Travaux Pratiques Réseaux

A. Habibi

Les sockets

Définitions, Analogies

Formellement, une *socket* est un point de communication bidirectionnel par lequel un processus pourra émettre ou recevoir des informations.

Moins formellement, au lieu de parler de la communication entre processus, parlons de la communication entre personnes par téléphone ou par courrier. Cette communication nécessite que les personnes disposent, soit d'un poste de téléphone, soit d'une boîte aux lettres. Le poste de téléphone et la boîte aux lettres sont des *points de communication* ayant des adresses connues du monde extérieur.

Les sockets sont l'analogue de ces "points de communications". Les processus sont l'analogue des individus. Lorsque deux processus veulent se communiquer des données, il faut que chacun d'eux dispose d'au moins une socket. Les sockets ne sont certes pas le seul moyen de communication entre deux processus mais un grand nombre d'applications sont fondées sur les sockets.

Les sockets sont un exemple de point d'accès aux services (*SAP*) de la couche Transport (*SAP4*).

1 Comment définir une socket ?

```
#include<sys/types.h>
#include<sys/socket.h>

int socket(
        int domaine,    /* AF_UNIX ou AF_INET        */
        int type,       /* SOCK_DGRAM ou SOCK_STREAM */
        int protocole   /* 0 : protocole par defaut  */
        );
```

Cette primitive est définie dans la librairie `libsocket.a` ou `libsocket.so`.

Les éléments de cette définition seront décrits dans la suite de cette section.

1.1 Le *descripteur* de la socket

1.1.1 Contexte général

Les sockets peuvent être utilisées sous *UNIX* mais aussi dans beaucoup d'autres types de systèmes et de machines. De façon générale, la primitive `socket` crée une socket et rend un entier appelé le *descripteur* de la socket créée. A partir de là, dans le processus qui a fait appel à cette primitive, la socket créée sera toujours désignée et identifiée par ce descripteur.

En particulier, pour manipuler une socket, pour fixer son adresse, pour pouvoir émettre un message sur elle ou y lire un message, le processus devra désigner la socket par son *descripteur*.

1.1.2 Sous *UNIX*

Ce qui est décrit ci-dessus est vrai en particulier pour des processus qui fonctionnent sous *UNIX*.

Mais dans le cas particulier d'un système *UNIX*, les sockets s'insèrent dans un ensemble d'autres objets partageant un grand nombre de propriétés. En particulier, le descripteur d'une socket est de même nature qu'un descripteur de fichier régulier, de terminal, ou de tube.

Tout processus *UNIX* a un tableau de descripteurs correspondant, soit à des fichiers réguliers (une région du disque), soit à des fichiers spéciaux (terminaux, tubes, sockets, etc). En particulier, certains de ces descripteurs sont alloués à l'avance. Le descripteur $d = 0$ correspond à l'entrée standard du processus, le descripteur $d = 1$ correspond à la sortie standard et $d = 2$ correspond à la sortie standard d'erreur. Les autres descripteurs sont alloués au fur et à mesure que le processus ouvre des fichiers, des terminaux, des tubes ou des sockets.

Ainsi, vu de l'extérieur du noyau *UNIX*, ces fichiers spéciaux peuvent souvent interagir avec leur environnement comme s'ils étaient des fichiers réguliers.

Dans beaucoup de cas, ces descripteurs sont obtenus par la primitive `open` et, en général, on peut lire et écrire dans un tube, un terminal ou une socket, de la même façon qu'on lit et écrit dans un fichier, c'est à dire en utilisant les primities `read` et `write` et en fournissant le descripteur du fichier, du terminal ou du tube.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *ref, int mode_ouverture, ... /* mode_t droits */);

#include <unistd.h>
ssize_t read(int desc, void *ptr, size_t nb_octets);

