0.017970000	192.168.1.45	192.168.1.70	TCP	68	1500→39277 [FIN, ACK] Seq=6 Ack=7 Win=29056 Len=0 TSval=5298095 TSecr=2259702
10	0.019662000	192.168.1.70	192.168.1.45	TCP	68	39277→1500 [ACK] Seq=7 Ack=7 Win=29312 Len=0 TSval=2259703 TSecr=5298095

Que nous pouvons analyser plus facilement en utilisant l'option *flowchart* dans wireshark :

FIGURE 4 – Flowchart de echo :



Les trois premiers paquets correspondent au mécanisme de *handshake*. Ensuite, on voit un paquet de taille 5 aller du client au serveur. Il s'agit du "echo" (4 caractères + le retour chariot).

Ensuite, le serveur envoi l'accusé de réception, avec le numéro d'acquittement 6 (5+1), puis il envoi la sa réponse à la requête, soit le message "echo". le numéro d'acquittement n'as pas changé, puisque le serveur n'as pas reçu d'autre données entre temps.
S'en suit alors le paquet 7 envoyé par le client qui accuse réception du retour effectué par le serveur. Le numéro d'acquittement est devenu 6 (1+6) pour le client (il a reçu 5 caractères).

Le client engage alors le processus de terminaison de la connexion. Il envoie un paquet vide avec le drapeau FIN. Le serveur lui renvoie alors un paquet similaire en incrémentant le numéro d'acquittement de 1. Enfin, un dernier paquet est transmis au serveur par le client en tant qu'accusé de réception du retour du paquet FIN. La connexion est alors terminée.

3 Application client/serveur chifoumi

3.1 Principe du chifoumi

Le but de ce programme est de faire jouer l'utilisateur, par l'application client, contre le serveur. Le client enverras son choix, pierre, feuille, ou ciseaux, au serveur, et le serveur choisiras aléatoirement l'un des 3.
On affiche les scores, et la partie se finie au bout de 3 victoires, après quoi la connexion seras perdue.

