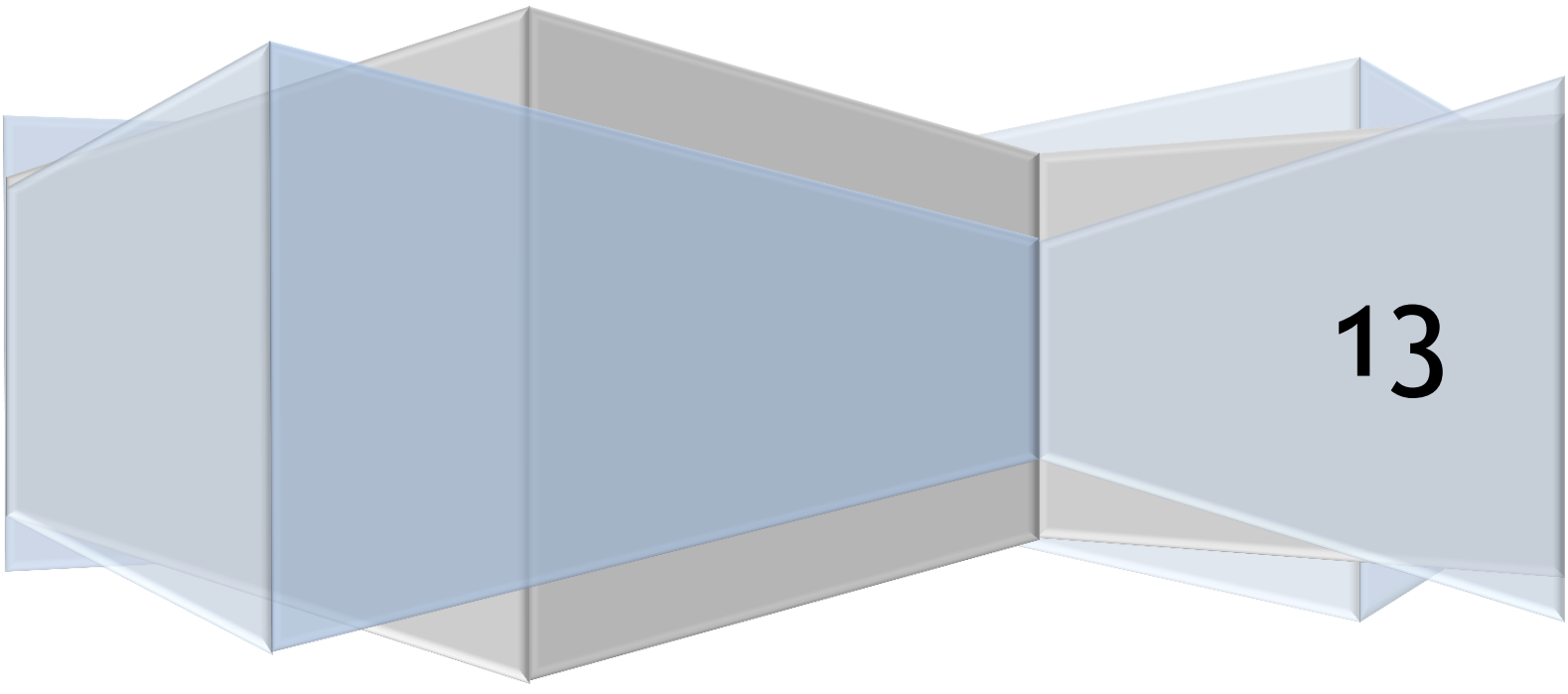
ESIEE Paris

INF 4201B
Informatique Distribuée

Rapport de TP : Sockets en UDP / TCP

Sébastien LE STUM



CONTENU

INTRODUCTION

L'utilisation des « sockets » comme moyen de communication entre systèmes est une méthode très pratique et très utilisée dans le monde de l'informatique.

Dans le cadre de l'unité, nous nous sommes focalisé principalement sur l'utilisation des sockets sur des systèmes à noyau UNIX. Les sockets sur Windows ont, quant à eux, des méthodes d'initialisation différentes et n'utilisent pas exactement les mêmes structures, bien que le fonctionnement global soit très similaire avec son équivalent UNIX.

Sur un système UNIX, les sockets ont deux modes de fonctionnement :

- AF_UNIX : Ce mode permet la communication entre deux processus d'un même système en utilisant un fichier tampon dans le système de fichier de la machine hôte. Il s'agit dans l'idée d'un fonctionnement similaire à un tube nommé.

- AF_INET : Ce mode permet quant à lui la communication entre deux processus de systèmes différents à travers une connexion Ethernet / internet.

Dans ce rapport, nous allons constamment nous trouver dans le mode AF_INET, qui nous permettra de contacter des machines distantes comme le proxy de l'ESIEE ou même un site extérieur.

Ensuite, l'ensemble de ce rapport concernera l'utilisation des sockets en mode UDP puis en mode TCP.

Le mode UDP représente une communication par blocs, sans aucune connexion persistante entre le client et le serveur. Cela implique qu'aucun contrôle n'est appliqué sur le flux transmis. En échange, ce protocole permet le transport de plus de données et de manière plus rapide.

Inversement, le mode TCP est un mode connecté : En effet, ce mode contient des structures et des éléments de contrôles (CRC, code d'erreur) afin de vérifier l'intégrité du message et de maintenir un lien temporaire entre le client et la machine distante.

