Unité d'Enseignement : Système et Réseaux

L3I parcours MIAGE

Gilles Menez

Université de Nice – Sophia-Antipolis
Département d'Informatique
email : menez@unice.fr
www : www.i3s.unice.fr/~menez

28 septembre 2021: V 1.1

Table des Matières

Communications Interprocessus

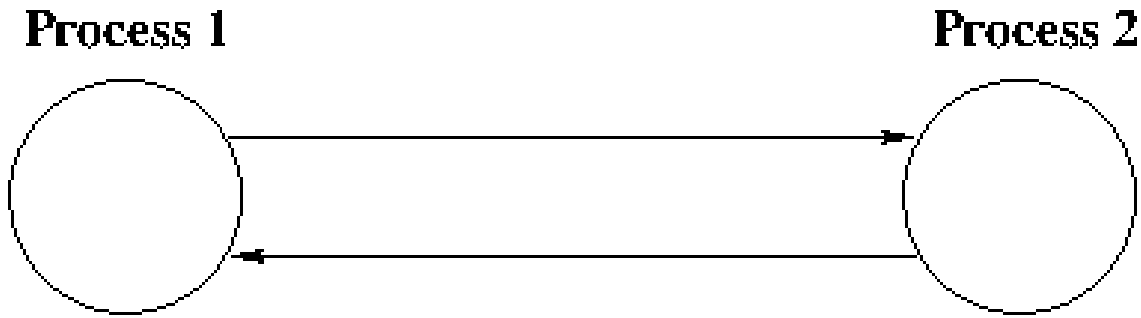
Avant de voir plus précisément comment échanger des informations entre processus, on évoque l'**organisation des programmes**

- reposant sur l'hypothèse/choix/utilisation de **"fonctionnalités partagées" entre processus distants et collaboratifs**

On va donc parler d'**"architecture client-serveur"** !

Problématique

L'objectif de la "communication interprocessus" est de permettre l'**échange de données entre deux processus** :



Cette problématique, telle que l'on va la considérer, recouvre les deux cas :

1. **communication locale** : les deux processus résident **sur la même machine** et **ne partagent pas** de données communes dans leur espace d'adressage ce qui exclut l'utilisation des thread et shm mais qui peut inclure les pipes.
2. **communication distante** : les deux processus résident **sur des machines différentes** reliées par un ou plusieurs réseaux de communication.

On n'oublie pas aussi que l'échange doit être **synchronisé** pour que le dialogue ait un sens.

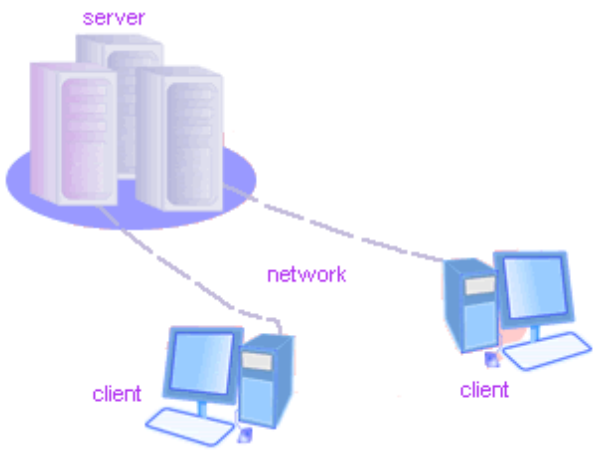
Architecture/Modélisation

Le souhait de faire communiquer des processus **doit nécessairement** s'accompagner d'une **réflexion sur l'architecture informatique** d'un tel système avec comme préoccupation :

transfert d'information **et** synchronisation.

Ainsi, beaucoup des premières applications "réseau" se fondent sur le **paradigme client-serveur** (" **two tiers - client/server architecture** ")

Historiquement ce type d'architecture logicielle est mis en place lorsqu'une application :

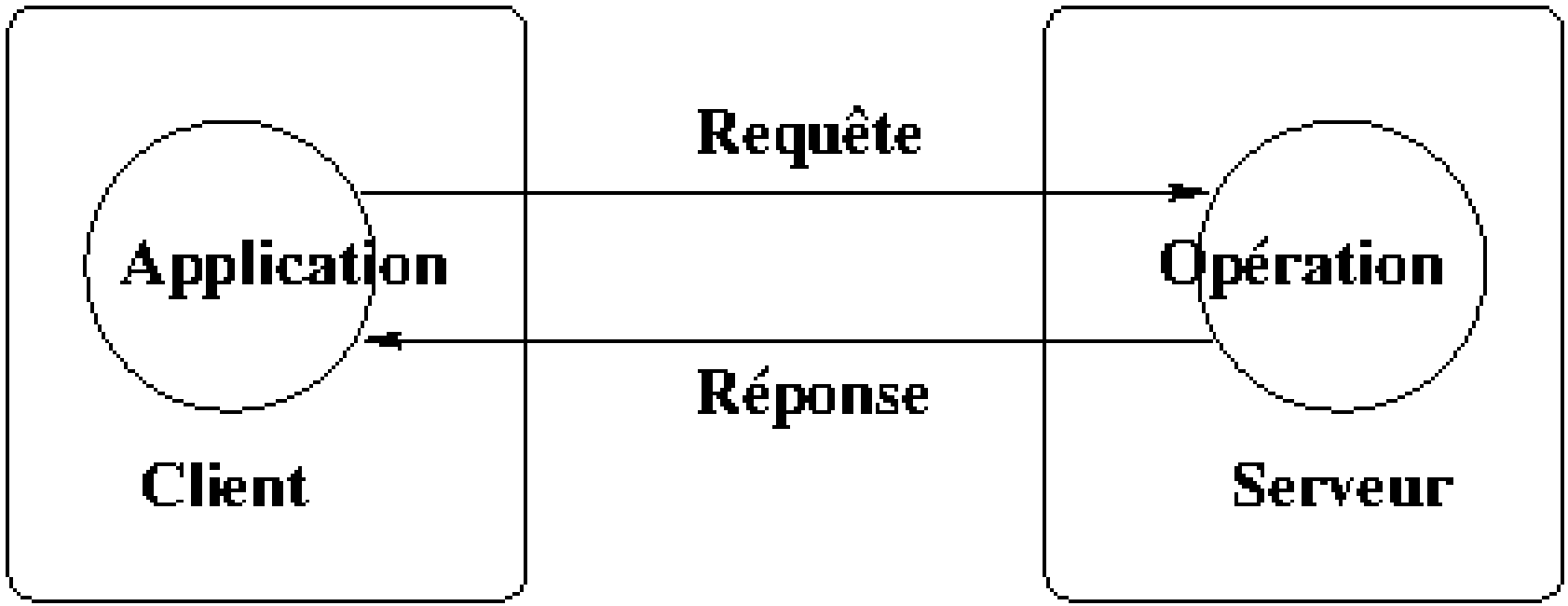


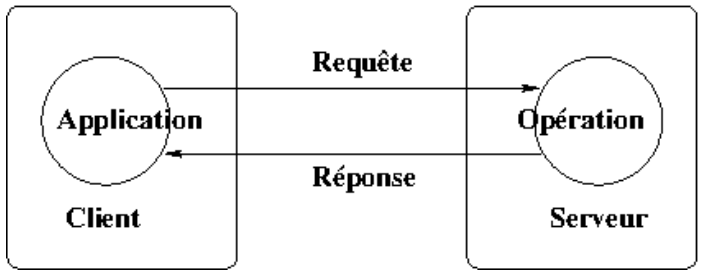
- ➢ prévoit de stocker d'innombrables données sur un serveur (puissant, onéreux . . .)
- ➢ tandis que le traitement et l'interface utilisateur échoit au logiciel client exécuté sur des ordinateurs individuels aux capacités/performances nettement plus modestes.

Chaque client gère sa session et chaque client doit être maintenu : **NOT** "scalable" !

Eléments de définition du modèle client-serveur

Clients et Serveurs sont des **entités distinctes fonctionnant de concert** sur un réseaux pour accomplir une tâche.





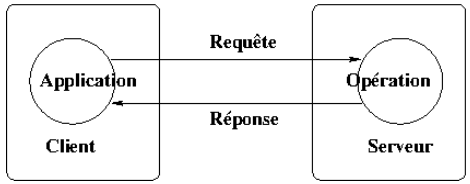
① Ce modèle induit l'**attribution de rôles aux entités** participantes :

➠ Le processus **serveur dispose de ressources** qu'il peut sélectivement partager. Il a une position plutôt "attentiste".

Il réalise une opération sur demande/requête d'un client et transmettant la réponse à ce client.

➠ Le(s) **client(s)** est un processus/application qui **initie un contact** et donc un dialogue avec le serveur dans l'objectif de **profiter des ressources** de ce dernier.

Il demande l'exécution d'une opération au serveur par envoi d'un message contenant le descriptif de l'opération à exécuter et "attend" la réponse à cette opération par un message en retour.



② Ce modèle suppose un **dialogue** :

➠ Les programmes **répartis** entre **processus clients et serveurs** communiquent par messages selon un "pattern" :

requêtes $\longleftrightarrow$ **réponses**.

La **Requête** est un message transmis par un client à un serveur décrivant l'opération à exécuter pour le compte de l'application cliente.

La **Réponse** est un message transmis par un serveur à un client suite à l'exécution d'une opération contenant les paramètres de retour de l'opération

➠ Ce dialogue (communication) se fait dans un langage et des régles de dialogues communs : c'est un **protocole de communication** de **niveau application**.

Notons que "la plupart du temps", un client ne dialogue qu'avec un serveur à la fois alors qu'un serveur est certainement appelé à gérer plusieurs connexions simultanément :

A **server** based network

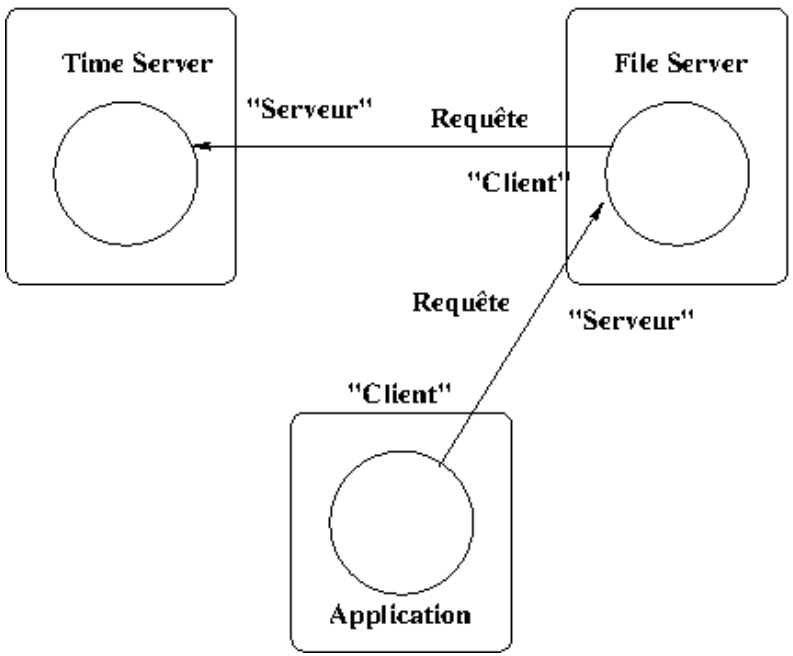
➢ Ceci conduit à une représentation **"centralisée"** des dialogues.

Il n'y a **pas de consensus "impératif" sur la définition exacte** du modèle client-serveur.

① Par exemple, **la présence d'un réseau** n'est pas forcément un critère pertinent :

➢ Clients et serveurs peuvent être **locaux**.