3.2 Code source du client

```
1
2  #include <netdb.h>
3  #include <netinet/in.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7  #include <sys/socket.h>
8  #include <sys/types.h>
9  #include <unistd.h>
10
11 #include "common.h"
12 #include "chifoumi.h"
13
14 int main(int argc, char * argv[]) {
15         //variables
16         struct sockaddr_in clientAddress, serverAddress;
17         struct hostent * serverHost;
18         int clientSocket, err;
19         void * line;
20         choice_t * choix_serveur, choix_joueur;
21         score_t * score;
22         win_t * win;
23         end_t * end;
24
25         //arguments du programme
26         if (argc != 2) {
27                 printf("%s <hostname>\n", argv[0]);
28                 exit(EXIT_FAILURE);
29         }
30
31         //resolution de l'hote
32         serverHost = gethostbyname(argv[1]);
33         if (serverHost == NULL) {
34                 printf("Unknown host %s.\n", argv[1]);
35                 exit(EXIT_FAILURE);
36         }
37         printf("Host %s found.\n", argv[1]);
38         serverAddress.sin_family = serverHost->h_addrtype;
39         memcpy(&serverAddress.sin_addr.s_addr, serverHost->
               h_addr_list[0], serverHost->h_length);
40         serverAddress.sin_port = htons(SERVER_PORT);
41
42         //creation du socket
43         clientSocket = socket(AF_INET, SOCK_STREAM, 0);
44         if (clientSocket < 0) {
45                 puts("Cannot open socket");
46                 exit(EXIT_FAILURE);
47         }
```

```
48          puts("Socket openned.");
49
50          //liaison a un port
51          clientAddress.sin_family = AF_INET;
52          clientAddress.sin_addr.s_addr = htonl(INADDR_ANY);
53          clientAddress.sin_port = htons(0);
54          err = bind(clientSocket, (struct sockaddr *) &
                clientAddress, sizeof(clientAddress));
55          if (err < 0) {
56                  puts("Cannot bind port %hu.");
57                  close(clientSocket);
58                  exit(EXIT_FAILURE);
59          }
60          puts("Port bound.");
61
62          //connection au serveur
63          err = connect(clientSocket, (struct sockaddr *) &
                serverAddress, sizeof(serverAddress));
64          if (err < 0) {
65                  printf("Cannot connect to server %s.\n", argv
                        [1]);
66                  close(clientSocket);
67                  exit(EXIT_FAILURE);
68          }
69          printf("Connected to %s.\n", argv[1]);
70
71          //traitement
72          send(clientSocket, "HELLO", 6, 0);
73          line = read_line(clientSocket);
74          if (line == NULL || strcmp(line, "OK") != 0) {
75                  puts("Communication error");
76                  close(clientSocket);
77                  exit(EXIT_FAILURE);
78          }
79          free(line);
80          send(clientSocket, "CHIFOUMI", 9, 0);
81          line = read_line(clientSocket);
82          if (line == NULL || strcmp(line, "OK") != 0) {
83                  puts("Communication error");
84                  close(clientSocket);
85                  exit(EXIT_FAILURE);
86          }
87          free(line);
88          memcpy(choix_joueur.CHOICE, "CHOICE", 6);
89          choix_joueur.SPACE = ' ';
90          choix_joueur.ZERO = '\0';
91          while (1) {
92                  printf("Entrez %hhu pour pierre, %hhu pour
                        feuille et %hhu pour ciseau :", PIERRE,
                        FEUILLE, CISEAU);
93                  scanf("%hhu", &choix_joueur.choice);
94                  while (choix_joueur.choice > 2) {
95                          puts("Invalide.");
96                          printf("Entrez %hhu pour pierre, %hhu
                                pour feuille et %hhu pour ciseau :
                                ", PIERRE, FEUILLE, CISEAU);
97                          scanf("%hhu", &choix_joueur.choice);
98                  }
99                  switch (choix_joueur.choice) {
100                         case PIERRE:
101                                 puts("Vous avez choisi PIERRE.