#include <unistd.h>
ssize_t write(int desc, void *ptr, size_t nb_octets);
```

Les points de suspension dans la définition de `open` signifient que la spécification des droits est optionnelle.

Dans le répertoire `habibi/public/ExemplesTP.d` les fichiers `Fichiers.c`, `OpenFile.c` et `Terminaux.c` donnent un exemple de l'utilisation de ces primitives respectivement dans le cas de fichiers réguliers et de terminaux.

Dans le cas des sockets, pour émettre un message il suffit souvent d'écrire le message dans la socket par la primitive `write` comme s'il s'agissait d'un fichier. De même, pour recevoir un message depuis une socket, il suffit quelquefois de lire dans la socket par la primitive `read` comme s'il s'agissait d'un fichier.

Certaines sockets, peuvent même être listées parmi les fichiers réguliers avec la commande `ls`.

1.2 Le *type* de la socket

1.2.1 SOCK_DGRAM *versus* SOCK_STREAM

Très grossièrement, dans le contexte de ce TP on parlera essentiellement des sockets de type `SOCK_DGRAM` et `SOCK_STREAM`. D'autres types possibles de sockets sont définis dans le fichier `<sys/socket.h>`.

1.2.2 Les datagrammes *versus* les streams

Les sockets de type `SOCK_DGRAM` transmettent des datagrammes alors que les sockets de type `SOCK_STREAM` transmettent un flot continu de caractères. Par exemple, si nous voulons transmettre la suite de caractères suivante : "`ABCDEFGHIJKLMNOPQRSTUVWXYZ`", et que cette suite est trop longue pour le desinataire et/ou pour l'expéditeur, alors elle sera émise en plusieurs écritures et/ou lue en plusieurs lectures.

Avec des sockets `SOCK_DGRAM`, si l'expéditeur envoie successivement `ABCDEFGHI` puis `JKLMNOP` et enfin `QRSTUVWXYZ`, alors le destinataire lira *nécessairement* le message en trois fois aussi (en trois *datagrammes*) et ces datagrammes contiendront `ABCDEFGHI`, `JKLMNOP` et `QRSTUVWXYZ`.

Par contre, avec des sockets `SOCK_STREAM`, les regroupements à la lecture et à l'écriture sont indépendants. Quelque soit le regroupement fait en écriture par l'expéditeur, (par exemple `ABCDEFGHI` puis `JKLMNOP` et enfin `QRSTUVWXYZ`), le destinataire pourra lire le message caractère par caractère (`A`, puis `B`, puis `C`, ...), en une seule fois (`ABCDEFGHIJKLMNOPQRSTUVWXYZ`), en groupant les caracères de la même manière que l'expéditeur, ou de manière complètement différente (par exemple `ABCDE` puis `FGIJKLM` puis `NOP` et enfin `QRSTUVWXYZ`.)

1.2.3 Mode connecté *versus* mode non-connecté

Par ailleurs et de manière indépendante, les sockets de type `SOCK_DGRAM` fonctionnent en mode non-connecté. Il s'ensuit que la destination d'un datagramme ne doit pas nécessairement être la même que celle du datagramme suivant. Ceci implique qu'à chaque émission de datagramme, on doit spécifier l'adresse de destination.

Par contre, les sockets `SOCK_STREAM` fonctionnent en mode connecté. La transmission de l'information est précédée d'une phase de *connexion* où sont fixées une fois pour toutes les adresses et autres paramètres régissant les échanges. Ensuite, il ne reste plus qu'à émettre et recevoir les données par `read` et par `write`. En particulier, contrairement à une socket de type `SOCK_DGRAM` (qui peut communiquer avec plusieurs interlocuteurs) une socket `SOCK_STREAM` ne communique qu'avec la socket à laquelle elle s'est connectée.

On compare quelquefois la communication en mode connecté à la communication téléphonique et la communication en mode non-connecté à la communication par courrier.

1.2.4 Fiable *versus* non-fiable

Par ailleurs, et de manière indépendante, les sockets de type `SOCK_DGRAM` offrent une fiabilité minimale, alors que `SOCK_STREAM` offrent une fiabilité maximale.

1.2.5 Attention ! ! !

"De manière indépendante" signifie que la transmission par datagramme n'est pas nécessairement non-fiable et que la transmission par caractères n'est pas nécessairement fiable. D'autre part, la transmission par datagramme n'implique pas le mode non-connecté. Par contre, la transmission par flot de caractères, elle, implique nécessairement le mode connecté.

Par exemple les sockets de type `SOCK_RDM` sont de type datagramme, fonctionnent en mode non-connecté, mais offrent une fiabilité maximale.

Les sockets de type `SOCK_SEQPACKET` sont aussi de type datagramme, mais fonctionnent en mode connecté et offrent aussi une fiabilité maximale.

1.3 Le *domaine* ou la *famille* d'une socket

1.3.1 global *versus* local : Analogie

Une boîte aux lettres n'a d'utilité que si d'autres personnes peuvent situer et désigner votre boîte aux lettres sans ambiguïté. Il faut qu'ils aient une adresse qui distinguent votre boîte aux lettres de toutes les autres qui seraient susceptibles d'être desservi par le même service de courrier.

S'il s'agit d'un courrier interne à une entreprise, une adresse comme "`Pierrot, batiment A bureau 322`" pourrait convenir. Si l'acheminement est assuré par la poste (pour une destination extérieure à l'entreprise), alors l'adresse ci-dessus n'est pas suffisante. Il y a beaucoup de bâtiments A et de bureaux 322 dans le monde. Il faut donc une adresse qui puisse vous situer dans le monde, comme par exemple

