Rapport de TP
Sockets

JOBE Ophélie
ADAM Baptiste

Sommaire :

JOBE Ophélie
ADAM Baptiste

Introduction

Ce TP a pour objectif de nous faire comprendre comment marche concrètement les sockets. Lorsqu'il s'agit de faire communiquer des machines entre elles, qu'elles soient connectés ou non, les sockets jouent un rôle primordial pour émettre et/ou recevoir des informations.

Nous verrons donc dans ce TP différent types et domaines de sockets ainsi que différents protocoles de communications via différents exercices en mode non connecté et connecté. nous irons même jusqu'à tester ces sockets pour faire communiquer deux machines et un serveur web pour voir comment cela se passe.

JOBE Ophélie
ADAM Baptiste

Communications en mode non connecté

Exercice 1

Le premier exercice consiste à créer une socket entre un programme serveur qui se trouvera sur un premier ordinateur et un programme client qui lui sera lancé sur un autre ordinateur.

Le serveur reçoit en paramètre le port sur lequel il écoutera les demandes du client. Le client reçoit en paramètre le nom de la machine sur lequel est le serveur, le port sur lequel il écoute et le message à lui faire passer.

```
1> ./serveur 12382
```

```
1> ./client pc5201i 12382 bonjour
```

Pour le serveur uniquement, nous commençons par définir une structure `sockaddr_in` qui nous servira a bind la socket. Nous la configurons en AF_INET, sur le port passé en paramètre (`atoi` est utilisé pour transformer la chaîne de caractère en entier, puis `htons` est utilisé pour s'assurer que le stockage se fait avec la bonne *Endianness*), avec l'adresse INADDR_ANY pour être associé avec toutes les adresses de la machine.

Ensuite, pour le serveur et pour le client, nous créons une socket avec les paramètres AF_INET et SOCK_DGRAM puisque nous voulons être en protocole UDP.

- AF_INET définie ici le domaine de la socket. La communication se fera ici sur des systèmes différents connectés par réseau internet mais pour ce premier exercice nous utiliserons le mode non connecté, c'est à dire UDP. Celui-ci ne permet pas de contrôle si il y a des erreurs dut par exemple à l'envoi de message sans savoir s'il a été reçu ou non.
- SOCK_DGRAM indique que le message sera structurés par un datagramme, la connexion entre les deux systèmes se terminera automatique après une communication entre eux.

Pour le serveur, il faut maintenant associer la socket avec la structure `sockaddr_in` pour que celle ci soit connectée au port que nous voulons. Le serveur est maintenant prêt à recevoir des messages du client.

Pour le client, il faut définir une structure `sockaddr_in` qui contiendra les informations du destinataire du message, donc du serveur. Nous le configurons donc avec les paramètres AF_INET et le port passé en paramètre qui a subi les même traitement que pour le serveur. Avec le nom de la machine du serveur passé en paramètre, nous obtenons l'adresse de la machine grâce à la commande `gethostbyname`, ce qui nous permet de remplir le champ correspondant de la structure. Le client est maintenant prêt à envoyer des messages au serveur.

JOBE Ophélie
ADAM Baptiste

serveur :

```c
int main(int argc, char* argv[]){
    if (argc != 2){
        printf("je veux des stats\n");
        return -1;
    }

    int res, soc, nbseq, envoie;
    struct sockaddr_in etiq;
    struct sockaddr_in ad_emet;
    char reception[100], char_pid[10], renvois[100];
    char **tabseq;
    socklen_t taille;
    pid_t pid;

    etiq.sin_family = AF_INET;
    etiq.sin_port = htons(atoi(argv[1]));
    etiq.sin_addr.s_addr = INADDR_ANY;
    soc = socket(AF_INET, SOCK_DGRAM, 0);
    if (soc == -1) {perror("error soc");}
    res = bind(soc, (struct sockaddr *)&etiq, sizeof(etiq));
    if (res == -1) {perror("error bind");}

    taille = sizeof(ad_emet);
    recvfrom(soc, &reception, sizeof(reception), 0, (struct sockaddr *)&ad_emet, &taille);
```

