

Exercice 1 :

Le but de cet exercice est de créer un client et un serveur communiquant en mode non connecté dans le domaine AF_INET.

Le client reçoit en argument sur la ligne de commande :

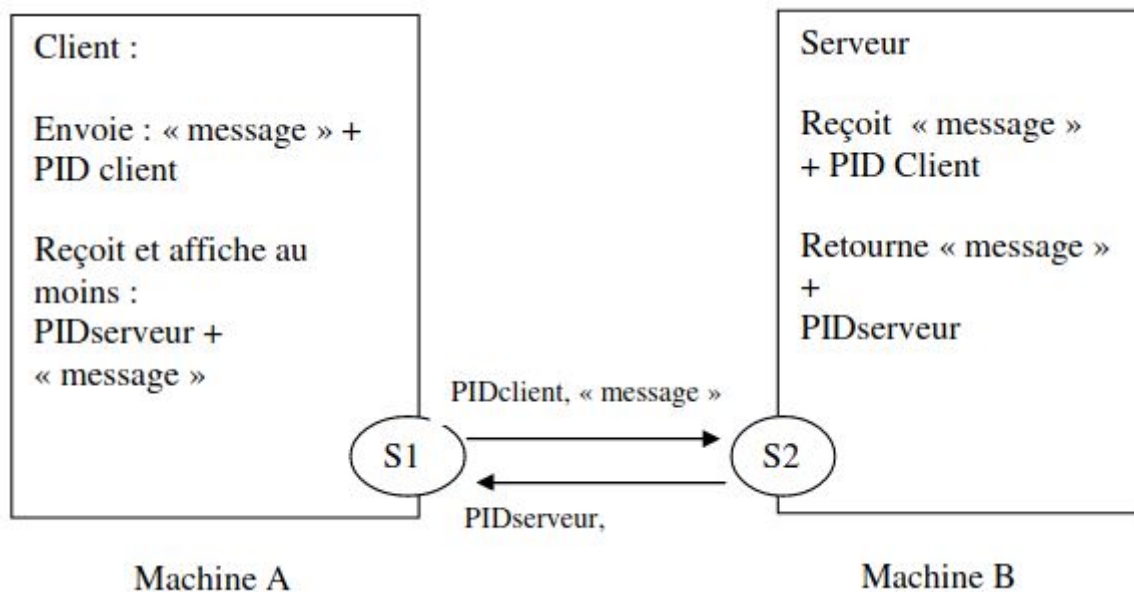
- Le port de ce serveur
- Le nom de la machine hébergeant le serveur
- Une chaîne de caractères, représentant le message à envoyer.

Le serveur reçoit en argument sur la ligne de commande :

- Le port sur lequel il va attendre les connexions des clients.

Le client doit envoyer au serveur son n° de PID ainsi que la chaîne de caractères passée en argument.

Le serveur quant à lui, reçoit les informations du serveur, les affiche, et retourne au client son propre numéro de PID ainsi que la chaîne reçue, qui sont alors affichées par le client.



Construction du serveur :

```
int main(int argc, char * argv[])
{
    if(argc < 1){
        perror("not enough arguments");
    }
    int port = atoi(argv[1]);
    char msg[1024];

    int serveur;
    struct sockaddr_in clientadd, serveuradd;
    int res;
    struct hostent* host;
```

Nous utilisons un entier permettant de stocker le port reçu en argument ainsi qu'une chaîne de caractère msg de taille 128 (choisie de façon totalement arbitraire).

L'entier serveur sera celui contenant la socket utilisée pour les communications, tandis que le res reçoit le bind de cette socket.

Nous créons les structures sockaddr_in pour les deux addresses, client et serveur, ainsi qu'une structure hostent* contenant l'hôte de connexion (host).

```
host = gethostbyname("localhost");
if(host == NULL)
{
    perror("gethost");
    return -1;
}
```

Dans un premier temps, nous initialisons l'hôte en localhost, puisque nous sommes en mode non connecté. Le test vérifie si cette initialisation n'a pas fonctionné, et interrompt le programme le cas échéant.

```
serveur = socket(AF_INET, SOCK_DGRAM, 0);
if (serveur == -1)
{
    perror("socket");
    return -1;
}
```