② Les formes de dialogues peuvent être plus complexes qu'une requête/réponse :

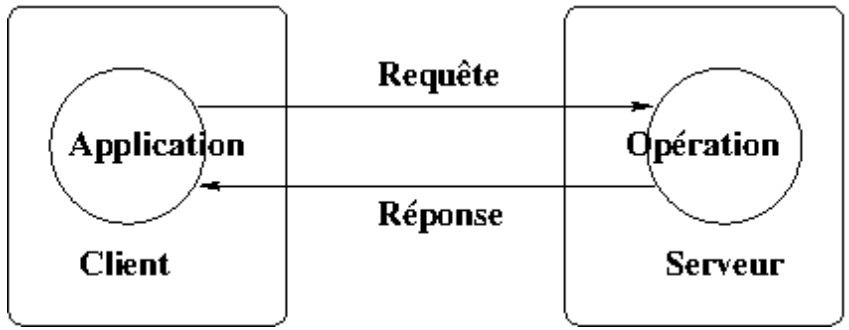


➢ le serveur étant lui même client d'un autre serveur (fournissant un temps atomique sur l'exemple).

=> Communication **"inter-server"** ou **"server-server"**.

Pourquoi le client-serveur ?

Le concept du client-serveur s'impose dés que l'on souhaite **décomposer / architecturer l'exécution d'une application** et faire en sorte que **différentes machines** (au sens large ... matérielles, logicielles) **participent efficacement à l'exécution** de l'application.



Synchroniser des activités : enjeux majeur !

La motivation principale sous-jacente du client-serveur est alors de résoudre le problème du rendez-vous et de permettre **une synchronisation des activités** grâce au schéma de **dialogue de type requête/réponse** et aux conventions/-règles qui l'accompagne !

Les intérêts de ce paradigme "C/S" sont multiples :

- **facilite l'exploitation (usability)** d'un système ou d'un composant.

  L'accès "par requêtes" aux fonctionnalités du composant est un mécanisme simple et efficace.

- **facilite la modification** d'un système ou un composant pour l'utiliser dans des applications ou des environnements **autres que ceux pour lesquels il a été conçu** : **flexibilité (flexibility)**.

  L'**abstraction** induite par l'accès par requêtes permet de favoriser la réutilisation et l'évolution.

- **interopérabilité (interoperability)** :

  La communication par requête/réponse permet, moyennant l'adoption du protocole sous jacent, de faciliter les échanges entre les systèmes ou les composants.

- **mise à l'échelle (scalability)** :

  facilite la mise à l'échelle du système ou du composant selon les dimensions du problème à résoudre. Il "suffit" d'instancier plus d'entités mais cela ne remet pas en cause le concept.

OS moderne : utilisation intensive des clients-serveurs

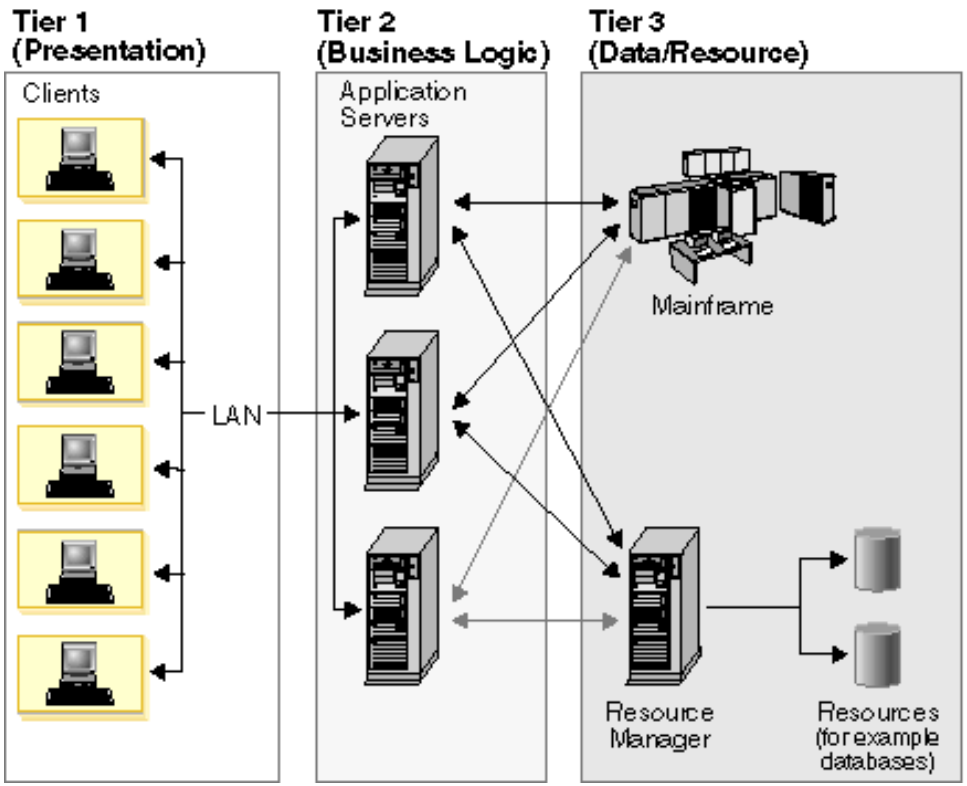
```
> netstat -a
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 localhost:domain        *:*                     LISTEN
tcp        0      0 *:ssh                   *:*                     LISTEN
tcp        0      0 localhost:ipp           *:*                     LISTEN
tcp        0      0 vtr.i3s.unice.fr:40736  improxy2-g27.free:imap2 ESTABLISHED
tcp        0      0 vtr.i3s.unice.fr:56335  star-slb-ecmp-01-:https ESTABLISHED
tcp        1      0 vtr.i3s.unice.fr:41094  mistletoe.canonica:http CLOSE_WAIT
tcp        0      0 vtr.i3s.unice.fr:56390  improxy3-g27.free:imap2 ESTABLISHED
tcp        0      0 vtr.i3s.unice.fr:60247  socrate.i3s.unice:imap2 ESTABLISHED
tcp6       0      0 [::]:ssh                [::]:*                  LISTEN
tcp6       0      0 ip6-localhost:ipp       [::]:*                  LISTEN
udp        0      0 localhost:domain        *:*
udp        0      0 *:mdns                  *:*
udp        0      0 *:54879                 *:*
udp6       0      0 [::]:53145              [::]:*
udp6       0      0 [::]:mdns               [::]:*
Active UNIX domain sockets (servers and established)
Proto RefCnt Flags       Type       State         I-Node   Path
unix  2      [ ACC ]     STREAM     LISTENING     10909    /tmp/ssh-BlHdtqsr1904/agent.1904
unix  2      [ ACC ]     STREAM     LISTENING     10917    /tmp/.ICE-unix/1904
unix  2      [ ACC ]     STREAM     LISTENING     9476     /var/run/dbus/system_bus_socket
unix  2      [ ACC ]     STREAM     LISTENING     11530    /tmp/keyring-mr5c0s/ssh
unix  2      [ ACC ]     STREAM     LISTENING     11532    /tmp/keyring-mr5c0s/pkcs11
unix  15     [ ]         DGRAM                    9482     /dev/log
unix  2      [ ACC ]     STREAM     LISTENING     9674     /tmp/.X11-unix/X0
unix  2      [ ACC ]     STREAM     LISTENING     10882    /tmp/keyring-mr5c0s/control
unix  2      [ ACC ]     STREAM     LISTENING     9509     /var/run/sdp
unix  2      [ ACC ]     STREAM     LISTENING     9522     /var/run/avahi-daemon/socket
unix  2      [ ACC ]     STREAM     LISTENING     6896     @/com/ubuntu/upstart
unix  2      [ ACC ]     STREAM     LISTENING     10567    /var/run/cups/cups.sock
unix  2      [ ACC ]     STREAM     LISTENING     9514     @/org/bluez/audio
```

Evolutions du client-serveur

① Renforcement de la frontière Clients/Serveurs :

Les fonctionnalités des serveurs croissent et tendent à faire plus que fournir des données.

➢ Ils traitent et analysent avant de répondre ≡ "Business Logic"



On passe ainsi :

➟ de serveurs de fichiers,

➟ à serveurs d'applications (Web par exemple),

➟ puis au paradigme des architectures 3-tiers lorsque l'opération de service devient plus complexe,

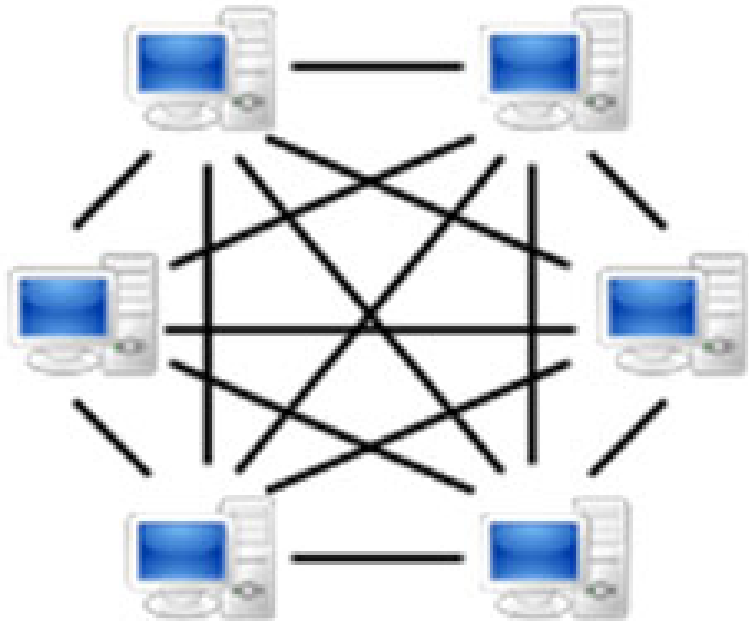
On répartit la charge / le travail induit par les requêtes.

➟ . . . 11 tiers du modèle de référence de l'IoT !

② Serveur et Client à la fois :

Le paradigme client/serveur se dérive en "peer to peer" dans une configuration/topologie **distribuée** (différence majeure !).

A **peer-to-peer** based network.

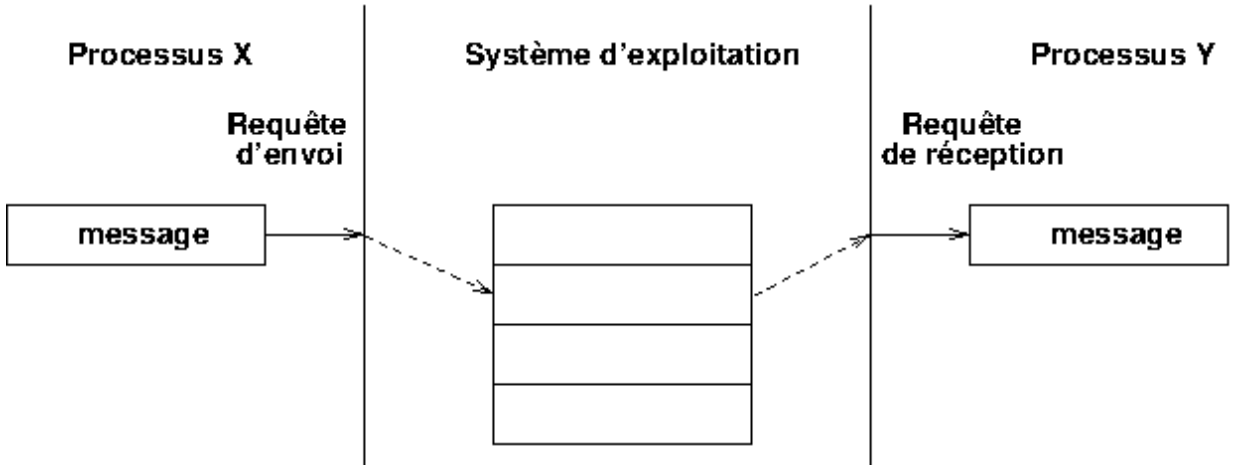


➤ Dès lors, la différence fonctionnelle est moins marquée, et on a un **comportement "plus symétrique"** des processus (et des machines) jouant à la fois le rôle d'un serveur et de clients.

Cas du Domaine Unix

La communication "inter processus" a aussi un sens pour des processus **résidants sur la même machine** et ne partageant pas de données communes dans leur espace d'adressage.