client :

```c
int main(int argc, char* argv[]){
    if (argc != 4){
        printf("je veux des stats\n");
        return -1;
    }

    struct sockaddr_in etiq;
    struct sockaddr_in ad_dest;
    int soc, envoie, nbseq;
    char nomMachine[100], char_pid[10], message[100], reception[100];
    char **tabseq;
    pid_t pid;
    socklen_t taille;

    // etiq.sin_family = AF_INET;
    // etiq.sin_port = htons(6500); //pas obligatoire
    // etiq.sin_addr.s_addr = INADDR_ANY;
     soc = socket(AF_INET, SOCK_DGRAM, 0);
     if (soc == -1) {perror("error soc");}
    // res = bind(soc, (struct sockaddr *)&etiq, sizeof(etiq));
    // if (res == -1) {perror("error bind");}

    ad_dest.sin_port = htons(atoi(argv[2]));
    printf("port dest : %s\n", argv[2]);
    ad_dest.sin_family = AF_INET;

    strcat(strcat(nomMachine, argv[1]), ".lan.esiee.fr");
    printf("nom dest : %s\n", nomMachine);
    struct hostent *lh = gethostbyname(nomMachine);
    if(lh == NULL){
        printf("Machine %s non trouvée\n", argv[1]);
        exit(0);
    }
    memcpy(&ad_dest.sin_addr.s_addr, lh->h_addr_list[0], lh->h_length);
    printf("adresse dest : %x\n", ad_dest.sin_addr.s_addr);
```

JOBE Ophélie
ADAM Baptiste

Le message que nous voulons envoyer contient le pid du processus, nous le concaténons donc avec le message passé en paramètre avec le caractère “|” comme séparateur. La fonction `sendto` envois le message au serveur grace a la structure `sockaddr_in` que nous avons créé préalablement. Néanmoins, cette fonction prends en paramètre une structure `sockaddr` et non `sockaddr_in`, il faut donc la convertir.

Lorsque le serveur reçoit le message grâce à la fonction `recvfrom`, il stocke les informations de l'émetteur dans une structure `sockaddr_in` préalablement créée mais non remplie (structure qu’il faut aussi convertir en `sockaddr` lorsqu’elle est passé en paramètre à la fonction).

Le message actuel est la concaténation du pid et du message proprement dit, il faut donc les séparer. Pour ce faire, nous avons utilisé la fonction `text2tabseq` que nous avons extraite d’un autre cours. Cette fonction permet de séparer une chaine de caractere en plusieurs séquences se terminant par le caractère passé en paramètre. Par exemple, en utilisant “|” comme paramètre :

- la chaîne `"pid|message|autrechose|"` sera séparée en `["pid", "message", "autrechose"]`
- alors que la chaîne `"pid|message|autrechose"` sera séparée en `["pid", "message"]` et `"autrechose"` sera perdu.

Une fois les différents composants du message séparés, le serveur les affiche puis envois une réponse à l’émetteur (le client) sous le même format mais avec son propre pid. Le client fait le même traitement.

serveur :

```
73      taille = sizeof(ad_emet);
74      recvfrom(soc, &reception, sizeof(reception), 0, (struct sockaddr *)&ad_emet, &taille);
75
76      tabseq = text2tabseq(reception, '|', &nbseq);
77
78      printf("reception : %s\n", reception);
79      printf("pid emeteur : %s\n", tabseq[0]);
80      printf("message recu : %s\n", tabseq[1]);
81      printf("-------------------------\n");
82      printf("-------- renvois --------\n");
83
84      pid = getpid();
85      printf("pid : %d\n", pid);
86
87      sprintf(char_pid, "%d|", pid);
88      strcat(strcat(strcat(renvois, char_pid), tabseq[1]), "|");
89      printf("message a renvoyer : %s\n", renvois);
90
91      envoie = sendto(soc, renvois, strlen(renvois), 0, (struct sockaddr *)&ad_emet, sizeof(ad_emet));
92      printf("renvoie : %d\n", envoie);
93
```