Nous initialisons ensuite notre socket serveur en AF_INET de type SOCK_DGRAM, utilisé justement dans le cas du mode non connecté. Le test a le même but que précédemment, à savoir vérifier si cette opération s'est déroulée correctement et d'interrompre le programme si nécessaire.

```
memset(&serveuradd.sin_zero, 0, sizeof(serveuradd));
serveuradd.sin_family = AF_INET;
serveuradd.sin_port = htons(port);
serveuradd.sin_addr.s_addr = htonl(INADDR_ANY);

res = bind(serveur, (struct sockaddr *)&serveuradd, sizeof(serveuradd));
if(res == -1) {
    perror("bind");
    return -1;
}
```

Ici nous préparons l'adresse du serveur en lui assignant le port en paramètre et une adresse disponible (via `htonl(INADDR_ANY)`).

Nous effectuons un `bind` de la socket serveur avec cette adresse par la suite, avec le même test pour assurer le fonctionnement.

```
unsigned int socklen = (unsigned int) sizeof(serveuradd);
recvfrom(serveur, msg, sizeof(msg), 0, (struct sockaddr *)&clientadd, &socklen); // RECEIVE
FROM

printf("%s\n", msg);
char pid[32];
sprintf(pid, " PID Serveur : %d", (int) getpid());
strcat(msg, pid);
printf("%s\n", msg);

sendto(serveur, msg, sizeof(msg), 0, (struct sockaddr *)&clientadd, sizeof(clientadd)); // send to
return 0;
}
```

Voilà ensuite le coeur du programme du serveur, à savoir recevoir le message du client via le `recvfrom` (nécessitant de préparer la longueur de l'adresse au préalable), d'y accoler le PID du serveur en effectuant une concaténation, puis de renvoyer le tout au client via le `sendto`.

Le programme se termine alors.

Construction du client :

```
int main(int argc, char* argv){  
    if(argc < 3){  
        perror("not enough arguments");  
        return -1;  
    }  
    int port = atoi(argv[1]);  
    int host_port = atoi(argv[2]);  
    char msg[1024];  
    sprintf(msg, "%s", argv[3]);  
    char msg2[1024];  
  
    char pid[32];  
    sprintf(pid, " client pid : %d", (int) getpid());  
    strcat (msg, pid);  
  
    int chaussette;
```

De la même manière que dans le serveur, nous utilisons un entier port pour stocker le port en argument du client. A cela s'ajoute un autre entier host_port pour stocker le port du serveur, ainsi qu'une chaîne de caractères msg contenant le message à envoyer. La char msg2 contiendra la chaîne de caractères renvoyée par le serveur.

Notons que nous concaténons à msg une chaîne de caractères pid contenant le PID du client afin d'envoyer en un coup le message du client et son pid.

Enfin le dernier entier "chaussette" (traduction française de sock, abréviation de socket) correspond à notre socket.

```
chaussette = socket(AF_INET, SOCK_DGRAM, 0);  
  
if(chaussette == -1){  
    perror("socket");  
    return -1;  
}
```

Nous l'initialisons de la même manière que pour le serveur.

```
// ----- client ad
struct sockaddr_in ad;
memset(&ad, 0, sizeof(ad));
ad.sin_family = AF_INET;
ad.sin_port = htons(port);
ad.sin_addr.s_addr = INADDR_ANY;

//----- host adress
struct hostent *host = gethostbyname( "localhost");
if(host == NULL){
    perror("host not found");
}

struct sockaddr_in ad_serv;
memset(&ad_serv, 0, sizeof(ad_serv));
ad_serv.sin_family = AF_INET;
ad_serv.sin_port = htons(host_port);
memcpy(&ad_serv.sin_addr.s_addr, host->h_addr, host->h_length);

// ----- bound
int bound;
bound = bind(chaussette, (struct sockaddr *) &ad, sizeof(ad));
if (bound == -1){
    perror("bind failure");
    return -1;
}
```