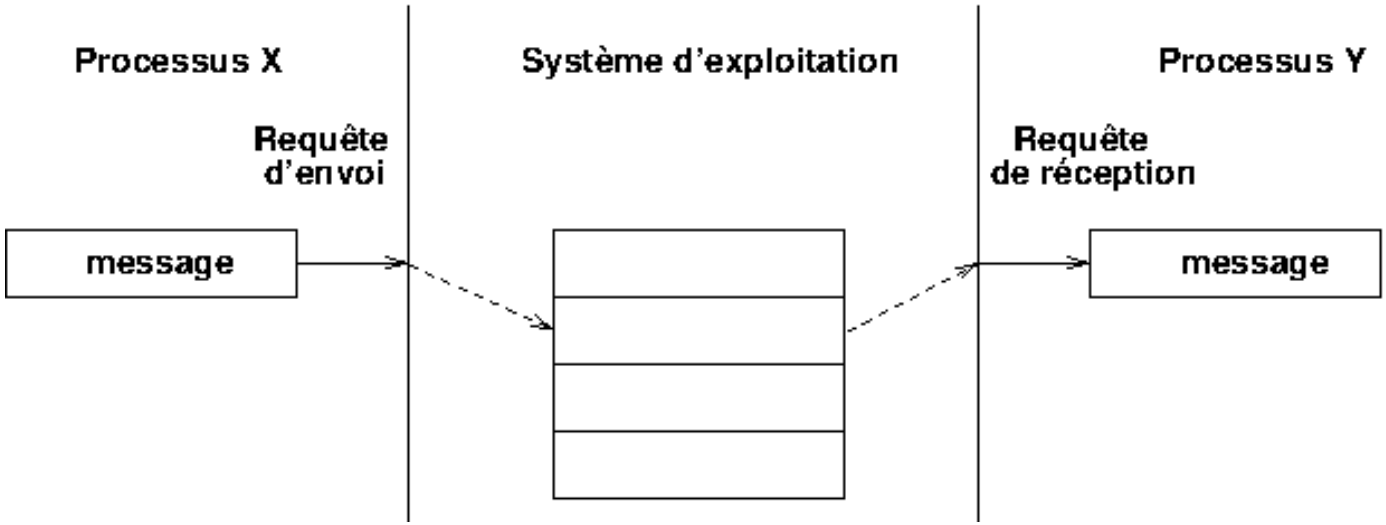
➢ Classiquement, **ils sollicitent les services du système d'exploitation** (auxquels ils appartiennent) pour réaliser une **double recopie des données à transmettre** :



① une première correspondant à un **envoi**, de l'espace d'adressage de l'émetteur vers le noyau,

② une seconde, correspondant à la **réception**, du noyau vers l'espace d'adressage du récepteur.

Ce schéma s'applique à la communication au travers de fichiers partagés, de tubes/pipe, de files de messages et de sockets du domaine UNIX.



- Les sockets locales (UNIX) ont comme avantage par rapport aux tubes (ordinaires), la possibilité des **communications bidirectionnelles et full-duplex**.

  Avec les tubes, pour établir une communication Full-duplex, chaque processus doit réciproquement créer un tube.

- Par contre, cette communication est plus lente que les tubes (cf [**?**] page 289)

SOCKETS

A ce stade du cours . . . nous avons décrit le modèle d'architecture logicielle :

Etudes des mécanismes de transport

- **Structrure des réseaux**
- **Interface de programmation**
- **Multimédia**

Etudes des architectures logicielles

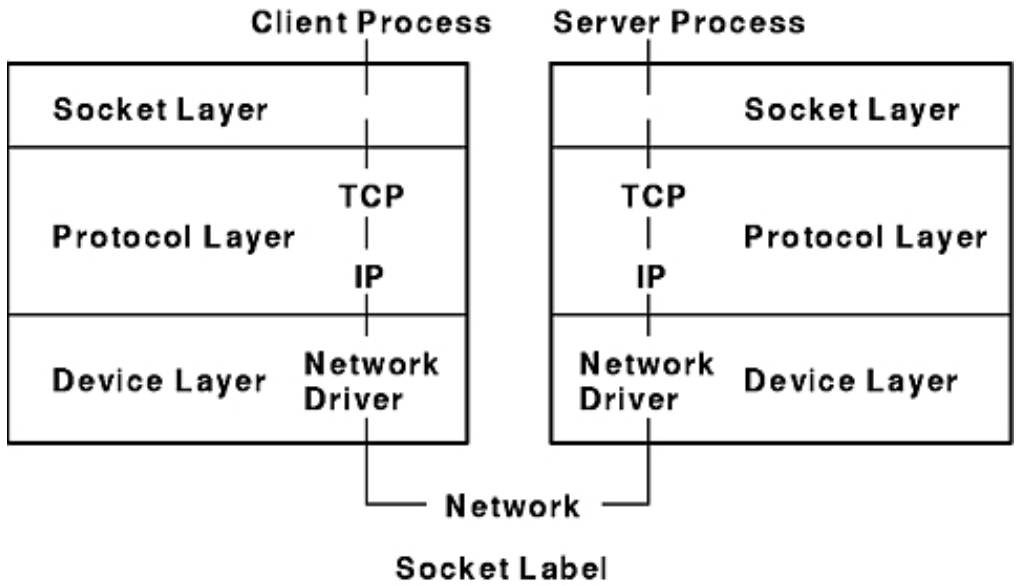
- **Architectures distribuées**
- **Appels de méthodes distants**
- **Architectures Web**

➟ Nous allons maintenant nous focaliser sur les échanges d'informations entre clients et serveurs en insistant sur une configuration comportant un réseau de communication.

– L'échange de données entre deux processus sur deux machines différentes –

Une API . . . pour échanger entre processus.

De façon analogue avec ce qui se passe dans le cadre des fichiers, (et des E/S en général), **la manipulation des émissions/réceptions réseaux va être réalisée au travers d'une interface de programmation (API)** :

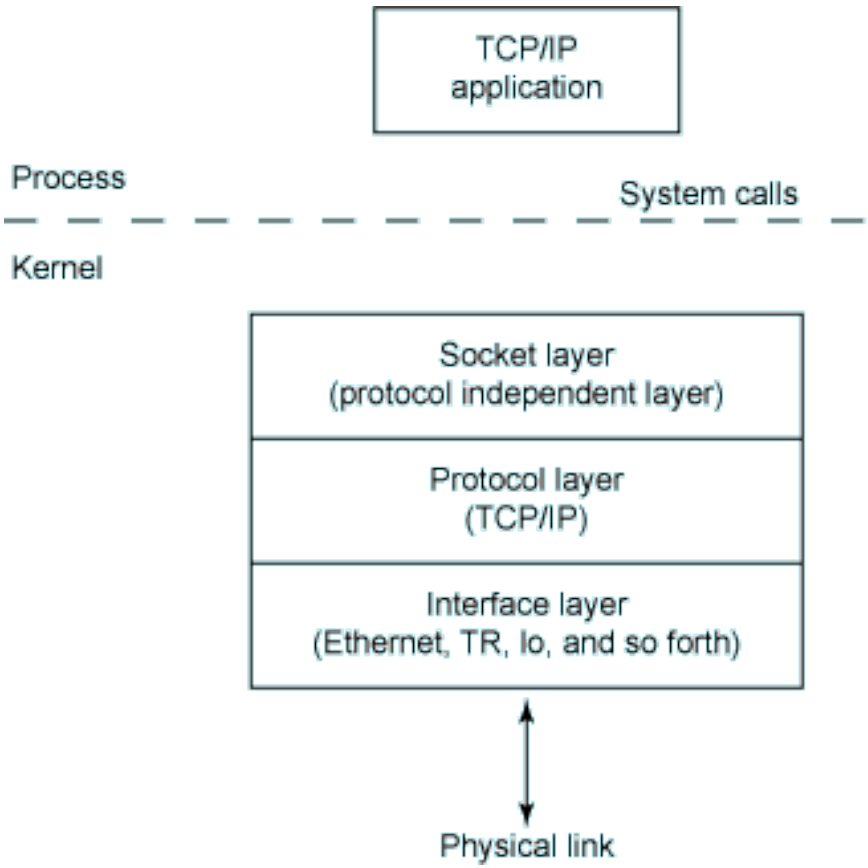


Cette API permet à une application/processus utilisateur de réaliser des échanges avec un processus distant.

➢ Elle **s'appuie** sur les couches inférieures : **la pile réseau** ("Protocol Layer").

API Réseau : Indépendance !

Afin d'éviter tout "verrouillage" ou dépendance, **cette interface de programmation ne fait pas partie** :



- du standard réseaux (par exemple TCP/IP)
- ou de la définition du système d'exploitation : Unix ou autre.

L'API Sockets est présente sous Windows, Unix, . . . en C, Python, Java, . . .

Standard "de facto"

Historiquement, dans le monde UNIX, deux API majeures ont été proposées :

① les **Berkeley Sockets**

② et la **System V Transport Layer Interface** (TLI) étendue à XTI.

Ces deux interfaces sont originellement écrites en langage C.

Avec le temps, **les sockets deviennent un standard** `de facto` via SUN, TEKTRONIX, DEC ...

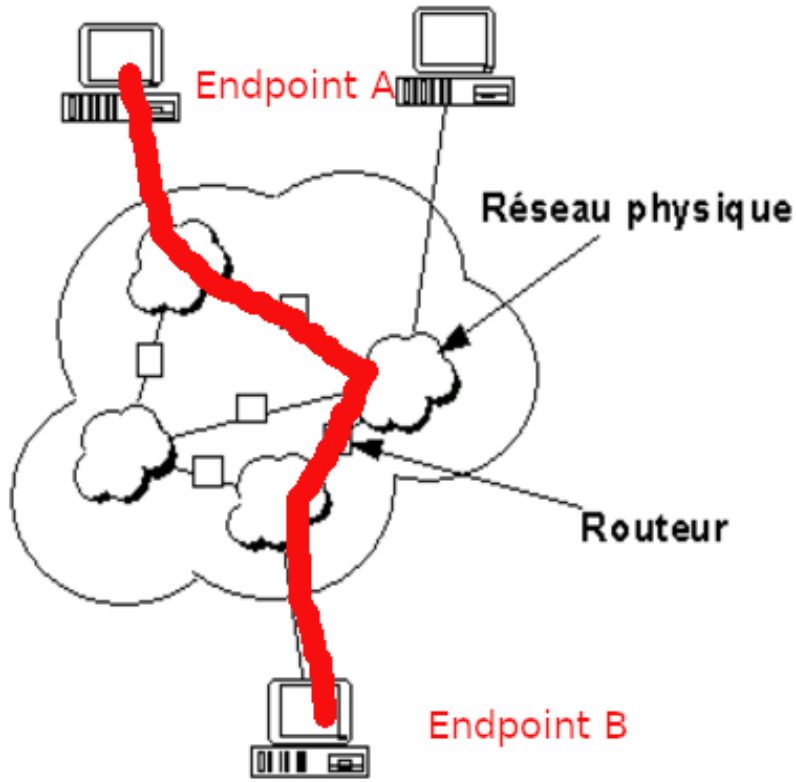
➢ Elles sont utilisées dans `telnet`, `rlogin`, `ftp`, `talk` ...

Cahier des charges de l'API I

C'est encore abstrait . . . mais on attend plusieurs choses d'une telle API :

① Créer un point de communication dans un processus :

- ➢ Allocation de ressource locales de communication . . . **idem fichiers**.
- ➢ Ces points sont les **endpoints** d'une communication "locale" ou "distante" :



Derrière la notion de "endpoints", il y la notion de "on ne sait pas trop", ce qui se passe entre ces points, dans le réseau.

On est utilisateur du réseau et pas acteur (du moins directement).

On n'est maître **que des extrémités** !

Cahier des charges de l'API II

② Préparer l'échange si le choix d'une approche connectée est fait ? :

- ➢ Initier une connexion (coté client)
- ➢ Attendre une connexion (coté serveur)

③ Echanger : a priori simple à faire MAIS méfiance !