Ainsi pour ce rapport, nous allons décrire explicitement chaque cas rencontré en indiquant le code source utilisé, appuyé par des traces d'exécution et conclure quant au fonctionnement des sockets et leurs utilisations pratiques dans un contexte de communication client / serveur.

COMMUNICATION EN MODE UDP

EXERCICE 1

Pour cet exercice simple, nous devions réaliser une communication UDP entre un client et un serveur. Le client doit envoyer son PID au serveur et un message passé en paramètre dans la ligne de commande. Dans un second temps, le serveur reçoit ces informations puis en envoie au client (PID du serveur + le message). Cette communication a pour but d'attester du fonctionnement des sockets en mode UDP et de le mettre en pratique.

Pour exécuter nos deux programmes il suffit :

- Pour le client : Fournir l'adresse du serveur (`localhost` ou `127.0.0.1`), le port utilisé et enfin le message que l'on souhaite envoyer.
- Pour le serveur : Fournir uniquement le port sur lequel le serveur sera en écoute, prêt à recevoir les informations du client.

Pour pouvoir initier la communication entre les deux programmes, il faut tout d'abord initialiser les sockets.

Code « client » et « serveur » :

```
struct sockaddr_in localAdd;
struct sockaddr_in serverAdd;
struct hostent* h_s;
int cSocket = socket(AF_INET, SOCK_DGRAM, 0);
if(cSocket == -1){
      perror("Erreur lors de la création de la socket");
      return 0;
}

localAdd.sin_family = AF_INET;
localAdd.sin_port = htons(5600);
localAdd.sin_addr.s_addr = INADDR_ANY;
memset(localAdd.sin_zero, 0, 8);
result = bind(cSocket, (struct sockaddr*) &localAdd,
sizeof(localAdd));
```

Le code ci-dessus est commun au serveur comme au client. En effet, il permet d'initialiser le socket et de le « *bind* », ce qui sous-entend de l'attacher au système pour le rendre prêt à être utilisé lors d'une communication.

Les deux structures contiennent les informations du socket « local » et du socket distant que l'on souhaite joindre.

L'argument `SOCK_DGRAM` permet de passer le socket en mode Datagramme c'est-à-dire en mode UDP. Il est également créé pour fonctionner en mode réseau grâce à l'argument `AF_INET`.

Une fois que le socket est créé et attaché, nous initialisons la structure contenant les informations du socket distant et ainsi commencer l'échange d'informations.

Code « Client » :

```
/*Envoi des données au serveur*/
fprintf(stdout,"Sent :\n");
sendto(cSocket, (void *) message1, sizeof(message1), 0, (struct
sockaddr*) &serverAdd, sizeof(serverAdd));
sendto(cSocket, (void *) pid, sizeof(pid), 0, (struct sockaddr*)
&serverAdd, sizeof(serverAdd));
      fprintf(stdout,"\tMessage : %s\n",argv[3]);
fprintf(stdout,"\tClient PID : %s\n",pid);

/*Reception de la réponse*/
fprintf(stdout,"Recieved :\n");
recvfrom(cSocket, message1, MESSAGE_SIZE, 0,(struct sockaddr*)
&serverAdd, &size);
recvfrom(cSocket, message2, MESSAGE_SIZE, 0,(struct sockaddr*)
&serverAdd, &size);
printf("\tMessage : %s\n\tServer PID : %s\n", message1, message2);
```

Code « Serveur » :

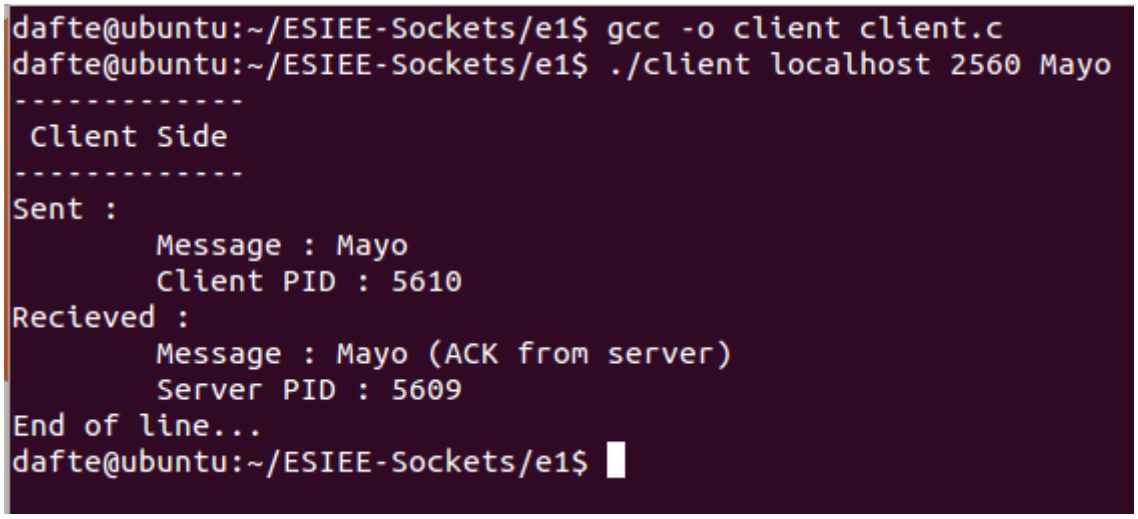
```
/*Reception des données du client*/
      fprintf(stdout,"Recieved :\n");
      recvfrom(sSocket, message1, MESSAGE_SIZE, 0,(struct sockaddr*)
&clientAdd, &taille);
      recvfrom(sSocket, message2, MESSAGE_SIZE, 0,(struct sockaddr*)
&clientAdd, &taille);
      fprintf(stdout,"\tMessage : %s\n\tClient PID : %s\n", message1,
message2);

sprintf(pid,"%d",getpid());

/*Envoi des informations au client*/
      fprintf(stdout,"Sent :\n");
      sprintf(message1,"%s%s",message1," (ACK from server)");
      sendto(sSocket, (void *) message1, sizeof(message1), 0, (struct
sockaddr*) &clientAdd, sizeof(clientAdd));
      sendto(sSocket, (void *) pid, sizeof(pid), 0, (struct
sockaddr*) &clientAdd, sizeof(clientAdd));
      fprintf(stdout,"\tMessage : %s\n",message1);
      fprintf(stdout,"\tServer PID : %s\n",pid);
```

