

Programmation des Réseaux IP - document de cours

Version originale : V. Ribaud - février 2002

Adaptation : P. Le Parc - décembre 2018

Département d'Informatique - Faculté des Sciences et Techniques

Université de Bretagne Occidentale

Décembre 2018

Table des matières

| | | |
|----------|--|-----------|
| I | Environnement et commandes de base | 7 |
| 1 | Les adresses et les fichiers de configuration | 9 |
| 1.1 | L'adressage | 9 |
| 1.1.1 | Le principe d'adressage d'origine | 9 |
| 1.1.2 | Adresses particulières | 10 |
| 1.1.3 | Adresses de multidistribution (classe D) | 11 |
| 1.1.4 | Adresses de classe E | 11 |
| 1.1.5 | Multidomicile | 11 |
| 1.2 | Agrégation d'adresses et sous-adressage | 11 |
| 1.2.1 | Sous-réseaux | 11 |
| 1.2.2 | Adressage agrégé ou sur-adressage | 12 |
| 1.3 | Routage | 13 |
| 1.3.1 | Tables de routage | 13 |
| 1.3.2 | L'interface IP / Ethernet | 13 |
| 1.4 | Les fichiers de configuration | 13 |
| 1.4.1 | Le fichier <code>/etc/hosts</code> | 13 |
| 1.4.2 | Le fichier <code>/etc/networks</code> | 13 |
| 1.4.3 | Le fichier <code>/etc/services</code> | 14 |
| 1.4.4 | Le fichier <code>/etc/protocols</code> | 14 |
| 1.4.5 | Cas des NIS | 14 |
| 1.5 | Les processus démons | 14 |
| 1.5.1 | Principes | 14 |
| 1.5.2 | Le super-démon <code>inetd</code> | 15 |
| 1.5.3 | Le fichier <code>/etc/inetd.conf</code> | 15 |
| 1.6 | Les fichiers d'équivalence UNIX | 15 |
| 1.6.1 | Le fichier <code>/etc/hosts.equiv</code> | 15 |
| 1.6.2 | Le fichier <code>/.rhosts</code> | 16 |
| 2 | Les commandes d'administration | 17 |
| 2.1 | La commande <code>ping</code> | 17 |
| 2.2 | La commande <code>arp</code> | 17 |
| 2.3 | La commande <code>ifconfig</code> | 17 |
| 2.4 | La commande <code>netstat</code> | 18 |
| 2.5 | La commande <code>nslookup</code> | 19 |
| 2.6 | La commande <code>finger</code> | 21 |
| 2.7 | Les commandes en <code>r</code> | 21 |
| 2.7.1 | La commande <code>rlogin</code> | 21 |

| | | |
|-------|--|----|
| 2.7.2 | La commande <code>rcp</code> | 21 |
| 2.7.3 | La commande <code>rsh</code> | 22 |
| 2.7.4 | La commande <code>ruser</code> | 22 |
| 2.8 | La commande <code>uname</code> | 22 |
| 2.9 | Les commandes <code>hostname</code> et <code>hostid</code> | 22 |

II Programmation réseaux 23

3 Les structures et les fonctions d'accès 25

| | | |
|-------|--|----|
| 3.1 | Les fichiers d'include | 25 |
| 3.2 | Les librairies standard | 25 |
| 3.3 | Les types Posix | 25 |
| 3.4 | Les adresses des machines | 25 |
| 3.5 | Les fonctions de manipulation des adresses | 26 |
| 3.5.1 | <code>inet_aton</code> | 26 |
| 3.5.2 | <code>inet_ntoa</code> | 27 |
| 3.6 | La représentation des nombres | 27 |
| 3.7 | La structure <code>hostent</code> | 27 |
| 3.8 | Les fonctions de consultation | 27 |
| 3.8.1 | <code>gethostname</code> | 27 |
| 3.8.2 | <code>gethostbyname</code> | 28 |
| 3.8.3 | <code>gethostbyaddr</code> | 28 |

4 Les sockets 29

| | | |
|-------|--|----|
| 4.1 | Introduction | 29 |
| 4.1.1 | Qu'est-ce qu'une socket ? | 29 |
| 4.1.2 | Type d'une socket | 29 |
| 4.1.3 | Domaine d'une socket | 30 |
| 4.2 | Primitives communes | 30 |
| 4.2.1 | La création d'une socket : <code>socket</code> | 30 |
| 4.2.2 | La suppression d'une socket : <code>close</code> | 31 |
| 4.2.3 | L'attachement d'une socket à une adresse : <code>bind</code> | 31 |
| 4.3 | La scrutation | 36 |
| 4.3.1 | Introduction | 36 |
| 4.3.2 | La primitive <code>select</code> | 36 |

5 La communication en mode connecté (TCP) 39

| | | |
|-------|--|----|
| 5.1 | Principes | 39 |
| 5.2 | Le point de vue du serveur | 39 |
| 5.2.1 | Introduction | 39 |
| 5.2.2 | L'ouverture du service : <code>listen</code> | 40 |
| 5.2.3 | La prise en compte d'une connexion : <code>accept</code> | 40 |
| 5.2.4 | Un exemple de serveur | 41 |
| 5.3 | Le point de vue du client | 44 |
| 5.3.1 | Introduction | 44 |
| 5.3.2 | La demande de connexion : <code>connect</code> | 44 |
| 5.3.3 | Un exemple de client | 45 |

| | | |
|----------|--|-----------|
| 5.4 | Le dialogue client/serveur | 47 |
| 5.4.1 | Introduction | 47 |
| 5.4.2 | L'envoi de caractères | 47 |
| 5.4.3 | La réception de caractères | 48 |
| 5.5 | La fermeture de la connexion | 49 |
| 5.5.1 | La primitive <code>close</code> | 49 |
| 5.5.2 | La primitive <code>shutdown</code> | 49 |
| 5.6 | Squelettes de serveur et de client | 49 |
| 5.6.1 | Le squelette d'un serveur créant un processus de service | 49 |
| 5.6.2 | Le squelette d'un client | 52 |
| 6 | La communication en mode non connecté (UDP) | 57 |
| 6.1 | Principes | 57 |
| 6.2 | Primitives d'envoi et de réception | 58 |
| 6.2.1 | Introduction | 58 |
| 6.2.2 | La primitive <code>sendto</code> | 58 |
| 6.2.3 | La primitive <code>recvfrom</code> | 59 |
| 6.3 | Un exemple | 59 |
| 6.4 | Les messages | 62 |
| 6.5 | Les pseudo-connexions | 62 |
| 6.5.1 | Introduction | 62 |
| 6.5.2 | Les primitives d'envoi et de réception | 62 |

Première partie

Environnement et commandes de base

Chapitre 1

Les adresses et les fichiers de configuration

1.1 L'adressage

Une ou plusieurs *adresses logiques* sont attribuées à chaque machine hôte. L'adresse INTERNET ou *adresse IP* est constituée d'une adresse de réseau et d'une adresse de machine sur ce réseau. Une adresse IPv4 occupe 4 octets ou 32 bits, une adresse IPv6 128 bits. Les adresses IPv4 sont en général données sous la forme $n1.n2.n3.n4$ (chacun des n représente la valeur d'un octet comprise entre 0 et 255).

1.1.1 Le principe d'adressage d'origine

Conceptuellement, chaque adresse IP est constituée d'une paire d'identificateur, le premier identifiant le réseau et le second identifiant une machine sur ce réseau. Dans le système d'adressage d'origine, à base de **classes**, les adresses IP devaient appartenir à une des classes A, B ou C (cf. tableau 1.1). Ceci a changé vers le milieu des années 90 avec l'arrivée de adresses sans classe (*classless*).

| CLASSE | INTERVALLE |
|--------|---|
| A | 0.0.0.0 à 127 .255.255.255 |
| B | 128 .0.0.0 à 191 .255.255.255 |
| C | 192 .0.0.0 à 223 .255.255.255 |
| D | 224 .0.0.0 à 239 .255.255.255 |
| E | 240 .0.0.0 à 255 .255.255.255 |

TABLE 1.1 – Intervalles des adresses des différents classes de réseaux

Adresses de classe A

Elles correspondent aux grands réseaux (relativement peu nombreux et abritant un très grand nombre de machines).

L'adresse réseau occupe 1 octet, l'adresse de la machine sur ce réseau 3 octets.

| | | | | |
|---|-------------------|---|------------------------|----|
| 0 | 1 | 7 | 8 | 31 |
| 0 | Adresse du réseau | | Identification machine | |

Adresses de classe B

Elles correspondent aux réseaux de taille moyenne.

L'adresse réseau occupe 2 octets, l'adresse de la machine sur ce réseau 2 octets.

| | | | | | |
|---|---|-------------------|----|----|------------------------|
| 0 | 1 | 2 | 15 | 16 | 31 |
| 1 | 0 | Adresse du réseau | | | Identification machine |

Adresses de classe C

Elles correspondent aux petits réseaux (très nombreux et abritant un petit nombre de machines).

L'adresse réseau occupe 3 octets, l'adresse de la machine sur ce réseau 1 octet.

| | | | | | | |
|---|---|---|-------------------|----|----|-------------|
| 0 | 1 | 2 | 3 | 23 | 24 | 31 |
| 1 | 1 | 0 | Adresse du réseau | | | Id. machine |

1.1.2 Adresses particulières

Par convention, l'identifiant 0 n'est attribué ni à aucun réseau ni à aucune machine. Ainsi, on peut utiliser une adresse IP ayant 0 comme identifiant de machine pour désigner le réseau lui-même (193.52.16.0 désigne le réseau où sont situées les machines Internet du département Informatique).

L'adresse dont tous les bits de l'identifiant de machine sont à 1 permet de désigner l'ensemble des machines de ce réseau, on l'appelle adresse de diffusion dirigée (*directed broadcast address*). Elle permet d'envoyer le même message à toutes les machines.

L'adresse de diffusion dirigée possède un identifiant de réseau, associée à un identifiant de machine, indiquant par sa valeur (tous les bits à 1) la diffusion. Une adresse de diffusion dirigée peut être interprétée sans ambiguïté parce qu'elle identifie le réseau cible sur lequel s'applique la diffusion mentionnée.

Une autre forme d'adresse de diffusion, appelée adresse de diffusion limitée ou adresse de diffusion locale, permet d'indiquer une diffusion locale, sur le réseau où se trouve la machine qui l'émet ou la reçoit, indépendamment de toute adresse IP.

Par convention l'adresse 127.0.0.1 est affectée à l'interface de bouclage (*loopback*). Tout ce qui est envoyé à cette adresse boucle et devient une réception. Cette adresse est normalement connue par le nom `INADDR_LOOPBACK`.

1.1.3 Adresses de multidistribution (classe D)

En plus de la remise à un destinataire unique (*unicast delivery*), dans laquelle un paquet atteint une unique machine, et de la remise en diffusion (*broadcast delivery*) où il s'agit de remettre un paquet à toutes les machines d'un certain réseau, il est possible d'avoir une remise multidestinataire (*multicasting*), dans laquelle un paquet est remis à un sous-ensemble spécifique de machines.

