1 Programmation sockets UDP/TCP

Vous trouverez les fichiers à utiliser dans ce TP dans le répertoire ~`coti/TR2`.

Identification d'une machine hôte sur internet Une machine peut être désignée de deux façons différentes :

1. 194.254.163.24 : adresse IP notation nombres et points (ou aussi ascii)
2. 0x18a3fec2 : adresse IP binaire (ici hexa pour voir les octets)

La forme 2 correspond à la structure `in_addr` qui est définie dans `netinet/in.h` généralement comme un entier long.

On n'utilise pas directement un entier long pour pouvoir modifier ultérieurement la longueur de l'adresse (par exemple pour IPv6) sans changer les programmes. De ce fait, toute manipulation d'une adresse doit mentionner la longueur de cette adresse.

Il existe également une troisième forme : `lipn.univ-paris13.fr`, par exemple, qui est appelée *nom symbolique* (ou aussi *nom DNS*). L'utilisation de cette dernière est liée à l'existence d'une équivalence dans le fichier `/etc/hosts` ou à la disponibilité d'un service de noms (DNS, NIS, ...). Cette troisième forme peut souvent être utilisée à la place de la première, car elles sont toutes deux des chaînes de caractères.

Fonctions de conversion Il est possible de passer de la forme 1 à la forme 2 et inversement à l'aide de *fonctions de conversion*.

```
in_addr_t inet_addr (const char *cp);
```

La fonction `inet_addr()` interprète une chaîne de caractères sous forme de notation pointée (de la forme 1).

```
int inet_aton (const char *cp, struct in_addr *inp);
```

`inet_aton()` convertit l'adresse Internet `cp` (notation nombres et points) en une donnée binaire (ordre réseau), et la place dans la structure pointée par `inp`. La fonction `inet_aton()` renvoie une valeur non nulle si l'adresse est valide, et 0 sinon.

```
char *inet_ntoa (struct in_addr in);
```

`inet_ntoa()` convertit l'adresse Internet `in` (notation binaire réseau) en une chaîne de caractères dans la notation nombres et points. La chaîne est renvoyée dans un buffer alloué statiquement, qui est donc écrasé à chaque appel.

En ce qui concerne le nom symbolique, il doit être traduit par le DNS sous une des formes 1 ou 2 avant de pouvoir être utilisé par les programmes. De plus, une machine (un hôte) peut être connue sous plusieurs noms symboliques différents et avoir plusieurs adresses IP différentes. C'est pourquoi toutes les informations concernant une machine sont rassemblées dans une même structure de données `hostent` qui est définie comme suit dans `netdb.h` :

```
struct hostent {
    char *h_name;                 /* Nom officiel de l'hôte */
    char **h_aliases;             /* Liste d'alias */
    int  h_addrtype;              /* Type d'adresse de l'hôte */
    int  h_length;                /* Longueur de l'adresse */
    struct in_addr **h_addr_list; /* Liste d'adresses */
}
#define h_addr h_addr_list[0]     /* pour compatibilité */
```

Les champs de la structure `hostent` sont :

`h_name` est le nom symbolique officiel de l'hôte.

`h_aliases` est une table, terminée par 0, d'autres noms symboliques de l'hôte.

`h_addrtype` est le type d'adresse (actuellement, toujours `AF_INET`).

`h_length` est la longueur, en octets, de l'adresse réseau.

`h_addr_list` est une table, terminée par 0, de pointeurs vers des adresses réseau pour l'hôte.

`h_addr` est la première adresse dans `h_addr_list` pour respecter la compatibilité ascendante.

Il est possible d'obtenir du DNS une structure `hostent` correspondant à une machine donnée, soit à partir d'un nom symbolique, soit à partir d'une adresse nombres et points, à l'aide des fonctions suivantes :

```
struct hostent *gethostbyname (const char *name);
```

`gethostbyname()` renvoie une structure de type `hostent` pour l'hôte `name`. La chaîne `name` est soit un nom symbolique d'hôte, soit une adresse IPv4 en notation pointée standard. Si `name` est une adresse, aucune recherche supplémentaire n'a lieu et `gethostbyname()` copie simplement la chaîne `name` dans le champ `h_name` et le champ équivalent `struct in_addr` dans le champ `h_addr_list[0]` de la structure `hostent` renvoyée.

```
struct hostent *gethostbyaddr (const struct in_addr * addr,
                               int len, int type);
```

`gethostbyaddr()` renvoie une structure du type `hostent` pour l'hôte dont l'adresse est `addr`. `len` est la longueur de cette adresse. Le seul type d'adresse valide est actuellement `AF_INET`.