Du point de vue du client, nous envoyons donc le PID obtenu par la fonction `getpid()` et le message écrit sur la ligne de commande qui a été mis en buffer. Une fois que l'envoi a été effectué, le client se place en écoute sur le port de communication et attend les informations du serveur.

Du point de vue serveur, l'inverse se passe. Le serveur est tout d'abord en attente de recevoir les informations. Ces informations, une fois reçues, sont transférées dans des buffers. Pour prouver que l'échange du message à eu lieu sans aucune altération, nous avons ajouté dans la réponse serveur la chaine de caractère : « ACK from server ». De ce

fait, nous pouvons confirmer que les informations envoyées l'ont été sans erreurs et que la réponse reçue par le client est bien celle provenant du serveur.

Ci-dessous se trouve une trace de l'exécution de ces deux programmes, dont le code source complet se trouve en annexe de ce rapport.



Aperçu de l'exécution du programme « Client »

Aperçu de l'exécution du programme « Serveur »

COMMUNICATION EN MODE TCP

Durant les exercices suivants, nous initialiserons les sockets utilisés en mode TCP, c'est-à-dire en utilisant l'argument `SOCK_STREAM` lors de la création du socket.

EXERCICE 2

Le principe de l'exercice 2 est d'analysé la requête envoyé par un navigateur. En effet, lors d'une communication entre un navigateur Web et un serveur, le navigateur envoie une requête HTTP. Cette requête contient les informations utiles demandées par l'utilisateur pour accéder à une page stockée sur le serveur.

Si le serveur dispose de cette page dans le bon répertoire, alors il répond en envoyant directement les informations au navigateur sous forme de code HTML. Sinon une réponse automatisée (ou non) est envoyée au navigateur pour signaler l'erreur rencontrée (comme l'erreur 404 signifiant : `Page Not Found`)

Grâce à notre embryon de serveur, nous pourrons ainsi étudier comment est structurée une requête HTTP.

Une requête de page HTTP ressemble à quelque chose prêt à ceci :

```
GET www.esiee.fr HTTP/1.1
Host: www.esiee.fr:80
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

Pour lancer ce programme nous fournissons en paramètre uniquement le port sur lequel le serveur devra se placer en écoute.

Code « Serveur » :

```
if(listen(lSocket,1) == -1){
      perror("Erreur lors du listen !\n");
      return -1;
}

lService = accept(lSocket,(struct sockaddr*) &clientAdd, &taille);
if(lService==-1){
      perror("Erreur lors du accept !\n");
      return -1;
}


read(lService, (void*) requete, sizeof(requete));
fprintf(stdout,"Request from browser  : %s", requete);
write(lService, (void*) reponse, sizeof(reponse));
```

```
fprintf(stdout,"Answer from server : %s \n",reponse);
close(lService);
close(lSocket);
```

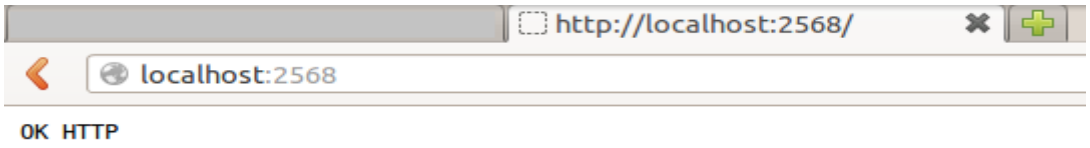
Pour placer le serveur en écoute, nous utilisons la fonction `listen()`. Une fois qu'un client s'est connecté au port d'écoute, nous utilisons la fonction `accept()` pour signifier que le serveur accepte la communication. A ce moment, le système effectue un `fork()` du programme pour pouvoir exécuter la requête indépendamment du programme principal.

Une fois fais, le programme lit la requête via `read()`, puis envoie une réponse au navigateur pour signifier que tout a fonctionné.

Ci-dessous se trouve une trace de l'exécution du programme et la réponse simple envoyée au navigateur utilisé dans cet exemple (Firefox). Le code source complet du programme se trouve en annexe de ce rapport.