Ce sont les adresse de classe D qui sont utilisées dans l'adressage multipoint (*multicast*). Chaque groupe de diffusion possède une adresse de diffusion de groupe (classe D) unique. Une machine peut rejoindre un groupe à tout moment et appartenir à plusieurs groupes. Son appartenance à un groupe détermine le fait qu'elle reçoive l'information destinée à ce groupe. Une machine peut envoyer des informations à un groupe sans y appartenir. Les 4 premiers bits sont utilisés pour désigner la classe, les 28 bits suivants pour numéroté les groupes.

| 0 | 1 | 2 | 3 | 4 | | 31 |
|---|---|---|---|--------------------------|--|----|
| 1 | 1 | 1 | 0 | Identification de groupe | | |

1.1.4 Adresses de classe E

Elles sont réservées pour un usage ultérieur.

Les 5 premiers bits sont utilisés pour désigner la classe, les 27 bits suivants pour l'instant inutilisés.

| 0 | 1 | 2 | 3 | 4 | 5 | | 31 |
|---|---|---|---|---|---------------------------------|--|----|
| 1 | 1 | 1 | 1 | 0 | Réservé pour un usage ultérieur | | |

1.1.5 Multidomicile

Traditionnellement le terme de machine multidomiciliée (*multihomed*) fait référence à une machine avec plusieurs interfaces réseaux, comme deux Ethernet par exemple ou un Ethernet et un modem. Chaque interface a son adresse IPv4 unique.

Il est désormais possible d'attribuer plusieurs adresses IP à la même interface, ce qui lui donne ainsi plusieurs interfaces logiques. Aussi la définition d'une machine multidomiciliée doit être étendue à une machine à plusieurs interfaces, sans faire la distinction si ces interfaces sont physiques ou logiques.

1.2 Agrégation d'adresses et sous-adressage

1.2.1 Sous-réseaux

La structure standard d'une adresse IP peut être modifiée localement en utilisant des bits d'adresse de machine comme bits d'adresse de réseau supplémentaires.

Ces nouveaux bits de réseau permettent de définir un réseau au sein d'un réseau de plus grande taille. Ce nouveau réseau est appelé sous-réseau. Par exemple, dans un réseau de classe B où les machines sont identifiées sur 16 bits, utiliser 3 bits pour identifier un sous-réseau et les 13 autres bits pour identifier une machine, permet d'identifier 8 (2^3) sous-réseaux de 8190 (2^{13}) machines.

L'application d'un masque de bits, appelé masque de sous-réseau, à l'adresse IP permet de définir un sous-réseau. Les bits du masque de sous-réseau sont à 1 si les bits correspondants font partie de l'identifiant du réseau. Les bits sont à 0 si ils font partie de l'identifiant de la machine.

Le masque de réseau associé aux adresses de classe B correspond à la valeur 255.255.0.0. En faisant un ET entre une adresse IP et ce masque, on obtient l'adresse du réseau. Par convention, on fait suivre l'adresse d'un réseau où le sous-adressage est utilisé par un / et du nombre de bits du masque de sous-réseau.

Le masque de sous-réseau le plus souvent utilisé par les réseaux de classe B est 255.255.255.0. Il permet d'agrandir la partie réseau d'une adresse de classe B d'un octet supplémentaire.

Les deux premiers octets définissent le réseau de classe B, le troisième l'adresse du sous-réseau, le quatrième la machine reliée au sous-réseau.

Par exemple, le département d'Informatique utilise un réseau de classe B pour la pédagogie (ce réseau n'est pas relié à l'Internet). Il est utilisé avec des sous-réseaux et se note 172.18.0.0/24.

1.2.2 Adressage agrégé ou sur-adressage

En 1993, il devint évident que la croissance d'Internet allait conduire à l'épuisement des adresses de classe B. En attendant l'arrivée d'une nouvelle version d'IP, il fallait trouver une solution.

Appelé adressage hors classe (*classless addressing*), adressage agrégé ou sur-adressage (*supernet addressing*), voire adressage de sur-réseau, cette technique est à l'opposé de celle des sous-réseaux. Au lieu d'utiliser une seule adresse IP pour plusieurs réseaux appartenant à une certaine organisation, le sur-adressage consiste à disposer d'un grand nombre d'adresses réseau IP pour une unique organisation (en général des adresses de classe C).

L'allocation d'un grand nombre d'adresses de classe C au lieu d'une adresse de classe B permet d'économiser des numéros de classe B, mais il apparaît un autre problème : au lieu d'avoir une seule entrée par entreprise, une table de routage en comporte alors un grand nombre.

La technique dite CIDR (*Classless Inter-Domain Routing* ou routage inter-domaine sans classe) permet de résoudre ce problème. De façon conceptuelle, CIDR fusionne ou agrège un ensemble contigu d'adresse de classe C en une seule entrée représentée par un couple (**adresse_de_réseau**, **compteur**) dans lequel **adresse_de_réseau** est la plus petite adresse de réseau du bloc et **compteur** donne le nombre total de réseau du bloc. Par exemple, (193.52.16.0, 3) permet de spécifier les trois adresses réseau 193.52.16.0, 193.52.17.0, 193.52.18.0.

En pratique, CIDR ne restreint pas les numéros de réseau aux adresses de classe C et n'a pas besoin de spécifier la taille du bloc comme un nombre entier. CIDR exige que chaque bloc d'adresse soit une puissance de deux et utilise un masque binaire pour identifier la taille du bloc.

Etant donné qu'identifier un bloc CIDR nécessite de définir deux items, une adresse et un masque, une notation abrégée a été définie (appelé notation slash). Cette notation indique la longueur du masque en décimal et utilise le signe slash (/) pour séparer cette valeur de l'adresse.

1.3 Routage

1.3.1 Tables de routage

Chaque machine du réseau (hôte ou passerelle) doit déterminer sa voie d'acheminement des datagrammes :

- si l'hôte de destination se trouve sur le même réseau local, les données sont transmises à l'hôte de destination,
- si l'hôte de destination se trouve sur un réseau distant, les données sont expédiées à une passerelle locale.

L'acheminement est donc déterminé en fonction de la partie réseau de l'adresse, selon les indications de la table de routage locale. Celle-ci est créée soit par l'administrateur-système, soit au moyen des protocoles de routage.

1.3.2 L'interface IP / Ethernet

Un réseau Ethernet identifie ses machines au moyen d'une adresse physique unique, appelée *adresse Ethernet*.

Le protocole spécifique ARP *Address Resolution Protocol* permet de retrouver l'adresse Ethernet correspondant à une adresse IP.

Le protocole RARP *reverse ARP* permet à une station sans disque ou un terminal X, ne connaissant pas sa propre adresse IP, de la demander par la diffusion d'un message Ethernet d'un type spécifié.

1.4 Les fichiers de configuration

1.4.1 Le fichier /etc/hosts

Il contient les informations relatives aux différentes machines du réseau local auquel appartient le système. Exemple :

```
193.52.16.26    kelenn-gw kelenn-gw.univ-brest.fr
```

Chaque ligne donne pour chaque site :

- l'adresse internet (193.52.16.26),
- le nom officiel (kelenn-gw),
- une liste d'alias (kelenn-gw.univ-brest.fr),
- un commentaire éventuel (apres un #).

1.4.2 Le fichier /etc/networks

Il constitue la base de données des réseaux connus. On y trouve une suite de lignes contenant :

- le nom officiel du réseau,
- son adresse internet,
- une liste d'alias,
- un commentaire éventuel (apres un #).

Du fait de la généralisation de l'adressage par nom, ce fichier n'est pratiquement plus utilisé et maintenu.

1.4.3 Le fichier `/etc/services`

Il donne la liste des services Internet connus. Exemple :

```
ftp          21/tcp
telnet       23/tcp
```

On y trouve une suite de lignes contenant :

- le nom du service (`ftp`),
- un numéro de port et un protocole (`21/tcp`)
- une liste d'alias,
- un commentaire éventuel (après un `#`).

1.4.4 Le fichier `/etc/protocols`

Il donne la liste des protocoles utilisés dans le réseau Internet. Exemple :

```
ip          0      IP      # internet protocol, pseudo protocol number
```

On y trouve une suite de lignes contenant :

- le nom officiel du protocole (`ip`),
- son numéro (`0`),
- une liste d'alias,
- un commentaire éventuel (après un `#`).

1.4.5 Cas des NIS

Lors de la gestion à l'aide des NIS, ces fichiers ne sont pas à jour. Les renseignements sont centralisés sur la machine-maître des NIS et s'obtiennent grâce à la commande `ypcat nom_de_fichier`. Exemple :

```
$ ypcat hosts
172.16.1.18 cisco6 cisco6.univ-brest.fr
193.52.16.18 cisco5 cisco5.univ-brest.fr
```

1.5 Les processus démons

1.5.1 Principes

L'invocation de services standard tels que FTP ou `rlogin` est réalisée par l'intermédiaire de commandes (`ftp`, `rlogin`).

Ces services sont invoqués sur une machine distante, il est donc nécessaires que des processus dédiés, appelés *démons*¹ s'exécutent sur la machine distante.

1. en anglais, daemon

1.5.2 Le super-démon `inetd`

On peut imaginer de lancer tous les processus-démons nécessaires au lancement du système, mais cela entraînerait l'existence permanente d'un nombre important de processus à l'état endormi.

On a préféré charger un processus serveur unique (le serveur `inetd`) de recevoir les demandes de services et d'activer le processus démon approprié à la demande.

Un exemplaire unique de ce super-démon est lancé, qui scrute les différents ports des services qu'il supervise. Lorsqu'une requête arrive sur l'un des points de communication associés aux ports, le super-démon crée un nouveau processus correspondant au service demandé (il arrive qu'il traite lui-même la requête).

1.5.3 Le fichier `/etc/inetd.conf`

Le contenu de ce fichier est lu au lancement du démon ou lorsque le démon reçoit le signal `SIGHUP`. Exemple :

```
ftp stream tcp      nowait root    /usr/etc/in.ftpd      in.ftpd
name dgram  udp      wait  root    /usr/etc/in.tnamed    in.tnamed
systat stream tcp    nowait root    /usr/bin/ps           ps -auwx
echo stream tcp      nowait root    internal
```

On y trouve une suite de lignes contenant :

- le nom du service (`ftp`),
- le type de point de communication utilisé (`stream` ou `dgram` qui dans le domaine Internet utilisent respectivement TCP et UDP),
- le protocole utilisé (`tcp` ou `udp`),
- une option `wait` ou `nowait` pour les services en mode `dgram` (l'option `wait` stipule qu'un seul serveur de ce type peut exister),
- un nom d'utilisateur qui sera le propriétaire du processus démon créé (`root`),
- la référence absolue du programme exécuté par le démon spécifique (par exemple `/usr/etc/in.ftpd/`) ; si la valeur est `internal` le service est réalisé par `inetd`,
- une liste de paramètres du programme à exécuter (par exemple des options).

1.6 Les fichiers d'équivalence UNIX

1.6.1 Le fichier `/etc/hosts.equiv`

Ce fichier définit les hôtes (machines) et utilisateurs fiables auxquels le droit d'accès aux commandes en `r` (cf. §2.7) est accordé. Ce fichier peut aussi définir les hôtes et utilisateurs auxquels le droit d'accès explicite n'est pas octroyé. Ne pas disposer des droits d'accès ne signifie pas nécessairement que l'utilisateur ne puisse pas accéder aux commandes ; cela signifie simplement que l'utilisateur doit spécifier un mot de passe avant de pouvoir y accéder.

Différents systèmes UNIX appartenant à un réseau local peuvent être déclarés donc équivalents du point de vue des `r`-commandes des utilisateurs, ce qui permet à un utilisateur connecté à un hôte d'accéder aux comptes utilisateurs de même nom sur un autre système qui aura précisé le nom de l'hôte dans son fichier `/etc/hosts.equiv`.