                                        ");
```

```
102                             break;
103                     case FEUILLE:
104                             puts("Vous avez choisi FEUILLE
                                    .");
105                             break;
106                     case CISEAU:
107                             puts("Vous avez choisi CISEAU.
                                    ");
108                             break;
109             }
110             send(clientSocket, &choix_joueur, sizeof(
                    choice_t), 0);
111             line = read_line(clientSocket);
112             if (line == NULL || !start_with(line, "CHOICE"
                    )) {
113                     puts("Communication error");
114                     close(clientSocket);
115                     exit(EXIT_FAILURE);
116             }
117             choix_serveur = line;
118             switch (choix_serveur->choice) {
119                     case PIERRE:
120                             puts("Le serveur a choisi
                                    PIERRE.");
121                             break;
122                     case FEUILLE:
123                             puts("Le serveur a choisi
                                    FEUILLE.");
124                             break;
125                     case CISEAU:
126                             puts("Le serveur a choisi
                                    CISEAU.");
127                             break;
128             }
129             free(line);
130             send(clientSocket, "OK", 3, 0);
131             line = read_line(clientSocket);
132             if (line == NULL || !start_with(line, "WIN"))
                    {
133                     puts("Communication error");
134                     close(clientSocket);
135                     exit(EXIT_FAILURE);
136             }
137             send(clientSocket, "OK", 3, 0);
138             win = line;
139             switch (win->winner) {
140                     case PLAYER:
141                             puts("Vous l'emportez.");
142                             break;
143                     case SERVER:
144                             puts("Le serveur l'emporte.");
145                             break;
146                     case NONE:
147                             puts("Egalite.");
148                             break;
149             }
150             free(line);
151             line = read_line(clientSocket);
152             if (line == NULL || !start_with(line, "SCORE")
                    ) {
153                     puts("Communication error");
154                     close(clientSocket);
```

```
155                         exit(EXIT_FAILURE);
156                 }
157                 send(clientSocket, "OK", 3, 0);
158                 score = line;
159                 printf("Scores :\t\tVous %hhu\t\tLe serveur %
                        hhu\n", score->player_score, score->
                        server_score);
160                 free(line);
161                 line = read_line(clientSocket);
162                 if (line == NULL || !start_with(line, "END"))
                        {
163                         puts("Communication error");
164                         close(clientSocket);
165                         exit(EXIT_FAILURE);
166                 }
167                 end = line;
168                 send(clientSocket, "OK", 3, 0);
169                 if (end->winner == PLAYER) {
170                         puts("Vous avez gagne la partie.");
171                         free(line);
172                         break;
173                 } else if (end->winner == SERVER) {
174                         puts("Le serveur a gagne la partie.");
175                         free(line);
176                         break;
177                 }
178                 free(line);
179         }
180
181         //fin du programme
182         close(clientSocket);
183         puts("Connection closed.");
184         exit(EXIT_SUCCESS);
185 }
```