De même, nous initialisons l'adresse du client, du serveur, l'hôte et le bind de notre socket de la même manière que pour le serveur.

```
printf("%s\n", msg);
sendto(chaussette, msg, sizeof(msg), 0, (struct sockaddr *) &ad_serv,
    sizeof(ad_serv));

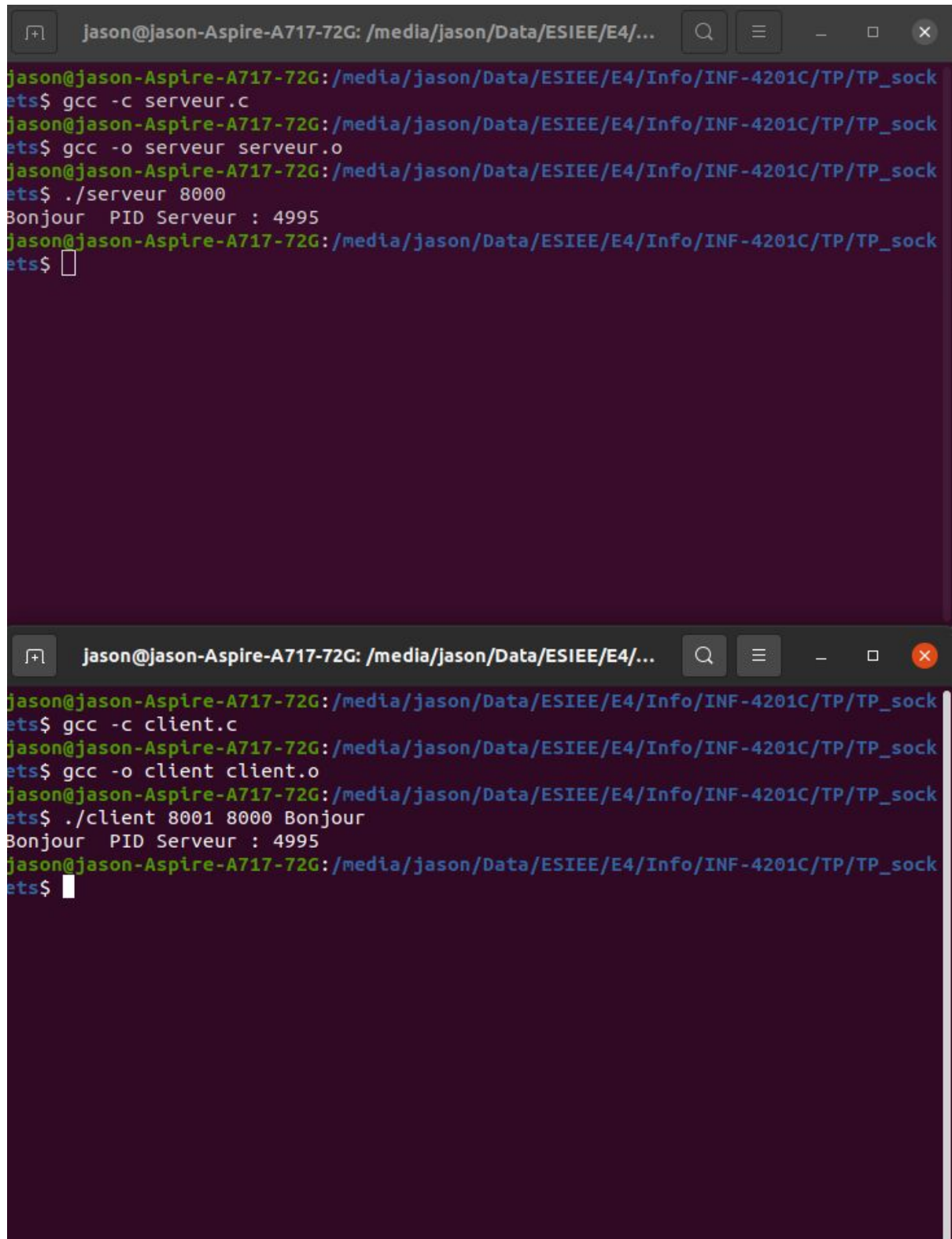
unsigned int socklen = (unsigned int) sizeof(ad_serv);
recvfrom(chaussette, msg2, sizeof(msg2), 0, (struct sockaddr *)&ad_serv, &socklen);

printf("%s\n", msg2);

return 0;
}
```

Enfin, le coeur du programme du client consiste à envoyer le message en paramètre ainsi que le pid du client stockés dans msg au moyen d'un sendto vers l'adresse du serveur, qui fait son traitement par la suite comme expliqué plus tôt. L'appel de recvfrom bloque jusqu'à ce que le serveur ait à son tour effectué son sendto. Le retour du serveur est stocké dans msg2, qui est alors affiché dans le terminal. Le programme se termine alors.