```
dafte@ubuntu:~/ESIEE-Sockets/e2$ ./serveur 2568
La requete du navigateur est  :
---------
 GET / HTTP/1.1
Host: localhost:2568
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:26.0) Gecko/20100
101 Firefox/26.0
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


---------
End of line...
dafte@ubuntu:~/ESIEE-Sockets/e2$
```

Aperçu de l'exécution du serveur



Aperçu de la réponse du serveur au navigateur

EXERCICE 3

Pour cet exercice, nous devions réaliser un embryon de navigateur. Pour se faire, nous disposions du squelette de la requête HTTP que nous avions récupéré précédemment. Il suffit ensuite, lors de l'exécution du programme de fournir la page que l'on souhaite atteindre puis le port à utiliser (port 80 pour un accès Internet).

Une fois fait, le client doit se connecter au port que l'on indique et demander grâce à notre requête HTTP la page souhaitée.

Code « client » :

```
      if(connect(lSocket, (struct sockaddr*) &serverAdd, sizeof(serverAdd))
== -1){
            perror("Echec lors de la connexion !");
            return -1;
      }

      sprintf(requete,"%s%s%s%s%s%s%s","GET /",argv[3]," HTTP/1.1\nHost:
",argv[1],":",argv[2],"\nUser-Agent: Mozilla/5.0 (X11; Linux x86_64;
rv:17.0) Gecko/20131029 Firefox/17.0\nAccept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\nAccept-
Language: en-US,en;q=0.5\nAccept-Encoding: gzip, deflate\nConnection:
Close\n\n");

      send(lSocket, (void*) requete, sizeof(requete),0);

      byteCount = recv(lSocket, (void*) reponse, sizeof(reponse),0);
      reponse[byteCount-1] = '\0';
      printf("Reponse du serveur : \n%s\n",reponse);
      close(lSocket);
```

Nous nous connectons dont au site voulu via notre socket. Ensuite nous écrivons notre requête en fonction des arguments fournis lors de l'exécution du programme. Enfin les données sont envoyées et le client attend la réponse du serveur à sa demande.

Pour éviter tout erreur de lecture du résultat nous plaçons un ‘\0’ à la fin des données reçues en réponse, ce symbole représentant la fin d’une chaine de caractère.

Ci-dessous se trouve la trace d’exécution de ce programme. Veuillez noter que le code source complet est disponible en annexe.

```
dafte@ubuntu:/host/Documents/GitHub/ESIEE-Sockets/e3$ gcc -o client client.c
dafte@ubuntu:/host/Documents/GitHub/ESIEE-Sockets/e3$ ./client www.esiee.fr 80 i
ndex.html
Reponse du serveur :
HTTP/1.1 302 Found
Date: Fri, 20 Dec 2013 08:12:38 GMT
Server: Apache/2.2.3 (Debian) mod_python/3.2.10 Python/2.4.4 mod_ssl/2.2.3 OpenS
SL/0.9.8c mod_perl/2.0.2 Perl/v5.8.8
Location: http://www.esiee.fr/index.php
Content-Length: 213
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>302 Found</title>
</head><body>
<h1>Found</h1>
<p>The document has moved <a href="http://www.esiee.fr/index.php">here</a>.</p>
</body></html>

dafte@ubuntu:/host/Documents/GitHub/ESIEE-Sockets/e3$
```

Aperçu de l’exécution de l’exercice 3

EXERCICE 4

Pour cet exercice, nous devons réaliser une base de serveur web. Ce serveur doit écouter sur deux ports afin de remplir deux fonctions :

- Si un client se connecte sur le port que l'on va nommer port HTTP, le serveur devra répondre en recherchant la page demandée par le client et le cas échéant, fournir soit la page en réponse, soit un retour erreur (404 par exemple)
- Si un client se connecte sur le port que l'on va nommer port LOG, le serveur devra quant à lui envoyer le log de connexion.

En effet, dans un second temps, le serveur doit journaliser l'ensemble des connexions établies sur le serveur dans un fichier `log_server.txt`.

L'écoute sur un ensemble de ports se fait via la fonction `select()`. Cette fonction spécifique permet d'observer l'activité d'un groupe de descripteurs de fichiers. Plus précisément, en ajoutant les descripteurs correspondant aux sockets que nous voulons utiliser dans une structure particulière, la fonction `select()` va être capable de détecter quel descripteur a été actif.

L'intérêt principal de cette méthode est que `select()` est une fonction non-bloquante. Tandis que `recvfrom()` bloque le programme tant qu'il n'a pas reçu de connexion, le fonctionnement de `select()` permet de signaler une activité via sa valeur de retour

- Si `select()` retourne -1, une erreur est survenue.
- Si `select()` retourne 0, aucun descripteur n'a été actif
- Si `select()` retourne une valeur positive n, alors il y a n descripteur qui ont été actifs qu'il va falloir traiter au cas par cas.

Pour signifier au navigateur que sa requête à aboutie nous envoyons l'en-tête HTTP correspondante : HTTP/1.0 200 OK. Nous gérons 3 cas distinct :

- Si la requête est incorrecte, alors le serveur stockera dans le log cette tentative infructueuse et enverra au client « ERREUR http »
- Si la requête est correcte, mais que la page n'est pas trouvée par le serveur, la tentative est loguée et la page « `404 not found` » est envoyée au client
- Si la requête est correcte et que la page est trouvée, alors nous affichons la page.

Pour ce qui est de la journalisation, il s'agit là d'une manipulation de fichiers en mode lecture lors de l'accès et en mode ajout lors de la journalisation serveur.

Le code étant d'une taille conséquente, il se situe dans son intégralité en annexe de ce rapport.

Ci-dessous se trouvent les traces d'exécution du serveur et les résultats obtenus :

serveur.c × client.c × log_server.txt ×

