Introduction

Il y a 2 modes de communications: le mode connecté et le mode *datagram*. Il y en 4 types:

1. en mode connecté à travers Internet : TCP
2. en mode connecté à travers Unix : TCP
3. en mode *datagram* à travers Internet : UDP
4. en mode *datagram* à travers Unix : UDP

TCP ou UDP sont 2 types de socket : `SOCK_STREAM` et `SOCK_DGRAM`.
Internet et Unix sont 2 familles de socket: `AF_INET` et `AF_UNIX`.

En mode `AF_INET`, ce sont les numéros de port qui donnent le point de rendez-vous.
En mode `AF_UNIX`, ce sont des noms de fichiers qui donnent le point de rendez-vous.

Les structures `sockaddr*`

Une socket est décrite par:

- un descripteur de fichier. C'est un entier qui servira aux opérations `read/write` ou `recvfrom/sendto` selon le protocole TCP ou UDP.
- les paramètres de connexion. Suivant la famille, c'est la structure:
  - **sockaddr_in** : de famille `AF_INET`:

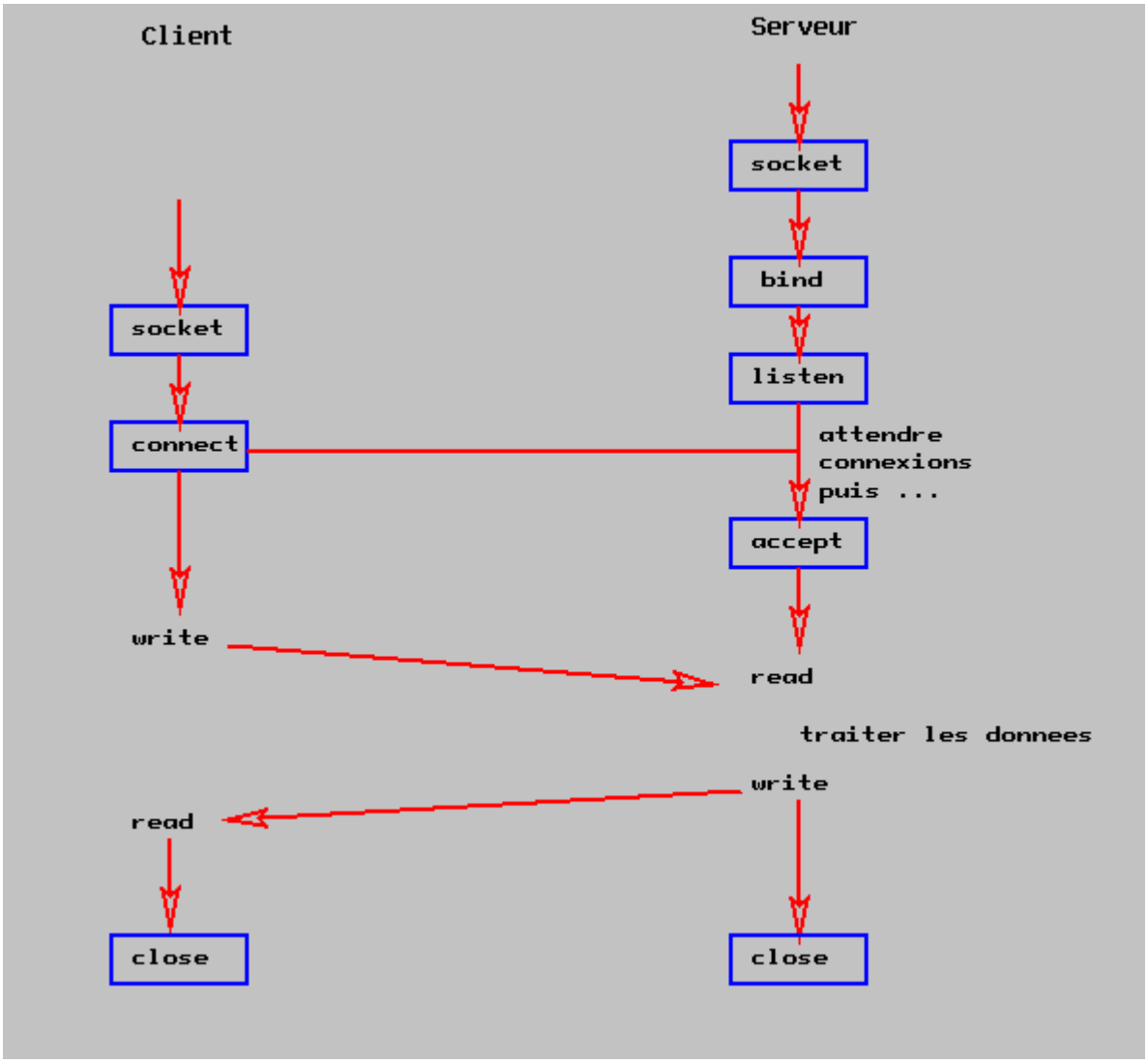
```
/* /usr/include/netinet/in.h */
struct in_addr {
    union {
            struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
            struct { u_short s_w1,s_w2; } S_un_w;
            u_long S_addr;
} S_un;
#define s_addr  S_un.S_addr     /* should be used for all code */
struct sockaddr_in {
    short   sin_family;         /* must be AF_INET */
    u_short sin_port;
    struct  in_addr sin_addr;
    char    sin_zero[8];
};
```