JOBE Ophélie
ADAM Baptiste

client :

```
    pid = getpid();
    printf("pid : %d\n", pid);

    sprintf(char_pid, "%d|", pid);
    strcat(strcat(strcat(message, char_pid), argv[3]), "|");
    printf("\nmessage a envoyer : %s\n", message);

    envoie = sendto(soc, message, strlen(message), 0, (struct sockaddr *)&ad_dest, sizeof(ad_dest));
    printf("envoie : %d\n", envoie);

    printf("-------------------------\n");
    printf("-------- réponse --------\n");

    taille = sizeof(ad_dest);
    recvfrom(soc, &reception, sizeof(reception), 0, (struct sockaddr *)&ad_dest, &taille);

    tabseq = text2tabseq(reception, '|', &nbseq);

    printf("reception : %s\n", reception);
    printf("pid emeteur : %s\n", tabseq[0]);
    printf("message recu : %s\n", tabseq[1]);
}
```

Affichage serveur :

```
reception : 2338|bonjour|
pid emeteur : 2338
message recu : bonjour
-------------------------
-------- renvois --------
pid : 2327
message a renvoyer : 2327|bonjour|
renvoie : 13
```

Affichage client :

```
port dest : 12382
nom dest : pc5209i.lan.esiee.fr
adresse dest : 9ebad793
pid : 2338
message a envoyer : 2338|bonjour|
envoie : 13
-------------------------
-------- réponse --------
reception : 2327|bonjour|
pid emeteur : 2327
message recu : bonjour
```

JOBE Ophélie
ADAM Baptiste

Communications en mode connecté

Exercice 2

Maintenant, nous voulons communiquer en mode connecté. Le processus est assez similaire à l'exercice 1, nous créons une structure `sockaddr_in` que nous remplissons de la même façon. Puis nous créons un socket avec les paramètres AF_INET et SOCK_STREAM puisque cette fois ci, nous voulons être en protocole TCP pour être en mode connecté. Enfin nous associons la socket avec le port avec un `bind`.

```
etiq.sin_addr.s_addr = INADDR_ANY;
soc = socket(AF_INET, SOCK_STREAM, 0); //
if (soc == -1) {perror("error soc");}
```

Il faut maintenant configurer la connection au client. Nous utilisons la fonction `listen` qui prends en paramètre une socket et le nombre maximum de connection que nous autorisons.

Avec la fonction `accept`, nous acceptons la demande du client et créons une socket qui lui est liée. C'est cette socket qui nous permettra de communiquer avec lui. A l'instar de `recvfrom`, `accept` stocke les informations du client dans une structure `sockaddr_in` préalablement créée mais non remplie.

Nous lisons le message envoyé par le client avec la fonction `read`. Il est possible de lire un message très long en plusieurs fois avec une boucle.

```
89   // listen/accept - connection du client
90   listener = listen(soc,BACKLOG);
91   if (listener == -1) {perror("error listen");}
92
93   printf("listen initialisé\n"); fflush(0);
94
95   taille = sizeof(ad_emet);
96   client = accept(soc, (struct sockaddr *)&ad_emet, &taille);
97
98   printf("client %d\n", client); fflush(0);
99
100  if (client == -1) {perror("error accept");}
101  //printf("client %d connecté\n",ad_emet.sin_addr.s_addr); fflush(0);
102
103  read_s = read(client, reception, sizeof(reception)-1);
104  reception[read_s] = '\0';
105  printf("%s\n", reception);
106  //recv_s = recv(client, reception, sizeof(reception)-1, MSG_PEEK);
107
108  // while(recv_s != 0){
109  //  //recv_s = recv(client, reception, sizeof(reception), MSG_PEEK);
110  //  read_s = read(client, reception, recv_s);
111  //  reception[read_s] = '\0';
112  //  printf("%s", reception);
113  //  recv_s = recv(client, reception, sizeof(reception)-1, MSG_PEEK);
114  //  //printf("- recv_s : %d\n", recv_s);
115  // }
```

JOBE Ophélie
ADAM Baptiste

Nous pouvons maintenant lire un message en mode connecté, mais notre but dans cet exercice est de comprendre la syntaxe d'une vraie requête http. Pour cela, nous nous connectons à notre serveur via un navigateur, cela envoie donc une (vraie) requête que nous pouvons afficher.