Exécution :



```

jason@jason-Aspire-A717-72G: /media/jason/Data/ESIEE/E4/...
jason@jason-Aspire-A717-72G:/media/jason/Data/ESIEE/E4/Info/INF-4201C/TP/TP_socks$ gcc -c serveur.c
jason@jason-Aspire-A717-72G:/media/jason/Data/ESIEE/E4/Info/INF-4201C/TP/TP_socks$ gcc -o serveur serveur.o
jason@jason-Aspire-A717-72G:/media/jason/Data/ESIEE/E4/Info/INF-4201C/TP/TP_socks$ ./serveur 8000
Bonjour  PID Serveur : 4995
jason@jason-Aspire-A717-72G:/media/jason/Data/ESIEE/E4/Info/INF-4201C/TP/TP_socks$

jason@jason-Aspire-A717-72G: /media/jason/Data/ESIEE/E4/...
jason@jason-Aspire-A717-72G:/media/jason/Data/ESIEE/E4/Info/INF-4201C/TP/TP_socks$ gcc -c client.c
jason@jason-Aspire-A717-72G:/media/jason/Data/ESIEE/E4/Info/INF-4201C/TP/TP_socks$ gcc -o client client.o
jason@jason-Aspire-A717-72G:/media/jason/Data/ESIEE/E4/Info/INF-4201C/TP/TP_socks$ ./client 8001 8000 Bonjour
Bonjour  PID Serveur : 4995
jason@jason-Aspire-A717-72G:/media/jason/Data/ESIEE/E4/Info/INF-4201C/TP/TP_socks$
```

On exécute dans deux fenêtres de terminal séparées le client et le serveur. Nous avons envoyé le message “Bonjour” au serveur et celui-ci nous répond en renvoyant le message ainsi que le pid du serveur. Les ports sont choisis pour être des ports habituellement inoccupés et donc toujours libres.

Exercice 2 :

Il s'agit cette fois de créer un serveur attendant une requête de connexion sur un port d'écoute (en argument) et qui affiche ensuite le flot de caractères émis par le client qui s'y connecte.

Explication du code :

```
int main(int argc, char * argv[])
{
    if(argc < 1){
        perror("not enough arguments");
    }
    int port = atoi(argv[1]);
    char msg[1024];

    int serveur;
    struct sockaddr_in serveuradd, clientadd;
    int res;
    struct hostent* host;
```

Il s'agit des mêmes variables que pour le serveur précédent, seule la longueur de la chaîne de caractères varie.

```
host = gethostbyname("localhost");
if(host == NULL)
{
    perror("gethost");
    return -1;
}

serveur = socket(AF_INET, SOCK_STREAM, 0);
if (serveur == -1)
{
    perror("socket");
    return -1;
}

memset(&serveuradd.sin_zero, 0, sizeof(serveuradd));
serveuradd.sin_family = AF_INET;
serveuradd.sin_port = htons(port);
//serveuradd.sin_addr.s_addr = INADDR_ANY;
memcpy(&serveuradd.sin_addr.s_addr, host->h_addr, host->h_length);
```


Nous avons, comme avec le serveur précédent, initialisé un hôte et notre socket, ainsi que l'adresse de notre serveur.

```
res = bind (serveur, (struct sockaddr *)&serveuradd, sizeof(serveuradd));
if(res == -1) {
    perror("bind");
    return -1;
}

res = listen (serveur, 1);
if(res == -1) {
    perror("listen");
    return -1;
}
printf("listening\n");
```

De même, on bind notre socket de la même manière que notre premier serveur.