```
 1 Client 127.0.0.1 connected the Thursday 19 December 2013 - 09:27:49. => requested page.html
 2 Client 127.0.0.1 connected the Thursday 19 December 2013 - 09:29:10. => 404
 3 Client 127.0.0.1 connected the Thursday 19 December 2013 - 09:29:10. => 404
 4 Client 127.0.0.1 connected the Thursday 19 December 2013 - 09:29:10. => 404
 5 Client 127.0.0.1 connected the Thursday 19 December 2013 - 09:30:44. => requested page.html
 6 Client 127.0.0.1 connected the Thursday 19 December 2013 - 09:30:44. => requested page.html
 7 Client 127.0.0.1 connected the Thursday 19 December 2013 - 09:30:44. => requested page.html
 8 Client 127.0.0.1 connected the Thursday 19 December 2013 - 09:32:12. => 404
 9 Client 127.0.0.1 connected the Thursday 19 December 2013 - 09:32:12. => 404
10 Client 127.0.0.1 connected the Thursday 19 December 2013 - 09:34:22. => requested page.html
11 Client 127.0.0.1 connected the Thursday 19 December 2013 - 09:30:44. => 404
12 Client 127.0.0.1 connected the Thursday 19 December 2013 - 09:35:27. => requested page.html
13 Client 127.0.0.1 connected the Thursday 19 December 2013 - 09:36:29. => 404
14 Client 127.0.0.1 connected the Thursday 19 December 2013 - 09:37:00. => ERREUR HTTP
15 Client 127.0.0.1 connected the Thursday 19 December 2013 - 09:37:00. => ERREUR HTTP
```

Aperçu du fichier « log_server.txt »

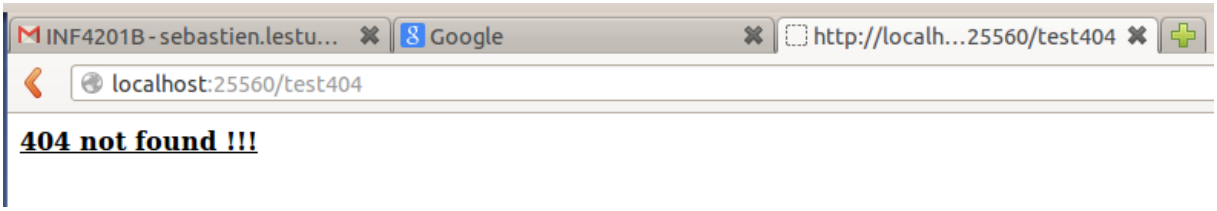
```
dafte@ubuntu:/host/Documents/GitHub/ESIEE-Sockets/e4$ ./serveur 2560 2561
```

http://localhost:2561/

localhost:2561

```
Client 127.0.0.1 connected the Thursday 19 December 2013 - 09:27:49. => requested page.html
Client 127.0.0.1 connected the Thursday 19 December 2013 - 09:29:10. => 404
Client 127.0.0.1 connected the Thursday 19 December 2013 - 09:29:10. => 404
Client 127.0.0.1 connected the Thursday 19 December 2013 - 09:29:10. => 404
Client 127.0.0.1 connected the Thursday 19 December 2013 - 09:30:44. => requested page.html
Client 127.0.0.1 connected the Thursday 19 December 2013 - 09:30:44. => requested page.html
Client 127.0.0.1 connected the Thursday 19 December 2013 - 09:30:44. => requested page.html
Client 127.0.0.1 connected the Thursday 19 December 2013 - 09:32:12. => 404
Client 127.0.0.1 connected the Thursday 19 December 2013 - 09:32:12. => 404
Client 127.0.0.1 connected the Thursday 19 December 2013 - 09:34:22. => requested page.html
Client 127.0.0.1 connected the Thursday 19 December 2013 - 09:30:44. => 404
Client 127.0.0.1 connected the Thursday 19 December 2013 - 09:35:27. => requested page.html
Client 127.0.0.1 connected the Thursday 19 December 2013 - 09:36:29. => 404
```

Aperçu du fichier « log_server.txt » via le navigateur



Aperçu de l'erreur 404 personnalisée en cas de recherche de page inexistante

Aperçu de la page « page.html » via le navigateur

EXERCICE 5

Etant donné que nous nous trouvons dans un réseau privé (celui de l'école) et éducatif, il est évident et primordial qu'un proxy filtre et redirige les requêtes allant vers le Web. Toutefois, ce proxy nous empêche de contacter directement les sites Web extérieurs.

Ainsi nous devons donc résoudre ce contretemps afin de signifier que notre requête doit être lue et déléguée au proxy afin de récupérer les données voulues.

Pour ce faire nous utilisons le même programme que pour l'exercice 2 à une petite différence prêt.

Voici la requête à utiliser pour « traverser » le proxy :