Les requêtes sont de ce format :

```
GET /index.html HTTP/1.1
Host: 127.0.0.1:12382
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101
Firefox/68.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: fr,fr-FR;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
\n\n
```

En utilisant `text2tabseq` avec comme caractère de fin " " (espace), nous pouvons isoler `/index.html` qui correspond au fichier à afficher. il ne nous reste plus qu'à ouvrir le fichier si il existe, récupérer son contenu et l'envoyer en réponse au client. Pour se faire, il nous faut maintenant comprendre la syntaxe d'une réponse d'un serveur, c'est ce que nous allons faire dans l'exercice 3.

```
pc5209i-l:~/homedir/E4/INF 4201C - Distributed programing/TP1/Exo 2 et 3> ./serv
eur 12382
listen initialisé
client 4
GET /index.html HTTP/1.1
Host: 127.0.0.1:12382
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: fr,fr-FR;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1


tabseq : /index.html
filename : ./index.html
message envoyé
```

JOBE Ophélie
ADAM Baptiste

Exercice 3

Pour comprendre à quoi ressemble une vrai réponse d'un serveur, il faut ce connecter a un vrai serveur en utilisant notre client.

Pour créer notre client en mode connecté, les étapes sont les même que dans l'exercice 1, sauf que la socket doit être créée avec les paramètres AF_INET et SOCK_STREAM pour être en mode TCP.

```
// creation socket
soc = socket(AF_INET, SOCK_STREAM, 0);
if (soc == -1) {perror("error soc");}
```

Une fois cela fait, il faut se connecter au serveur avec la fonction `connect` qui prend en paramètres la socket et la structure remplie avec les informations du serveur.

Grâce au informations de l'exercice 2, nous pouvons créer une requête correcte et l'envoyer à un site. Nous avons utilisé **dest5.lan.esiee.fr** qui est un serveur de l'intranet de l'ESIEE qui renvois une simple page http. Une fois la requête envoyée avec la fonction `write`, qui prend en paramètre une socket (connectée grâce au `connect`) et le message a envoyer, nous pouvons lire la réponse du serveur avec un `read` et l'afficher.

```
85  // connection au serveur
86  connection = connect(soc, (struct sockaddr *)&ad_dest, sizeof(ad_dest));
87  if(connection == -1){perror("error connect");}
88
89  // creation du message
90  mess_debut = "GET /";
91  mess_fin =" HTTP/1.1\nHost: 127.0.0.1:12382\nUser-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) 
92  message = malloc(sizeof(char)*(strlen(mess_debut)+strlen(mess_fin)+strlen(argv[3])+1));
93  strcat(strcat(memcpy(message, mess_debut, strlen(mess_debut)), argv[3]), mess_fin);
94
95  // envois du mesage
96  write_s = write(soc, message, strlen(message));
97  printf("message envoyé\n");
98
```

Une réponse d'un serveur est de ce format :

```
HTTP/1.1 200 OK
Date: Fri, 06 Dec 2019 13:44:44 GMT
Server: Apache/2.2.22 (Mandriva Linux/PREFORK-0.1mdv2010.2)
Last-Modified: Wed, 20 Feb 2013 13:32:32 GMT
ETag: '26876-130-4d627fd88cb70
Accept-Ranges: bytes
Content-Length: 304
      Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html

<html>
<body>
<!-- $Id: index.html 92365 2007-09-23 14:04:27Z oden $ -->
<!-- $HeadURL:
http://svn.mandriva.com/svn/packages/cooker/apache-conf/current/SOURCES/ind
ex.html $ -->
<h1>DIST5.esiee.fr</h1>
<h1>   Dpt info     </h1>
<h1>contact C.DIETRICH</h1>
<h1>phone : +33 145 926 654</h1>
</body>
</html>
```

La dernière partie (en italique) correspond à la page internet à afficher, c'est ce qu'il faudra changer en allant chercher le bon fichier basé sur requête du client.