  - **sockaddr_un** : de famille `AF_UNIX`:

```
/* /usr/include/sys/un.h */
struct  sockaddr_un {
        short   sun_family;     /* AF_UNIX */
        char    sun_path[108];      /* path name (gag) */
};
```

Protocole TCP

Le schéma est le suivant:



Serveur et client : appel `socket`

Créer une socket, i.e. le descripteur de fichier:

- en C:

```
int sockfd;
sockfd = socket(AF_INET, SOCK_STREAM, 0);
        /* et l'autre variante : AF_UNIX */
if (sockfd < 0) /* error */
```

- en Perl:

```
require "sys/socket.ph";
socket(S,&AF_INET,&SOCK_STREAM,0) || # error;
```

Le dernier paramètre (ici 0) sert à spécifier quand on ne veut pas des protocoles classiques. Exemple: pour traiter des paquets ICMP, ce 0 devrait être remplacé par `IPPROTO_ICMP`.

Client : appel `connect`

Le client demande à se connecter au port P de la machine M.

- en C :

```
struct sockaddr_in that;
bzero(&that,sizeof(that)); /* en BSD */
memset(&that,0,sizeof(that)); /* en System V */
that.sin_family = AF_INET;
that.sin_port = htons(7000);
that.sin_addr.s_addr = inet_addr("129.199.129.1");
if (connect(sockfd, &that, sizeof(that)) < 0) /* error */
```

L'appel `htons` (*Host TO Network Short*) met en format réseau l'entier court 7000.
L'appel `inet_addr` génère le format ad hoc d'une adresse IP depuis une chaîne de caractères. L'appel gethostent est une autre manière de retrouver les numéros via des noms.

- en Perl, la notion de structure n'existe pas. On utilise donc la fonction `pack` pour créer une variable style chaîne de caractères (en fait une suite d'octets) qui correspond à la structure `sockaddr_in`:

```
$sockaddr = "S n a4 x8";
        # S = unsigned short, n = short in network order
        # a4 = 4 caracteres, x8 = 8 NULL
        # espaces non significatifs
$port = 7000;
$thataddr = pack("C4",129,199,129,1);
$that = pack($sockaddr,&AF_INET, $port, $thataddr);
connect(S,$that);
```

NB: avec la famille `AF_UNIX` tout est pareil sauf:

- la structure qui est `sockaddr_un`
- on copie le nom de fichier dedans:

```
that.sun_family = AF_UNIX;
strcpy(that.sun_path,"le/nom/du/fichier");
```

Client : `read/write`

Le client se contente ensuite de faire des entrées/sortie classiques sur le descripteur de fichier, `sockfd` en C, `S` en Perl. Il y a néanmoins un piège: se méfier des bufférisations si l'on veut dialoguer!