```
GET www.esiee.fr HTTP/1.1
Host: www.esiee.fr:80
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Proxy-connection: keep-alive
```

Grâce à l'argument « Proxy-connection », nous sommes donc capables d'accéder aux sites externes. Il faut toutefois préciser le nom de domaine en entier sous peine de recevoir des erreurs.

CONCLUSION

A travers l'ensemble de ce travail, nous avons pu donc développer de manière plus profonde notre compréhension du fonctionnement des sockets.

La gestion des réponses, des buffers et des statuts de chaque client sont en effet primordiaux pour le bon fonctionnement de l'ensemble du système.

L'exercice 4 est selon nous le parfait exemple d'application des sockets, permettant à un programme de gérer des connections simultanées et également d'établir une journalisation, élément essentiel pour les statistiques d'accès au serveur, mais également d'un point de vue sécurité.

ANNEXES

ANNEXE : EXERCICE 1

CLIENT.C

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <errno.h>
#include <sys/un.h>
#include <netinet/in.h>
#include <netdb.h>

#define h_addr h_addr_list[0]
#define MESSAGE_SIZE 500

void usage(void){
        fprintf(stderr,"Utilisation : ./client \"nom du serveur\" \"Port utilisé\" \"Chaine
à envoyer\"\n");
}

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

        if(argc < 4){
                usage();
                return -1;
        }
        struct sockaddr_in localAdd;
        struct sockaddr_in serverAdd;
        struct hostent* h_s;

        int cSocket,result;
        int size = sizeof(serverAdd);

        char pid[5];
        char message1[MESSAGE_SIZE];
        char message2[MESSAGE_SIZE];

        /*Socket du client*/
        cSocket = socket(AF_INET, SOCK_DGRAM, 0);
        if(cSocket == -1){
                perror("Erreur lors de la création de la socket");
                return 0;
        }

        localAdd.sin_family = AF_INET;
        localAdd.sin_port = htons(5600);
        localAdd.sin_addr.s_addr = INADDR_ANY;
        memset(localAdd.sin_zero, 0, 8);

        result = bind(cSocket, (struct sockaddr*) &localAdd, sizeof(localAdd));

        if(result == -1){
                perror("Erreur lors du bind!\n");
                return 0;
        }

        /*Adresse du serveur distant à joindre*/
        h_s = gethostbyname(argv[1]);
        if(h_s == NULL){
```

```
                perror("Erreur : gethostbyname()\n");
                return 0;
        }
        serverAdd.sin_family = AF_INET;
        serverAdd.sin_port = htons(atoi(argv[2]));
        memcpy(&serverAdd.sin_addr.s_addr, h_s->h_addr, 4);

        fprintf(stdout,"-------------\n");
        fprintf(stdout," Client Side \n");
        fprintf(stdout,"-------------\n");
        sprintf(pid,"%d",getpid());
        strcpy(message1,argv[3]);

        /*Envoi des données au serveur*/
        fprintf(stdout,"Sent :\n");
        sendto(cSocket, (void *) message1, sizeof(message1), 0, (struct sockaddr*)
&serverAdd, sizeof(serverAdd));
        sendto(cSocket, (void *) pid, sizeof(pid), 0, (struct sockaddr*) &serverAdd,
sizeof(serverAdd));
          fprintf(stdout,"\tMessage : %s\n",argv[3]);
        fprintf(stdout,"\tClient PID : %s\n",pid);

        /*Reception de la réponse*/
        fprintf(stdout,"Recieved :\n");
        recvfrom(cSocket, message1, MESSAGE_SIZE, 0,(struct sockaddr*) &serverAdd, &size);
        recvfrom(cSocket, message2, MESSAGE_SIZE, 0,(struct sockaddr*) &serverAdd, &size);
        printf("\tMessage : %s\n\tServer PID : %s\n", message1, message2);

        printf("End of line...\n");
        return 0;
}
```

SERVEUR.C

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <errno.h>
#include <sys/un.h>
#include <netinet/in.h>

#define MESSAGE_SIZE 500

void usage(void){
        fprintf(stderr,"Utilisation : ./serveur \"Port utilisé\"\n");
}

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

        if(argc < 2){
                usage();
                return -1;
        }

        struct sockaddr_in serverAdd;
             struct sockaddr_in clientAdd;

        int sSocket;
          int taille = sizeof(clientAdd);

        char pid[5];
        char message1[MESSAGE_SIZE];
        char message2[MESSAGE_SIZE];


        /*Création de la socket d'écoute*/
        sSocket = socket(AF_INET, SOCK_DGRAM, 0);

        if(sSocket == -1){
                perror("Erreur de création de la socket : N°"+ errno);
                return 0;
        }

        serverAdd.sin_family = AF_INET;
        serverAdd.sin_port = htons(atoi(argv[1]));
        serverAdd.sin_addr.s_addr = INADDR_ANY;
        memset(serverAdd.sin_zero, 0, 8);

        if(bind(sSocket, (struct sockaddr*) &serverAdd, sizeof(serverAdd)) == -1) {
                perror("Erreur lors du bind !\n");
                return 0;
        }

        fprintf(stdout,"-------------\n");
        fprintf(stdout," Server Side \n");
        fprintf(stdout,"-------------\n");