```
          Pierre Lachapelle
           Societe Taarna
       batiment A, bureau 322
   5057 rue Sainte Catherine Ouest
     H2V 3J7 Montreal (Quebec)
               CANADA
```

Par contre, si à vos interlocuteurs potentiels vous dites "Ma boîte aux lettres se trouve `juste après la porte d'entrée à droite sous les escaliers`" vous ne les aidez pas beaucoup (ou alors vous ne tenez pas à recevoir du courier de leur part.) Ce que vous leur avez donné, c'est l'adresse que *VOUS* utilisez pour aller chercher votre courrier. Mais cette adresse n'est utile à personne d'autre. Il existe des centaines de milliers de foyers qui ont des boîtes aux lettres "`juste après l'entrée, à droite sous les escaliers`".

1.3.2 Revenons à nos sockets

Là aussi, il faut une adresse qui puisse situer la socket sans ambiguité dans un certain domaine. Les sockets peuvent être utilisées sur différents domaines. A chaque fois il est impératif que l'adresse soit unique dans le cadre de ce domaine. Donc à chaque domaine correspond un système d'adressage. Les sockets peuvent s'adapter à tous les formats d'adressage.

Le descripteur : une adresse ?

Le descripteur d'une socket (celui fourni par la primitive `socket`) sert à désigner la socket, mais seulement dans le cadre d'un même processus. En effet, ce descripteur n'est unique que dans le cadre du processus qui a ouvert la socket (et éventuellement des processus fils). Par exemple, si processus P_1 ouvre une socket et obtient le descripteur $d = 3$, alors aucun autre fichier, terminal ou tube ouvert par P_1 ne pourra avoir le même descripteur. Mais par ailleurs une socket, un fichier, un terminal ou un tube ouvert par un autre processus pourra très bien être désigné par le descripteur $d = 3$. Ainsi ne serait-ce que dans le cadre d'une même machine, il peut y avoir énormément de sockets avec le descripteur $d = 3$. Donc si un processus P_2 veut se connecter sur la socket de P_1 pour échanger des informations, il ne lui sera d'aucun secours de savoir que le processus P_1 la désigne par le descripteur $d = 3$.

Le descripteur est l'analogue de l'adresse “`après l'entrée, à droite sous les escaliers`”. Autrement dit, le descripteur est une *adresse locale au processus*. Nous n'utiliserons jamais le descripteur autrement que de manière interne à un processus.

Les *familles d'adresse*

S'il s'agit d'échanger des informations seulement entre des processus d'un même système *UNIX*, alors c'est seulement dans ce système que l'adresse doit être unique. Ainsi, on peut situer une socket en la liant à un répertoire comme un fichier régulier. Son adresse n'est rien d'autre qu'une chaîne de caractère, qui montre sa position dans cette arborescence. Par exemple “`/usr/people/dupont/socket1`”. Il s'agit de l'analogue de l'adresse “`Pierrot, batiment A, bureau 322`”.

Ce domaine et ce format d'adresse sont désignés par la constante symbolique `AF_UNIX`.

Si au contraire, il s'agit d'échanger des informations entre des processus tournant sur des machines distantes, alors chaque socket active doit avoir une adresse unique sur tout le réseau. Si le réseau en question est *INTERNET*, alors chaque socket active doit avoir une adresse unique au monde. Nous allons voir que dans le domaine *INTERNET*, l'adresse d'une socket est composée de l'adresse IP de la machine sur laquelle elle se trouve (l'adresse IP est unique au monde) et aussi d'un numéro de port qui permet de distinguer cette socket des autres sockets actives de la même machine. Il s'agit donc de l'analogue de l'adresse générale : “`5057 rue Sainte Catherine Ouest ... Montréal CANADA`”

Ce domaine et ce format d'adresse sont désignés par la constante symbolique `AF_INET`.

Ces constantes symboliques sont définies, ainsi que bien d'autres, dans le fichier `<sys/socket.h>`. Citons `AF_APPLETALK` pour *AppleTalk*, `AF_CCITT` pour le monde *CCITT* et les réseaux *X25*, et `AF_OSI` pour le monde *OSI*, etc. “`AF`” signifie *Address Family*.

Dans la suite de ce TP, nous nous intéresserons essentiellement à la famille `AF_INET` et très accessoirement à la famille `AF_UNIX`.

1.3.3 Le protocole sous-jacent

Typiquement, l'échange de sockets peut s'appuyer sur différents protocoles (*IP*, *TCP*, *UDP*, *ICMP*, etc.) Dans le fichier `/usr/include/netinet/in.h` on trouve la définition d'un certain nombre de constantes symboliques : `IPPROTO_IP`, `IPPROTO_TCP`, `IPPROTO_UDP`, `IPPROTO_ICMP`, etc chacune désignant un protocole.

1.4 En pratique ...

Par exemple pour allouer une socket de la famille *INTERNET*, de type `SOCK_STREAM` et avec le protocole *TCP*, il faudrait appeler comme suit la primitive socket :