- en C, il faut faire des lectures et reconstituer soi-même des lignes. On va donc lire caractère par caractère jusqu'au séparateur:

```
while (buffer non plein) {
        if ( (rc = read(fdsock, &c, 1)) == 1) {
                *ptr++ = c;
                if (c == '\n') break;
        }
}
```

- en Perl, on se mettra en non-bufférisé et on lit des lignes via `<S>`:

```
select(S); $| = 1;      # sic...
```

Serveur : appel `bind`

Le serveur remplit une structure `sockaddr_in` où il indique sur quel port il attend:

- en C:

```
struct sockaddr_in this;
bzero(&this,sizeof(this));
this.sin_family = AF_INET;
this.sin_port = htons(7000);
this.sin_addr.s_addr = htonl(INADDR_ANY);
if (bind(sockfd,&this,sizeof(this) < 0) /* error */
```

- en Perl:

```
$sockaddr = "S n a4 x8";
$port = 7000;
$this = pack($sockaddr,&AF_INET,$port,"\0\0\0\0");
bind(S,$this) || # error;
```

Serveur : appel `listen`

On indique au système que l'on attend des appels. Un paramètre supplémentaire indique la taille d'une file d'attente que peut gérer le système: quand cette file d'attente est pleine, les demandes de connexions supplémentaires sont ignorées, donc elles génèrent un *time-out*.

- en C:

```
if (listen(s,5) < 0) /* error */
```

- en Perl:

```
listen(S,5) || # error...
```

Serveur : appel `accept`

Cet appel fournit une nouvelle socket, laissant donc ainsi libre la socket d'origine pour d'autres appels: c'est cette nouvelle socket qui est utilisée pour le dialogue entre le client et le serveur.

- en C :

```
int newsockfd, lg;
struct sockaddr_in that;
lg = sizeof(that);
newsockfd = accept(sockfd, &that, &lg);
```

- en Perl:

```
$that = accept(NS,S);
```

La structure `that`, à décoder comme il se doit avec `unpack` en Perl, contient le port et l'adresse IP de la machine cliente: filtrer, comptabiliser sont donc possibles.

Serveur : traiter...

Il y a essentiellement 2 méthodes de traitement:

- je traite moi-même. Ceci a un sens si le traitement est rapide, car pendant ce temps, la file d'attente des demandes de connexions peut s'allonger.
- je me dépêche de créer un processus-fils qui traite lui-même:
  - en C:

```
for (;;) {
```

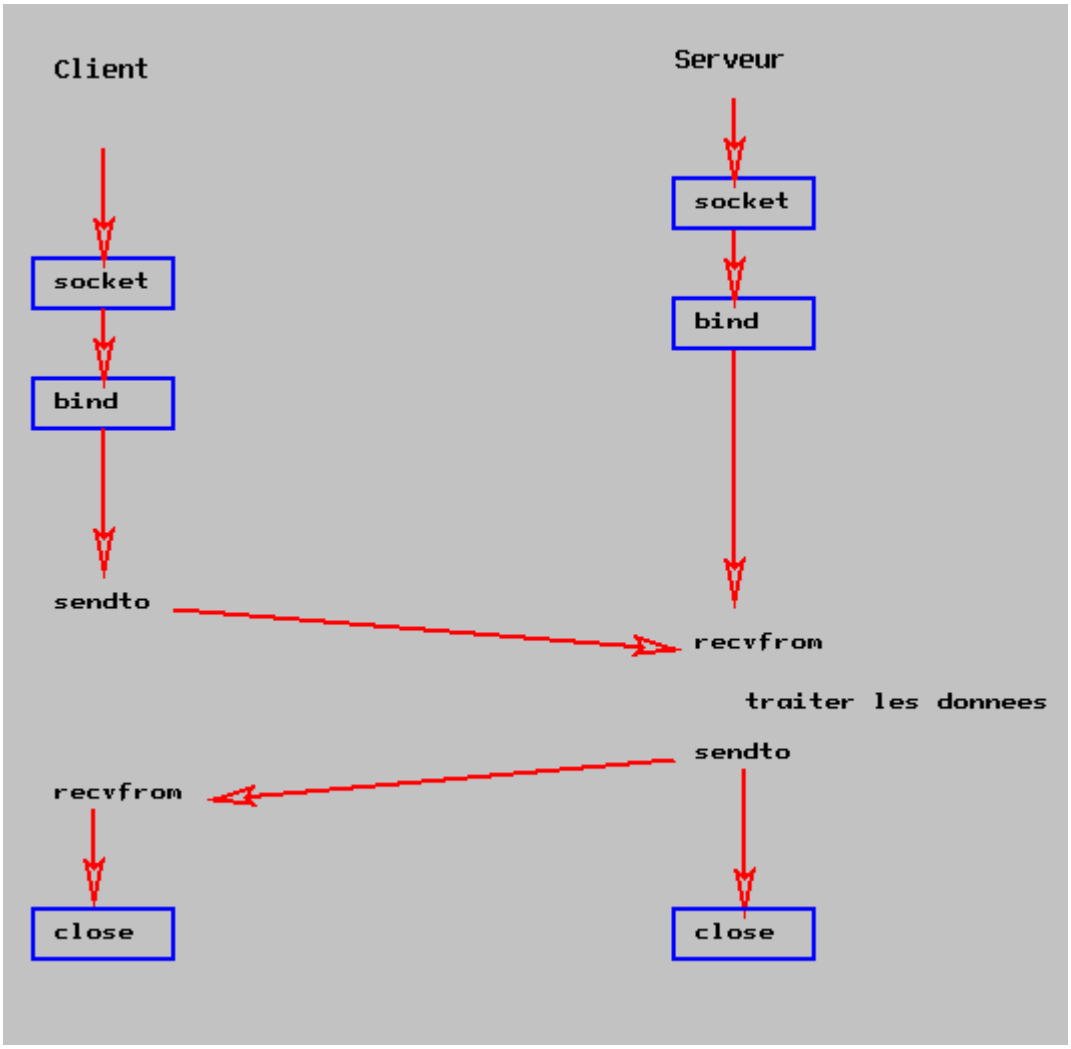
```
                newsockfd = accept(sockfd, &that, &lg);
                if (newsockfd < 0) /* accept error */
                if ((pid = fork()) < 0) /* fork error ! */
                if (pid == 0) {
                        close(sockfd);
                        /* traiter newsockfd */
                        close(newsockfd);
                        exit(0);
                }
                close(newsockfd);
        }
```