On initialise également l'écoute avec un test pour couper le programme en cas de dysfonctionnement.

```
while(1){

    unsigned int ad_len = sizeof(struct sockaddr_in);
    int clientaccept = accept (serveur, (struct sockaddr*) &clientadd, &ad_len);
    if(res == -1) {
        perror("accept");
        return -1;
    }
    //----- lire -----
    printf("accepting\n");
    int lire = read(clientaccept, msg, sizeof(msg));
    if(lire == -1){
        perror("lire");
        return -1;
    }
    else if(lire == 0){
        perror("nothing to read");
        exit(1);
    }
    //----- write -----
    printf("%s", msg);
    write(clientaccept, msg, sizeof(msg));
}
//----- connection is DONE;

return 0;
}
```

Voici le cœur du programme.

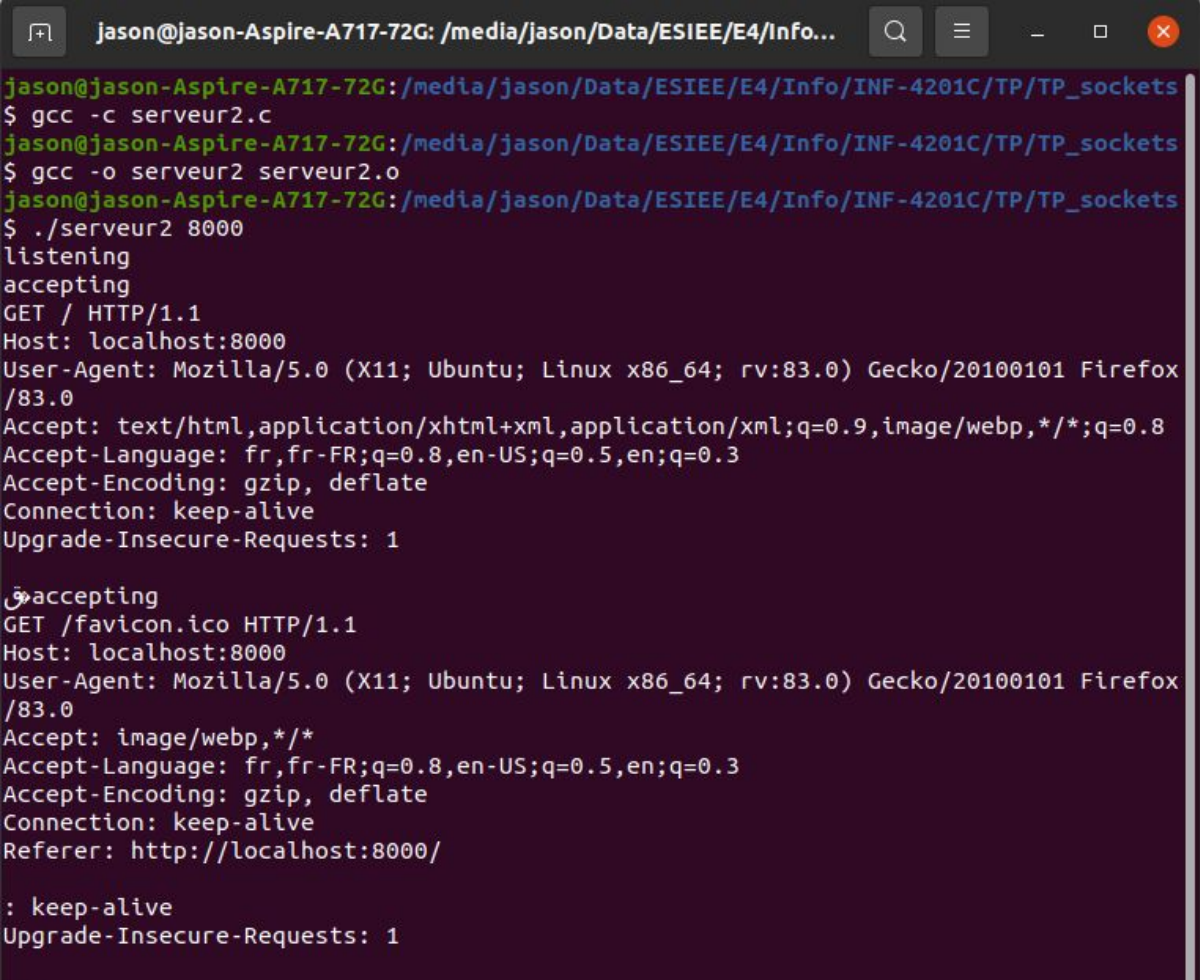
Dans cette boucle, on initialise une requête avec le même test habituel pour interrompre le programme en cas d'erreur.