```
desc_sock = socket(SOCK_STREAM, AF_INET, IPPROTO_TCP);
```

Si la création de la socket a été possible, alors `desc_sock` est un descripteur de cette socket, sinon `desc_sock` vaut -1.

REMARQUE IMPORTANTE : Toutes les manips jusqu'à la manip 7bis seront effectuées sur les *Suns* (*purdey* ou *steed*)

$MANIP1$ Ecrivez un programme `Ouvre9Sockets.c` qui ouvre :

1. une socket de la famille *UNIX*, de type `SOCK_STREAM` avec le protocole *TCP*.
2. une socket de la famille *UNIX*, de type `SOCK_DGRAM` avec le protocole *UDP*.
3. une socket de la famille *UNIX*, de type `SOCK_DGRAM` avec le protocole par défaut (désigné par 0).
4. une socket de la famille *UNIX*, de type `SOCK_STREAM` avec le protocole par défaut (désigné par 0).
5. une socket de la famille *INTERNET*, de type `SOCK_DGRAM` avec le protocole *UDP*.
6. une socket de la famille *INTERNET*, de type `SOCK_STREAM` avec le protocole *TCP*.
7. une socket de la famille *INTERNET*, de type `SOCK_DGRAM` avec le protocole *TCP*.
8. une socket de la famille *INTERNET*, de type `SOCK_STREAM` avec le protocole *UDP*.
9. et une dernière de la famille *INTERNET*, de type `SOCK_RAW` avec le protocole *IP*.

Est-ce que ces ouvertures ont effectivement eu lieu ? Lesquelles ? Lorsqu'une ouverture échoue, comment pourrait-on en connaître la raison ?

En fait, le choix n'est pas indépendant du domaine de la socket et de son type. Souvent il n'existe qu'un seul protocole possible pour un type et un domaine donnés. En particulier, dans le cadre de ce TP (celui du domaine `AF_INET`), avec le type `SOCK_DGRAM` on ne peut utiliser que le protocol *UDP* et avec le type `SOCK_STREAM` on ne peut utiliser que le protocol *TCP*. Donc pour nous, le choix du protocole n'en est pas vraiment un. Dorénavant pour la primitive `socket`, au lieu de spécifier un nom de protocole, nous entrerons 0. De cette façon le système choisit le protocole par défaut prévu pour ce type de socket et pour ce domaine.

Donc dans toute la suite, les deux commandes utilisées seront :
soit `desc_dtgram = socket(AF_INET, SOCK_DGRAM, 0);`
soit `desc_stream = socket(AF_INET, SOCK_STREAM, 0);.`

1.5 La commande `netstat`

La commande `netstat` permet d'avoir des informations sur l'activité réseau du système local.

En particulier, `netstat -f unix` donne la liste de toutes les sockets *actives* du domaine `AF_UNIX`.

D'autre part, et surtout, `netstat -f inet` donne la liste de toutes les sockets *actives* de la famille *INTERNET* et plus particulièrement les sockets *TCP* connectées à une socket distante. `netstat -f inet` fournit l'adresse IP et le numéro de port de la socket locale et de la socket distante. Elle fournit également des informations propres au protocole *TCP* comme la taille des fenêtre d'émission, celle de la fenêtre de réception ainsi que la taille des files d'attente dans les deux cas.

La commande `netstat -a -f inet` est une commande qui, en plus des informations précédentes fournit les numéros de port où des sockets *TCP* attendent une connexion (par exemple les serveurs) ainsi que les numéros de port où des sockets *UDP* attendent des messages.

Sous *Linux*, pour avoir les mêmes fonctions, il faut en gros remplacer l'option `-f` par l'option `-A`.

$MANIP2$

1. Ecrire un programme *OuvreSocketUnix* qui ouvre une socket *UNIX* quelconque. Avant, pendant et après l'exécution de ce programme, lancer la commande `netstat -a -f unix`. Pouvez-vous situer la socket que vous avez ouverte ?
2. Même chose mais avec la commande `ls` au lieu de `netstat -a -f unix`
3. Ecrire un nouveau programme *OuvreSocketInternet* qui ouvre une socket *INTERNET* quelconque. Avant, pendant et après l'exécution de ce programme, lancer la commande `netstat -a -f inet`. Pouvez-vous situer la socket que vous avez ouverte ?
4. Même chose mais avec la commande `ls` au lieu de `netstat -a -f inet`
5. Conclusion ?

2 L'attribution d'une adresse à la socket : la primitive `bind`

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int socket_desc, const struct sockaddr *adr, int long_adr);
```

Cette fonction retourne -1 en cas d'échec.

Quelque soit le domaine d'adressage, la structure `sockaddr` est toujours composé d'un entier court qui donne la famille d'adresse (`AF_UNIX`, `AF_INET` ou autres) mais ce qui suit cet entier court dépend de la famille d'adresses. D'ailleurs il peut y avoir autant de formats et de longueurs différents qu'il existe de familles d'adresse. C'est pourquoi l'utilisation de ces adresses par `bind` nécessite que l'on spécifie le paramètre `long_adr`, i.e. la longueur (en octets) de l'adresse `adr`. En général pour ce paramètre, on se contente de spécifier `sizeof(adr)`.

2.1 Domaine `AF_UNIX`

Pour la famille `AF_UNIX`,

```
#include <sys/un.h>
struct sockaddr_un{
        short   sun_family;     /* AF_UNIX   */
        char    sun_path[108];  /* reference */
        };
```

> $MANIP3$ Ecrire un nouveau programme *BindUnix* qui crée une socket *UNIX* quelconque et lie cette socket à une position dans l'arborescence avec la primitive `bind`.
>
> Mêmes questions qu'auparavant : lancer avant, pendant et après l'exécution du programme, les commandes `ls` et `netstat -a -f unix`.

2.2 Domaine `AF_INET`

Dans la suite, nous nous intéresserons aux adresses de la famille `AF_INET`.

```
#include <netinet/in.h>

struct in_addr{u_long s_addr};

struct sockaddr_in{
        short   sin_family;       /* AF_INET                    */
        u_short sin_port;         /* numero du port associ\'e      */
        struct  in_addr sin_addr; /* adresse internet de la machine */
        char    sin_zero;         /* un champ de 8 caracteres nuls  */
        };
```

Les éléments de cette structure d'adresse sont développés dans ce qui suit.

2.2.1 L'adresse *IP* d'une machine

Toutes les machines disposent d'une adresse unique au monde et qui est composée d'un entier long ou, plus précisément, de quatre entiers n_1, n_2, n_3, n_4 compris chacun dans l'intervalle $[0, 255]$ et notés selon le format : $n_1.n_2.n_3.n_4$.

L'adresse IP de la machine *acroe.imag.fr* à Grenoble est 147.171.149.150et celle de la machine *bezier.iro.umontreal.ca* à l'université de Montréal est 132.204.26.144.

Pour connaître l'adresse IP d'une machine à partir de son nom `nom_machine` on utilise la commande `nslookup`.

```
nslookup nom_machine
```

Cette commande rend d'abord le nom et l'adresse IP de la machine sur laquelle se trouve la table de correspondance entre les noms et les adresse IP (le *"serveur de noms"*), et ensuite l'adresse IP de la machine `nom_machine`.