- ➢ Envoyer ou recevoir des données
- ➢ Déterminer quand des données arrivent
- ➢ Générer des données urgentes
- ➢ Gérer l'occurence de données urgentes
- ➢ Terminer une connexion
- ➢ Gérer l'occurence d'une déconnexion du site distant
- ➢ Abréger une communication
- ➢ Gérer des conditions d'erreur
- ➢ Libérer des ressources locales de communication

Origines et démarche de l'API

A l'heure de la réflexion, deux approches sont possibles pour les concepteurs des API "réseaux" :

① **Développer** entièrement (**concepts nouveaux**) une bibliothèque système permettant l'accés au fonctions réseaux,

② **Dériver** les accés **traditionnels** aux E/S (fichiers, claviers, . . .) pour concevoir l'accés au réseau.

Les sockets se développent dans la seconde philosophie :

Elles constituent une généralisation du mécanisme Unix d'accès aux fichiers :

➢ **"Enrichir la sémantique des descripteurs de fichiers avec des options et ajouter à l'API quelques fonctions"**.

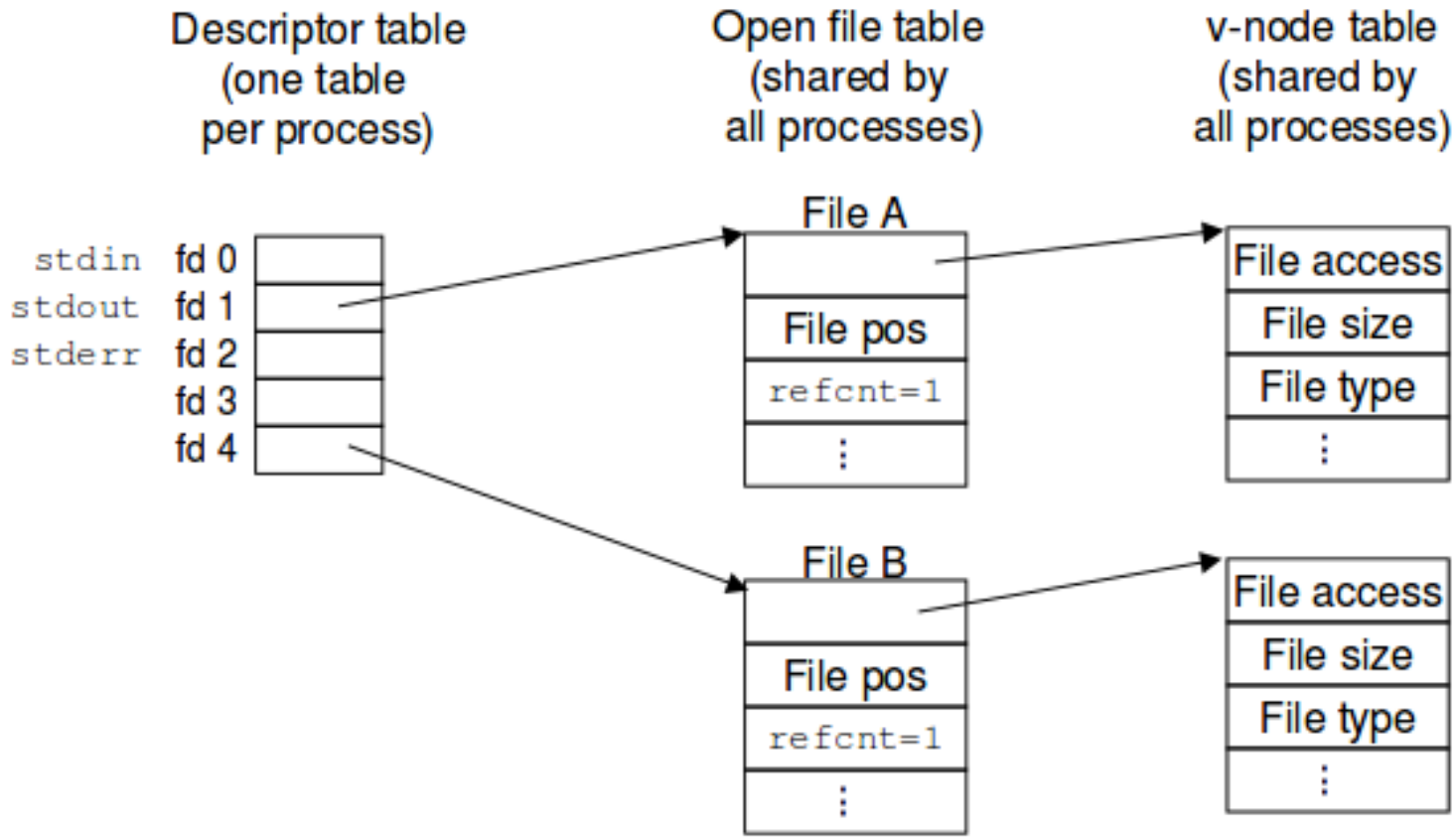
Ceci constituera une petite entorse à la compatibilité des I/O de Unix, mais met en place une certaine **orthogonalité** :

➢ Ainsi un code "réseau/socket" peut beaucoup ressembler à un code "fichier" :

"Envoyer sur un réseau" ≡ "Ecrire dans un fichier"

File I/O : File descriptor

On se rappelle que le concept de fichier repose sur le concept fondamental de **descripteur de fichier** qui permet d'accéder aux différentes tables (partagées par les processus) contenant les informations sur les fichiers ouverts :



- **"Descriptor table"** : **Chaque processus a sa propre table de descripteurs** dont les entrées sont indexées par les descripteurs de fichier ouverts par le processus

  Chaque descripteur de cette table permet de pointer sur une entrée de la table des fichiers.

- **"File table"** : Cette table contient **l'ensemble des fichiers ouverts**. Elle est partagée par tous les processus.

  Chaque entrée de cette table contient un ensemble d'informations : la position courante dans le fichier, un compteur du nombre de références (nombre de descripteurs pointant) sur ce fichier, un pointeur sur la table des v-node, etc

  Fermer un descripteur décrémente le compteur associé au fichier. Mais le noyau ne détruit l'entrée dans la table des fichiers que lorsque ce compteur vaut 0 !

- **"v-node ou i-node table"** : Chaque entrée de cette table (aussi partagée par tous les processus) contient des **informations qui discriminent les fichiers stockés dans le système de fichiers** : créateur, type d'accès, droits, . . .

  On accède à ces informations par la structure "stat" : cf `stat(2)`

Files I/O : API

Au niveau de l'API des fichiers Unix, il y a **six appels système** :

- open(), close(), read(), write(), lseek(), ioctl()

```
1   #include <stdio.h>
2   #include <sys/types.h>
3   #include <sys/stat.h>
4   #include <string.h>
5   #include <stdlib.h>
6   #include <fcntl.h>
7   #include <sys/param.h>
8   /*-------------------------------------*/
9   void copie(char *src, char * dst){
10    int fd1,fd2;
11    char buff[256];
12    int n;
13    fd1 = open(src,O_RDONLY);
14    fd2 = open(dst,O_WRONLY|O_CREAT|O_TRUNC,S_IRWXU);
15    while(n=read(fd1,buff,sizeof(buff))){
16      write(fd2,buff,n);
17    }
18    close(fd1);
19    close(fd2);
20  }
21  /*-------------------------------------*/
22  int main(int argc, char *argv[]){
23    copie( "foo.raw", "goo.raw");
24  }
25  /*-------------------------------------*/
```

- Tous ces appels utilisent un **file descriptor**.

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/param.h>
#define PAGESIZE 100

int init_open(char * filename){ /* fill a file if it does not exist */
  int fd, nbytes;
  char buf[] ="abcdefgh";
  if ((fd = open("foo.raw",O_RDWR|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR)) < 0){
    fprintf(stderr,"Error is %s (errno=%d) \n", strerror(errno), errno ); exit(1);
  }
  nbytes = write(fd,buf,sizeof(buf));
  lseek(fd, 0, SEEK_SET);
  return fd;
}