- en Perl:

```
        for(;;) {
                $that = accept(S,NS);
                if (($pid = fork) < 0) # fork error...
                if ($pid == 0) {
                        close(S);
                        .... traiter NS ...
                }
                close(NS);
        }
```

Protocole UDP

Le schéma est le suivant:



Serveur et client : appel `socket`

Créer une socket, i.e. le descripteur de fichier:

- en C:

```
int sockfd;
sockfd = socket(AF_INET, SOCK_DGRAM, 0);
        /* et l'autre variante : AF_UNIX */
if (sockfd < 0) /* error */
```

- en Perl:

```
require "sys/socket.ph";
socket(S,&AF_INET,&SOCK_DGRAM,0) || # error;
```

Client : appel `bind`

Il prend une adresse quelconque du côté client:

- en C :

```
struct sockaddr_in &this;
bzero(&this,sizeof(this));
this.sin_family = AF_INET;
this.sin_port = htons(0);
this.sin_addr.s_addr = hotnl(INADDR_ANY);
```

```
if (bind(sockfd,&this,sizeof(this)) < 0) /* error */
```

- en Perl:

```
$this = pack($sockaddr,&AF_INET,0,"\0\0\0\0");
bind(S,$this) || # error
```

Client : appels `sendto` et `recvfrom`

Ces 2 appels permettent en une fois de spécifier buffer et destination des paquets.

- en C:

```
struct sockaddr_in that, newthat;
bzero(&that,sizeof(that));
that.sin_family = AF_INET;
that.sin_port = htons(7000);
that.sin_addr.s_addr = inet_addr("129.199.96.11");
.... on veut emettre buffer sur n octets ...
if (sendto(sockfd,buffer,n,0,&that,sizeof(that) < 0) /* error */
n = recvfrom(sockfd,buffer,sizeof(buffer),0,&newthat,sizeof(newthat));
```

- en Perl:

```
socket(S,&AF_INET,&SOCK_DGRAM,0);
$port = 7000;
$thataddr = pack("C4",129,199,96,11);
$that = pack($sockaddr,&AF_INET,$port,$thataddr);
send(S,$msg,0,$that);
$newthat = recv(S,$buf,100,0);
```

`newthat` est rempli avec les paramètres réseau (port, adresse IP) du message reçu.