1.6.2 Le fichier `/.rhosts`

Dans le cas où des machines n'ont pas été déclarés équivalentes par les administrateurs systèmes, les utilisateurs ont la possibilité de définir personnellement des accès aux `r`-commandes sur différentes machines par l'intermédiaire du fichier `.rhosts` de leur *home directory*.

Si le fichier `.rhosts` de l'utilisateur `arthur` de la machine `ma_machine` contient les lignes :

```
autre_machine toto
une_autre_machine toto
```

l'utilisateur `toto` sur les machines `autre_machine` ou `une_autre_machine` pourra s'identifier sur la machine `ma_machine` sous le nom `arthur` sans avoir à donner le mot de passe de cet utilisateur.

N.B. : c'est un "trou de sécurité important", à utiliser avec précaution.

Chapitre 2

Les commandes d'administration

Toutes ces commandes sont à exécuter avec `man` sous la main ...

2.1 La commande ping

Elle permet de tester si une machine donnée est active. Exemple :

```
$ ping kelenn-gw
kelenn-gw.univ-brest.fr is alive
```

2.2 La commande arp

Elle permet de visualiser le contenu de la table du protocole ARP. Exemple :

```
$ arp -a
kelenn-gw.univ-brest.fr (193.52.16.26) at 8:0:20:74:f9:df
odet (172.16.2.2) at 8:0:20:20:83:31
```

Lorsqu'aucune résolution d'adresse n'a été faite dernièrement, il n'y a pas d'entrée. En utilisant La commande `ping`, on force la conversion d'adresse et on peut ensuite visualiser l'adresse demandée.

```
$ arp terre
terre (193.52.16.59) -- no entry
$ ping terre
terre is alive
$ arp terre
terre (193.52.16.59) at 8:0:20:2:c8:38
```

2.3 La commande ifconfig

Elle permet de spécifier (pour les administrateurs) ou d'obtenir des informations sur les interfaces du système.

Le champ **Name** contient le nom attribué à l'interface avec `ifconfig`. Une astérisque (*) indique que l'interface n'a pas été activée.

Le champ **MTU** contient le Maximum Transmit Unit.

Le champ **Net/Dest** indique le réseau, le sous-réseau (auquel cas il faut appliquer le masque) ou l'hôte auquel l'interface permet d'accéder.

Le champ **Address** contient l'adresse IP attribuée à cette interface.

Le champ **Ipkts** indique le nombre de paquets que cette interface a reçu.

Le champ **Ierrs** indique le nombre de paquets endommagés que cette interface a reçu.

Le champ **Opkts** indique le nombre de paquets que cette interface a envoyé.

Le champ **Oerrs** indique le nombre de paquets générant une erreur que cette interface a envoyé.

Le champ **Collis** indique le nombre de collisions Ethernet détectées par cette interface.

Le champ **Queue** indique le nombre de paquets figurant dans la file d'attente et attendant leur transmission par cette interface (normalement à 0).

L'option `-r` donne les informations sur les tables de routage :

```
$ netstat -r
Routing tables

Destination          Gateway                Flags    Refcnt  Use      Interface
venan                 paludenn-gw           UGHD     0        238      le0
default               cisco5                UG       10       67909    le0
193.52.16.0           penfeld-gw            U        36       24917    le0
172.16.2.0            penfeld               U         3       167449    le1
```

L'option `-s` donne des statistiques par protocole ou sur le routage (avec `-r`).

Enfin, avec l'option `-a`, on visualise toutes les sockets actives (on peut restreindre au domaine Unix `-f unix` ou au domaine internet `-f inet`).

2.5 La commande `nslookup`

Elle permet de se connecter au serveur de noms utilisé par défaut ou à un serveur de nom particulier et de l'interroger pour obtenir des informations sur les domaines (liste des machines) et sur les machines (adresses).

Elle s'utilise en mode interactif ou en mode non interactif.

```
$ nslookup
Default Server:  cassis-gw.univ-brest.fr
Address:  192.70.100.29

> kelenn
Server:  cassis-gw.univ-brest.fr
Address:  192.70.100.29

Name:    kelenn.univ-brest.fr
Address:  172.16.1.24
```

```

> kelenn-gw
Server:  cassis-gw.univ-brest.fr
Address: 192.70.100.29

Name:    kelenn-gw.univ-brest.fr
Address: 193.52.16.26

> whitehouse.gov
Server:  cassis-gw.univ-brest.fr
Address: 192.70.100.29

Non-authoritative answer:
Name:    whitehouse.gov
Address: 198.137.241.30

> exit

```

La commande `help` donne la liste des actions permises :

```

Commands:      (identifiers are shown in uppercase, [] means optional)

NAME           - print info about the host/domain NAME using default server
NAME1 NAME2    - as above, but use NAME2 as server
help or ?      - print help information
exit           - exit the program
set OPTION      - set an option
  all          - print options, current server and host
  [no]debug    - print debugging information
  [no]d2       - print exhaustive debugging information
  [no]defname   - append domain name to each query
  [no]recurse  - ask for recursive answer to query
  [no]vc       - always use a virtual circuit
domain=NAME    - set default domain name to NAME
root=NAME      - set root server to NAME
retry=X        - set number of retries to X
timeout=X      - set time-out interval to X
querytype=X    - set query type to one of A,CNAME,HINFO,MB,MG,MINFO,MR,MX
type=X        - set query type to one of A,CNAME,HINFO,MB,MG,MINFO,MR,MX
server NAME    - set default server to NAME, using current default server
lserver NAME   - set default server to NAME, using initial server
finger [NAME]  - finger the optional NAME
root          - set current default server to the root
ls NAME [> FILE] - list the domain NAME, with output optionally going to FILE
view FILE      - sort an 'ls' output file and view it with more

```

2.6 La commande finger

Elle permet de donner des informations sur les utilisateurs d'une machine donnée ou du domaine NIS courant.

En particulier, cette commande liste les informations contenus dans les fichiers `/etc/passwd` `./plan` `./project`, indique le dernier login, quand le courrier a été lu pour la dernière fois ...

Exemple :

```
$ finger gire
Login name: gire                      In real life: Sophie Gire
Directory: /home/ens/gire            Shell: /bin/csh
Last login Wed Jul 30 12:43 on console
New mail received Mon Oct 13 14:35:44 1997;
  unread since Mon Aug 18 18:21:11 1997
Plan:
  Etre toute ma vie en "mise en disponibilite pour convenances personnelles"
  et tout le temps tout le temps tout le temps tout le temps
  faire des patates a l'ail a John Cassavetes
```

Le démon fingerd doit être actif sur la machine.

```
$ finger president@whitehouse.gov
[whitehouse.gov]
```

```
Finger service for arbitrary addresses on whitehouse.gov is not
supported. If you wish to send electronic mail, valid addresses are
"PRESIDENT@WHITEHOUSE.GOV", and "VICE-PRESIDENT@WHITEHOUSE.GOV".
```

2.7 Les commandes en r

Les commandes en `r` (pour *remote-command*) permettent d'exécuter des commandes à distance sur d'autres machines. Pour des raisons de sécurité, certains sites désactivent l'utilisation des commandes en `r`.

2.7.1 La commande rlogin

Elle permet de se connecter sur les machines du réseau local (identique à `telnet`).

```
uneMachine $ rlogin autreMachine
Password:
autreMachine $
```

2.7.2 La commande rcp

Elle permet la copie de fichiers à partir et à destination des autres machines du réseau local.

```
uneMachine $ rcp autreMachine: fichierDistant fichierLocal
```

2.7.3 La commande rsh

Elle permet de transmettre une commande à une autre machine du réseau local, en vue de son exécution. La sortie standard et les erreurs standard sont renvoyées à la machine locale.

```
uneMachine $ rsh autreMachine ls -l /tmp
total 4
drwx-----  2 ribaud  users          4096 Feb 26 07:06 xdvijjHQZh
```

2.7.4 La commande ruser

Elle liste les utilisateurs connectés sur les machines du réseau local.

```
$ rusers
kelenn3      root
kelenn-gw.un ballot ballot ballot ballot
```

2.8 La commande uname

Elle permet d'obtenir des informations sur le système local.

```
$ uname -a
SunOS penfeld-g 4.1.3_U1 2 sun4m
```

2.9 Les commandes hostname et hostid

Elles permettent d'obtenir le nom de la machine et un identificateur unique attribué par le constructeur.

```
$ hostname ; hostid
penfeld-gw
8074f8a8
```

Deuxième partie

Programmation réseaux

Chapitre 3

Les structures et les fonctions d'accès

3.1 Les fichiers d'include

La plupart des définitions de structures et déclarations de fonctions de ce chapitre sont contenues dans les fichiers d'include :

```
#include<sys/types.h>          /* types de base */
#include<sys/socket.h>         /* definitions de base des sockets */
#include<netdb.h>              /* fonctions d'information sur le reseau */
#include<netinet/in.h>         /* sockaddr_in and co */
#include <arpa/inet.h>          /* fonctions de manipulation d'adresse */
```

3.2 Les librairies standard

La compilation dans les environnements Linux ne nécessite pas d'utilisation explicite de librairies. Par contre sous solaris, les différentes fonctions se trouvent dans les librairies `ns1`, `socket` et `resolv`.

3.3 Les types Posix

Le type de données `in_addr_t` doit être un entier sans signe d'au moins 32 bits, le type de données `in_port_t` doit être un entier sans signe d'au moins 16 bits et le type de données `sa_family_t` peut être n'importe quel entier sans signe.

La table 3.1, inspirée de [?], liste ces trois types ainsi que d'autres types Posix.1g qui seront utilisés.

3.4 Les adresses des machines

En IPv4, l'adresse Internet d'une machine est un entier long (4 octets).

Historiquement la structure suivante était utilisée :

```
#include<netinet/in.h>
struct in_addr {
    u_long s_addr; };
```

| TYPE DE DONNÉES | DESCRIPTION | FICHIER D'EN-TÊTE |
|--------------------------|---|-----------------------------------|
| <code>int8_t</code> | Entier 8-bit signé | <code><sys/types.h></code> |
| <code>uint8_t</code> | Entier 8-bit sans signe | <code><sys/types.h></code> |
| <code>int16_t</code> | Entier 16-bit signé | <code><sys/types.h></code> |
| <code>uint16_t</code> | Entier 16-bit sans signe | <code><sys/types.h></code> |
| <code>int32_t</code> | Entier 32-bit signé | <code><sys/types.h></code> |
| <code>uint32_t</code> | Entier 32-bit sans signe | <code><sys/types.h></code> |
| <code>sa_family_t</code> | Famille d'adresse | <code><sys/socket.h></code> |
| <code>socklen_t</code> | Entier 32-bit sans signe | <code><sys/socket.h></code> |
| <code>in_addr_t</code> | Adresse IPv4, normalement un <code>uint32_t</code> | <code><netinet/in.h></code> |
| <code>in_port_t</code> | Port TCP ou UDP, normalement un <code>uint16_t</code> | <code><netinet/in.h></code> |

TABLE 3.1 – Types de données Posix

En Posix, l'adresse internet est :

```
#include<netinet/in.h>
struct in_addr {
    uint32_t s_addr; };
```

Sur Sun, on trouve la définition suivante, compatible avec la première, et permettant d'accéder à différentes interprétations :

```
#include<netinet/in.h>
struct in_addr {
    union {
        struct { uint8_t s_b1,s_b2,s_b3,s_b4; } _S_un_b;
        struct { uint16_t s_w1,s_w2; } _S_un_w;
        uint32_t _S_addr;
    } _S_un;
#define s_addr _S_un._S_addr          /* should be used for all code */
};
```

Sur certains Linux (qui devraient pourtant être Posix ...), on trouve la définition suivante :

```
#include<netinet/in.h>
struct in_addr {
    unsigned int s_addr; };
```

3.5 Les fonctions de manipulation des adresses

3.5.1 `inet_aton`

```
#include <arpa/inet.h>
int inet_aton(char *cp, struct in_addr *inp);
```

Cette fonction convertit la chaîne de caractères `cp` (représentant une adresse en notation pointée) en une adresse sous la forme d'un entier 32-bit pointée par `inp`. La fonction renvoie 1 si l'appel a réussi, 0 sinon.