On effectue ensuite une read de cette requête pour récupérer le flot de caractères du client, avec le même test habituel, et un autre test pour interrompre le programme si il n'y a rien à lire.

On affiche ensuite le flot de caractères, puis on write sur cette requête le flot de caractères.

Exécution :

Sur le terminal :

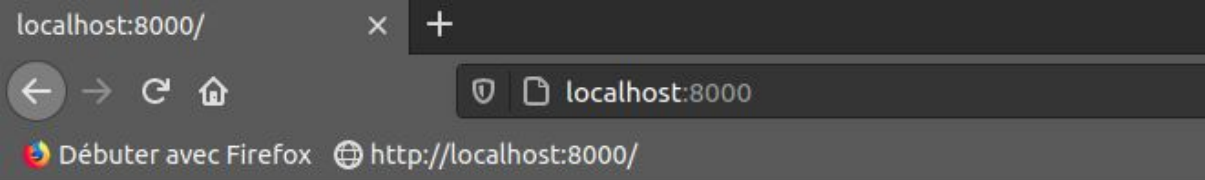


```
jason@jason-Aspire-A717-72G: /media/jason/Data/ESIEE/E4/Info...
jason@jason-Aspire-A717-72G:/media/jason/Data/ESIEE/E4/Info/INF-4201C/TP/TP_sockets$ gcc -c serveur2.c
jason@jason-Aspire-A717-72G:/media/jason/Data/ESIEE/E4/Info/INF-4201C/TP/TP_sockets$ gcc -o serveur2 serveur2.o
jason@jason-Aspire-A717-72G:/media/jason/Data/ESIEE/E4/Info/INF-4201C/TP/TP_sockets$ ./serveur2 8000
listening
accepting
GET / HTTP/1.1
Host: localhost:8000
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Firefox/83.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;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

accepting
GET /favicon.ico HTTP/1.1
Host: localhost:8000
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Firefox/83.0
Accept: image/webp,*/*
Accept-Language: fr,fr-FR;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Connection: keep-alive
Referer: http://localhost:8000/

: keep-alive
Upgrade-Insecure-Requests: 1
```

Sur la page :



```

GET / HTTP/1.1
Host: localhost:8000
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Firefox/83.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;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
  
```

The screenshot shows a Firefox browser window with the address bar set to 'localhost:8000/'. Below the address bar, the status bar indicates 'Débuter avec Firefox' and the URL 'http://localhost:8000/'. The main content area displays the raw HTTP request details, including the method 'GET', the path '/', and the protocol 'HTTP/1.1'. It also shows the host 'localhost:8000', the user agent 'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Firefox/83.0', and various headers like 'Accept', 'Accept-Language', 'Accept-Encoding', 'Connection', and 'Upgrade-Insecure-Requests'.

On constate donc que la requête est bien reçue sur le navigateur et qu'elle est bien renvoyée au serveur.

Cependant, on remarque des caractères en plus, qui sont des caractères vides. Cela vient du fait que notre taille pour notre chaîne msg est probablement trop grande.

Exercice 3:

Pour l'exercice 3, le but est d'écrire un client capable d'envoyer une requête http sur un serveur web demandant l'affichage d'un fichier html.

Le client reçoit donc en argument, le nom du serveur ou l'on se connecte, le numéro de port du serveur, et le nom du fichier que l'on souhaite afficher.

La requête http sera copié collé de celle trouvée sur l'exercice 2.

Explication du code:

Exemple de ligne de compilation :

```
./client 80 intra.esiee.fr index.html
```

Nous allons déjà préparer le programme en créant notre requête html, et en préparant un socket pour communiquer avec le serveur.

```
20 |     sprintf(msg, "GET /%s HTTP/1.1\  
21 | Host: localhost:8000\  
22 | User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:83.0) Gecko/20100101 Firefox/83.0\  
23 | Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8\  
24 | Accept-Language: fr,fr-FR;q=0.8,en-US;q=0.5,en;q=0.3\  
25 | Accept-Encoding: gzip, deflate\  
26 | Connection: keep-alive\  
27 | Upgrade-Insecure-Requests: 1\  
28 | Cache-Control: max-age=0, filename);  
29 |
```

```
//----- socket -----  
printf("socket \n");  
int s = socket(AF_INET,SOCK_STREAM,0);  
if (s == -1)  
{  
    perror("socket");  
    return -1;  
}
```