$MANIP4$ Donner l'adresse IP de la machine *pascal.u-strasbg.fr* ainsi que le nom et l'adresse IP du serveur de nom.

Donner aussi l'adresse IP de la machine *ling.ohio-state.edu* du département de linguistique de l'université d'Ohio aux Etats-Unis.

Dans un programme, pour avoir l'adresse IP à partir du nom, il faut utiliser la fonction :

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

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

Cette fonction a été définie dans la librairie `/usr/lib/libnsl.so.1`. La structure `hostent`, elle, est définie dans le fichier `/usr/include/netdb.h` :

```
struct  hostent {
        char    *h_name;       /* Official name of host              */
        char    **h_aliases;   /* alias list                         */
        int     h_addrtype;    /* host address type (tjrs AF_INET)  */
        int     h_length;      /* length of address                  */
        char    **h_addr_list; /* list of addresses from name server */
#define h_addr  h_addr_list[0] /* address, for backward compatiblity */
};
```

L'adresse IP de la machine n'est autre que les 4 premiers caractéres du tableau `h_addr_list`. Donc pour connaître l'adresse IP, il faut lire 4 bytes (un `u_long`) à partir de `h_addr`. Si `HostEnt` est l'hote dont nous voulons avoir l'adresse IP dans `u_long AdresseIP`, alors on ecrira :

```
#include<string.h>
struct hostent HostEnt; (...)
memcpy(&AdresseIP, HostEnt.h_addr, HostEnt->h_length);
```

Cependant, dans le cas particulier de la fonction `bind`, l'adresse de la machine *locale* peut être donnée par le nombre `INADDR_ANY`. Cette valeur permet d'associer la socket à toutes les adresses possibles de la machine locale, sans passer par la fonction `gethostbyname`.

Par contre, pour l'adresse de la machine distante, on ne fera pas l'economie d'appels à la fonction `gethostbyname`.

$MANIP5$

1. A l'aide des fonctions précédentes, écrivez la fonction `u_long IPAddrByName(char *)` qui, à partir d'un nom de machine, vous donnera directement l'adresse IP sous forme de `u_long`. Essayez-en le bon fonctionnement sur les mêmes hôtes que pour la manip précédente. Est-ce que les résultats concordent ? Expliquez.
2. Ecrivez une autre fonction, `LongToAdresseIP` qui, à partir d'une adresse IP (de type entier long) fournit les quatre entiers n_1, n_2, n_3 et n_4 compris entre 0 et 255 et tels que l'adresse IP s'écrit sous la forme $n_1.n_2.n_3.n_4$.

2.2.2 La notion de *port*

Acquis : la communication entre les deux machines

On dispose d'un système (un protocole de la couche 3 comme *IP*) qui, étant donnés l'adresse *adr* d'une machine (e.g. son adresse IP) et un paquet de données, peut transmettre le paquet de données à la machine d'adresse *adr*.

Problème : la communication entres les processus

Tel quel, un processus P_e expéditeur de données fonctionnant sur la machine expéditrice M_e sait que ses données arriveront à la machine de destination M_d, mais à l'intérieur de cette machine, il n'y a aucun moyen de savoir qui lira ces données. La machine de desination ne sait pas non plus à quel processus rendre ces données.

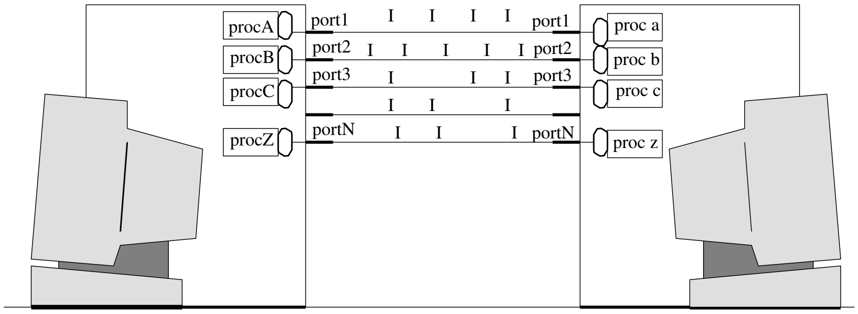

FIG. 1: Chaque processus lirait OU écrirait dans un fichier tampon qui serait transmis sur un des N câbles. Les câbles ne transporteraient que de l'information I.

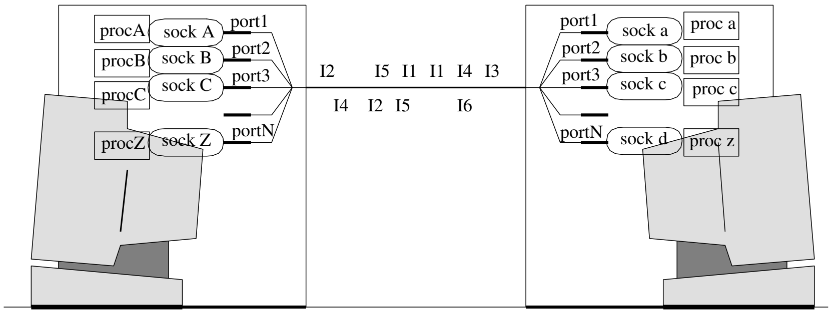

FIG. 2: Chaque processus lit ET écrit dans une socket qui est attachée à un port interne. Ensuite, les messages sont mélangés et transmis ensemble. Un moyen de trier les messages à l'arrivée est de faire accompagner chaque message par le numéro de port de destination. ($I1, I2$, ... etc). Les messages qui arrivent sur le port sont alors lisibles sur les sockets d'arrivée.