3.5.2 inet_ntoa

```
#include <arpa/inet.h>
char * inet_ntoa(struct in_addr in);
```

Cette fonction convertit une adresse sous la forme d'un entier 32-bit nommée `in` en une adresse en notation pointée. La fonction renvoie l'adresse de la chaîne si l'appel a réussi, NULL sinon.

3.6 La représentation des nombres

La représentation des nombres sur des machines hétérogènes peut poser des problèmes d'interprétation (Little-Endian ou Big-Endian). Aussi, une représentation standard est adoptée : celle dite *Big Endian* où les octets de poids fort sont les plus à gauche.

Des fonctions de conversion (en général macro-définies) sont fournies dans le fichier `<netinet/in.h>`.

`htnol` et `ntohl` permettent la manipulation des adresses, et `htnos` et `ntohs` permettent celles des numéros de port :

```
u_short ntohs(u_short); /* network to host short */
u_short htons(u_short); /* host to network short */
u_long  ntohl(u_long);  /* network to host long */
u_long  htonl(u_long);  /* host to network long */
```

3.7 La structure hostent

Elle correspond à la liste des informations relatives à une machine au retour d'un appel à l'une des fonctions `gethostname`, `gethostbyname`, `gethostbyaddr`, `gethostent` :

```
#include<sys/socket.h>
#include<netdb.h>
struct hostent {
    char    *h_name;           /* official name of host */
    char    **h_aliases;       /* pointer to array of pointers to alias names */
    int      h_addrtype;        /* host address type : AF_INET or AF_INET6 */
    int      h_length;          /* length of address : 4 or 16 */
    char    **h_addr_list;     /* ptr to array of ptrs with IPv4 or IPv6 addrs */
#define h_addr h_addr_list[0] /* first address in list */
};
```

3.8 Les fonctions de consultation

3.8.1 gethostname

On a souvent besoin du nom de la machine locale.

```
#include <unistd.h>
int gethostname(char *name, size_t len);
```

Cette fonction permet de récupérer à l'adresse **name** le nom de la machine locale (terminé par un `\0`). **len** contient la taille de la zone pointée par **name**. La fonction renvoie 0 si l'appel a réussi, -1 sinon.

3.8.2 gethostbyname

Il est possible d'obtenir une structure **hostent** associée à une machine de nom donné en paramètre : si un serveur de nom est actif, il est interrogé ; sinon les NIS et en dernier le fichier `/etc/hosts`.

```
#include<sys/types.h>
#include<sys/socket.h>
#include<netdb.h>
struct hostent *gethostbyname(const char *name);
```

Le pointeur renvoyé pointe en zone statique et chaque appel écrase le résultat du précédent : il faut donc recopier les résultats avec **memcpy** par exemple.

3.8.3 gethostbyaddr

Lorsqu'on connaît l'adresse Internet, on peut aussi obtenir une structure **hostent**.

```
#include<sys/types.h>
#include<sys/socket.h>
#include<netdb.h>
struct hostent *gethostbyaddr(const char *addr, int len, int type);
```

Dans le cas d'une adresse Internet v4, **addr** pointe sur un entier long, et donc **len** est égal à `sizeof(long)` et **type** est égal à `AF_INET`.

Chapitre 4

Les sockets

4.1 Introduction

4.1.1 Qu'est-ce qu'une socket ?

Une *socket* est un point de communication par lequel un processus peut émettre ou recevoir des informations.

A l'intérieur d'un processus, une socket sera identifiée par un descripteur de même nature que ceux identifiant les fichiers (c'est-à-dire dans le même ensemble). Cette propriété est essentielle car elle permet la redirection des fichiers d'entrée-sortie standard (descripteurs 0, 1 et 2) sur des sockets et donc l'utilisation d'applications standards sur le réseau.

Cela signifie également que tout nouveau processus (créé par **fork**) hérite des descripteurs de son père.

A toute socket est associée dans le système un objet ayant la structure **socket**, définie dans le fichier standard `<sys/socketvar.h>` qui contient l'ensemble des caractéristiques de cet objet.

4.1.2 Type d'une socket

Le *type* d'une socket détermine la sémantique des communications qu'elle permet de réaliser.

L'ensemble des propriétés d'une communication dépend essentiellement du protocole de transport utilisé.

Dans le monde Internet, on utilise essentiellement TCP et UDP, bien qu'on puisse utiliser IP et que l'interface des sockets aie été étendue pour l'utilisation de protocoles de transport conformes au modèle OSI.

Une communication nécessitant l'échange de messages complets et structurés emploie une socket de *type* **SOCK_DGRAM** (datagramme), le protocole sous-jacent est UDP.

Une communication nécessitant l'échange de flots d'information emploie une socket de *type* **SOCK_STREAM** (stream ou flot), le protocole sous-jacent est TCP.

Du fait de la nature des protocoles TCP et UDP, une socket de *type* **SOCK_DGRAM** s'utilise en mode non-connecté et sans garantie de fiabilité et une socket de *type* **SOCK_STREAM** s'utilise en mode connecté et avec garantie du maximum de fiabilité.

Les constantes symboliques `SOCK_DGRAM` et `SOCK_STREAM` sont définies dans le fichier standard `<sys/socket.h>`.

4.1.3 Domaine d'une socket

Le *domaine* d'une socket détermine l'ensemble des autres points de communication qu'elle permet d'atteindre (et donc les processus qui les utilisent).

Il existe plusieurs domaines dont les deux principaux dans le monde Unix sont :

- `AF_UNIX` : le domaine *local* qui permet d'atteindre les processus s'exécutant sur la même machine,
- `AF_INET` : le domaine de l'*internet* qui permet d'atteindre les processus s'exécutant sur une des machines de l'internet.

Dans le domaine `AF_UNIX`, une socket est désignée de la même manière que les fichiers et l'adresse d'une telle socket est définie dans le fichier standard `<sys/un.h>` comme correspondant à la structure suivante :

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

Dans le domaine `AF_INET`, une socket appartient à une machine et est identifiée par un *port*, l'adresse d'une telle socket est définie dans le fichier standard `<netinet/in.h>` comme correspondant à la structure suivante :

```
#include <netinet/in.h>
struct sockaddr_in {
    short    sin_family;    /* AF_INET */
    u_short  sin_port;      /* numero du port associe */
    struct   in_addr sin_addr; /* adresse internet de la machine */
    char     sin_zero[8];   /* tableau de 8 caracteres nuls */
};
```

En Posix, on a la définition suivante :

```
#include <netinet/in.h>
struct sockaddr_in {
    uint8_t    sin_len;    /* longueur de la structure (16) */
    sa_family_t sin_family; /* AF_INET */
    in_port_t  sin_port;    /* numero du port associe */
    struct     in_addr sin_addr; /* adresse internet de la machine */
    char       sin_zero[8]; /* tableau de 8 caracteres nuls */
};
```

4.2 Primitives communes

4.2.1 La création d'une socket : `socket`

Tout processus souhaitant communiquer doit demander à son système local la création d'une socket en spécifiant le type, le domaine et le protocole.

Pour un domaine et un type, il n'existe souvent qu'un seul type de protocole utilisable, dans ce cas on choisit 0 pour le protocole et le système choisit le protocole adapté.

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(
    int domain,      /* AF_UNIX, AF_INET, ... */
    int type,        /* SOCK_DGRAM, SOCK_STREAM, ... */
    int protocol     /* 0 protocole par défaut */
);
```

La valeur de retour est un descripteur sur la socket nouvellement créée si cette création est possible et -1 en cas d'erreur. En cas d'erreur, **errno** vaut :

- **EACCES** : incompatibilité type / protocole,
- **EMFILE** : table des descripteurs pleine,
- **EPROTONOSUPPORT** : protocole non-supporté,
- ... (peut dépendre du système utilisé).

4.2.2 La suppression d'une socket : close

```
#include <unistd.h>
int close(int fd);
```

Une socket est supprimée lors de la fermeture du **dernier descripteur** permettant d'y accéder (appel-système **close**).

Cette suppression libère toutes les ressources allouées.

Cette primitive peut être bloquante dans le cas d'utilisation d'une socket de type **SOCK_STREAM** dans le domaine **AF_INET** au cas où le tampon d'émission n'est pas vide.

4.2.3 L'attachement d'une socket à une adresse : bind

Une différence essentielle avec les fichiers est que le *nommage* d'une socket est une opération différente de sa création : ouverture et création vont de pair comme les *tubes*, mais il est ensuite possible et souvent nécessaire d'attacher une adresse de son domaine à l'objet créé au moyen de la primitive **bind**.

Sans nommage explicite de la socket, on se retrouve dans une situation analogue à celle de la création d'un *tube* sans nom :

seuls les processus ayant hérité du descripteur sur la socket peuvent lire ou écrire sur la socket (car elle est *duplex*), mais personne ne pourrait envoyer de l'extérieur, de données sur la socket (dans le monde réel, essayez de vous faire appeler dans une cabine téléphonique dont vous ne connaissez pas le numéro ...).

Un moyen de désigner la socket indépendamment de tout contexte (en dehors de son domaine) est fourni par **bind** :

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(
    int sockfd,                /* descripteur de la socket à attacher */
    const struct sockaddr *my_addr, /* pointeur sur l'adresse à utiliser */
    socklen_t my_addr_len);
```

```
int addrlen          /* longueur de l'adresse */
);
```

La structure `sockaddr` est générique :

```
#include <sys/socket.h>
struct sockaddr {
    u_short sa_family;          /* address family */
    char     sa_data[14];       /* up to 14 bytes of direct address */
};
```

Il faut, en fonction du domaine, lui substituer la structure correspondante (voir 4.1.3).

N.B. Pour les appels `bind`, `accept`, `connect` et `sendto`, l'utilisation de la structure du domaine provoque un warning à la compilation¹, du genre "warning : passing arg 2 of 'bind' from incompatible pointer type". Pour éviter ce warning, il faudrait faire une conversion (*cast*) à l'appel.

La valeur de retour est 0 si cet attachement est possible et -1 en cas d'erreur. En cas d'erreur, `errno` vaut :

- `EACCES` : adresse protégée,
- `EADDRINUSE` : adresse déjà utilisée,
- `EADDRNOTAVAIL` : adresse incorrecte,
- `EBADF` : `s` est un descripteur invalide,
- ... (peut dépendre du système utilisé).

Cas du domaine `AF_UNIX`

L'attachement d'une socket à une adresse du domaine `AF_UNIX` ne peut se faire que si la référence correspondante n'existe pas (elle peut d'ailleurs être utilisée par autre chose qu'une socket : un fichier régulier, un tube nommé, ...).