        /*Reception des informations du client*/
        fprintf(stdout,"Recieved :\n");
        recvfrom(sSocket, message1, MESSAGE_SIZE, 0,(struct sockaddr*) &clientAdd,
&taille);
        recvfrom(sSocket, message2, MESSAGE_SIZE, 0,(struct sockaddr*) &clientAdd,
&taille);
        fprintf(stdout,"\tMessage : %s\n\tClient PID : %s\n", message1, message2);

        sprintf(pid,"%d",getpid());
```

```
        /*Envoi des informations au client*/
        fprintf(stdout,"Sent :\n");
        sprintf(message1,"%s%s",message1," (ACK from server)");
        sendto(sSocket, (void *) message1, sizeof(message1), 0, (struct sockaddr*)
&clientAdd, sizeof(clientAdd));
        sendto(sSocket, (void *) pid, sizeof(pid), 0, (struct sockaddr*) &clientAdd,
sizeof(clientAdd));
        fprintf(stdout,"\tMessage : %s\n",message1);
        fprintf(stdout,"\tServer PID : %s\n",pid);

        printf("End of line...\n");
        return 0;
}
```

ANNEXE : EXERCICE 2

SERVEUR.C

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <errno.h>
#include <sys/un.h>
#include <netinet/in.h>

void usage(void){
        fprintf(stderr,"Utilisation : ./serveur \"Port utilisé\"\n");
}

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

        if(argc < 2){
                usage();
                return -1;
        }

        struct sockaddr_in serverAdd;
        struct sockaddr_in clientAdd;

        int lSocket, lService, res;
        int taille = sizeof(clientAdd);
        char requete[2000];
        char reponse[7] = "OK HTTP";

        lSocket = socket(AF_INET, SOCK_STREAM, 0);

        if(lSocket == -1){
                perror("Erreur lors de la création de la socket !\n");
                return -1;
        }

        serverAdd.sin_family = AF_INET;
        serverAdd.sin_port = htons(atoi(argv[1]));
        serverAdd.sin_addr.s_addr = INADDR_ANY;
        memset(serverAdd.sin_zero, 0, 8);
        res = bind(lSocket, (struct sockaddr*) &serverAdd, sizeof(serverAdd));

        if(res == -1){
                perror("Erreur lors du bind !\n");
                return -1;
        }

        /* En écoute */
        if(listen(lSocket,1) == -1){
                 perror("Erreur lors du listen !\n");
                return -1;
        }

        lService = accept(lSocket,(struct sockaddr*) &clientAdd, &taille);

        if(lService==-1){
                 perror("Erreur lors du accept !\n");
                return -1;
        }


        read(lService, (void*) requete, sizeof(requete));
```

```
        fprintf(stdout,"Request from browser  : \n---------\n%s---------\n", requete);
        write(lService, (void*) reponse, sizeof(reponse));
        fprintf(stdout,"Answer from server : %s \n",reponse);
        close(lService);
        close(lSocket);
        fprintf(stdout,"End of line...\n");

        return 0;
}
```

ANNEXE : EXERCICE 3

CLIENT.C

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <errno.h>
#include <sys/un.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdlib.h>
#define h_addr h_addr_list[0]

void usage(void){
        fprintf(stderr,"Utilisation : ./client \"nom du serveur a joindre\" \"Port
utilisé\" \"Fichier demandé\"");
}

int main(int argc, char* argv[])
{
        if(argc < 4){
                usage();
                return -1;
        }

        struct sockaddr_in clientAdd;
        struct sockaddr_in serverAdd;
        struct hostent* h_s;

        char requete[1000];
        char reponse[65536];

        int lSocket, result;
        int byteCount;

        /*Socket du client*/
        lSocket = socket(AF_INET, SOCK_STREAM, 0);
        if(lSocket == -1){
                perror("Erreur lors de la création de la socket !\n");
                return 0;
        }

        clientAdd.sin_family = AF_INET;
        clientAdd.sin_port = htons(9000);
        clientAdd.sin_addr.s_addr = INADDR_ANY;
        memset(clientAdd.sin_zero, 0, 8);

        result = bind(lSocket, (struct sockaddr*) &clientAdd, sizeof(clientAdd));
        if(result == -1){
                perror("Erreur lors du bind!\n");
                return -1;
        }


        /*Adresse du serverAdd distant à joindre*/
        h_s = gethostbyname(argv[1]);
        if(h_s == NULL){
                perror("Erreur avec le résultat de gethostbyname()\n");
                return -1;
        }

        serverAdd.sin_family = AF_INET;
```

```
        memcpy(&serverAdd.sin_addr.s_addr, h_s->h_addr, 4);
        serverAdd.sin_port = htons(atoi(argv[2]));

        if(connect(lSocket, (struct sockaddr*) &serverAdd, sizeof(serverAdd)) == -1){
                perror("Echec lors de la connexion !");
                return -1;
        }

        sprintf(requete,"%s%s%s%s%s%s%s","GET /",argv[3]," HTTP/1.1\nHost:
",argv[1],":",argv[2],"\nConnection: Keep-Alive\n\n");

        send(lSocket, (void*) requete, sizeof(requete),0);