Remarque : l'ordre des octets des processeurs i80x86 est LSB (Least significant Byte first ou rangement en mémoire des octets de poids faible en premier ou "little endian"), alors que l'ordre des octets sur l'Internet est MSB (Most Significant Byte first ou rangement en mémoire des octets de poids fort en premier ou "big endian"). De ce fait, toutes les structures de données associées à internet sont en big endian et il faut procéder à des inversions d'ordre Host-to-Network avec `htons()` chaque fois que l'on remplit une telle structure, ou Network-to-Host avec `ntohs()` chaque fois que l'on lit une telle structure.

```
unsigned short int htons (unsigned short int hostshort);
```

`htons()` convertit un entier court (`short`) `hostshort` depuis l'ordre des octets de l'hôte vers celui du réseau.

```
unsigned short int ntohs (unsigned short int netshort);
```

`ntohs()` convertit un entier court (`short`) `netshort` depuis l'ordre des octets du réseau vers celui de l'hôte.

La forme 1 et le nom symbolique sont des chaînes de caractères, donc des `char *`.

Mécanisme d'accès à un service transport Le mécanisme d'accès transport est généralement (sous UNIX ou WINDOWS) une entité de type `socket`. Fonctionnellement, c'est l'analogue d'un tube (`pipe`), mais distribué sur plusieurs machines et avec différents protocoles de transport (UDP, TCP, autres). Une socket est créée par un appel de la fonction `socket()` :

```
int socket(int famille, int type, int protocole);
```

`socket()` crée un point de communication, et renvoie un descripteur, c'est-à-dire un numéro dans la table des fichiers ouverts.

`famille` permet de sélectionner la famille de protocoles à employer :

– `AF_INET` : communication internet
– `AF_UNIX` : communication locale
– `AF_ISO` : communication ISO

`type` permet de sélectionner le protocole :
– `SOCK_STREAM` : mode connecté (TCP pour internet)
– `SOCK_DGRAM` : mode datagramme (UDP pour internet)

`protocole` permet de sélectionner le protocole à utiliser pour la socket, s'il n'est pas déjà déterminé par la famille et le type. Pour les protocoles de la famille internet, ce paramètre vaut IP (=0).

Remarque : après sa création, une socket **n'est pas associée à un SAP**. Cette association doit être faite ensuite.

Une socket peut être fermée à l'aide de la fonction `close()` (fonction standard de fermeture d'un fichier).

Descripteur d'un point d'accès à un service de transport (extrémité en terminologie unix, TSAP en terminologie OSI).

Un point d'accès à un service de transport est désigné par un couple (identification de machine, identification de port) nommé aussi extrémité. Toutefois les identifications de machine et de port sont spécifiques à chaque famille de protocoles. La structure de données générique `sockadrr` permet d'unifier les traitements dans les programmes :

```
struct sockaddr {
    unsigned short int sa_family; /* famille d'adresses réseau */
    unsigned char sa_data[14];    /* adresse proprement dite */
};
```

Cette structure générique doit être rédéfinie pour chacune des familles de protocoles. Pour la famille internet, il faut utiliser `sockaddr_in` de `netinet/in.h` :

```
struct sockaddr_in {
    unsigned short sin_family;
    in_port_t sin_port;
    struct in_addr sin_addr;
    unsigned char sin_zero[sizeof (struct sockaddr) -
                           sizeof (unsigned short int) -
                           sizeof (in_port_t) -
                           sizeof (struct in_addr)];
};
```

`sin_family` indique la famille internet, `sin_port` le numéro de port, `sin_addr` l'adresse (format binaire).

```
struct in_addr {
    in_addr_t s_addr;
};
```

`sin_zero[]` est du remplissage pour atteindre la taille générique.