On visualise les sockets du domaine `AF_UNIX` avec la commande `netstat` :

```
% netstat -f unix
Active UNIX domain sockets
Address Type  Recv-Q Send-Q Vnode      Conn Refs Nextref Addr
ff64668c stream    0      0      0 ff65f18c    0      0
ff656a8c stream    0      0      0 ff674e0c    0      0 /tmp/.X11-unix/X0
ff64688c dgram     0      0 ff13d6b8    0      0      0 /dev/log
```

Une socket de ce domaine apparaît avec un `s` en début des permissions.

```
% ls -l /tmp/.X11-unix/X0
srwxrwxrwx  1 moi          0 Nov  3 08:07 /tmp/.X11-unix/X0
% ls -l /dev/log
srw-rw-rw-  1 root        0 Sep 19 09:57 /dev/log
```

La primitive `socketpair` permet de créer deux sockets *non nommées* et des les associer (utilisable pour les deux types de sockets `SOCK_DGRAM` et `SOCK_STREAM`).

1. C n'est pas un langage-objet, `struct sockaddr_in` et `struct sockaddr_un` devraient être des sous-types de `struct sockaddr`...


```
#include <sys/types.h>
#include <sys/socket.h>
int socketpair(
    int domain, /* AF_UNIX */
    int type, /* SOCK_DGRAM, SOCK_STREAM */
    int protocol, /* 0 */
    int sv[2] /* tableau de 2 entiers qui contiendra les sockets */
);
```

Les avantages par rapport à un tube nommé POSIX sont que la paire de sockets permet une communication duplex (alors qu'il faudrait deux tubes nommés et quatre descripteurs) et qu'on peut choisir son mode de communication `SOCK_DGRAM` ou `SOCK_STREAM` ce qui conduit à des sémantiques différentes.

Cas du domaine `AF_INET`

Rappelons le format des structures utilisées :

```
#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 associe */
    struct in_addr sin_addr; /* adresse internet de la machine */
    char sin_zero[8]; /* tableau de 8 caracteres nuls */
};
```

Les champs peuvent avoir, à l'appel de `bind`, des valeurs particulières :

- La valeur `INADDR_ANY` de `sin_addr.s_addr` permet d'associer la socket à toutes les adresses IP possibles de la machine (particulièrement important pour les machines "passerelles" : la socket pourra être indifféremment contactée sur n'importe quel des réseaux auquel appartient la passerelle),
sinon il faut utiliser les primitives `gethostname` et `gethostbyname` pour connaître l'adresse de la machine locale.
- L'attachement à un numéro de port particulier est indispensable si ce numéro est public et connu (exemple d'un service dont les clients connaissent le numéro).
Il existe des cas où le numéro de port peut être attribué par le système, en spécifiant 0 dans le champ `sin_port`. Ceci peut conduire le processus à ne pas connaître le port qu'il utilise, il peut le demander au système au moyen de la primitive `getsockname`.

```
#include <sys/types.h>
#include <sys/socket.h>
int getsockname(
    int s, /* descripteur de la socket */
    struct sockaddr *name, /* pointeur sur une structure adresse
                           a remplir par getsockname */
    int *len);
```

```
int *namelen          /* pointeur sur la taille de l'adresse */  
);
```

A l'appel `*namelen` est la taille de la structure adresse réservée pour récupérer le résultat, au retour `*namelen` aura pour valeur la longueur effective de l'adresse.

Attention : les valeurs des champs `sin_addr.s_addr` et `sin_port` sont renvoyées en format réseau (voir 3.6).

N.B. Les noms `name` et `namelen` sont mal choisis, ce ne sont pas des noms de machines mais des adresses. Les noms corrects sont `my_addr` et `addrlen` (comme pour `bind`).

Une fonction de création et d'attachement de socket

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

int creer_socket(int type, int *ptr_port, struct sockaddr_in *ptr_adresse)
{
    struct sockaddr_in adresse;
    int desc;
    int longueur = sizeof(struct sockaddr_in);

    /* cr'eatation de la socket */
    if ((desc = socket(AF_INET, type, 0)) == -1) {
        perror("Creation socket impossible\n");
        return -1;
    }

    /* pr'eparation de l'adresse locale : port + toutes les @ IP */
    adresse.sin_family=AF_INET;
    adresse.sin_addr.s_addr=htonl(INADDR_ANY);
    adresse.sin_port=htons(*ptr_port);

    /* attachement de la socket a' l'adresse locale */
    if (bind(desc, &adresse, longueur) == -1) {
        perror("Attachement de la socket impossible\n");
        close(desc);
        return -1;
    }

    /* dans le cas ou' *port == 0, le syste'me a attribu'e un port */
    /* on r'ecupe're ce num'ero de port avec getsockname */
    if (ptr_adresse != NULL)
        getsockname(desc, ptr_adresse, &longueur);

    /* mise sous forme hote du numero de port */
    *ptr_port = ntohs(ptr_adresse->sin_port);

    return desc;
}
```

4.3 La scrutation

4.3.1 Introduction

Dans un certain nombre de situations, un processus est amené à communiquer (de manière non régulière) avec différentes entités du système et utilise pour cela des objets de nature quelconque (fichiers, tubes, sockets, ...). Les opérations qu'il est amené à réaliser sont très souvent bloquantes et dans le cas d'un serveur, par exemple, il est essentiel que les requêtes soient prises en compte le plus rapidement possible après qu'elles aient été transmises. Le processus est donc amené à scruter ce qui se passe sur différents descripteurs *sans prendre le risque d'être bloqué*. Il existe deux possibilités :

- si les descripteurs ont été positionnés en mode non-bloquant au moyen de la primitive `fcntl` ou `ioctl` :

```
int on=1; ... ; ioctl(desc, FIONBIO; &on); ... } /* ou FIONBIO */
```

on peut essayer de lire alternativement sur chacun des descripteurs (*polling*) ;
- sinon on peut demander à être averti lorsqu'une opération peut être réalisée sur l'un des descripteur au moyen de la primitive `select`, le processus appelant cette primitive se met alors en attente sur des événements relatifs aux lectures et écritures sur un ensemble de descripteurs.

4.3.2 La primitive select

Les ensembles de descripteurs

Le type `fd_set` prédéfini dans le fichier standard `<sys/types.h>` permet de définir des ensembles de descripteurs qui seront utilisés comme paramètres de la primitive `select`. Différentes macros définies dans ce fichier permettent la manipulation des `fd_set`.

| <i>Prototype de la fonction</i> | <i>Effet</i> |
|--|---|
| <code>FD_ZERO(fd_set *ptr_set)</code> | <code>*ptr_set = {}</code> |
| <code>FD_CLR(int desc, fd_set *ptr_set)</code> | <code>*ptr_set = *ptr_set - {desc}</code> |
| <code>FD_SET(int desc, fd_set *ptr_set)</code> | <code>*ptr_set = *ptr_set + {desc}</code> |
| <code>FD_ISSET(int desc, fd_set *ptr_set)</code> | valeur non nulle : desc appartient à <code>*ptr_set</code> |

La primitive select

Cette primitive permet de scruter plusieurs descripteurs (fichiers, tubes, sockets, ...) , en se mettant en attente des événements relatifs aux lectures et écritures.

Trois types d'événements sont susceptibles d'être examinés par la primitive `select` :

- la possibilité de lire sur un descripteur donné,
- la possibilité d'écrire sur un descripteur donné,
- l'existence d'une condition exceptionnelle sur une ressource d'un descripteur pour lequel ce concept existe (comme par exemple l'arrivée d'un message urgent sur une socket du domaine Internet et de type `SOCK_STREAM`).

```
int select(
int width,          /* ensemble de descripteurs a examiner 0, 1, ..., width - 1 */
fd_set *readfds,    /* ensemble de descripteurs en lecture */
fd_set *writefds,    /* ensemble de descripteurs en ecriture */
fd_set *exceptfds,  /* ensemble de descripteurs en exception */
struct timeval *timeout /* temporisation */
);
```

Interprétation des paramètres en entrée

readfds pointe en entrée sur un ensemble de descripteurs sur lequel on souhaite pouvoir réaliser une lecture.

Une valeur **NULL** de ce paramètre correspond à l'ensemble vide (rien à surveiller en lecture).

L'existence dans l'ensemble correspondant d'un descripteur sur un fichier régulier non verrouillé en mode non exclusif de manière non impérative provoque un retour immédiat de la primitive **select** (en effet, la lecture sur un tel fichier n'est pas bloquante).

writefds pointe en entrée sur un ensemble de descripteurs sur lequel on souhaite pouvoir réaliser une écriture.

Une valeur **NULL** de ce paramètre correspond à l'ensemble vide (rien à surveiller en écriture).

L'existence dans l'ensemble correspondant d'un descripteur sur un fichier régulier non verrouillé impérativement (que ce soit en mode partagé ou exclusif) provoque un retour immédiat de la primitive **select** (en effet, l'écriture sur un tel fichier n'est pas bloquante).

exceptfds pointe en entrée sur un ensemble de descripteurs sur lequel on souhaite la réalisation d'un test de condition exceptionnelle.

Une valeur **NULL** de ce paramètre correspond à l'ensemble vide (rien à surveiller).

width doit au moins être égal à la valeur du plus grand descripteur appartenant à l'un des trois ensembles précédents augmentée de 1. Il indique que l'opération porte sur les **width** descripteurs (0, 1, ..., **width** -1).

timeout pointe sur une structure de type **timeval** qui définit un temps maximal d'attente avant que l'une des opérations souhaitées soit possible.

Une valeur **NULL** correspond à un temps d'attente infini.

Retour de la primitive

Un processus appelant est bloqué jusqu'à ce que l'une des conditions suivantes se réalise :

- L'un des événements attendus sur *un des descripteurs de l'un des ensembles* s'est produit. Les ensembles ***readfds**, ***writefds** et ***exceptfds** sont mis à jour et contiennent les descripteurs sur lesquels l'opération correspondante est possible : la macro **FD_ISSET** permet de tester l'appartenance des descripteurs à ces ensembles.
En cas d'utilisation de **select** dans une boucle, il est donc nécessaire de réinitialiser les trois ensembles avant de rappeler la primitive **select**.
La valeur de retour est le nombre total de descripteurs sur lesquels une opération est possible.

- Le temps d'attente maximum s'est écoulé.
La valeur de retour est alors 0, et les trois ensembles sont également mis à jour.
- Un signal capté par le processus est survenu : la valeur de retour est alors -1 et la variable **errno** a la valeur **EINTR**.

En cas d'erreur, la primitive renvoie -1 et **errno** vaut :

- **EBADF** : **s** est un descripteur invalide,
- **EINTR** : le processus appelant **select** a reçu un signal avant qu'un des événements attendus soit arrivé, et le signal a interrompu l'appel-système,
- **EFAULT** : un des pointeurs a une valeur incorrecte,
- **EINVAL** : le temps a une valeur incorrecte.

Chapitre 5

La communication en mode connecté (TCP)

5.1 Principes

C'est le mode de communication associé aux sockets de type `SOCK_STREAM`. Il permet d'échanger des séquences de caractères continues et non structurées en messages.

Dans le cas du domaine `AF_INET`, le protocole sous-jacent est TCP, la communication est donc fiable.

Ainsi que l'indique son nom, le mode connecté nécessite l'établissement d'une connexion (un *circuit virtuel*) entre deux points.

Alors que le mode de communication par datagrammes présente un aspect symétrique dans la mesure où chacun des points de transport (les sockets) sont dans le même état et peut prendre l'initiative de la connexion, le mode connecté met en lumière la dissymétrie des deux processus impliqués :

- l'une des deux entités, en position de *serveur*, est en attente passive de demande de connexion,
- l'autre entité, en position de *client*, doit prendre l'initiative de la connexion en demandant au serveur s'il accepte la connexion.

Après cette phase initiale, on retrouve la symétrie : chaque extrémité peut envoyer et recevoir des caractères.

5.2 Le point de vue du serveur

5.2.1 Introduction

Le rôle du serveur est passif pendant l'établissement de la connexion :

- le serveur doit disposer d'une *socket d'écoute* ou *socket de rendez-vous* attachée au port TCP correspondant au service (et donc supposé connu des clients).
- le serveur doit aviser le système auquel il appartient qu'il est prêt à accepter les demandes de connexion, sur la socket de rendez-vous (par la primitive `listen`).
- puis le serveur se met en attente de connexion sur la socket de rendez-vous (par la primitive `bloquante accept`).

- lorsqu'un client prend l'initiative de la connexion, le processus serveur est **débloqué** et une nouvelle socket, appelée *socket de service*, est créée :
c'est cette socket de service qui est connectée à la socket du client.
- le serveur peut alors déléguer le travail à un nouveau processus créé par **fork** et reprendre son attente sur la socket de rendez-vous,
ou traiter lui-même la demande (mais il risque de faire attendre des nouveaux clients).

5.2.2 L'ouverture du service : listen