3.3 Code source du serveur

```
1
2  #include <arpa/inet.h>
3  #include <netinet/in.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7  #include <sys/socket.h>
8  #include <sys/time.h>
9  #include <sys/types.h>
10 #include <time.h>
11 #include <unistd.h>
12
13 #include "common.h"
14 #include "chifoumi.h"
15
16 #define MAX_CLIENTS 5
17
18 int main() {
19         //variables
20         struct sockaddr_in serverAddress, clientAddress;
21         char * ipClient;//pour l'affichage de l'adresse
22         int serverSocket, clientSocket, err;
23         socklen_t clientAddrLen = sizeof(clientAddress);
```

```
24          void * line;
25          choice_t * choix_joueur, choix_serveur;
26          score_t score;
27          win_t win;
28          end_t end;
29
30          //creation du socket
31          serverSocket = socket(AF_INET, SOCK_STREAM, 0);
32          if (serverSocket < 0) {
33                  puts("Cannot open socket.");
34                  exit(EXIT_FAILURE);
35          }
36          puts("Socket openned.");
37
38          //liaison au port
39          serverAddress.sin_family = AF_INET;
40          serverAddress.sin_addr.s_addr = htonl(INADDR_ANY);
41          serverAddress.sin_port = htons(SERVER_PORT);
42          err = bind(serverSocket, (struct sockaddr *) &
                serverAddress, sizeof(serverAddress));
43          if (err < 0) {
44                  printf("Cannot bind port %hu.\n", SERVER_PORT)
                        ;
45                  close(serverSocket);
46                  exit(EXIT_FAILURE);
47          }
48          printf("Port %hu bound.\n", SERVER_PORT);
49
50          //ecoute du port
51          err = listen(serverSocket, MAX_CLIENTS);
52          if (err < 0) {
53                  printf("Cannot listen port %hu.\n",
                            SERVER_PORT);
54                  close(serverSocket);
55                  exit(EXIT_FAILURE);
56          }
57          printf("Listening port %hu.\n", SERVER_PORT);
58
59          while (1) {
60                  //attente de connection
61                  clientSocket = accept(serverSocket, (struct
                            sockaddr *) &clientAddress, &clientAddrLen)
                            ;
62                  if (clientSocket < 0) {
63                          puts("Cannot accept connection.");
64                          close(serverSocket);
65                          exit(EXIT_FAILURE);
66                  }
67                  ipClient = inet_ntoa(clientAddress.sin_addr);
68                  printf("New connection from %s.\n", ipClient);
69
70                  //traitement
71                  srand(time(NULL));
72                  memcpy(choix_serveur.CHOICE, "CHOICE", 6);
73                  choix_serveur.SPACE = ' ';
74                  choix_serveur.ZERO = '\0';
75                  memcpy(score.SCORE, "SCORE", 5);
76                  score.SPACE = ' ';
77                  score.ZERO = '\0';
78                  score.player_score = 0;
79                  score.server_score = 0;
80                  memcpy(win.WIN, "WIN", 3);
```

```
81                  win.SPACE = ' ';
82                  win.ZERO = '\0';
83                  memcpy(end.END, "END", 3);
84                  end.SPACE = ' ';
85                  end.ZERO = '\0';
86                  line = read_line(clientSocket);
87                  if (line == NULL || strcmp(line, "HELLO") !=
                        0) {
88                      puts("Communication error.");
89                      close(clientSocket);
90                      break;
91                  }
92                  free(line);
93                  send(clientSocket, "OK", 3, 0);
94                  line = read_line(clientSocket);
95                  if (line == NULL || strcmp(line, "CHIFOUMI")
                        != 0) {
96                      puts("Communication error.");
97                      close(clientSocket);
98                      break;
99                  }
100                 free(line);
101                 send(clientSocket, "OK", 3, 0);
102                 while (1) {
103                     line = read_line(clientSocket);
104                     if (line == NULL || !start_with(line,
                            "CHOICE")) {
105                         break;
106                     }
107                     choix_joueur = line;
108                     choix_serveur.choice = rand() % 3;
109                     send(clientSocket, &choix_serveur,
                            sizeof(choice_t), 0);
110                     if (((choix_joueur->choice + 1) % 3)
                            == choix_serveur.choice) {
111                         win.winner = SERVER;
112                         score.server_score++;
113                     } else if (((choix_serveur.choice + 1)
                             % 3) == choix_joueur->choice) {
114                         win.winner = PLAYER;
115                         score.player_score++;
116                     } else {
117                         win.winner = NONE;
118                     }
119                     free(line);
120                     line = read_line(clientSocket);
121                     if (line == NULL || strcmp(line, "OK")
                             != 0) {
122                         break;
123                     }
124                     free(line);
125                     send(clientSocket, &win, sizeof(win_t)
                            , 0);
126                     line = read_line(clientSocket);
127                     if (line == NULL || strcmp(line, "OK")
                             != 0) {
128                         break;
129                     }
130                     free(line);
131                     send(clientSocket, &score, sizeof(
                            score_t), 0);
132                     line = read_line(clientSocket);
```

```
133                         if (line == NULL || strcmp(line, "OK")
                                != 0) {
134                             break;
135                         }
136                         free(line);
137                         if (score.player_score == 3) {
138                             end.winner = PLAYER;
139                         } else if (score.server_score == 3) {
140                             end.winner = SERVER;
141                         } else {
142                             end.winner = NONE;
143                         }
144                         send(clientSocket, &end, sizeof(end_t)
                                , 0);
145                         line = read_line(clientSocket);
146                         if (line == NULL || strcmp(line, "OK")
                                != 0) {
147                             break;
148                         }
149                         free(line);
150                     }
151 
152                     //fermeture de la connection
153                     while (read_line(clientSocket) != NULL);
154                     printf("Connection with %s closed.\n",
                            ipClient);
155             }
156 }
```