Le principe des voies indépendantes

Pour qu'un processus P_e sur M_e puisse communiquer avec un processus P_d sur la machine M_d, il est nécessaire d'avoir une voie de communication sur laquelle P_e est le seul processus émetteur et sur laquelle P_d est le seul processus récepteur.

Si on ne veut pas inclure d'information de contrôle dans les paquets d'information alors entre deux machines M_e et M_d en communication il ne doit y avoir qu'une paire de processus en communication uni-directionnelle.

Si l'on veut avoir une communication bidirectionnelle ou avoir plusieurs processus en communication, alors une solution consisterait peut-être à équiper les deux machines de plusieurs sorties i.e. plusieurs prises pour le réseau (cf figure 1). Ainsi on peut relier les deux machines par plusieurs voies indépendantes. Lorsque le processus P_e désire émettre, il doit d'abord se réserver un des ports de la machine et s'assurer que son processus interlocuteur s'est également réservé un port relié au sien. Alors la communication peut commencer.

La notion de *port*. Le multiplexage des paquets

En fait on a retenu une solution plus souple et plus pratique. On ne peut pas se passer du principe des voies indépendantes. Deux processus en communication nécessitent nécessairement une voie réservée à ces deux processus. Mais une *voie* ne signifie pas nécessairement un *câble* et un *port* ne signifie pas nécessairement une *prise*.

Lorsqu'un processus désire émettre, il crée une socket et l'attache sur un des ports internes de sa machine. Lorsqu'il émet un message, celui-ci est mélangé avec les informations émises par d'autres ports, et l'ensemble des messages est véhiculé ensemble (multiplexage). Mais ceci nécessite qu'on puisse les séparer à l'arrivée. Un moyen est de faire accompagner chaque message par le numéro de port de destination. (Un autre moyen est d'établir une connexion.) A l'arrivée des messages, ils sont triés et dirigés chacun vers un port. Alors les messages peuvent être lus par les processus destinataires dans les sockets.

En réalité, le port interne n'est pas une prise métallique interne, ce n'est qu'un emplacement mémoire où on peut lire et écrire. Ainsi, le nombre de ports qu'on peut utiliser est beaucoup moins limité que s'il fallait utiliser des prises physiques.

D'autre part, si l'on émet sur le port numéro n, il n'est nullement besoin que l'interlocuteur reçoive le message sur le port n. Il peut le recevoir sur n'importe quel port libre, du moment qu'on peut en connaître le numéro.

Quels numéros de port ?

Lorsque vous voulez émettre ou recevoir, il faut ouvrir une socket et l'attacher à un port de la machine. Mais peut-on s'attacher sur n'importe quel port ?

En réalité certains ports sont des ports correspondant à des services. (echo(7), telnet(23), finger(79), ftp(21), etc.) Bref, les numéros de port jusqu'à 511 sont réservés à des services *INTERNET*. Mais au delà, les numéros de port 512 à 1023 sont également réservés mais à des services spécifiquement *UNIX*, comme who(513), talk(517).

En général on choisit un numero de port supérieur à 1024. la commande `netstat -f inet` permet de connaître l'ensemble des numéros de port attachés.

Par ailleurs, si une socket n'est pas attachée et qu'on commence à envoyer des données dessus, elle est automatiquement attachée à un port libre. Pour récupérer son adresse, il faut utiliser la fonction :

```
#include <sys/types.h>
#include <sys/socket.h>
int getsockname (int desc, struct sockaddr *name, int *namelen);
```

Cette fonction retourne -1 en cas d'échec.

Avant l'appel de cette fonction, il faut disposer d'un objet de type `sockaddr`, quelque soit son contenu. A l'appel de `getsockname`, c'est son adresse et sa longueur qu'on donne comme deuxième et troisième paramètres de la fonction. Au retour, ces paramètres auront les valeurs correspondant à l'adresse de la socket concernée.

***MANIP*6**

1. Ecrire un programme qui, étant donné un numéro de port `nb_port` passé en argument, crée une socket *INTERNET* quelconque (à vous de choisir le type) et la lie au port numéro `nb_port` de la machine locale.
2. Mêmes questions habituelles : En lançant les commandes `ls` et `netstat -a -f inet` avant, pendant et après l'exécution, vérifier si ces commandes ont pris votre socket en compte.
3. Essayez divers numéros de port. Est-ce que le numéro de port indiqué correspond effectivement au numéro de port que vous avez spécifié ?
4. Que se passe-t-il lorsque vous spécifiez un numéro de port déjà utilisé par ailleurs ?
5. Que se passe-t-il lorsque le port demandé est le port 0 ? Est-ce que la socket a été attachée ? Est-ce que le numéro de port est 0 ?

La manip suivante consiste à écrire une fonction qui sera utilisée très fréquemment dans la suite du TP.