```
#include <sys/types.h>
#include <sys/socket.h>
int listen(int s,          /* socket de rendez-vous */
int backlog              /* nombre maximum de connexions pendantes */
);
```

Cette primitive permet à un serveur de déclarer un service ouvert auprès de son entité TCP locale.

Après cet appel, l'entité TCP locale commence à recevoir les demandes de connexions (appelées des *connexions pendantes*), le paramètre **backlog** définit la taille d'une file d'attente où sont mémorisées les connexions pendantes.

La valeur de retour est 0 en cas de réussite. En cas d'erreur, la valeur de retour est -1 et **errno** vaut :

- **EBADF** : **s** est un descripteur invalide,
- **ENOTSOCK** : **s** n'est pas une socket,
- **EOPNOTSUPP** : type de socket incorrect.

5.2.3 La prise en compte d'une connexion : accept

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int s,          /* socket de rendez-vous */
struct sockaddr *addr, /* pointeur sur l'adresse de la socket connectée */
int *addrlen             /* pointeur sur la longueur de l'adresse */
);
```

Cette primitive permet à un serveur d'extraire une connexion *pendante* de la file d'attente associée à la socket de rendez-vous **s**.

Ainsi le serveur prend en compte un nouveau client : une nouvelle socket *de service* est créée, est connectée au client et son descripteur est renvoyé par la primitive **accept**.

L'adresse de la socket du client avec lequel la connexion est établie est écrite dans la zone pointée par **addr** si sa valeur est différente de NULL (la longueur est alors donnée par ***addrlen**).

Un appel à **accept** est normalement *bloquant* s'il n'y a aucune connexion pendante. En cas de succès, il renvoie un entier non-négatif, constituant un descripteur pour la nouvelle socket. En cas d'erreur, la valeur de retour est -1 et **errno** vaut :

- **EBADF** : **s** est un descripteur invalide,
- **EINTR** : le processus appelant **accept** a reçu un signal avant qu'un des clients attendus soit arrivé, et le signal a interrompu l'appel-système,

- ENOTSOCK : s n'est pas une socket,
- EOPNOTSUPP : type de socket incorrect
- EWOULDBLOCK : la socket est non-bloquante et il n'y a pas de connexions pendantes à accepter,
- ... (peut dépendre du système utilisé).

5.2.4 Un exemple de serveur

```
/* Lancement d'un serveur :  serveur_tcp port */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define TRUE 1

main (int argc, char **argv)
{
    int socket_RV, socket_service;
    char buf[256];
    int entier_envoye;
    struct sockaddr_in adresseRV;
    int lgadresseRV;
    struct sockaddr_in adresseClient;
    int lgadresseClient;
    struct hostent *hote;
    unsigned short port;

    /* creation de la socket de RV */
    if ((socket_RV = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        perror("socket");
        exit(1);
    }

    /* preparation de l'adresse locale */
    port = (unsigned short) atoi(argv[1]);

    adresseRV.sin_family = AF_INET;
    adresseRV.sin_port = htons(port);
    adresseRV.sin_addr.s_addr = htonl(INADDR_ANY);
    lgadresseRV = sizeof(adresseRV);
```

```
/* attachement de la socket a l'adresse locale */
if ((bind(socket_RV, (struct sockaddr *) &adresseRV, lgadresseRV)) == -1)
{
    perror("bind");
    exit(3);
}

/* declaration d'ouverture du service */
if (listen(socket_RV, 10)==-1)
{
    perror("listen");
    exit(4);
}

/* boucle d'attente de connexion */
while (TRUE)
{
    printf("Debut de boucle\n");
    fflush(stdout);

    /* attente d'un client */
    lgadresseClient = sizeof(adresseClient);
    socket_service=accept(socket_RV, (struct sockaddr *) &adresseClient, &lgadresseClient);
    if (socket_service==-1 && errno==EINTR)
    { /* reception d'un signal */
        continue;
    }
    if (socket_service==-1)
    { /* erreur plus grave */
        perror("accept");
        exit(5);
    }

    /* un client est arrive */
    printf("connexion acceptee\n");
    fflush(stdout);

    /* lecture dans la socket d'une chaine de caractères */
    if (read(socket_service, buf, 256) < 0)
    {
        perror("read");
        exit(6);
    }
    printf("Chaine recue : %s\n", buf);
    fflush(stdout);

    /* ecriture dans la socket d'un entier */
}
```

```
entier_envoye = 2006 ;
if (write(socket_service, &entier_envoye, sizeof(int)) != sizeof(int))
    {
        perror("write");
        exit(7);
    }
printf("Entier envoye : %d\n", entier_envoye);

close(socket_service);
}
close(socket_RV);
}
```

5.3 Le point de vue du client

5.3.1 Introduction

Le rôle du client est actif pendant l'établissement de la connexion :

- le client doit disposer d'une socket attachée à un port TCP quelconque.
- le client doit construire l'adresse du serveur, dont il doit connaître le numéro de port de la socket de rendez-vous, (il obtient éventuellement l'adresse IP du serveur à partir de son nom par la primitive `gethostbyname`).
- puis le client demande la connexion de sa socket locale à la socket de rendez-vous du serveur (par la primitive **bloquante** `connect`).
- lorsque le serveur accepte la connexion, le processus client est **débloqué** et peut dialoguer avec le serveur.

5.3.2 La demande de connexion : `connect`

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockfd,          /* socket locale */
struct sockaddr *serv_addr,    /* pointeur sur l'adresse du serveur */
int addrlen                    /* longueur de l'adresse */
);
```

Contrairement à ce qui se passait dans le cas d'une demande de pseudo-connexion (voir 6.5), le processus appelant `connect` est bloqué jusqu'à ce que la négociation entre les deux entités TCP concernées soit achevée (il est cependant possible de rendre l'appel non bloquant).

La demande de connexion réussit, et la primitive renvoie 0 si les conditions suivantes sont réalisées :

1. les paramètres sont localement corrects,
2. il existe une socket de type `SOCK_STREAM` attachée à l'adresse `*name` (dans le même domaine que la socket locale `s`) et cette socket est dans l'état `LISTEN` (c'est-à-dire qu'un appel à `listen` a été réalisé pour cette socket),
3. la socket locale `s` n'est pas déjà connectée,
4. la socket d'adresse `*name` n'est pas actuellement utilisée dans une autre connexion (sauf paramétrage exceptionnel de cette socket),
5. la file des connexions pendantes de la socket distante n'est pas pleine.

Dans le cas où l'une des quatre premières conditions n'est pas remplie, la fonction renvoie -1 et `errno` vaut :

- `EBADF` : `s` est un descripteur invalide,
- `EINTR` : le processus appelant `connect` a reçu un signal avant que la connexion aie eu lieu, et le signal a interrompu l'appel-système,
- `ENOTSOCK` : `s` n'est pas une socket,
- `EOPNOTSUPP` : type de socket incorrect
- `EISCONN` : la socket locale est déjà connectée,
- `EADDRINUSE` : la socket distante est déjà connectée,
- ... (peut dépendre du système utilisé).

Si la file d'attente est pleine, le comportement est particulier :

- Si la socket locale est en mode bloquant, le processus est bloqué. La demande de connexion est itérée pendant un certain temps : si au bout de ce laps de temps la connexion n'a pas pu être établie, la demande est abandonnée. La fonction renvoie alors -1 et **errno** vaut ETIMEDOUT.
- Si la socket locale est en mode non-bloquant, le retour est immédiat avec la valeur de retour -1 et **errno** qui vaut EINPROGRESS. Cependant ce retour ne correspond pas à une véritable erreur, la demande de connexion n'est pas abandonnée mais automatiquement réitérée comme dans le mode bloquant.

Pour déterminer si la connexion a pu être établie, le processus pourra utiliser la primitive **select** en testant le descripteur en écriture.

Si une autre demande de connexion est formulée avant que cet essai de connexion soit terminé, la fonction renvoie -1 et **errno** vaut EALREADY.

5.3.3 Un exemple de client

```
/* Lancement d'un client : client_tcp port machine_serveur */
```

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

```
main (int argc, char **argv)
{
    int sock;
    char buf[256];
    int entier_recu;
    struct sockaddr_in adresse_serveur;
    struct hostent *hote;
    unsigned short port;
```

```
/* creation de la socket locale */
if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
{
    perror("socket");
    exit(1);
}
```

```
/* recuperation de l'adresse IP du serveur (a partir de son nom) */
if ((hote = gethostbyname(argv[2])) == NULL)
{
    perror("gethostbyname");
}
```

```
    exit(2);
}

/* preparation de l'adresse du serveur */
port = (unsigned short) atoi(argv[1]);

adresse_serveur.sin_family = AF_INET;
adresse_serveur.sin_port = htons(port);
bcopy(hote->h_addr, &adresse_serveur.sin_addr, hote->h_length);
printf("L'adresse du serveur en notation pointee %s\n", inet_ntoa(adresse_serveur.sin_addr));
fflush(stdout);

/* demande de connexion au serveur */
if (connect(sock, (struct sockaddr *) &adresse_serveur, sizeof(adresse_serveur))==-1)
{
    perror("connect");
    exit(3);
}
/* le serveur a accepte la connexion */
printf("connexion acceptee\n");
fflush(stdout);

/* ecriture dans la socket d'une chaine */
strcpy(buf, "Bonne année");
if (write(sock, buf, strlen(buf)+1) != strlen(buf)+1)
{
    perror("write");
    exit(4);
}
printf("Chaine envoyée : %s\n", buf);
fflush(stdout);