3.4 Analyse des trames réseau

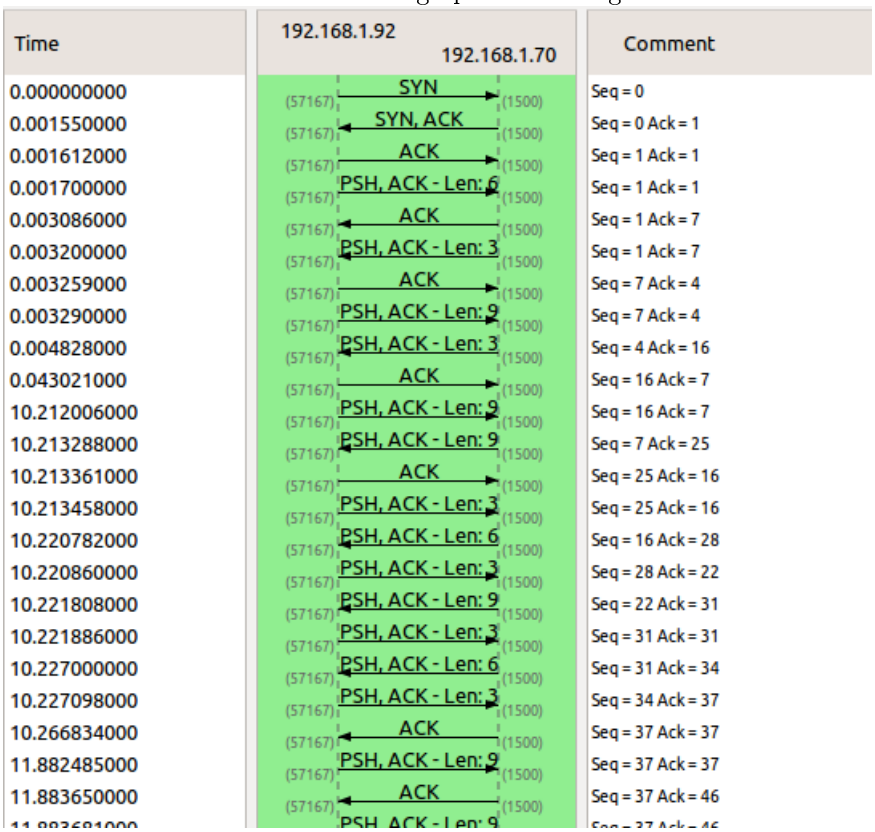
Voici ce qu'afficheras le client (pour le premier tour de jeu) :

FIGURE 5 – client chifoumi :

```
➜  chifoumi git:(master) ✗ ./q11_client 192.168.1.70
Host 192.168.1.70 found.
Socket openned.
Port bound.
Connected to 192.168.1.70.
Entrez 0 pour pierre, 1 pour feuille et 2 pour ciseau : 1
Vous avez choisi FEUILLE.
Le serveur a choisi CISEAU.
Le serveur l'emporte.
Scores :               Vous 0          Le serveur 1
```

On peut regarder ensuite le flowchart de wireshark pour cet échange :

FIGURE 6 – flowgraph de l'echange :



Comme précédemment, les trois premiers échanges correspondent au mécanisme de *handshake*. Ensuite, on constate les échanges de "HELLO", et "CHIFOUMI", toujours suivis d'un message d'"acknowledgement".

4 Échange de données entre deux machines en UDP

4.1 Objectif

Le but de cette partie est d'échanger des chaines de caractères entre deux machine, avec une application client/serveur, et en utilisant le protocole UDP. Comme il est dit dans l'introduction, ce mode nécessiteras l'adresse de destination pour chaque envoi de données. Il n'y aura pas non plus d'accusé de réception.

4.2 Code source du client

```c
/* Exemple de client UDP */

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SUCCESS 0
#define ERROR 1
#define MAX_MSG 100

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

    int sockfd;
    int port;
    int i;
    struct sockaddr_in servaddr;

    //Verification du nombre d'arguments
    if(argc < 4)
    {
        printf("Usage : %s <server> <port> <data1> <data2> ... <dataN>\n", argv[0]);
        exit(ERROR);
    }

    //Creation de la socket
    sockfd=socket(AF_INET,SOCK_DGRAM,0);
    if(sockfd < 0)
    {
        perror("Cannot open socket :");
        exit(ERROR);
    }

    //Initialisation des champs de la structure servaddr
    bzero(&servaddr,sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr=inet_addr(argv[1]);
    port = atoi(argv[2]);
    servaddr.sin_port=htons(port);

    for(i=3; i < argc; i++)
    {
        //Envoi du mot
        sendto(sockfd,argv[i],strlen(argv[i]),0, (struct sockaddr *)&servaddr,sizeof(servaddr));


    }

    exit(SUCCESS);

}
```