Pendant son fonctionnement une socket est associée à deux extrémités : une extrémité locale et une extrémité distante (extrémité du serveur). Une socket est associée à une extrémité locale soit par une demande d'association explicite grâce à la fonction `bind()` dont un des arguments est un `sockadrr` qui doit être préalablement rempli, soit automatiquement par le système à la suite d'une interaction avec une extrémité distante (fonctions `connect()`, `recvfrom()` ou `sendto()`). Lorsqu'une socket a été associée automatiquement à une extémité locale, il est possible d'obtenir la structure `sockadrr` correspondante grâce à la fonction `getsockname()`.

```
int bind(int sockfd, struct sockaddr *local_ext,
         socklen_t sockaddrlen);
```

`bind()` associe la socket `sockfd` à l'extrémité locale contenue dans `local_ext`, qui doit avoir été remplie au préalable. `sockaddrlen` indique la longueur en octets de la structure pointée par `local_ext`.

Traditionnellement, cette opération est appelée *assignation d'un nom à une socket*. Le terme de nom doit être compris ici comme l'identification d'une extrémité. Dans le cas d'une socket internet c'est un triplet <famille internet, numéro de port, adresse internet>.

```
int connect(int sockfd, struct sockaddr *serv_ext,
            socklen_t sockaddrlen);
```

La fonction `connect()` est principalement utilisée pour les processus clients orientés connexion.

Si la socket est du type `SOCK_DGRAM`, cette fonction indique le correspondant avec lequel la socket doit communiquer, c'est l'extrémité à laquelle les datagrammes seront envoyés, et la seule extrémité depuis laquelle les datagrammes seront reçus.

Si la socket est du type `SOCK_STREAM`, cette fonction tente de se connecter à une autre socket dont l'adresse doit être indiquée par `serv_ext`. Cette autre socket doit être dans le même domaine que la socket initiale.

```
int sendto(int s, const void *msg, size_t len, int flags,
           const struct sockaddr *to, socklen_t tolen);
```

La fonction `sendto()` permet de transmettre un message à destination d'une autre socket.

```
ssize_t write(int fildes, void *buf, size_t nbyte);
```

La fonction `write()` essaie d'écrire `nbyte` octets dans le fichier référencé par le descripteur `fildes` à partir de la chaîne `buf`. Elle renvoie le nombre d'octets effectivement écrits.

```
int recvfrom(int s, void *buf, int len, unsigned int flags,
             struct sockaddr *from, socklen_t *fromlen);
```

La fonction `recvfrom()` est utilisée pour recevoir des messages depuis une socket `s`, et peut servir à la lecture de données que la socket soit orientée connexion ou non.

```
ssize_t read(int fildes, void *buf, size_t nbyte);
```

La fonction `read()` essaie de lire `nbyte` octets à partir du descripteur de fichier `fildes` et place la chaîne lue dans `buf`. Elle renvoie le nombre d'octets effectivement lus.

```
int getsockname(int s, struct sockaddr *local_tsap,
                socklen_t *namelen)
```

La fonction `getsockname()` renseigne l'identificateur d'extrémité `sockaddr local_ext` de la socket indiquée. Le paramètre `namelen` doit être initialisé pour indiquer la taille de la zone mémoire pointée par `name`. En retour, il contiendra la taille effective (en octets) du nom renvoyé.

```
int listen (int s, int backlog);
```

`listen()` informe le système du désir d'accepter des connexions entrantes et de la limite de la file d'entrée. Les connexions seront ensuite acceptées avec `accept()`.

```
int accept(int sock, struct sockaddr *adresse,
           socklent_t *longueur);
```

`accept()` extrait la première connexion de la file des connexions en attente, crée une nouvelle socket avec les mêmes propriétés que `sock` et alloue un nouveau descripteur de fichier pour cette socket. L'argument `adresse` est un paramètre-résultat qui est renseigné avec l'adresse de l'entité se connectant, telle qu'elle est connue par la couche de communication. Le format exact du paramètre `adresse` est fonction du domaine dans lequel la communication s'établit. Le paramètre-résultat `longueur` est renseigné avec la longueur (en octets) de l'adresse retournée. Ce paramètre doit initialement contenir la longueur du paramètre `adresse`. Si `adresse` est `NULL`, rien n'est écrit. S'il n'y a pas de connexion en attente dans la file, et si la socket est bloquante, `accept()` se met en attente d'une connexion. Si la socket est non-bloquante, et qu'aucune connexion n'est présente dans la file, `accept()` retourne une erreur. Une socket acceptée ne peut pas être utilisée pour accepter de nouvelles connexions. La socket originale `sock` reste ouverte.