Ensuite, on cherchera à retrouver l'adresse de notre serveur. Pour cela on utilisera "gethostbyname()". Il ne restera alors plus qu'à préciser le port où l'on se connecte et de se connecter.

```
int connection = connect(s, (struct sockaddr*) &serveuradd, sizeof(serveuradd));
if (connection == -1)
{
    perror("connection failed");
    return -1;
}
```

Une fois ici, il ne reste plus qu'à envoyer la requête via notre socket, et de lire la réponse:

```
//----- write -----
if(write(s, msg, sizeof(msg)) == -1){
    perror("write error");
    return -1;
}

printf("read begin\n");
//----- read -----
int read_desc;
while(read(s, msg2, sizeof(msg2)) > 0)
{}

printf("%s\n", msg2);
```

Il ne restera alors plus qu'à close() la socket.

Résultat

Cependant les résultats sur cet exercice ne sont pas bons:

Si le client arrive parfaitement à communiquer avec le serveur de l'exercice 2, lorsque l'on essaye de communiquer avec intra.esiee.fr en demandant l'affichage de index.html par exemple, on tombe sur ça:

```
magicmike@DESKTOP-RBBL42I:/mnt/c/Users/Xx_pussyslayerVR_xX/Desktop/Esiee/e4/hpc/exo2$ g++ client2.c -o client
magicmike@DESKTOP-RBBL42I:/mnt/c/Users/Xx_pussyslayerVR_xX/Desktop/Esiee/e4/hpc/exo2$ ./client 80 intra.esiee.fr index.html
socket
host acquisition
host acquired
connexion
write begin
read begin
^C
```

Le programme reste bloqué sur read(), impliquant donc que la page ne répond pas à notre requête.

Sauf dans un cas, si l'on fait en sorte que la requête http soit erronée, alors le serveur web nous renvoie un message d'erreur sur msg2:

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>400 Bad Request</title>
</head><body>
<h1>Bad Request</h1>
<p>Your browser sent a request that this server could not understand.<br />
</p>
</body></html>
```

Nous pensons que le problème vient du read, peut-être avons-nous mal compris ce qu'il faut faire après avoir envoyé la requête. Cependant, comme ce message d'erreur s'affiche lorsque l'on envoie une requête fausse, nous pensons que la connexion avec le serveur web se fait bien, et qu'il reçoit bien notre requête, et que c'est l'affichage qui ne marche pas.

EXERCICE 4

Pour l'exercice 4, le but est de créer un serveur web capable de communiquer sur deux ports, sur l'un d'entre eux, les clients qui se connectent écriront un message daté dans un fichier log_file, et sur l'autre, les clients récupéreront log_file.

Le serveur recevra en argument les deux ports, d'abord celui sur lequel il va lire le message et le mettre dans le fichier, puis celui sur lequel il restituera log_file.

Explication du code

Pour la compilation :

```
./serveur4 8000 8001
```

Nous créons d'abord le fichier log_file (et on le remplace s'il existe déjà). Puis nous préparons deux sockaddr, une pour chaque port:

```
// ----- serveuradd -----  
memset(writeadd.sin_zero, 0, sizeof(writeadd));  
writeadd.sin_family = AF_INET;  
writeadd.sin_port = htons(portWrite);  
writeadd.sin_addr.s_addr = htonl(INADDR_ANY);  
  
memset(readadd.sin_zero, 0, sizeof(readadd));  
readadd.sin_family = AF_INET;  
readadd.sin_port = htons(portRead);  
readadd.sin_addr.s_addr = htonl(INADDR_ANY);
```

Ensuite nous créons deux sockets, et chacune d'entre elle sera bind avec une des deux sockaddr.