Serveur : appel `bind`

Le serveur déclare le port de réception:

- en C :

```
struct sockaddr_in &this;
int sockfd;
if ((sockfd = socket(AF_INET,SOCK_DGRAM,0) < 0) /* error */
bzero(&this,sizeof(this));
this.sin_family = AF_INET;
this.sin_port = htons(7000);
this.sin_addr.s_addr = hotnl(INADDR_ANY);
```

- en Perl:

```
$this = pack($sockaddr,&AF_INET,7000,"\0\0\0\0");
bind(S,$this) || # error
```

Serveur : appels `sendto` et `recvfrom`

L'ordre est inverse du client, les principes sont identiques:

- en C:

```
struct sockaddr_in that;
n = recvfrom(sockfd,buffer,sizeof(buffer),0,&that,sizeof(that));
... traiter le buffer ...
if (sendto(sockfd,buffer,n,0,&that,sizeof(that) < 0) /* error */
```

- en Perl:

```
$newthat = recv(S,$buf,100,0);
... traiter buf ...
send(S,$buf,0,$that);
```

Des exemples en C

- Makefile
- inet.h
- unix.h
- client-tcp-inet.c
- server-tcp-inet.c
- client-tcp-unix.c
- server-tcp-unix.c
- client-udp-inet.c
- server-udp-inet.c
- client-udp-unix.c
- server-udp-unix.c

et les fichiers accessoires:

- dg-cli.c
- dg-echo.c
- err_dump.c
- readline.c
- readn.c
- str-cli.c
- str-echo.c
- writen.c

Un exemple en Perl

Récupérer la liste des newsgroups

Tester?

Le mieux est de faire un dialogue avec du texte. Ainsi on peut simuler un client avec:

```
$ telnet HOST PORT
```

Exemple:

```
telnet clipper echo
telnet clipper discard
telnet clipper daytime
telnet clipper finger
```

```
        # mettre ici une ligne vide ou le nom de quelqu'un
```

Exemple de dialogue: `rmt` permet de piloter une bande à distance, ce qui peut servir aux sauvegardes. Les commandes sont:

- `'Opathname\nmode\n' : ouvrir le fichier`
- `'Cpathname\n' : fermer`
- `'Wcount\ndata' : ecrire count octets pris dans data`

et les réponses sont simples:

- 'Ax' : tout va bien, x est un code de retour.
- 'Ex' : erreur x.

Des routines annexes

gethostent

Ou comment ne pas manipuler des numéros de machines, mais plutôt des noms....
Il y a 3 sources d'informations possibles: le fichier `/etc/hosts`, les pages jaunes (`ypcat hosts`), les serveurs de noms appelés DNS (*Domain Name Services*). La configuration du système permet de rendre transparent les appels à ces 3 sources. Il reste 2 appels:

- en C:

```
struct hostent *gethostbyname(name);
struct hostent *gethostbyaddr(addr, len, type);
struct    hostent {
        char *h_name;  /* official name of host */
        char **h_aliases;   /* alias list */
        int  h_addrtype;    /* address type */
        int  h_length; /* length of address */
        char **h_addr_list; /* list of addresses from name server */
};
```

- en Perl:

```
($name,$aliases,$addrtype,$length,@addrs) = gethostbyname($hostname);
# $aliases est la liste des noms separes par un espace
# souvent une adresse suffit donc on ecrit:
($name,$aliases,$addrtype,$length,$addr) = gethostbyname($hostname);
# et $addr contient l'adresse cherche
```

getservbyname

Ou comment retrouver des numéros de services à partir des noms.

setsockopt et getsockopt

Pour positionner des options sur une socket. 2 exemples:

- `SO_KEEPALIVE`: quand on a connecté une socket en mode stream, aucune information n'est échangé entre les 2 parties. Ainsi il se peut que le client meurt sans que le serveur soit mis au courant; ainsi des ressources restent mobilisés sur le serveur pour rien. L'option `SO_KEEPALIVE` demande au système de vérifier régulièrement que le client est toujours là. Ceci se

fait toutes les 2 heures.
- `SO_LINGER` : quand on appelle `close`, doit-on considérer ce qui n'est pas encore transmis comme inutile ou doit-on le transmettre?

getsockname

Pour retrouver les paramètres réseau de son côté.

getpeername

Pour retrouver les paramètres réseau du côté distant.