JOBE Ophélie
ADAM Baptiste

Exercice 4

Maintenant que nous savons à quoi ressemble une réponse d'un serveur, nous allons pouvoir répondre à notre client dans notre propre serveur. Dans l'exercice 2, nous avions déjà isolé le fichier à afficher. Nous pouvons maintenant le lire ligne par ligne avec la fonction `getline` et stocker le tout dans un tableau de `char`, puisque nous utilisons la même variable a chaque itération, ce tableau grandit de taille a chaque fois, il faut donc lui ré-allouer assez de mémoire a chaque fois.

Une fois que le fichier est lu en entier, nous pouvons concaténer le résultat avec la première partie de la réponse comme vu dans l'exercice 3.

```
136
137         /*****************************************/
138         // construction de la réponse
139         tabseq = text2tabseq(reception, ' ', &nbseq); // tabseq[1] = fichier voulu
140
141             // aller chercher le fichier correspondant
142         filename = malloc(sizeof(char)*(strlen(tabseq[1])+strlen(".")));
143         if(strcmp(tabseq[1], "/") == 0)
144             strcat(memcpy(filename, ".", strlen(".")), "/index.html");
145         else
146             strcat(memcpy(filename, ".", strlen(".")), tabseq[1]);
147
148         printf("CLIENT - filename :%s\n", filename);
149         file = fopen(filename, "r");
150         if(file){
151             read_l = getline(&line, &len, file);
152             rep_fin = malloc(sizeof(char)*strlen(line));
153             memcpy(rep_fin, line, strlen(line));
154             while ((read_l = getline(&line, &len, file)) != -1) {
155                 alloc_tab(rep_fin, strlen(rep_fin)+strlen(line));
156                 strcat(rep_fin, line);
157             }
158         }
159         else
160             perror("CLIENT - fopen");
161         fclose(file);
162
163             // constrcution du String reponse
164         reponse = malloc(sizeof(char)*(strlen(rep_debut)+strlen(rep_fin)));
165         strcat(memcpy(reponse, rep_debut, strlen(rep_debut)), rep_fin);
166
167         // envois du mesage
168         write_s = write(client, reponse, strlen(reponse));
169         if(write_s == -1) perror("CLIENT - write");
170         else printf("CLIENT - réponse envoyé\n");
171
```

Nous voulons aussi qu'on client puisse se connecter sur un autre port pour voir les logs du serveur (les clients qui se sont connecté, a quelle date et quel fichier il ont demandé). Pour ce faire, nous avons utilisé `fork` pour "dédoubler" le processus. Le processus père gérera les requêtes classique et le processus fils gérera les logs. L'initialisation est exactement la même pour les deux, il faut juste utiliser le bon port au bon endroit.

Processus père :

```
91      ppid = fork();
92      if(ppid == 0){
93          /*****************************************/
94          /* ecoute sur le premier port (argv[1]) */
95          /* renvois le fichier demandé           */
96          /*--------------------------------------*/
97          /*             PORT CLIENT              */
```

Quelque soit la requête du client, le processus fils se contente d'ouvrir le fichier **log_file.txt** et d'en extraire les lignes. Comme ce n'est pas un fichier html, il faut le mettre en forme pour que l'affichage dans un navigateur se fasse correctement (une ligne = un log), donc lors de la lecture du fichier, il faut penser à insérer les balises html nécessaires. Une fois que c'est fait, il suffit d'envoyer la réponse de la même manière que précédemment expliqué.

```
226         /*************************************/
227         // construction de la reponse (lecture de log_file.txt)
228         file = fopen("./log_file.txt", "r");
229         if(file){
230             read_l = getline(&line, &len, file);
231             rep_fin = malloc(sizeof(char)*(strlen(line))+strlen("<html>\n<body>\n<p>"));
232             strcat(memcpy(rep_fin, "<html>\n<body>\n<p>", strlen("<html>\n<body>\n<p>")), line);
233             while ((read_l = getline(&line, &len, file)) != -1) {
234                 alloc_tab(rep_fin, strlen(rep_fin)+strlen(line)+strlen("</p>\n<p>"));
235                 strcat(strcat(rep_fin, "</p>\n<p>"), line);
236             }
237             alloc_tab(rep_fin, strlen(rep_fin)+strlen("</p>\n</body>\n</html>"));
238             strcat(rep_fin, "</p>\n</body>\n</html>");
239         }
240         else
241             perror("LOG - fopen");
242         fclose(file);
243
244         reponse = malloc(sizeof(char)*(strlen(rep_debut)+strlen(rep_fin)));
245         strcat(memcpy(reponse, rep_debut, strlen(rep_debut)), rep_fin);
246
247         printf("%s\n", reponse);
248
249         // envois du mesage
250         write_s = write(client, reponse, strlen(reponse));
251         if(write_s == -1)perror("LOG - write");
252         else printf("LOG - log_file envoyé\n");
```