Exercice 1.1 : Traduction nom symbolique vers adresse IP et inversement

Question 1 : Étudier le programme suivant qui, à partir d'un nom symbolique d'une machine donné dans la ligne de commande, affiche son nom officiel, ses alias, et sa ou ses adresses IP en notation pointée.

```
#include <stdio.h>
#include <netdb.h>

int main(int argc, char** argv) {
 struct hostent *host;
 int i;

 if (argc != 2) {
     fprintf(stderr,"usage: %s host\n",argv[0]);
     exit(1);
 }
 host = gethostbyname(argv[1]);
 printf("Nom officiel : %s\n",host->h_name);
 for (i = 0; host->h_aliases[i] != NULL; i++)
     printf("Alias : %s\n",host->h_aliases[i]);
 for (i = 0; host->h_addr_list[i] != NULL; i++)
     printf("Adresse : %s\n",
            inet_ntoa(*(struct in_addr *)host->h_addr_list[i]));
 return 0;
}
```

Question 2 : Étudier le programme ci-après qui fait la même chose que le précédent, mais à partir de l'adresse IP en notation pointée.

```
#include <stdio.h>
#include <netdb.h>
#include <sys/socket.h>

int main(int argc, char** argv) {
 struct hostent *host;
 struct in_addr addr;
 int i;

 if (argc != 2) {
```

```
    fprintf(stderr,"usage: %s addr\n",argv[0]);
    exit(1);
 }
 addr.s_addr = inet_addr(argv[1]);
 host = gethostbyaddr((char *)&addr,sizeof(addr),AF_INET);
 printf("Nom officiel : %s\n",host->h_name);
 for (i = 0; host->h_aliases[i] != NULL; i++)
    printf("Alias : %s\n",host->h_aliases[i]);
 for (i = 0; host->h_addr_list[i] != NULL; i++)
    printf("Adresse : %s\n",
           inet_ntoa(*(struct in_addr *)host->h_addr_list[i]));
 return 0;
}
```

Exercice 1.2 : Client/serveur UDP

1. Compléter les programmes suivants pour faire communiquer un serveur et un client UDP.

```
 1  /* serveuru.c  (serveur UDP) */
 2
 3  #include <stdio.h>
 4  #include <errno.h>
 5  #include <string.h>
 6  #include <netinet/in.h>
 7
 8  char* id = 0;
 9  short port = 0;
10  int sock = 0; /* socket de communication */
11  int nb_reponse = 0;
12
13  int main(int argc, char** argv) {
14   int ret;
15   struct sockaddr_in serveur; /* SAP du serveur */
16
17   if (argc!=3) {
18       fprintf(stderr,"usage: %s id port\n",argv[0]);
19       exit(1);
20   }
21   id = argv[1];
22   port = atoi(argv[2]);
23   if ((sock = socket(.....................)) == -1) {
24       fprintf(stderr,"%s: socket %s\n",argv[0],strerror(errno));
25       exit(1);
26   }
27   serveur.sin_family = .......;
28   serveur.sin_port = ...........;
29   serveur.sin_addr.s_addr = ..........;
30   if (bind(.............................................) < 0) {
31       fprintf(stderr,"%s: bind %s\n",
32               argv[0],strerror(errno));
33       exit(1);
34   }
35   while (1) {
```

```
36        struct sockaddr_in client; /* SAP du client */
37        int client_len = sizeof(client);
38        char buf_read[1<<8], buf_write[1<<8];
39
40        ret = recvfrom(.....................
41                       ....................................);
42        if (ret <= 0) {
43            printf("%s: recvfrom=%d: %s\n",
44                   argv[0],ret,strerror(errno));
45            continue;
46        }
47        printf("serveur %s a recu le message %s de %s:%d\n",id,buf_read,
48               ..............................................);
49        sprintf(buf_write,"serveur#%2s reponse%03d#",
50                id,nb_reponse++);
51        ret = sendto(.....................................
52                     ....................................);
53        if (ret <= 0) {
54            printf("%s: sendto=%d: %s\n",
55                   argv[0],ret,strerror(errno));
56            continue;
57        }
58        sleep(2);
59    }
60  return 0;
61  }
```

```
1   /* clientu.c (client UDP) */
2
3   #include <stdio.h>
4   #include <errno.h>
5   #include <netinet/in.h>
6   #include <string.h>
7
8   char* id = 0;
9   short sport = 0;
10  int sock = 0; /* socket de communication */
11
12  int main(int argc, char** argv) {
13   struct sockaddr_in moi; /* SAP du client */
14   struct sockaddr_in serveur; /* SAP du serveur */
15   int nb_question = 0;
16   int ret,len;
17   int serveur_len = sizeof(serveur);
18   char buf_read[1<<8], buf_write[1<<8];
19
20   if (argc != 4) {
21       fprintf(stderr,"usage: %s id host sport\n",argv[0]);
22       exit(1);
23   }
24   id = argv[1];
25   sport = atoi(argv[3]);
26   if ((sock = socket(...................)) == -1) {
27       fprintf(stderr,"%s: socket %s\n",argv[0],strerror(errno));
```

```
28          exit(1);
29      }
30      len = sizeof(moi);
31      getsockname(.............................);
32      serveur.sin_family = .......;
33      serveur.sin_port = ............;
34      ..................................&serveur.sin_addr);
35      while (nb_question < 3) {
36          char buf_read[1<<8], buf_write[1<<8];
37
38          sprintf(buf_write,"#%2s=%03d",id,nb_question++);
39          printf("client %2s: (%s,%4d) envoie a ",
40                 id,.....................................);
41          printf("(%s,%4d): %s",
42                 ................................................,
43                 .........);
44          ret = sendto(...................................,
45                       .....................................);
46          if (ret <= 0) {
47              printf("%s: erreur dans sendto (num=%d, mess=%s)\n",
48                      argv[0],ret,strerror(errno));
49              continue;
50          }
51          len = sizeof(moi);
52          getsockname(.............................);
53          printf("client %2s: (%s,%4d) recoit de ",
54                 id,.....................................);
55          ret = recvfrom(.................,
56                         .....................................);
57          if (ret <= 0) {
58              printf("%s: erreur dans recvfrom (num=%d, mess=%s)\n",
59                     argv[0],ret,strerror(errno));
60              continue;
61          }
62          printf("(%s,%4d) : %s\n",.......................,
63                 .............................);
64      }
65      return 0;
66  }
```