***MANIP*7**

1. Ecrire la fonction :

   ```
   int GetAndBindSocket(
           int type,       /* SOCK_DGRAM ou SOCK_STREAM           */
           int nb_port,    /* numero de port                      */
           struct sockaddr_in *adresse_loc
           );              /* l'adresse effective de la socket    */
                           /* allouee */
   ```

   qui, étant donnés un numéro de port (éventuellement 0), et un type (`SOCK_DGRAM` ou `SOCK_STREAM`), ouvre une socket *INTERNET* de ce type, l'attache au port numéro de port demandé, rende l'adresse `adresse_loc` où la socket a été effectivement attachée et vous retourne un descripteur de la socket.
2. Est-ce que l'adresse `adresse_loc` fournie par la fonction s'accorde toujours avec le numéro de port donné dans la liste de la commande `netstat -a -f inet` ?

2.3 Quelques mots sur la portabilité

A priori, la structure des sockets a été faite pour offrir la meilleure portabilité et pour pouvoir être utilisée dans une grande variété de machines et de systèmes.

2.3.1 *Big endians* vs *Little endians*

Ce qui nous posera peut-être un peu plus de problèmes est la structure des entiers courts et longs en mémoire. A l'intérieur d'une même machine, la structure des entiers n'a pas à être connue par l'utilisateur. Du moment que toutes les opérations à l'intérieur d'un système sont cohérentes tout se passe bien.

Par contre, lorsqu'il faut s'échanger des informations entre machines qui n'ont pas les mêmes conceptions de l'organisation d'un entier, alors il risque d'y avoir des problèmes.

En général, les entiers 32 bits sont divisés en deux parties ayant chacune 16 bits, et ces 16 bits sont divisées à leur tour en deux parties ayant chacun 8 bits. L'organisation de ces bytes (i.e. ces groupes de 8 bits) est la même pour la plupart des architectures. Donc pour la lecture et l'écriture des caractères, il n'y a aucun problème.

Par contre, pour écrire un entier court ou long, les avis divergent. Dans certaines architectures, les octets de poids fort sont placés aux adresses les plus basses, alors que les octets de poid faible sont placés aux adresses les plus élevées. Dans d'autres architectures, l'organisation est exactement le contraire. Les premiers sont appelés les *big endians* et les seconds sont les *little endians*.

Il s'agit d'une analogie avec *Les Voyages de Gulliver* roman de Jonathan Swift qui imagine deux royaumes (justement les *little endians* et les *big endians*) qui se livrent une guerre sans fin, parce que les *big endians* ouvrent les œufs par le grand bout (the big end) alors que les *little endians* les ouvrent par le petit bout (the little end).

Pour écrire l'entier court 1, les big endians écrivent 00000000 00000001 alors que les little endians écrivent 00000001 00000000.

Les processeurs 680x0 de *Motorola* (et donc les *Macintosh*), les *PA-RISC* de *Hewlett- Packard*, et les *SuperSPARC* de *Sun* sont des big endians.

Par contre, les processeurs 80X86 et Pentium de *Intel* et les *Alpha RISC* de *DEC* sont des little endians.

Les processeurs *MIPS* de *Silicon Graphics* et *PowerPC* de *IBM/Motorola* sont des big endians et little endians à la fois *(bi-endians)*.

2.3.2 Comment concilier les deux ?

Sur le réseau on ne peut admettre qu'un seul type d'organisation. (En fait il s'agit du format des big endians)

Donc si on travaille sur des little endians, et lorsqu'on envoie autre chose que des caractères, et en particulier des entiers courts ou longs, (comme par exemple les adresses IP et les numéros de port) il faut changer leur format au format réseau avant de construire la socket.

En réalité, pour produire un code portable d'une architecture à une autre, on a recours à des fonctions qui transforment les entiers courts (s) et les entiers longs (l) du format *hôte* (h) en format réseaux (n) ou inversement. Chez les big endians, ces fonctions ne font rien et chez les little endians ces fonctions changent le format des entiers.

Les quatre fonctions qui en correspondantes sont :

```
#include <sys/types.h>
#include <netinet/in.h>

u_long htonl(u_long hostlong);
u_short htons(u_short hostshort);
u_long ntohl(u_long netlong);
u_short ntohs(u_short netshort);
```

Il faut penser à les utiliser surtout pour les adresses IP (entiers longs) et les numéros de port (entiers courts). En particulier, les fonctions `gethostbyname` et `getsockname` rendent des adresses IP et des numéros de port qui sont au format réseau. D'autre part, la fonction `bind` prend en paramètre une adresse de socket au format réseau.

MANIP7bis

1. Essayer le programme écrit à la Manip 5 sur une big endian (comme *steed* ou *purdey*) et sur une little endian (comme *emma* ou *tara*). Expliquez les résultats obtenus. Que faudrait-il faire pour que les résultats soient identiques sur les deux machines ?