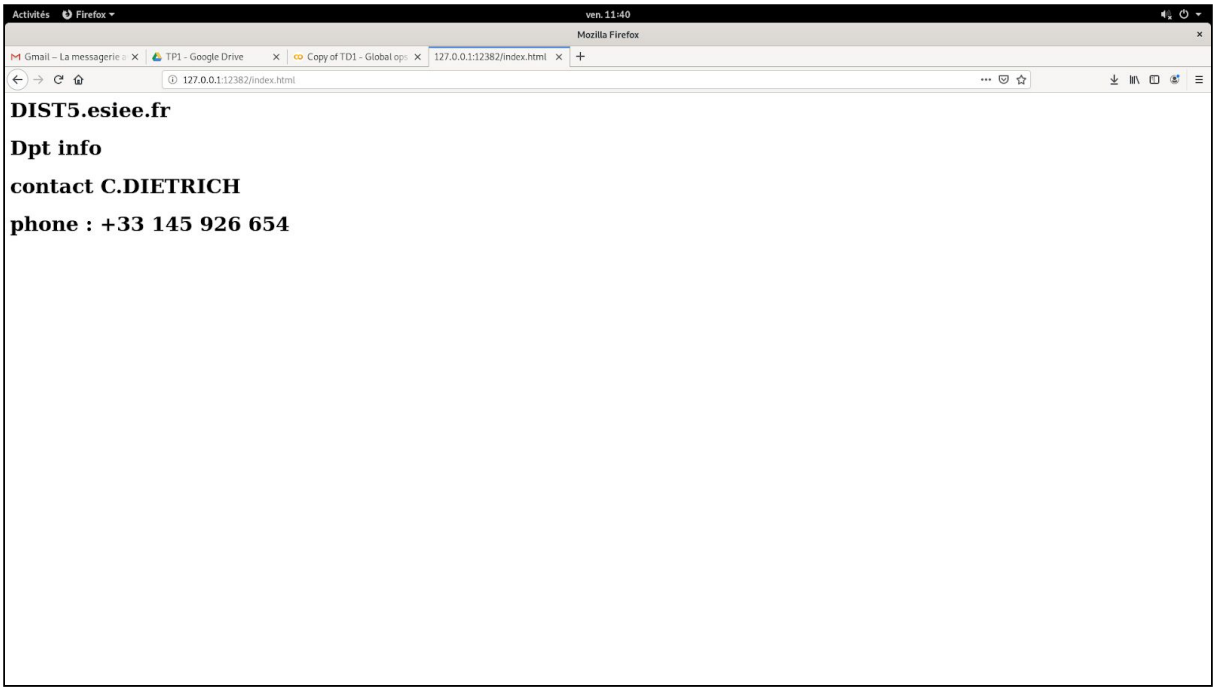
Nous pouvons lire les logs du serveur, mais pour l'instant, aucun logs ne sont enregistré. C'est le processus père que doit se charger de ca. Une fois que la connection avec un client a été établie, il faut récupérer la date courante grâce à la fonction `time`. Cette fonction nous retourne un timestamp, un format pas très parlant pour nous simple humains. Avec la fonction `ctime`, nous obtenons une chaîne de caractères avec la date dans un format plus classique que nous pourrons interpréter. Il nous suffit alors d'écrire le tout (l'adresse, le fichier demandé et la date) à la fin du fichier **log_file.txt** en l'ouvrant en mode `append` (`"a"`).

```
172         /*************************************/
173         // enregistrement de ce log dans log_file.txt
174         timestamp = time(NULL);
175
176         file = fopen("./log_file.txt", "a");
177         if(file){
178             fprintf(file, "%u, %s, %s", ad_emet.sin_addr.s_addr, filename, ctime(&timestamp));
179         }
180         else
181             perror("CLIENT - fopen (log)");
182         fclose(file);
```