/* lecture dans la socket d'un entier */
if (read(sock, &entier_recu, sizeof(int)) < 0)
{
    perror("read");
    exit(5);
}
printf("Entier reçu : %d \n", entier_recu);
fflush(stdout);
}
```

5.4 Le dialogue client/serveur

5.4.1 Introduction

Une fois la connexion établie, la communication redevient symétrique. Les deux processus peuvent échanger des suites de caractères en *duplex* par l'intermédiaire des deux sockets connectées.

Une différence essentielle est que, dans la communication via des sockets de type `SOCK_DGRAM`, le découpage de l'envoi en messages n'est pas préservée à la réception (une lecture peut provenir de plusieurs écritures).

Par ailleurs dans le cas du domaine `AF_INET`, le protocole sous-jacent (TCP) garantit que les caractères seront reçus dans le même ordre que celui de leur envoi, à l'exception de la possibilité d'adresser un caractère dit urgent ou hors bande (*Out Of Band*).

5.4.2 L'envoi de caractères

L'interface standard `write`

Une première solution est d'utiliser `write` :

```
#include <sys/types.h>
#include <sys/uio.h>
int write(int fd,          /* descripteur */
const void *buf,          /* adresse du buffer a envoyer */
int count                 /* nombre d'octets maximum \a envoyer */
);
```

La fonction renvoie le nombre de caractères envoyés (éventuellement 0) et -1 en cas d'erreur.

L'interface standard `send`

```
#include <sys/types.h>
#include <sys/socket.h>
int send(int s,
const void *msg,
int len,
unsigned int flags
);
```

Les trois premiers paramètres sont identiques à ceux de `write`. La primitive `send` permet pour les sockets de type `SOCK_STREAM` d'utiliser l'option `MSG_OOB` dans le domaine `AF_INET` qui indique que les données à transmettre ont un caractère urgent.

La fonction renvoie le nombre de caractères envoyés (éventuellement 0). Dans le cas où le tampon d'émission est plein, le processus est bloqué. Dans le cas où la connexion a été fermée par l'autre extrémité, on se trouve dans la même situation que l'écriture dans un *tube* sans lecteur : le processus écrivain reçoit le signal `SIGPIPE` ce qui entraîne sa terminaison (comportement par défaut).

En cas d'erreur, la valeur de retour est -1 et `errno` vaut :

- EBADF : `s` est un descripteur invalide,
- EINTR : le processus appelant `send` a reçu un signal avant que les caractères puissent être "bufferisés" pour être envoyées, et le signal a interrompu l'appel-système,
- ENOTSOCK : `s` n'est pas une socket,
- EWOULDBLOCK : la socket est non-bloquante et l'opération demandée serait bloquante,
- ... (peut dépendre du système utilisé).

5.4.3 La réception de caractères

L'interface standard `read`

Une première solution est d'utiliser `read` :

```
#include <sys/types.h>
#include <sys/uio.h>
int read(int fd,          /* descripteur */
void *buf,               /* adresse du buffer pour recevoir les donnees */
int count                /* nombre d'octets maximum \ 'a recevoir */
);
```

La fonction renvoie le nombre de caractères reçus (éventuellement 0) et -1 en cas d'erreur.

L'interface standard `recv`

```
#include <sys/types.h>
#include <sys/socket.h>
int recv(int s,
void *msg,
int len,
unsigned int flags
);
```

Les trois premiers paramètres sont identiques à ceux de `read`.

Le dernier paramètre a soit la valeur 0, soit une combinaison bit à bit des constantes symboliques `MSG_PEEK` (pour consulter sans extraire) et `MSG_OOB` dans le domaine `AF_INET` (pour lire une donnée urgente).

La fonction renvoie le nombre de caractères reçus (éventuellement 0, ce qui indique que la connexion a été fermée). Dans le cas où le tampon de réception est vide, le processus est bloqué.

En cas d'erreur, la valeur de retour est -1 et `errno` vaut :

- EBADF : `s` est un descripteur invalide,
- EINTR : le processus appelant `send` a reçu un signal avant que les caractères puissent être reçus, et le signal a interrompu l'appel-système,
- ENOTSOCK : `s` n'est pas une socket,
- EWOULDBLOCK : la socket est non-bloquante et l'opération demandée serait bloquante,
- ... (peut dépendre du système utilisé).

5.5 La fermeture de la connexion

5.5.1 La primitive `close`

L'appel de la primitive `close` sur le dernier descripteur d'une socket entraîne la suppression de la socket.

Cependant, dans le cas d'une socket de type `SOCK_STREAM` du domaine `AF_INET`, s'il existe encore des données dans le tampon d'émission de la socket, le système continue d'essayer de les acheminer avant de libérer complètement les ressources (il est d'ailleurs possible de rendre cette primitive bloquante).

5.5.2 La primitive `shutdown`

```
#include <sys/types.h>
#include <sys/socket.h>
int shutdown(
int s,
int how
);
```

Elle permet de rendre compte, au niveau de la fermeture d'une socket, de l'aspect *duplex* de la communication.

Le paramètre `how` détermine le niveau de fermeture souhaité :

- 0 : la socket n'accepte plus d'opérations de lecture : un appel à `read` ou à `recv` renverra 0 ;
- 1 : la socket n'accepte plus d'opérations d'écriture : un appel à `write` ou à `send` provoquera la génération d'un exemplaire du signal `SIGPIPE`, et s'il est capté, une valeur de retour -1 (`errno = EPIPE`) ;
- 2 : la socket n'accepte plus ni opérations de lecture, ni opérations d'écriture.

5.6 Squelettes de serveur et de client

Les squelettes qui suivent peuvent être utilisés pour fabriquer plus facilement des serveurs et des clients TCP.

5.6.1 Le squelette d'un serveur créant un processus de service

Le code de ce serveur-type, une fois un client accepté, confie la gestion du dialogue avec ce client à un processus dédié à cette gestion (grâce à `fork` puis `exec`).

Le lancement du serveur sera réalisé avec deux paramètres :

- le premier correspond au numéro du port d'attachement de la socket d'écoute (de rendez-vous),
- le deuxième correspond à la référence du fichier binaire exécuté par un processus de service : dans ce processus, la socket sera associée à son entrée standard (qui est donc redirigée sur la socket de service avant le recouvrement par `exec`).

```
/* Lancement d'un serveur :  serveur_tcp port reference */
```

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

```
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define TRUE 1

struct sigaction action;

int creer_socket(int type, int *ptr_port, struct sockaddr_in *ptr_adresse)
{
    struct sockaddr_in adresse;
    int desc;
    int longueur = sizeof(struct sockaddr_in);

    if ((desc = socket(AF_INET, type, 0)) == -1) {
        perror("Creation socket impossible\n");
        return -1;
    }

    adresse.sin_family=AF_INET;
    adresse.sin_addr.s_addr=htonl(INADDR_ANY);
    adresse.sin_port=htons(*ptr_port);

    if (bind(desc, &adresse, longueur) == -1) {
        perror("Attachement de la soccket impossible\n");
        close(desc);
        return -1;
    }

    if (ptr_adresse != NULL)
        getsockname(desc, ptr_adresse, &longueur);
    return desc;
}

void eliminer_zombie(int sig){
    printf("terminaison d'un processus de service\n");
    wait(NULL);
}

main(int argc, char *argv[]){
    struct sockaddr_in adresse;
    int lg_adresse;
    int port;
```

```
int socketRV, socket_service;
if (argc!=3){
    fprintf(stderr, "Nombre de parametres incorrect\n");
    exit(2);}

/* test d'existence du programme a executer pour le service */
if (access(argv[2], X_OK)!=0){
    fprintf(stderr, "Fichier %s non executable\n", argv[2]);
    exit(2);}

/* detachement du terminal */
if (fork()!=0) exit(0);
setsid();
printf("serveur de pid %d lance\n", getpid());

/* handler de signal SIGCHLD */
action.sa_handler=eliminer_zombie;
sigaction(SIGCHLD, &action, NULL);

/* creation et attachement de la socket d'ecoute */
port=atoi(argv[1]);
lg_adresse=sizeof(adresse);
if ((socketRV=creer_socket(SOCK_STREAM, &port, &adresse))== -1){
    fprintf(stderr, "Creation socket ecoute impossible\n");
    exit(2);}

/* declaration d'ouverture du service */
if (listen(socketRV, 10)==-1){
    perror("listen");
    exit(2);}

/* boucle d'attente de connexion */
while(TRUE){
    socket_service=accept(socketRV, &adresse, &lg_adresse);

    /* reception d'un signal (probablement SIGCHLD) */
    if (socket_service== -1 && errno==EINTR)
        continue;

    /* erreur plus grave */
    if (socket_service== -1){
        perror("accept");
        exit(2);}
    printf("connexion acceptee\n");

    /* lancement du processus de service */
    if (fork()==0){
```

```

/* il n'utilise plus la socket d'ecoute */
close(socketRV);
/* redirection de l'entree standard sur socket_service */
dup2(socket_service, STDIN_FILENO);
close(socket_service); /* descripteur inutile */
execlp(argv[2], argv[2], NULL);
perror("execlp");
exit(2);}

/* le serveur principal n'utilise pas socket_service */
close(socket_service);
}
}

```

5.6.2 Le squelette d'un client

Le code de ce client-type pourra se connecter à un serveur-type donné plus haut. On suppose qu'une fois la connexion avec le serveur établie, le client exécute la fonction `client_service` qui reçoit en premier paramètre le descripteur de la socket du client, en second le nombre de paramètres spécifiques et en troisième un vecteur de ces différents paramètres.

Le lancement du client sera réalisé avec plusieurs paramètres :

- le premier est le nom de la machine où s'exécute le serveur,
- le deuxième correspond au numéro du port du service contacté,
- les différents paramètres spécifiques.

```

/* Lancement d'un client : client_tcp serveur port liste_parametres */

```

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void client_service(int socket_client, int argc, char *argv[]);

int creer_socket(int type, int *ptr_port, struct sockaddr_in *ptr_adresse)
{
    struct sockaddr_in adresse;
    int desc;
    int longueur = sizeof(struct sockaddr_in);

    if ((desc = socket(AF_INET, type, 0)) == -1) {
        perror("Creation socket impossible\n");
        return -1;
    }
}

```

```
}

adresse.sin_family=AF_INET;
adresse.sin_addr.s_addr=htonl(INADDR_ANY);
adresse.sin_port=htons(*ptr_port);

if (bind(desc, &adresse, longueur) == -1) {
perror("Attachement de la socket impossible\n");
close(desc);
return -1;
}

if (ptr_adresse != NULL)
getsockname(desc, ptr_adresse, &longueur);
return desc;
}

main(int argc, char *argv[]){
int port;
int socket_client;
struct hostent *hp;
struct sockaddr_in adresse_serveur, adresse_client;

if (argc<3){
fprintf(stderr, "Nombre de parametres incorrect\n");
exit(2);}
/* test d'existence du serveur */
if ((hp=gethostbyname(argv[1]))==NULL){
fprintf(stderr, "Serveur %s inconnu\n", argv[1]);
exit(2);}

/* creation et attachement de la socket du client (port quelconque) */
port=0;
if ((socket_client=creer_socket(SOCK_STREAM, &port, &adresse_client))===-1){
fprintf(stderr, "Creation socket client impossible\n");
exit(2);}
printf("client sur le port %d\n", ntohs(adresse_client.sin_port));

/* preparation de l'adresse du serveur */
adresse_serveur.sin_family=AF_INET;
adresse_serveur.sin_port=htons(atoi(argv[2]));
memcpy(&adresse_serveur.sin_addr.s_addr, hp->h_addr, hp->h_length);

/* demande de connexion au serveur */
if (connect(socket_client, &adresse_serveur, sizeof(adresse_serveur))===-1){
perror("connect");
exit(2);}
```

```
printf("connexion acceptee\n");  
client_service(socket_client, argc-3, argv+3);  
}
```


Chapitre 6

La communication en mode non connecté (UDP)

6.1 Principes

Quelque soit le domaine (`AF_UNIX` ou `AF_INET`), les principales caractéristiques de la communication par datagrammes sont :

- lorsqu'un datagramme est envoyé, l'expéditeur n'obtient aucune information sur l'arrivée de son message à destination,
- les limites des messages sont préservées.