```
//----- socketWrite -----  
s1w = socket(AF_INET, SOCK_STREAM, 0);  
if (s1w == -1)  
{  
    perror("socket");  
    return -1;  
}  
  
res = bind (s1w, (struct sockaddr *)&writeadd, sizeof(writeadd));  
if(res == -1) {  
    perror("bind");  
    return -1;  
}
```

Après avoir utiliser listen() sur les deux sockets il nous faudra preparer box, un fd_set, pour le select:

```
FD_ZERO(&box);  
FD_SET(s1w, &box);  
FD_SET(s2r, &box);  
  
printf("select \n");  
res = select(FD_SETSIZE, &box, 0, 0, 0);  
printf("select out \n");  
if(res<0) {  
    perror("select() a echoué");  
}
```

En effet, il faut que les deux ports puissent être actifs en même temps. Le but du select() ici est d'attendre qu'une des deux connexions de box (donc s1w et s2r) se fasse avant d'exécuter la suite du code.

FD_ZERO() et FD_SET() sont ici après le while car sinon la dernière connexion aurait fixé le select sur un des deux sockets.

une fois là, le code va écrire le message dans logfile si s2r est sollicité:

```
while(readDesc > 0)
{
    printf("%d, %s \n", readDesc, buffer);

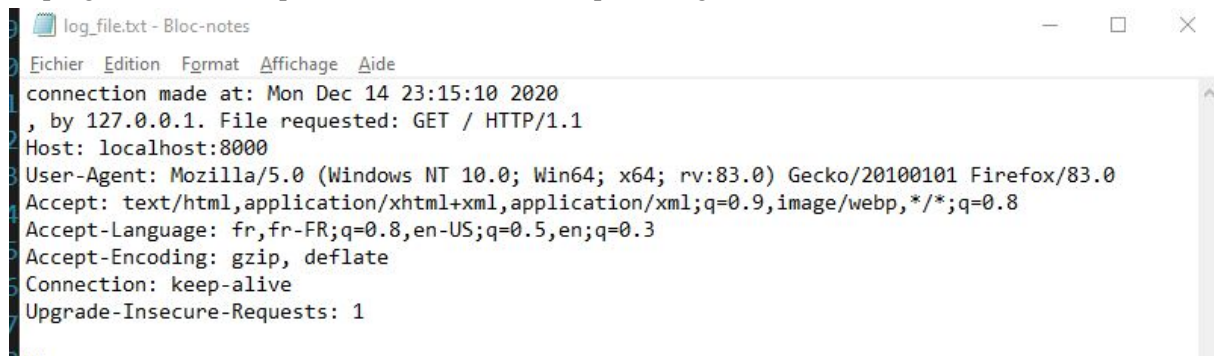
    fputs(buffer, fPtr);
    readDesc = read(writer, buffer, sizeof(buffer));
}
```

et envoyer logfile si s1w est sollicité:

```
while(fgets(buffer, sizeof(buffer), fPtr) != NULL){
    write(reader, buffer, sizeof(buffer));
}
```

Resultats:

Le programme marche plutôt bien: Voici un exemple de logfile si l'on se connecte via mozilla



```
log_file.txt - Bloc-notes
Fichier  Edition  Format  Affichage  Aide
1 connection made at: Mon Dec 14 23:15:10 2020
2 , by 127.0.0.1. File requested: GET / HTTP/1.1
3 Host: localhost:8000
4 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:83.0) Gecko/20100101 Firefox/83.0
5 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
6 Accept-Language: fr,fr-FR;q=0.8,en-US;q=0.5,en;q=0.3
7 Accept-Encoding: gzip, deflate
8 Connection: keep-alive
9 Upgrade-Insecure-Requests: 1
```

Cependant, voici ce qu'il affiche lorsqu'il envoie le fichier:

```

connection made at: Mon Dec 4 23:15:10 2020
♦nt: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:83.0) Gecko/20100101 Firefox/83.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;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

GET / HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:83.0) Gecko/20100101 Firefox/83.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;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

```

On peut voir que le message est bien écrit, et est bien restitué, cependant il y a beaucoup de bugs d'affichage dû à la trop grande taille du buffer.

Si l'on essaye de le faire marcher avec le client de l'exercice 3, légèrement adapté pour qu'il ne puisse soit qu'écrire, soit que lire (client4.c) nous avons ceci qui s'affiche dans log_file:

```
connection made at: Mon Dec 14 23:48:56 2020
, by 127.0.0.1. File requested: test
```

(pour tester avec client4.c, la ligne de compilation est :

```
./client 8000 localhost test
```