2. Lancez le serveur. Que se passe-t-il ?
3. La commande `netstat` permet de voir les sockets ouvertes sur le système et leur état. En vous aidant du manuel de `netstat`, cherchez des informations sur la socket ouverte par votre serveur.
4. Lancez Wiresharke et capturez tout le traffic UDP dirigé vers le port sur lequel votre serveur écoute.
5. Lancez votre client. Que se passe-t-il côté serveur ?
6. Regardez ce qui a été capturé par Wireshark. Que constatez-vous ?
7. Après l'exécution de votre client, regardez à nouveau l'état de la socket du serveur avec `netstat`. Que constatez-vous ?

Exercice 1.3 : Client/serveur TCP

1. Compléter les programmes suivants pour faire communiquer un serveur et un client TCP.

```
1   /* serveurt.c (serveur TCP) */
2
3   #include <stdio.h>
4   #include <errno.h>
5   #include <netinet/in.h>
6   #include <string.h>
7
8   #define NBECHANGE 3
9
10  char* id = 0;
11  short port = 0;
12  int sock = 0; /* socket de communication */
13  int nb_reponse = 0;
14
15  int main(int argc, char** argv) {
16   struct sockaddr_in serveur; /* SAP du serveur */
17
18   if (argc != 3) {
19       fprintf(stderr,"usage: %s id port\n",argv[0]);
20       exit(1);
21   }
22   id = argv[1];
23   port = atoi(argv[2]);
24   if ((sock = socket(.....................)) == -1) {
25       fprintf(stderr,"%s: socket %s\n",argv[0],strerror(errno));
26       exit(1);
27   }
28   serveur.sin_family = .......;
29   serveur.sin_port = ...........;
30   serveur.sin_addr.s_addr = ..........;
31   if (bind(.............................................) < 0) {
32       fprintf(stderr,"%s: bind %s\n",argv[0],strerror(errno));
33       exit(1);
34   }
35   if (listen(......) != 0) {
36       fprintf(stderr,"%s: listen %s\n",argv[0],strerror(errno));
37       exit(1);
38   }
39   while (1) {
40       struct sockaddr_in client; /* SAP du client */
41       int len = sizeof(client);
42       int sock_pipe; /* socket de dialogue */
43       int ret,nb_question;
44
45       sock_pipe = accept(.................................);
46       for (nb_question = 0 ; nb_question < NBECHANGE ;
47           nb_question++) {
48           char buf_read[1<<8], buf_write[1<<8];
49
50           ret = read(.....................);
51           if (ret <= 0) {
52               printf("%s: read=%d: %s\n",
```

```
53                      argv[0], ret, strerror(errno));
54                  break;
55              }
56              printf("serveur %s recu de (%s,%4d) : %s\n",id,
57                     ......................,
58                     ............................);
59              sprintf(buf_write,"#%2s=%03d#",id,nb_reponse++);
60              ret = write(.....................................);
61              if (ret <= 0) {
62                  printf("%s: write=%d: %s\n",
63                         argv[0], ret, strerror(errno));
64                  break;
65              }
66              sleep(2);
67          }
68          close(sock_pipe);
69      }
70      return 0;
71  }
```

```
1   /* clientt.c (client TCP) */
2
3   #include <stdio.h>
4   #include <errno.h>
5   #include <netinet/in.h>
6   #include <string.h>
7
8   #define NBECHANGE 3
9
10  char* id = 0;
11  short sport = 0;