La communication avec UDP (échange de datagrammes) est réalisée par l'intermédiaire de sockets de type `SOCK_DGRAM` dans le domaine `AF_INET`.

Un processus souhaitant émettre un message à destination d'un autre doit :

- d'une part disposer d'un point de communication local (descripteur de socket sur le système local obtenu avec la primitive `socket`, et éventuellement nommé avec la primitive `bind`),
- d'autre part connaître une adresse (adresse IP et numéro de port) sur le système auquel appartient son interlocuteur (en espérant que cet interlocuteur dispose d'une socket **attachée** à cette adresse ... ce type de communication ne permettant pas d'en être certain).

Les sockets sont utilisés en mode non connecté : en principe, toute demande d'envoi doit comporter l'adresse de la socket destinataire (bien qu'il soit possible d'établir une *pseudo-connexion* entre deux sockets de type `SOCK_DGRAM`).

6.2 Primitives d'envoi et de réception

6.2.1 Introduction

Un processus désirant communiquer avec le monde extérieur par l'intermédiaire d'une socket de type `SOCK_DGRAM` doit réaliser, selon les circonstances, un certain nombre des opérations suivantes :

- demander la création d'une socket dans le domaine adapté aux applications avec lesquelles il souhaite communiquer,
- demander éventuellement l'attachement de cette socket sur un port convenu ou un port quelconque selon qu'il joue un rôle de serveur attendant des requêtes ou celui de client prenant l'initiative d'interroger un serveur,
- construire l'adresse de son interlocuteur en mémoire : tout client désirant s'adresser à un serveur doit en connaître l'adresse et donc la préparer en mémoire (éventuellement avec la primitive `gethostbyname` pour obtenir l'adresse IP à partir du nom) ; s'il s'agit d'un serveur, il recevra l'adresse de son émetteur avec chaque message ;
- procéder à des émissions et des réceptions de messages ; les sockets du type `SOCK_DGRAM` sont en mode non connecté ; chaque demande d'envoi s'accompagne de la spécification complète de l'adresse du destinataire (bien qu'il soit possible de réaliser des "pseudo-connexions" cf. paragraphe 6.5).

6.2.2 La primitive `sendto`

```
#include <sys/types.h>
#include <sys/socket.h>
int sendto(
int s,                /* descripteur de la socket d'émission */
const void *msg,      /* adresse du message a envoyer */
int len,              /* longueur du message */
unsigned int flags,   /* option = 0 pour le type SOCK_DGRAM */
const struct sockaddr *to, /* pointeur sur l'adresse du destinataire */
int tolen             /* longueur de l'adresse de la socket destinataire */
);
```

Un appel à `sendto` correspond à la demande d'envoi, via la socket `s`, du message pointé par `msg` de longueur `len` (éventuellement nulle), à destination de la socket attachée à l'adresse pointée par `to` de longueur `tolen`.

La valeur de retour est le nombre de caractères envoyé en cas de réussite, et -1 en cas d'échec. Seules les erreurs locales sont détectées, notamment il n'y a aucune détection qu'une socket est effectivement attachée à l'adresse destinataire (si c'est faux, le message sera perdu et l'émetteur n'en sera pas avisé).

En cas d'erreur, `errno` vaut :

- `EBADF` : `s` est un descripteur invalide,
- `EINTR` : le processus appelant `sendto` a reçu un signal avant que les données aient été "bufferisés", et le signal a interrompu l'appel-système,
- `EINVAL` : longueur d'adresse incorrecte,
- ... (peut dépendre du système utilisé).

6.2.3 La primitive `recvfrom`

```
#include <sys/types.h>
#include <sys/socket.h>
int recvfrom(
    int s,                /* descripteur de la socket de reception */
    void *buf,            /* adresse de recuperation du message reçu */
    int len,              /* taille de la zone allouee a l'adresse buf */
    unsigned int flags,    /* 0 ou MSG_PEEK */
    struct sockaddr *from, /* pointeur pour recuperer l'adresse d'expedition */
    int *fromlen           /* pointeur sur la longueur de l'adresse
                           de la socket d'expedition */
);
```

Cette primitive permet à un processus d'extraire un message sur une socket dont il possède un descripteur, s'il existe un message (sinon l'appel est bloquant).

Le rôle du paramètre `len` est de donner la longueur de la taille réservée pour mémoriser le message (des caractères seront perdus si cette taille est inférieure à la longueur du message).

Dans le cas où `from` est différent de `NULL`, au retour `from` pointe sur l'adresse d'expédition et `*fromlen` contient la longueur de cette adresse.

En cas d'erreur, `errno` vaut :

- `EBADF` : `s` est un descripteur invalide,
- `EINTR` : le processus appelant `recvfrom` a reçu un signal avant que les données aient été reçus, et le signal a interrompu l'appel-système,
- `EWOULDBLOCK` : la socket est non-bloquante et l'opération demandée serait bloquante,
- ... (peut dépendre du système utilisé).

La valeur de retour est le nombre de caractères reçus en cas de réussite, et -1 en cas d'échec.

6.3 Un exemple

L'"émetteur" envoie la chaîne de caractères "salut" et attend la réponse. Le "receveur" attend l'envoi et répond "tchao".

Code de l'"émetteur" Utilisation : emetteur port machine_distante

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

main(int argc, char **argv)
{
    int sock, envoye, recu;
    char buf[256];
    struct sockaddr_in adr;
    int lgadr;
    struct hostent *hote;

    /* cr'eaton de la socket */
    if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("socket"); exit(1);}

    /* recherche de l'@ IP de la machine distante */
    if ((hote = gethostbyname(argv[2])) == NULL){
        perror("gethostbyname"); exit(2);}

    /* pr'eparation de l'adresse distante : port + la premier @ IP */
    adr.sin_family = AF_INET;
    adr.sin_port = htons(atoi(argv[1]));
    bcopy(hote->h_addr, &adr.sin_addr, hote->h_length);
    printf("L'adresse en notation pointee %s\n", inet_ntoa(adr.sin_addr));

    /* echange de datagrammes */
    strcpy(buf, "salut");
    lgadr = sizeof(adr);
    if ((envoye = sendto(sock, buf, strlen(buf)+1, 0, &adr, lgadr)) != strlen(buf)+1) {
        perror("sendto"); exit(1);}
    printf("salut envoye\n");
    lgadr = sizeof(adr);
    if ((recu = recvfrom(sock, buf, 256, 0, &adr, &lgadr)) == -1) {
        perror("recvfrom"); exit(1);}
    printf("j'ai recu %s\n", buf);
}
```

Code du "receveur" Utilisation : `receveur port`

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

main (int argc, char **argv)
{
    int sock,recu,envoye;
    char buf[256], nomh[50];
    struct sockaddr_in adr;
    int lgadr;
    struct hostent *hote;

    /* cr'creation de la socket */
    if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("socket"); exit(1);}

    /* r'ecup'eration du nom de la machine pr'esente */
    if (gethostname(nomh, 50) == -1) {
        perror("gethostbyname"); exit(1);}
    printf("Je m'execute sur %s\n", nomh);

    /* pr'eparation de l'adresse locale : port + toutes les @ IP */
    adr.sin_family = AF_INET;
    adr.sin_port = htons(atoi(argv[1]));
    adr.sin_addr.s_addr = htonl(INADDR_ANY);

    /* attachement de la socket a' l'adresse locale */
    lgadr = sizeof(adr);
    if ((bind(sock, &adr, lgadr)) == -1) {
        perror("bind"); exit(1);}

    /* 'echange de datagrammes */
    lgadr = sizeof(adr);
    if ((recu = recvfrom(sock, buf, 256, 0, &adr, &lgadr)) == -1) {
perror("recvfrom"); exit(1);}
    printf("j'ai recu %s\n", buf);
    strcpy(buf, "tchao");
    if ((envoye = sendto(sock, buf, strlen(buf)+1, 0, &adr, lgadr)) != strlen(buf)+1) {
        perror("sendto"); exit(1);}
    printf("reponse envoyee ...adios\n");
}
```

6.4 Les messages

Les primitives `sendmsg` et `recvmsg` permettent d'envoyer et de recevoir un message dont le contenu est constitué de fragments non contigus en mémoire.

Les différents fragments, de type `struct iovec` (défini dans le fichier standard `<sys/uio.h>`) sont stockés dans un tableau `msg_iov` d'éléments de type `struct iovec`, ce tableau appartient à une structure de type `struct msghdr` (définie dans le fichier standard `<sys/socket.h>`).

Les fragments sont rassemblés en un seul message avant envoi, et lors de la réception le message reçu est récupéré sous forme de fragments qui sont stockés selon les indications données par une structure de type `struct msghdr`.

6.5 Les pseudo-connexions

6.5.1 Introduction

Chaque demande d'émission d'un message nécessite la spécification de l'adresse de la socket destinataire.

Il est possible de mémoriser dans la structure `socket` associée à une socket l'adresse d'une socket *paire* vers laquelle seront automatiquement dirigés les messages émis, sans préciser l'adresse à chaque envoi (ceci ne change rien aux caractéristiques de la communication qui sont celles d'UDP).

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int s,
struct sockaddr *serv_addr,
int addrlen
);
```

La primitive `connect` permet d'associer à une socket locale, identifiée par `s` à la socket d'adresse `*serv_addr` (de longueur `addrlen`).

Tout nouvel appel à `connect` annule la demande précédente, et si `serv_addr` a la valeur `NULL`, la socket `s` n'est plus associée à aucune autre.

6.5.2 Les primitives d'envoi et de réception

Une fois pseudo-connectée, une socket ne permet plus d'émettre vers des sockets différentes de celle de la socket *paire*.

En particulier, toute demande d'envoi par `sendto`, même avec l'adresse de la socket *paire*, échouera (valeur `EINVAL` de `errno`).

De même, si un message arrive d'une autre socket, toute tentative de réception entraînera une erreur (valeur `EISCONN` de `errno`) ; il faudra rompre la connexion pour lire ce message.

La destination étant implicite, l'envoi de message se fait soit par la primitive standard sur les descripteurs : `write` ou par la primitive spécifique aux sockets :

```
#include <sys/types.h>
#include <sys/socket.h>
int send(int s,
const char *msg,
int len,
unsigned int flags
);
```

Par rapport à `write`, la primitive `send` n'apporte rien pour les sockets de type `SOCK_DGRAM` dans la mesure où aucune option n'est supportée.

La réception de message peut se faire par la primitive standard sur les descripteurs : `read`, dans ce cas, s'agissant d'une communication en mode datagramme, un appel à `read` provoquera l'extraction d'un message complet.

On peut utiliser la primitive spécifique aux sockets :

```
#include <sys/types.h>
#include <sys/socket.h>
int recv(int s,
char *buf,
int len,
unsigned int flags
);
```

L'emploi de la primitive `recv` permet de consulter la socket sans rien extraire, en utilisant l'option `MSG_PEEK`.