2. Essayer le programme écrit à la Manip 6 sur sur une little endian (comme *emma* ou *tara*). Situez votre socket dans la liste fournie par la commande `netstat -a -A inet`.
3. Changer la fonction `GetAndBindSocket` (Manip 7) de façon à ce qu'elle tienne compte des passages de formats hôte ↔ réseau.
4. Ecrire les deux fonctions :

```
typedef struct AdrIP_t
{ unsigned char n1, n2, n3, n4;
} AdresseIP_t;

struct sockaddr_in WriteNetworkAddress(
     char *HostName,            /* le nom du hote */
     short port                 /* le num de port en format hote */
     );

void ReadNetworkAddress(
     struct sockaddr_in n_adr,  /* l'adresse sous format reseau    */
     AdrIP_t *IP,               /* resultat sous forme n1.n2.n3.n4 */
     short *port                /* le num de port format hote      */
     );
```

   - La structure `AdrIP_t` représente une adresse IP, non sous la forme d'un entier long, mais sous la forme de 4 entiers non-signés compris entre 0 et 255.
   - La fonction `ReadNetworkAddress` devra rendre le numéro de port de la socket sous format hôte, ainsi que l'adresse IP de la socket sous la forme habituelle des quatre entiers compris entre 0 et 255 à partir de l'adresse de la socket sous format réseau.
   - La fonction `WriteNetwokAddress` devra au contraire, rendre une adresse de socket au format réseau, à partir du nom du hôte et de son numéro de port en format hôte.

   Elles devront prendre en charge la préparation et la lecture des adresses, ainsi que tous les changements de format nécessaires, de façon à ce que dans la suite des manips, nous n'ayons pas à nous soucier de ce problème.

Dans la suite, on validera ce travail en expérimentant les manips suivantes entre des machines hétérogènes. L'une sera de type big endian alors que l'autre sera de type little endian.

3 La communication par datagrammes

3.1 Les primitives `sendto` et `recvfrom`

La primitive `sendto` permet d'envoyer un message à partir d'une socket locale vers une socket distante.

```
int sendto(
        int desc,               /* descripteur socket   */
        void *message,          /* message a envoyer    */
        int longueur,           /* longueur du message */
        int option,             /* 0                    */
        struct sockaddr *ptr_adresse, /* adresse destinataire */
        int longueur_adresse /* longueur de cette adresse */
        );
```

Cette primitive rend -1 en cas d'échec.

La primitive `recvfrom` permet de recevoir un message sur une socket locale et venant d’une socket distante.

```
int recvfrom(
        int desc,               /* descripteur socket     */
        void *message,          /* adresse de reception   */
        int longueur,           /* taille zone reserv\'ee */
        int option,             /* 0 ou MSG_PEEK          */
        struct sockaddr *ptr_adresse, /* adresse emetteur */
        int *long_adresse        /* zone adresse           */
        );
```

Cette primitive rend -1 en cas d’échec.

Lorsqu’avant l’appel, l’adresse `ptr_adresse` est non-nulle, alors après l’appel, la variable contient la valeur de l’adresse de la socket émettrice.

Contrairement au cas de la primitive `sendto`, il y a plusieurs options possibles. Avec l’option `MSG_PEEK` le contenu du message est effectivement copiée à l’adresse `message`, mais n’est pas extrait du tampon. Le prochain appel à `recvfrom` portera sur le même message. Par contre, l’adresse de la socket émettrice n’est pas recopiée (elle ne l’est qu’au moment de l’extraction effective du message).

Ces primitives fonctionnent aussi bien avec les sockets *UNIX* qu’avec des sockets *INTERNET*.

***MANIP*8** Ecrire deux programmes qui ouvrent une socket *UNIX* de type *SOCK_DGRAM* et l’attachent à un emplacement quelconque.

1. Un des programmes est l’émetteur et émet un message vers l’autre socket.
2. L’autre programme doit recevoir ce message, identifier la socket émettrice, et afficher le message et l’emplacement de la socket émettrice.
3. A présent, le message à envoyer doit être envoyé au clavier. Donc sa longueur n’est pas connue à l’avance.
4. Donner le nombre de caractères envoyés pour un ensemble de messages de tailles différentes.
5. Comment faire pour que le nombre de caractères envoyés sur le réseau corresponde aux seuls caractères utiles ?
6. Comment réagit la primitive `recvfrom` lorsqu’il n’y a rien à lire sur la socket distante ?
7. Réciproquement, comment réagit la primitive `sendto` lorsque le récepteur n’est pas en train d’attendre un message ?
8. Expliquez en substance comment on peut utiliser ces propriétés pour synchroniser deux processus.

***MANIP*9** A présent on utilise des sockets *INTERNET* pour communiquer entre des machines distantes. Ecrire deux programmes, l’un émetteur et l’autre récepteur.

1. Le récepteur doit ouvrir une socket et l’attacher sur un port libre de la machine locale. Il doit afficher le numéro du port *port_loc* effectivement alloué par le système. Ensuite, avec la primitive `recvfrom`, le récepteur doit commencer à écouter le réseau en attendant un message venant d’un émetteur quelconque. Dès qu’il reçoit un message, il affiche le message à l’écran, ainsi que l’adresse IP et le numéro de port de l’émetteur. Il ne s’arrête que lorsqu’il reçoit un message constitué d’un simple point “.” .
2. L’émetteur doit recevoir comme argument un nom d’hôte *hote_dist* et un numéro de port *port_dist* sur cet hôte. On pourra trouver l’adresse IP de cette machine avec la fonction :
   `u_long IPAddrByName(char *)`
   écrite à la manip 5. Etant donnés ces arguments, l’émetteur doit ouvrir une socket et l’attacher sur un port libre (afficher le numéro de port effectivement attribué par le système). Ensuite, émettre un (des) message(s) de longueur quelconque (par exemple saisis au clavier) vers l’hôte *hote_dist* sur le port numéro *post_dist*. Si le message envoyé est constitué d’un simple “.” alors après la transmission du message, l’émetteur et le récepteur s’arrˆ