/* Print the number of 'a' in the file ? */
int main(int argc, char *argv[]){
  int fd, nbytes, nb, i;
  char buf[PAGESIZE];       /*A convenient buffer size*/
  nb=0;
  if ((fd = open("foo.raw", O_RDONLY)) < 0){
    fd = init_open("foo.raw");
  }
  while ((nbytes = read(fd,buf,1)) > 0){ /* read(fd,buf,sizeof(buf))) > 0)*/
    for (i = 0; i < nbytes; i++)
      if (buf[i] == 'a' )
    nb++;
  }
  close (fd);
  printf("%d 'a' found in file \n", nb);
}
```

Files vs Sockets

Les fichiers et les sockets partagent donc les mêmes "fondements".

Il **FAUT** être au point sur les fichiers :

- Maîtrise de l'API
- Gestion des descripteurs, notamment sur les fichiers standards d'E/S (redirections, duplications, ...),
- Héritage (fork()),
- Récupération des erreurs,
- ...

N'hésitez pas à poser des questions en TP ! ! !

Particularités E/S réseau

Différences/"suppléments sémantiques", liés à la présence d'un réseau :

① La communication est **typée** !

➤ Soit **connection-oriented** : Ce type s'apparente à une entrée/sortie de type fichier.

On doit "ouvrir" une connexion avec un destinataire, comme on ouvre un fichier, et la communication (/ les transferts) **a toujours lieu sur le ce lien** avec ce destinaire.

Ce type de communication sera **sans** pertes !

➤ Soit **connectionless** : Il n'y a pas de destinaire **ouvert**, le point de communication peut permettre d'atteindre plusieurs destinataires différents (i.e processus **peer**).

Un peu comme si le même descripteur de fichier permettait d'écrire sur différents fichiers (à chaque appel de write()).

Pour ce type de communication, le transfert d'information peut échouer : **perte réseau** !

Et, **il n'est pas prévu que l'API nous informe de la perte** ...

② **Non-symétrie** de la relation client-serveur :

On verra que le comportement vis à vis des extrémités de communication est différent suivant que l'on est serveur ou client.

➠ Par exemple, dans le cas d'une communication **connection-oriented**,
- ➢ Il y a celui qui provoque la connexion
- ➢ et celui qui la gère.

Dans le cas des fichiers, tous les processus accèdent de la "même" façon au descripteur (dont le comportement est constant et unique).

③ Le lien sur lequel les données vont être acheminer, c'est à dire une **connexion réseau, est définie par cinq informations** :

(protocol, local-addr, local-process/port, foreign-addr, foreign-process/port)

Exemple :

(tcp , 128.10.0.3 , 1500 , 128.10.0.7 , 21)

Ce qui signifie que le lien de communication sur lequel le transfert d'information aura lieu, correspond à une connexion exploitant le protocole "tcp" entre les machines 128.10.0.3 et 128.10.0.7 utilisant respectivement les ports 1500 et 21.

Dans le cas des fichiers, on dirait que le point de communication (file descriptor) établit un lien avec un fichier.

- ✔ Le fichier étant identifié par son nom.
- ✔ Il n'y a pas de "protocole" de transfert : on transfert des octets !

④ **Taille des enregistrements** :

Les **entrées/sorties sur fichier** sont assimilables à des streams/flux (un tuyau non percé entre deux lieux).

A la taille des buffers systèmes près, la **granularité des transferts** (c'est à dire la taille du buffer utilisée par l'application) (cf [**?**] page 115) ne modifie pas le mécanisme de transfert vers le fichier destinataire :

- ➠ Cela **n'a donc pas d'importance sur le succès du transfert**.
- ➠ Mais cela peut tout de même influer sur les performances.

Par contre, **pour certains protocoles réseaux, la granularité des messages peut avoir une grande importance** :

- ➢ Selon la taille des buffers transmis, certains mécanismes vont s'activer et modifieront les caractéristiques du transfert.

  "Fragmentation IP" . . . Ca vous parle ? . . . Ca devrait !

⑤ "Protocol agnostic" :

L'interface (API) doit permettre d'accéder à différents types de protocoles :

- ✔ TCP,
- ✔ UDP,
- ✔ XNS,
- ✔ Appletalk, ...

Il est important de bien souligner que les sockets permettent d'accéder à une interface avec un réseau de communication qui n'est pas forcément Internet (TCP/IP) !

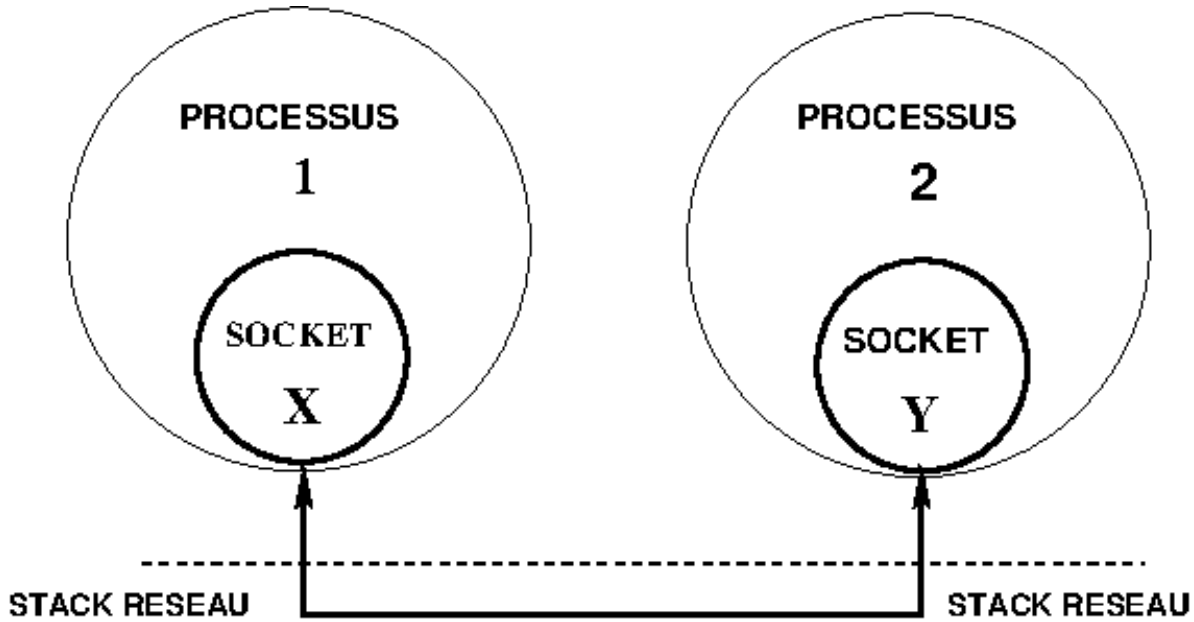
Pour les fichiers, il n'y a pas de "protocole d'accés" et donc pas de protocoles d'accés "différents" !

- ✔ Le "write()" ou le "read()" se font de manière "constante".

Point de communication

Les sockets sont des **points de communication** qui permettent à plusieurs processus, éventuellement distants, d'échanger des informations :

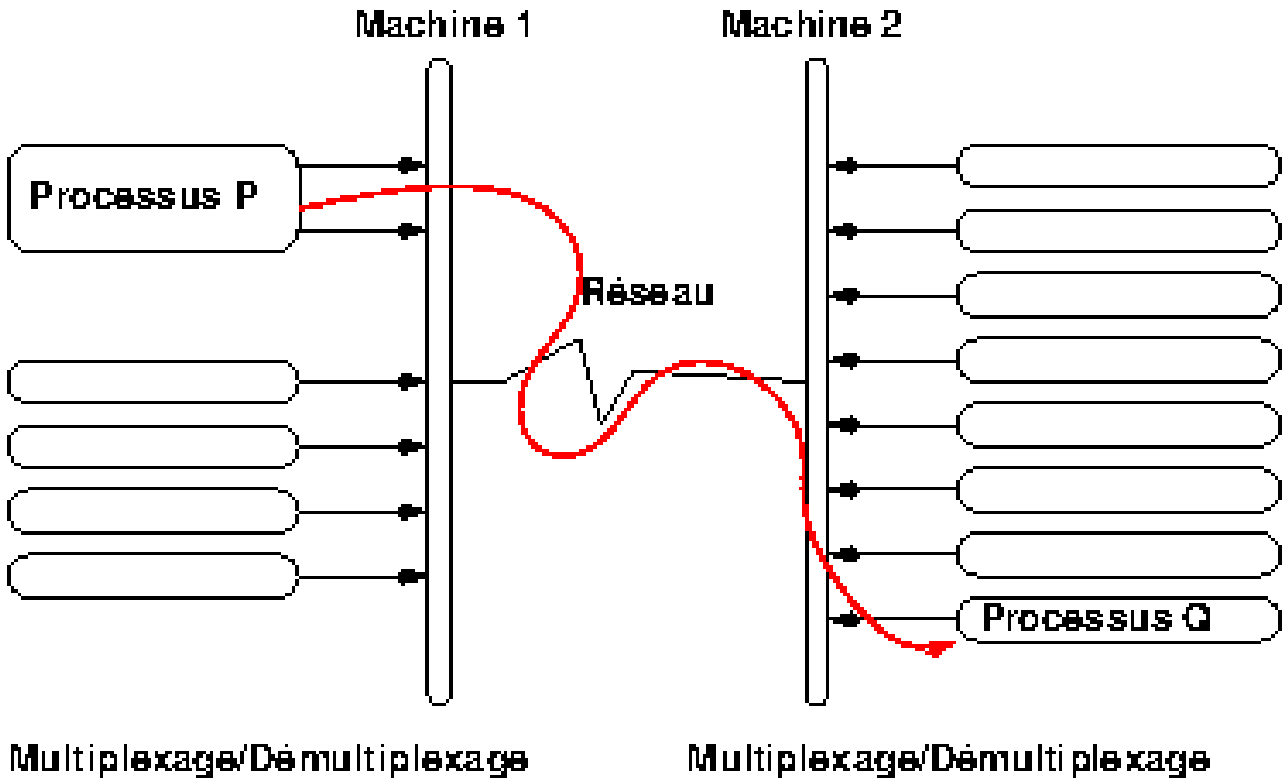
➢ Cette communication peut être bi-directionnelle ($\neq$ pipes).



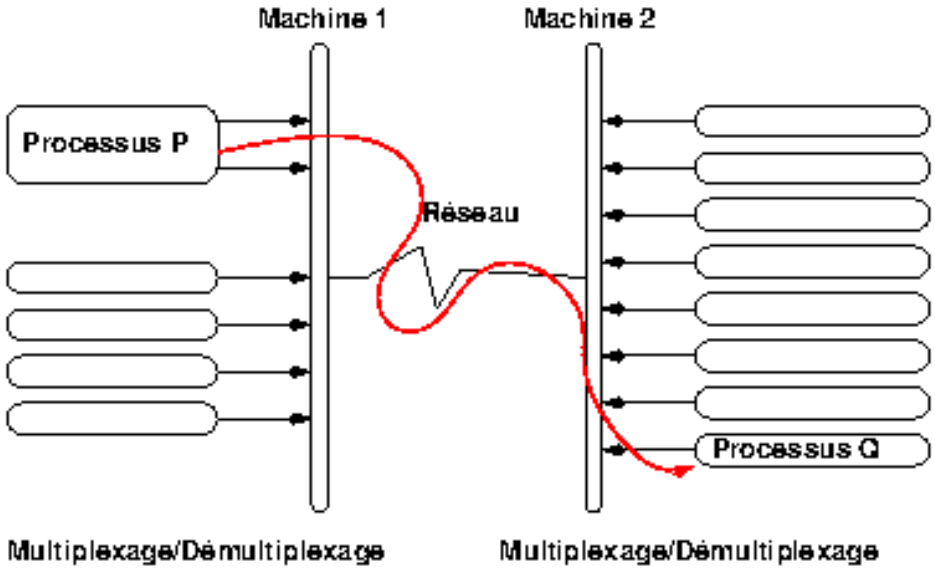
Mais identifier un point de communication demande plus d'information qu'identifier une machine : L'adresse IP ne va pas suffire !

Identification d'un point de communication

D'un point de vue "réseau", le rôle de l'OS d'une machine est de **distribuer les paquets qu'il recoit aux applications destinatrices** : c'est le **démultiplexage**.



Ainsi, beaucoup d'applications différentes, de la même machine, sont susceptibles d'utiliser le réseau au même instant !



Une machine (i.e son OS) réceptionnant un message à destination de son seul nom (i.e par exemple son adresse logique) ne pourrait identifier l'application destinatrice.

➠ Cette seule information **ne permet donc pas à l'OS de cette dernière de démultiplexer correctement le message**.

L'application destinataire d'une requête ou d'une réponse ne peut donc pas être "uniquement" identifié par le nom d'une machine !

Notion de port

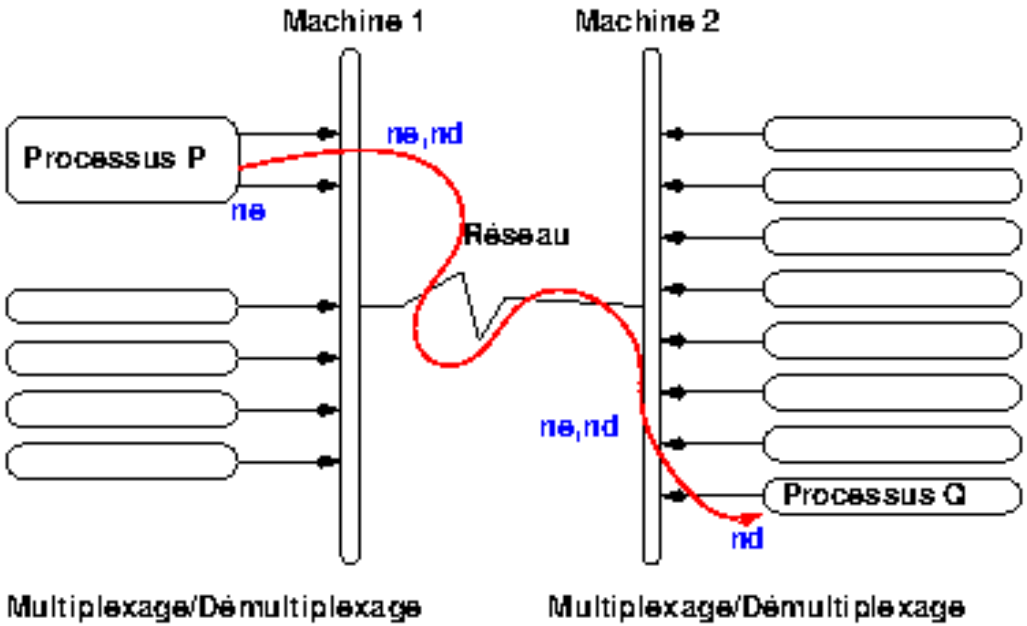
Le numéro du processus ne permet pas non plus de résoudre le problème car il est choisi par le système et donc trop inconstant.

- ➠ De plus, de la même façon qu'un processus peut travailler simultanément sur plusieurs fichiers, un processus pourrait aussi souhaiter simultanément dialoguer avec plusieurs interlocuteurs.

Il faut un nouveau concept : "Le port".

Numéros de ports

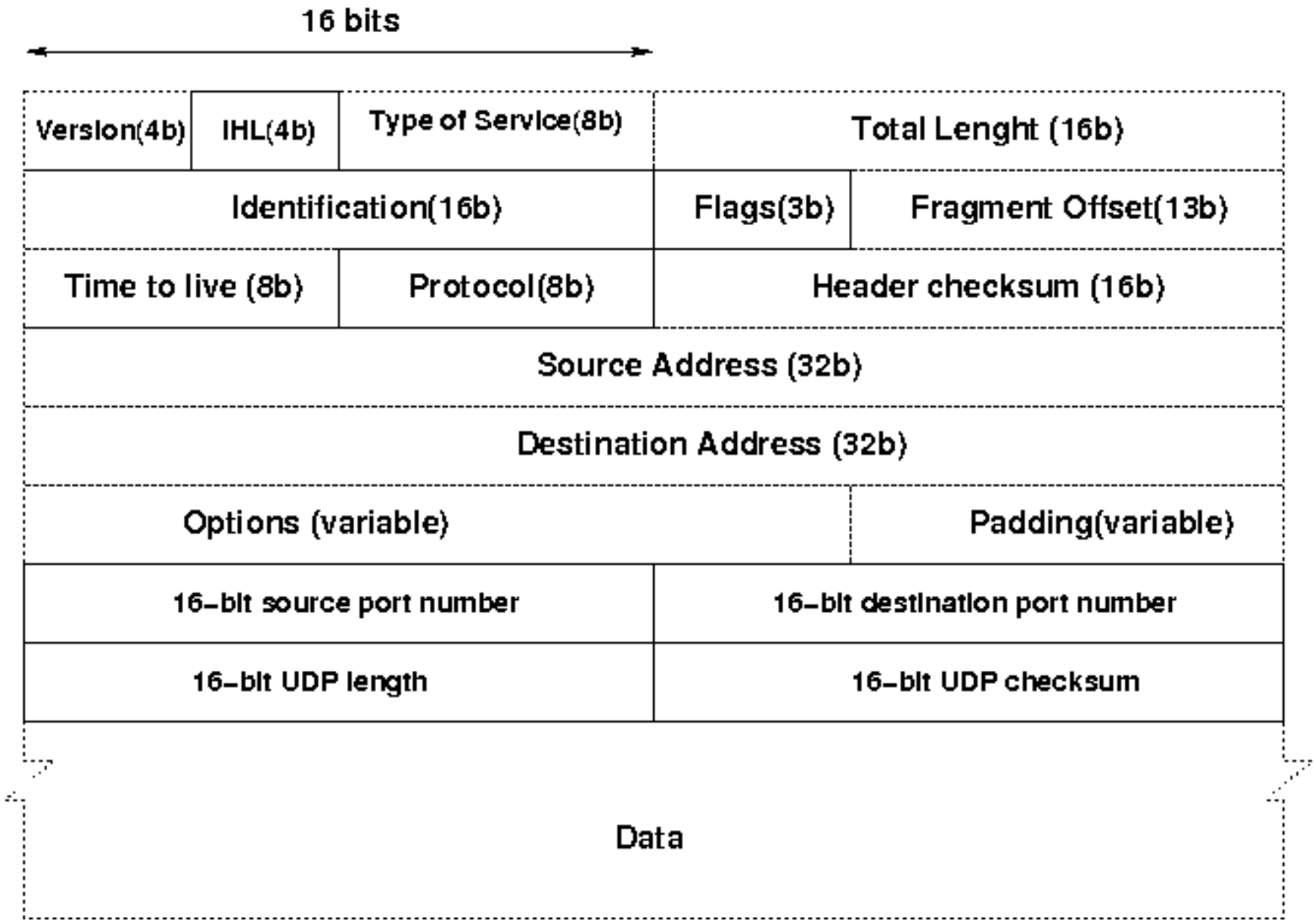
Pour résoudre ce problème d'identification, **une application réseau** émet via un **point de communication** qui est identifié au niveau de l'OS par un numéro sur 16 bits dit " **numéro de port** ".



➟ **L'application destinaire est**, elle aussi, munie d'un tel point de communication lui même **qualifié par numéro de port** (trés probablement différent).

Les **paquets** qui vont transiter entre les machines contiennent les numéros des ports source et destination . . . en plus des adresses réseaux de chacune des machines : UDP et TCP !

On voit bien ces informations dans le datagramme UDP :



Numéro de port et Service

Les communications Inter Processus (IPC : Inter Process Communication) s'effectuent donc entre des paires (idenfifiant machine + identifiant port) qui caractérisent les **points de communication** :

(machine 1 : port ne) <——————> (machine 2 : port nd)

Exemple :

(seneque.unice.fr : 20349) <——————> (danube.unice.fr : 3447)

On peut aussi désigner l'application par un **nom de service** qui est un **identificateur associé à un numéro de port** : par exemple www (port 80).

Le point de communication est alors défini par une paire :

(host : nom de service)

Exemple utilisant numéro et nom :

danube.unice.fr : 10456 <——————> www.inria.fr : www

La correspondance entre les noms des services et leurs numéro de ports est contenu dans le fichier :

`/etc/services`

