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);
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 :

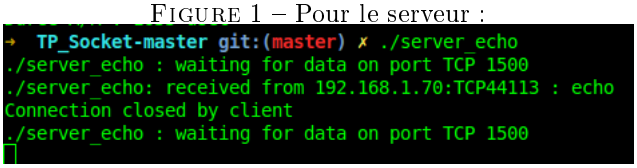
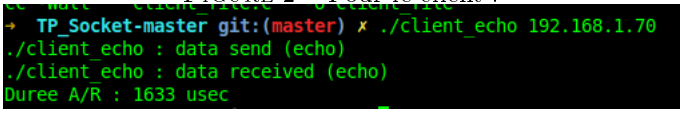
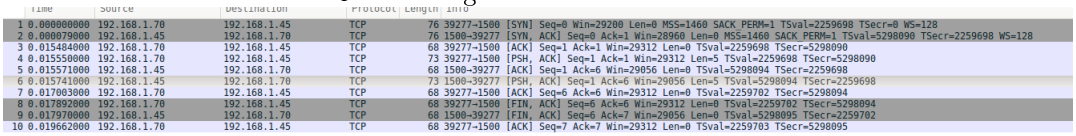


FIGURE 2 – Pour le client :



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

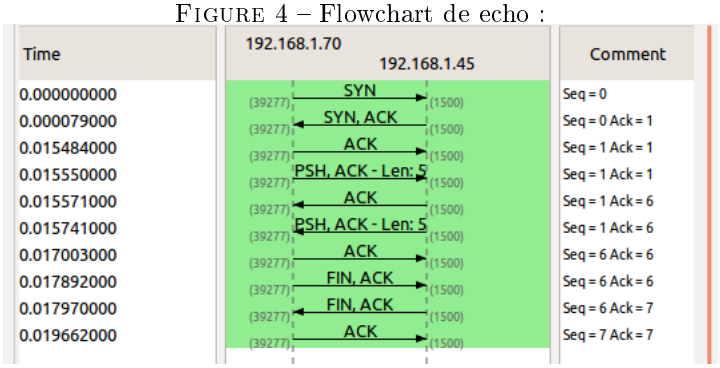
FIGURE 3 – Affichage dans wireshark :



	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.