        byteCount = recv(lSocket, (void*) reponse, sizeof(reponse),0);
        printf("Reponse du serveur : \n%s\n",reponse);
        close(lSocket);
        return 0;
}
```

ANNEXE : EXERCICE 4

SERVEUR.C

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/un.h>
#include <netinet/in.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <pthread.h>
#include <time.h>
#include <arpa/inet.h>

#define TAILLE_MAX 256

void usage(void){
        fprintf(stderr,"Utilisation : ./serveur \"Port HTTP\" \"Port LOG\"\n");
}

int main(int argc, char* argv[])
{
        if(argc <3){
                usage();
                return -1;
        }
        fd_set rdfs;
        FILE* logfile;
        FILE* page;

            time_t timestamp;

        struct sockaddr_in httpAdd, logAdd, clientAdd;

        int httpSocket, logSocket;
        int tailleHttp = sizeof(httpAdd);
        int tailleLog = sizeof(logAdd);
        int tailleClient = sizeof(clientAdd);
        int res = 0;

        char dbuffer[256] ="";
        char buffer[TAILLE_MAX];
        char reponse[1024] ="";
        char* log = (char*)malloc(sizeof(char)*65536);
        char requete[1000] ="";
        char ipbuffer[256] ="";

        httpSocket = socket(AF_INET, SOCK_STREAM,0);
        logSocket = socket(AF_INET, SOCK_STREAM,0);
        if(httpSocket == -1 || logSocket == -1){
                perror("Erreur lors de la création des sockets !\n");
                return -1;
        }


        httpAdd.sin_family = AF_INET;
        httpAdd.sin_port = htons(atoi(argv[1]));
        httpAdd.sin_addr.s_addr = INADDR_ANY;
        memset(httpAdd.sin_zero, 0, 8);

        logAdd.sin_family = AF_INET;
        logAdd.sin_port = htons(atoi(argv[2]));
        logAdd.sin_addr.s_addr = INADDR_ANY;
        memset(logAdd.sin_zero, 0, 8);

        if (bind(logSocket, (struct sockaddr *) &logAdd, sizeof(logAdd)) == -1) {
```

```
        perror("Erreur lors du bind (LOG) !");
        return -1;
}

if(listen(logSocket,1) == -1){
         perror("Erreur lors du listen (LOG)!\n");
        return -1;
}

if (bind(httpSocket, (struct sockaddr *) &httpAdd, sizeof(httpAdd)) == -1) {
        perror("Erreur lors du bind (HTTP) !");
        return -1;
}

if(listen(httpSocket,1) == -1){
         perror("Erreur lors du listen (HTTP) !\n");
        return -1;
}

    while(1) {
         FD_ZERO(&rdfs);

         /* Ajout du socket pour les requètes HTTP */
         FD_SET(httpSocket, &rdfs);

         /* Ajout du socket pour l'accès aux logs */
         FD_SET(logSocket, &rdfs);

         int max = select(FD_SETSIZE, &rdfs, NULL, NULL, NULL);

         if(max == -1){
             perror("Erreur lors du select()");
             return -1;
         } else {
                 if(max == 0){
                         fprintf(stdout,"Rien \n");
             } else {
                  if(FD_ISSET(httpSocket, &rdfs)){
                      int httpService = accept(httpSocket,(struct sockaddr*) &clientAdd,
&tailleClient);
                      if(httpService == -1){
                          perror("Erreur lors de accept() (HTTP)");
                          return -1;
                      }
                      read(httpService, (void*) requete, sizeof(requete));
                      if(strstr(requete, "HTTP") == NULL){
                          sprintf(reponse, "ERREUR HTTP");
                          res = 1;
                      } else {
                          if(strstr(requete,"page.html") == NULL){
                              sprintf(reponse,"HTTP/1.1 404 Not Found\r\nConnection : keep-
alive\r\n\r\n");
                              sprintf(reponse, "%s<html><body><b><u>404 not found
!!!</u></b></body></html>\r\n",reponse);
                              res = 2;
                 } else {
                              page = fopen("page.html","r");
                              if(page != NULL){
                                  sprintf(reponse,"HTTP/1.1 200 OK\r\nConnection : keep-
alive\r\n\r\n");
                                  while(fgets(buffer,TAILLE_MAX,page) != NULL){
                                      sprintf(reponse,"%s%s",reponse,buffer);
                    }
                              } else {
                                  fprintf(stderr,"Erreur lors de la lecture de la page\n");
                                  return -1;
                              }
                              fclose(page);
                          }
                      }
                      write(httpService, (void*) reponse, strlen(reponse));

                      timestamp = time(NULL);
```

```
                        strftime(dbuffer, sizeof(dbuffer), "%A %d %B %Y - %X.",
        localtime(&timestamp));
                        logfile = fopen("log_server.txt","a");
                        fprintf(logfile, "Client %s connected the %s =>
        ",inet_ntoa(clientAdd.sin_addr),dbuffer);
                          switch(res){
                            case 0:
                                fprintf(logfile,"requested page.html\n");
                                break;
                            case 1:
                                fprintf(logfile, "HTTP ERROR\n");
                                break;
                            case 2:
                                fprintf(logfile, "404\n");
                                break;
                        }
                        res = 0;
                        fclose(logfile);
                        close(httpService);

                    /*S'il y a une activité sur le port LOG */
                    } else if(FD_ISSET(logSocket, &rdfs)){
                        int logService = accept(logSocket,(struct sockaddr*) &clientAdd,
        &tailleClient);
                        if(logService == -1){
                            perror("Erreur lors de accept() (HTTP)");
                            return -1;
                        }
                        logfile = fopen("log_server.txt","r");
                        if(logfile != NULL){
                            while(fgets(buffer,TAILLE_MAX,logfile) != NULL){
                                strcat(log,buffer);
                            }
                        } else {
                            fprintf(stderr,"Erreur lors de la lecture du fichier de log !\n");
                            return -1;
                           }
                        fclose(logfile);
                        write(logService, (void*) log, strlen(log));
                        close(logService);
                    }
                }
            }
            memset(buffer, 0, sizeof(buffer));
            memset(log,0,sizeof(log));
            memset(reponse,0,sizeof(reponse));
        }
    close(httpSocket);
    close(logSocket);
}
```