```
tcpmux          1/tcp                           # TCP port service multiplexer
echo            7/tcp
echo            7/udp
discard         9/tcp           sink null
discard         9/udp           sink null
systat          11/tcp          users
daytime         13/tcp
daytime         13/udp
netstat         15/tcp
qotd            17/tcp          quote
msp             18/tcp                          # message send protocol
msp             18/udp
chargen         19/tcp          ttytst source
chargen         19/udp          ttytst source
ftp-data        20/tcp
ftp             21/tcp
fsp             21/udp          fspd
ssh             22/tcp                          # SSH Remote Login Protocol
ssh             22/udp
telnet          23/tcp
smtp            25/tcp          mail
```

L'affectation des numéros de ports inférieurs à 1024 est effectuées par l'IANA (Internet Assigned Numbers Authority).

```
tcpmux          1/tcp                           # TCP port service multiplexer
echo            7/tcp
echo            7/udp
discard         9/tcp           sink null
discard         9/udp           sink null
systat          11/tcp          users
daytime         13/tcp
daytime         13/udp
netstat         15/tcp
qotd            17/tcp          quote
msp             18/tcp                          # message send protocol
msp             18/udp
chargen         19/tcp          ttytst source
chargen         19/udp          ttytst source
ftp-data        20/tcp
ftp             21/tcp
fsp             21/udp          fspd
ssh             22/tcp                          # SSH Remote Login Protocol
ssh             22/udp
telnet          23/tcp
smtp            25/tcp          mail
```

Leur politique est d'affecter deux protocoles à un service lorsqu'il lui assigne un numéro de port.

➢ C'est pourquoi la plupart des entrées sont doubles même si elles n'utilisent souvent qu'un protocole.

Numéros de ports : http://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers

Les numéros de ports sont codés sur 16 bits : 65536 possibilités.

- ✔ Les ports 0 à 1023 sont les **ports reconnus** ou réservés (**Well Known Ports**) :

  Ils sont, de manière générale, **réservés aux processus système** (démons) ou aux programmes exécutés par des utilisateurs privilégiés.

  Un administrateur réseau peut néanmoins lier des services aux ports de son choix.

- ✔ Les ports 1024 à 49151 sont appelés **ports enregistrés** (**Registered Ports**) :

  Ces ports sont **assignés à une application par IANA** sur demande d'une entité/société. Sur la plupart des systèmes, ces ports peuvent être utilisés par un utilisateur ordinaire.

- ✔ Les ports 49152 à 65535 sont les **ports dynamiques et/ou privés** (**Dynamic and/or Private Ports**).

  Ces ports **ne peuvent pas être enregistrés** dans IANA. Par contre ils peuvent servir de **port éphémère** ou de ports privés.

Port éphémère

Un serveur est comme une "cible" pour les clients et donc son numéro de port doit être connu pour être atteint !

Par contre, un client **ne se préoccupe généralement pas du numéro de port qu'il utilise pour émettre ou recevoir**.

➢ Seule la garantie de l'unicité du numéro de port utilisé est important pour lui dans la mesure où deux points de communications **ne peuvent pas** utiliser le même port.

On va voir plus tard qu'une entité **peut savoir** quel numéro de port est utilisé par le point de communication d'où provient l'information.

➠ Ainsi **les réponses du serveurs se feront sur le port utilisé par le client**.

Les ports des clients peuvent donc être des **ports éphémères** :

✔ C'est à dire dont l'utilisation est limitée dans le temps.

✔ Rien n'empêchant une application cliente d'utiliser des numéros de ports différents à chaque activation.

port=0 donc éphémère !

Pour créer un port éphémère, il est possible de laisser le système choisir le port en l'**identifiant comme le port de numéro 0** :