12  int sock = 0; /* socket de communication */
13
14  int main(int argc, char** argv) {
15      struct sockaddr_in moi; /* SAP du client */
16      struct sockaddr_in serveur; /* SAP du serveur */
17      int nb_question = 0;
18      int ret,len;
19
20      if (argc != 4) {
21          fprintf(stderr,"usage: %s id serveur port\n",argv[0]);
22          exit(1);
23      }
24      id = argv[1];
25      sport = atoi(argv[3]);
26      if ((sock = socket(.....................)) == -1) {
27          fprintf(stderr,"%s: socket %s\n",argv[0],strerror(errno));
28          exit(1);
29      }
30      serveur.sin_family = AF_INET;
31      serveur.sin_port = htons(sport);
32      inet_aton(.....................................);
33      if (connect(.......................................... ) < 0) {
34          fprintf(stderr,"%s: connect %s\n",argv[0],strerror(errno));
```

```
35        exit(1);
36    }
37    len = sizeof(moi);
38    getsockname(.............................);
39    for (nb_question = 0 ; nb_question < NBECHANGE ;
40         nb_question++) {
41        char buf_read[1<<8], buf_write[1<<8];
42
43        sprintf(buf_write,"#%2s=%03d",id,nb_question);
44        printf("client %2s: (%s,%4d) envoie a ",
45               id,.....................................);
46        printf(" (%s,%4d) : %s\n",.....................,
47               ............................);
48        ret = write(............................);
49        if (ret <= strlen(buf_write)) {
50            printf("%s: erreur dans write (num=%d, mess=%s)\n",
51                   argv[0],ret,strerror(errno));
52            continue;
53        }
54        printf("client %2s: (%s,%4d) recoit de ",
55               id,.....................................);
56        ret = read(.................);
57        if (ret <= 0) {
58            printf("%s:  erreur dans read (num=%d, mess=%s)\n",
59                   argv[0],ret,strerror(errno));
60            continue;
61        }
62        printf("(%s,%4d) : %s\n",inet_ntoa(serveur.sin_addr),
63               ntohs(serveur.sin_port),buf_read);
64    }
65    close(sock);
66    return 0;
67  }
```

2. Lancez le serveur. Que se passe-t-il ?
3. La commande `netstat` permet de voir les sockets ouvertes sur le système et leur état. En vous aidant du manuel de `netstat`, cherchez des informations sur la socket ouverte par votre serveur.
4. Lancez Wiresharke et capturez tout le traffic TCP dirigé vers le port sur lequel votre serveur écoute.
5. Lancez votre client. Que se passe-t-il côté serveur ?
6. Regardez ce qui a été capturé par Wireshark. Combien de paquets sont passés dans la communication ? Que constatez-vous ?
7. Après l'exécution de votre client, regardez à nouveau l'état de la socket du serveur avec `netstat`. Que constatez-vous ?
8. Modifiez votre programme serveur pour ajouter l'instruction `sleep()` *avant* la fermeture de la socket pour endormir votre serveur pendant quelques secondes pendant que la socket est ouverte. Vous vous aiderez si besoin du manuel de `sleep()` (`man 3 sleep`).
9. Relancez votre serveur et le client. Avant la fin de l'exécution du client, regardez l'état des sockets associées au serveur avec `netstat`. Que constatez-vous ? Combien y en a-t-il ?