4.3 Code source du serveur

```c
/* Exemple de serveur UDP */
```

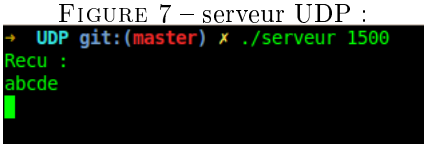
```
2
3  #include <sys/socket.h>
4  #include <netinet/in.h>
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <string.h>
8
9  #define SUCCESS 0
10 #define ERROR 1
11 #define MAX_MSG 100
12
13 int main(int argc, char** argv)
14 {
15
16     int sockfd,n;
17     int port;
18     struct sockaddr_in servaddr,cliaddr;
19     socklen_t len;
20     char mesg[MAX_MSG];
21
22     //Verification du nombre d'arguments
23     if(argc != 2)
24     {
25         printf("Usage : %s <port>\n", argv[0]);
26         exit(ERROR);
27     }
28
29     //Creaction de la socket
30     sockfd=socket(AF_INET,SOCK_DGRAM,0);
31     if(sockfd<0)
32     {
33         perror("Cannot open socket ");
34         exit(ERROR);
35     }
36
37     //Initialisation des champs de la structure servaddr
38     bzero(&servaddr,sizeof(servaddr));
39     servaddr.sin_family = AF_INET;
40     servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
41     port = atoi(argv[1]);
42     servaddr.sin_port=htons(port);
43
44     //Bind
45     if(bind(sockfd,(struct sockaddr *)&servaddr,sizeof(
           servaddr)) < 0)
46     {
47         perror("Can't bind local adresse ");
48         return(ERROR);
49     }
50
51
52
53     for (;;)
54     {
55
56         len = sizeof(cliaddr);
57                 //Reception des mots
58         n = recvfrom(sockfd,mesg,MAX_MSG,0,(struct sockaddr *)
               &cliaddr,&len);
59         mesg[n] = 0;
60         printf("Recu :\n");
61         printf("%s\n",mesg);
```

```
62
63
64          }
65
66          exit(SUCCESS);
67      }
```

4.4 Analyse par Wireshark

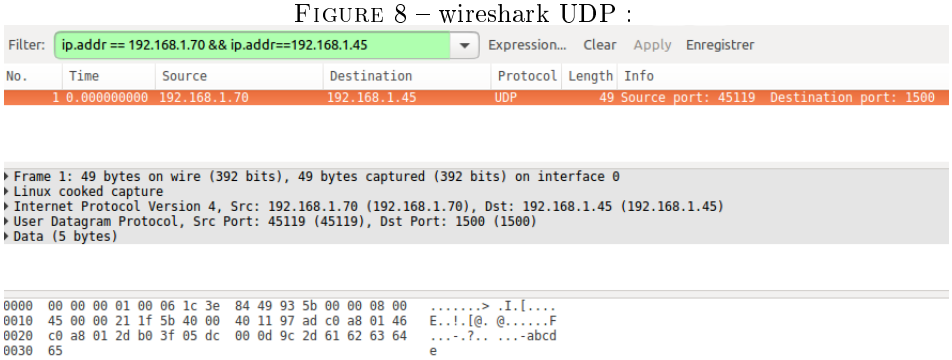
Depuis la machine avec le serveur, on obtient ceci (en exécutant "./client 192.168.1.45 1500 abcde" sur la machine avec le client) :

FIGURE 7 – serveur UDP :



Puis on regarde à l'aide de wireshark les données utiles transmises :

FIGURE 8 – wireshark UDP :



On constate que l'on a bien 5 bytes de données transmis, qui correspondent aux 5 derniers octets transmis : on voit qu'il s'agit du code ASCII hexadécimal des lettres abcde.

5 Conclusion

Lors de ce TP, la principale difficulté que j'ai rencontré à été que j'ai voulu rentrer tout de suite dans le code, sans être parfaitement à l'aise avec la théorie de ces protocoles. Ce qui m'as le plus aidé à été d'écrire sur une feuille le déroulement des échanges (pour le protocole TCP).
Ce TP m'auras servi à comprendre le fonctionnement des sockets, et à savoir utiliser leurs primitives. Le fait de regarder plus précisément les échanges avec Wireshark à également aidé à expliciter les différences entre les protocoles.