```
1        int port = 0;
2        adresse.sin_port = htons(port);
```

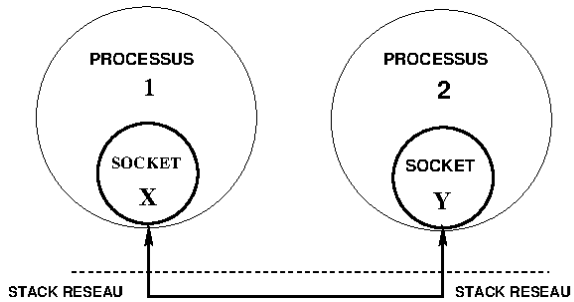
Laisser le système choisir, c'est éviter les conflits induits par l'utilisation successive d'un numéro de port.

➢ En utilisant la fonction `getsockname()`, **on peut ensuite connaître le numéro de port** qui a été choisi lors de l'attachement de l'adresse à la socket (cf inc.c à la ligne 54).

Client ONLY ! ! !

Sémantique Utilisateur

Chaque entité (i.e : processus) impliquée dans une communication devra donc disposer d'un (ou plusieurs) point de contact/communication : une **socket**.



Lorsque l'on crée une socket, on doit spécifier :

- la **forme de communication** souhaitée : DGRAM, STREAM, ...

  c'est son "type" / la **"sémantique utilisateur"**
- et la **famille de protocoles** pour l'implémenter : UNIX, INET, ...

  c'est son "domaine"

Il ne faut pas oublier que ceci est paramétrable car les sockets **NE SONT PAS UNE EXCLUSIVITE INTERNET** !

Lettre ou Téléphone ?

Sémantique utilisateur ?

On peut faire une analogie avec :

- ➢ une boîte aux lettres ou un poste téléphonique.

Une socket n'est rien d'autre dans le système que l'équivalent de l'un des ces objets :

- ✔ Il s'agit d'un point de communication **bidirectionnel** par lequel un processus pourra émettre ou recevoir des informations.
- ✔ Informations contenues dans **des lettres ou des appels téléphoniques**.

Le choix d'une "sémantique utilisateur" va influer sur les propriétés de la communication :

- ➢ fiabilité,
- ➢ séquencement de l'information,
- ➢ établissement d'une connexion, . . .

Le **sémantique utilisateur choisie répond aux questions/spécifications des échanges** :

① Quelles sont les **unités de données** transmises ?

- ➢ Plutôt téléphone . . . Choisir de transmettre les données comme une séquence (suite continue et non structurée) d'octets (**Stream**),
- ➢ Plutôt lettre . . . Former des groupements structurés d'octets (**Paquets**/**Datagrammes**) donnés comme tel à la couche inférieure.

② Est ce que l'on peut **perdre des données ?**

- ➢ Plutôt téléphone . . . Des formes de communications garantissent que les données arrivent dans l'ordre et sans perte.
- ➢ Plutôt lettre . . . D'autres peuvent se permettre de perdre les données ou alors de les dupliquer ou encore de les inverser.

③ Est ce qu'il s'agit d'une **communication avec un seul partenaire ?** . . . Plutôt téléphone !

④ ou le même point de communication doit-il permettre d'**échanger avec plusieurs partenaires ?** . . . Plutôt lettre !

Type d'une socket

Le type d'une socket contribue à déterminer la sémantique utilisateur des communications qu'elle permet de réaliser.

➢ Les différents types de socket figurent dans $< sys/socket.h >$.

Vous n'y retrouverez pas forcément tous ceux qui suivent :

Nom Symbolique	**Orientation et Caractéristiques principales de la socket**
SOCK_RAW	échange de datagrammes à un niveau bas de protocole (IP par exemple)
SOCK_DGRAM	transmission de datagrammes structurés en mode non–connecté avec une fiabilité minimale
SOCK_STREAM	échange de séquences continues de caractères, généralement en mode connecté avec garantie du maximum de fiabilité
SOCK_RDM	échange de datagrammes en mode non–connecté en garantissant un maximum de fiabilité
SOCK_SEQPACKET	échange de datagrammes en mode connecté en garantissant un maximum de fiabilité

Le domaine (≡ address family) d'une socket

Une autre caractéristique importante des points de communication, outre la sémantique utilisateur, concerne **l'ensemble des autres points qu'ils permettent d'atteindre**.

➢ Pour poursuivre l'analogie avec les téléphones/lettres, c'est l'équivalent de la notion de postes et timbres **nationaux ou internationaux**.

C'est le **domaine** de la socket (aussi appelé **Address Family**) qui décide des points accessibles et de l'ensemble des protocoles utilisables pour la communication : AF_UNIX, AF_INET, AF_INET6, ...

Par conséquent, le domaine induit la forme de l'**identification réseau** de la socket : adresse IPv4, Ipv6 ... ou autre ?

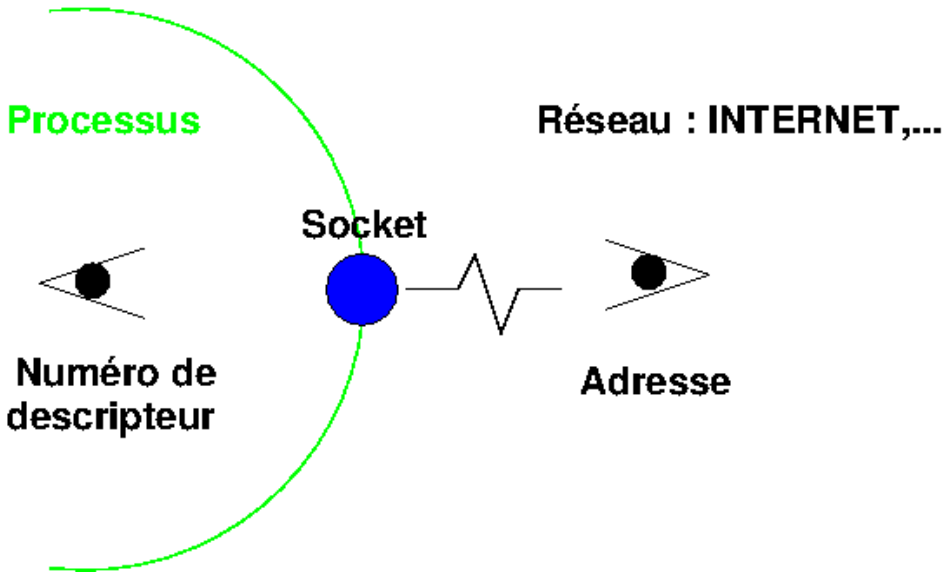
Ne pas confondre les identifications !

➢ Une socket est identifiée localement au processus par un descripteur de socket : Mais cette identification (/descripteur) n'est que **locale** !

Là on parle d'adresse.

Identification sur le réseau

Afin que les autres processus puissent la nommer, **il faut pouvoir lui attribuer** une identification/**adresse** à usage externe.



Le **domaine** d'une socket définit **le format des adresses possibles et l'ensemble des sockets avec lesquelles la socket pourra communiquer**.

➢ Il **définit également une famille de protocoles** utilisables.

Address Family (AF) (≡ Domaine)

"The sockets layer is a protocol agnostic interface that provides a set of common functions to support a variety of different protocols."[**?**]

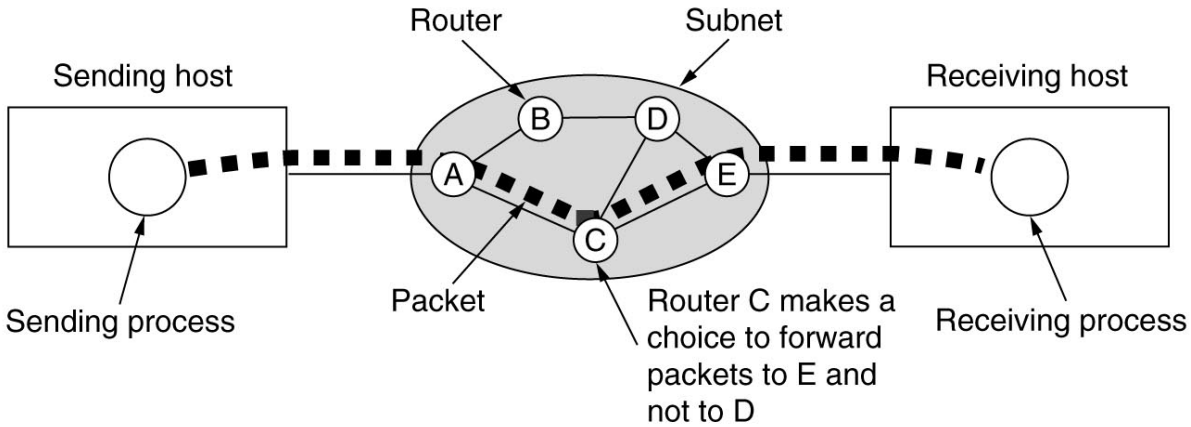
Il existe donc plusieurs domaines (ou **Address Family**) potentiels :

Nom symbolique	**Domaine**	**Format C des adresses**
AF_UNIX	Local (Loopback)	sockaddr_un ($< sys/un.h >$)
AF_INET	Internet	sockaddr_in ($< netinet/in.h >$)
AF_NS	XEROX NS	sockaddr_ns ($< netns/ns.h >$)
AF_OSI	monde OSI	
AF_SNA	SNA – IBM	
AF_CCITT	CCITT, X25	
AF_DECnet	DECnet	
AF_APPLETALK	Apple Talk	

Autant de struct (C) définies dans des header files.

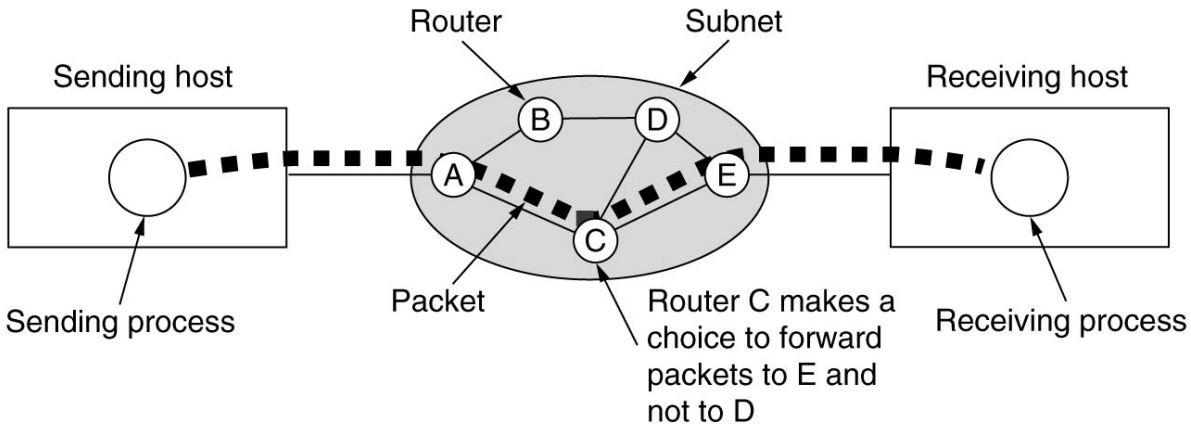
Protocoles "transports" du domaine AF_INET

Les **sockets s'appuient sur la couche transport** et le domaine Internet propose deux protocoles de communications dans cette couche :



① `UDP` (User Datagram Protocol) . . . "Les processus s'envoient des cartes postales".

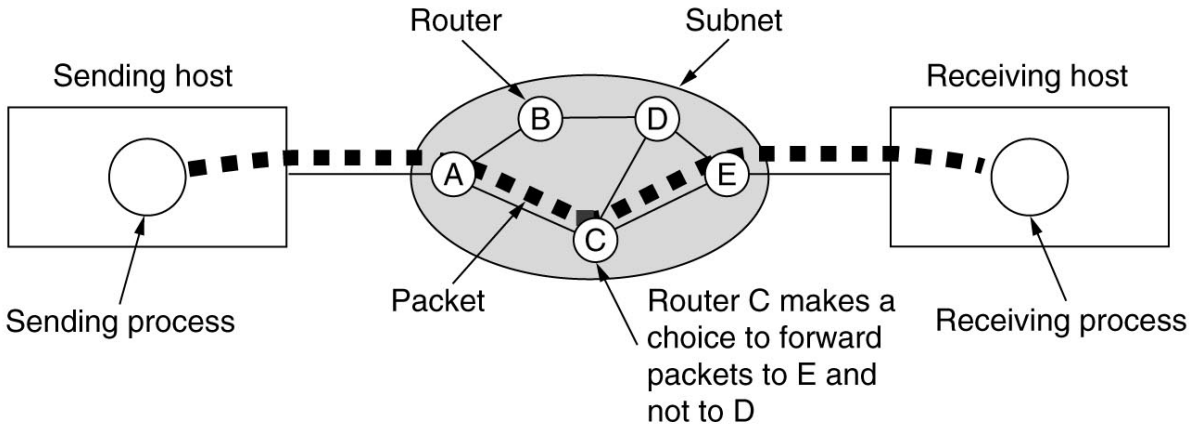
- ➢ La notion de **datagram** est associée à un service de type **connectionless**.
- ➢ UDP ressemble beaucoup à IP, les ports en plus !
- ➢ Les messages, les cartes postales ou encore **datagrams** sont transmis indépendamment d'un système à l'autre.



Cette indépendance suppose que **chaque message contient toutes les informations nécessaires à sa livraison**.

Caractéristiques d'utilisation :

- Ce type de transport **n'est pas fiable** et **l'ordre d'émission des paquets n'est pas nécessairement celui des recéption**.
- Ce mode permet de construire ses propres règles de QOS et de plus il autorise la mise en oeuvre plus souple d'applications **multicast**.



② `TCP` (Transmission Control Protocol) . . . "Les processus utilisent le téléphone".

La notion de **stream** sous–entend un flôt d'informations non interprétées :

➢ On place les données dans un tuyau non-percé !

Caractéristiques d'utilisation :

➢ Il n'y a **pas de constitution d'enregistrement** (et par conséquent pas de notion de **record boundaries**) par l'application.

Mais attention => TCP segmente !

➢ Elle assure, de plus une **transmission fiable et ordonnée**.

Choix du protocole

L'API socket permet, évidemment, de supporter/choisir les deux protocoles de transport Internet UDP et TCP.

➢ La figure qui suit montre **les combinaisons valides** ainsi que les protocoles **sélectionnés par la paire** ("sémantique ou type vs domaine ou family") :

Sémantique / Domaine	AF_UNIX	**AF_INET**	AF_NS
SOCK_STREAM	Yes	**TCP**	SPP
SOCK_DGRAM	Yes	**UDP**	IDP
SOCK_RAW		IP	Yes
SOCK_SEQPACKET			SPP

En fonction de l'OS, de la machine et de ses interfaces physiques, des autorisations de l'utilisateur, **toutes les combinaisons (domaine, types) ne sont pas forcément permises**.

Adressage et Socket

On a vu que plusieurs domaines peuvent être associés à une socket.

- AF_UNIX : local
- AF_INET : Internet V4
- AF_INET6 : Internet V6
- ...

La syntaxe du nom de la socket, c'est à dire de son adresse, dépend logiquement du domaine dans lequel elle souhaite communiquer :

- Dans le domaine Unix, un nom de fichier.
- Dans le domaine INET, un entier sur 32 bits.
- Dans le domaine INET6, un entier sur 128 bits.
- ...

Adresses dans le domaine Unix

Il s'agit d'une structure :

① dont le premier champ correspond à la **famille/domain** de l'adresse.

② Et le second, au **chemin du fichier** qui va servir de lieu de communication.