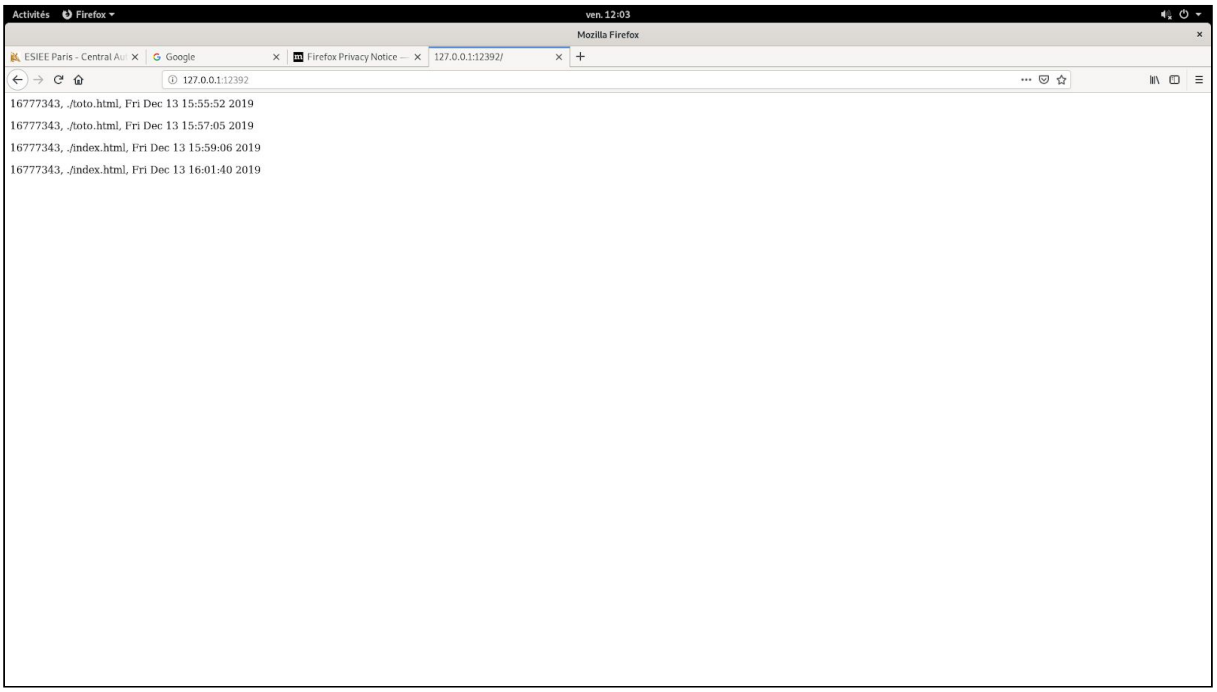
JOBE Ophélie
ADAM Baptiste

Affichage client :

(la page index.html affiché est la même que celle récupérée sur ***dest5.lan.esiee.fr***)



Affichage log :



Conclusion

Pour conclure, nous pouvons donc dire que le choix des sockets se fait selon nos besoins en transmission d'informations. En effet, le mode TCP permet d'envoyer des messages plus long qui pourront être lu en plusieurs fois, ce qui n'est pas le cas du mode UDP. L'avantage du mode connecté est aussi qu'il peut traiter plusieurs clients en même temps.

De plus, nous avons découvert comment créer une requête http et sa réponse, ce qui est la base de la communication entre un serveur et ses clients.