```
1  /* Definitions for UNIX IPC domain. */
2  #include <sys/un.h>
3  struct  sockaddr_un { /* Adresse dans le domaine UNIX */
4    /* domaine/famille  = AF_UNIX */
5    short sun_family;
6    /* reference          */
7    char  sun_path[108];
8   };
```

Dans un code C, l'**initialisation de la structure** sera de la forme :

```
1  /* Variable de type "Adresse de socket Unix" */
2  struct sockaddr_un addrserv;
3
4  /* RAZ de l'adresse Unix */
5  memset(&addrserv, 0, sizeof(addrserv));
6
7  /* Initialisation de l'adresse Unix */
8  addrserv.sun_family = AF_UNIX;
9  strcpy(addrserv.sun_path,"./f_toto.su");
```

Le nom prend la forme d'un nom de fichier qui sert de lieu d'échange.

- L'attachement de l'adresse à la socket ne peut se faire que si la référence correspondante (i.e. le fichier) n'existe pas déjà.
- La suppression de la référence associée à une socket est réalisée
  - ✔ soit au niveau externe par la commande `rm` ,
  - ✔ soit dans une application par un appel à la primitive `unlink`

Adresses dans le domaine Internet

Le format de l'adresse d'une socket du domaine Internet comporte :

- Un **numéro de port** `sin_port` ... un entier 16 bits.
- et une **adresse Internet** `sin_addr` ... une autre structure qui ne contient qu'un entier 32 bits.

```
1  #include <netinet/in.h> /* Def. for INTERNET domain. */
2
3  /* Adresse Internet d'une machine ---------------*/
4  struct in_addr {
5      u_long s_addr; /* Un entier long unsigned */
6    };
7
8    /* Adresse Internet d'une socket ---------------*/
9  struct  sockaddr_in {
10     short   sin_family; /* Domaine AF_INET */
11     u_short sin_port;   /* Numero du port : Format reseau! */
12     struct in_addr sin_addr; /* Adr.Internet : Format reseau! */
13     char    sin_zero[8];     /* 8 caracteres inutilises */
14 };
```

C'est l'identifiant dans le monde Internet de la socket.

➢ Elle peut désormais être la "cible" d'une communication provenant d'une autre socket.

Dans un code C, l'initialisation de cette adresse est réalisée comme suit :

```
1  struct sockaddr_in adresse; /* Variable  adresse */
2  int port = 50000; /* Zone des ports prives ! */
3    ......
4  adresse.sin_family = AF_INET;
5  adresse.sin_addr.s_addr = htonl(INADDR_ANY);
6  adresse.sin_port = htons(port);
```

Reste à expliquer pas mal de choses : les "hton..." et le INADDR_ANY ...

Adresse dans le domaine AF_INET6

IPv6 addresses have a human-readable format that can be represented as a string of characters.

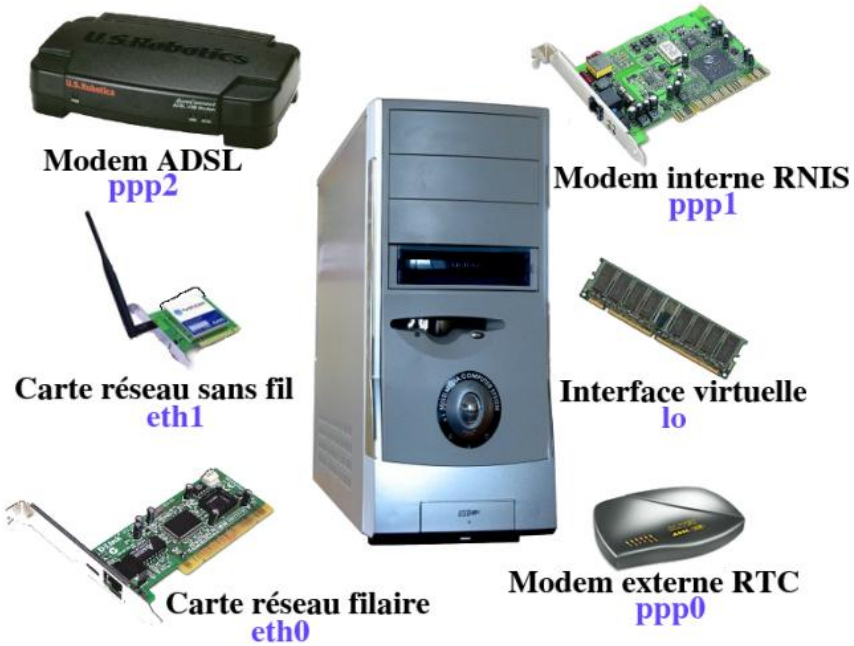
The details of IPv6 addresses are out of scope of this section but here are some examples :

- The : :1 IPv6 address identifies the computer on which the current program is running.
- The 2001 :6a8 :308f :9 :0 :82ff :fe68 :e520 IPv6 address identifies the computer serving the https://beta.computer-networking.info website.

```
1 struct sockaddr_in6 adresse;      // Variable adresse
2 memset(&adresse, 0, sizeof(adresse)); // RAZ
3 adresse.sin6_family = AF_INET6;  // IPv6 address domain
4 adresse.sin6_port = htons(55555);// on port 55555
5 inet_pton(AF_INET6, "::1",
6           &adresse.sin6_addr); // ::1 IPv6 address
```

Interfaces physiques d'une machine

Les interfaces physiques d'une machine lui permettent de réaliser ces connexions physiques avec par exemple :



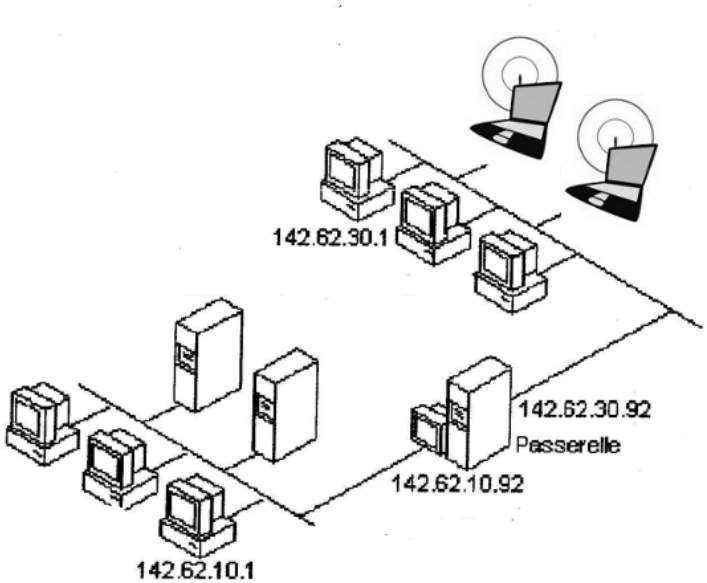
- LAN/Ethernet donc filaire
- Wifi donc sans fil
- Bluetooth donc sans fil
- ...

Un acronyme important est celui de NIC : "Network Interface Card".

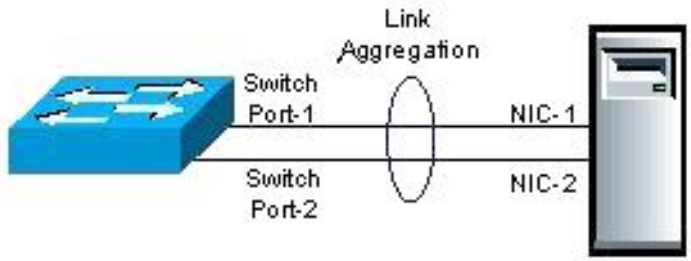
- Ne pas oublier que la connexion nécessite un matériel adapté . . . même si la carte est intégrée à la machine.

Combien d'Interfaces ?

Une machine communicante est nécessairement connectée, avec ou sans fil, à (au moins) un réseau.



➠ Mais rien n'empêche de la connecter à plusieurs réseaux différents (au niveau logique ou physique).



➠ Ou plusieurs fois au même (Link Aggregation), pour augmenter le débit et aussi être redondant (si défaillance).

Au final : Autant d'adresses que d'interfaces connectées !

INADDR_ANY

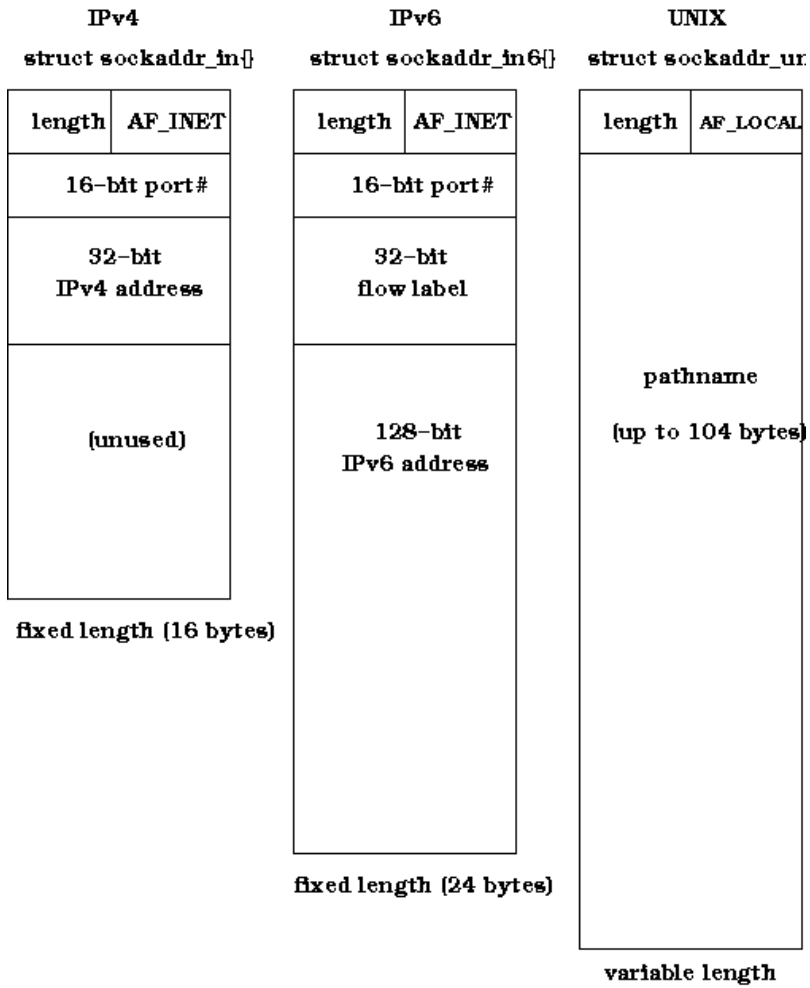
```
1        ...
2        adresse.sin_addr.s_addr = htonl(INADDR_ANY);
3        adresse.sin_port = htons(port);
4        ...
```

La valeur INADDR_ANY de *sin_addr.s_addr* permet d'associer à la socket **toutes les adresses possibles de la machine**.

- ✔ Utile si la machine est une passerelle, la socket peut être connectée sur les différents réseaux auxquels la machine est connectée.
- ✔ Mais aussi sur une machine quelconque, l'utilisation de cette valeur permet de faire l'économie de la recherche de l'adresse de la machine locale.
- ✔ **Enfin (et surtout ? !), cela évite d'écrire "en dur" une adresse particulière qui "verrouillerait" le code source/compilé sur une machine précise !**

Bilan des adresses

Des structures différentes **et** des longueurs différentes :



Conseil : Faire travailler le système "au maximum" pour les remplir !

Adresse et Endianess

Historiquement, l'adresse Internet d'une machine est représentée par un entier long (32 bits) non signé.

#IP	1	.	2	.	255	.	4
0x	01		02		FF		04

Soit en base 10 : 16973572_{10}

C'est pourquoi au niveau du code une adresse Internet est définie ainsi :

```
1    struct  in_addr {
2      u_long s_addr; /* Un entier long */
3      };
```

Or la représentation de tout ce qui est plus long qu'un octet, et donc à fortiori d'un entier long, est "**dépendant machine**".

Deux organisations mémoire sont possibles :

#IP	Valeur Binaire	Format Endian
132.227.70.77	84 C3 46 4D	Big (standard)
132.227.70.77	4D 46 C3 84	Little

① Dans le cas **"Big Endian"** :

l'octet de poids fort est stocké à l'adresse A et l'octet de poids faible est socké à l'adresse A+3.

Ce format suit celui de l'écriture : $132_{10} = 84_{16}, \ldots$

② Dans le cas **"Little Endian"** :

l'octet de poids fort est stocké à l'adresse A+3 et l'octet de poids faible est socké à l'adresse A.

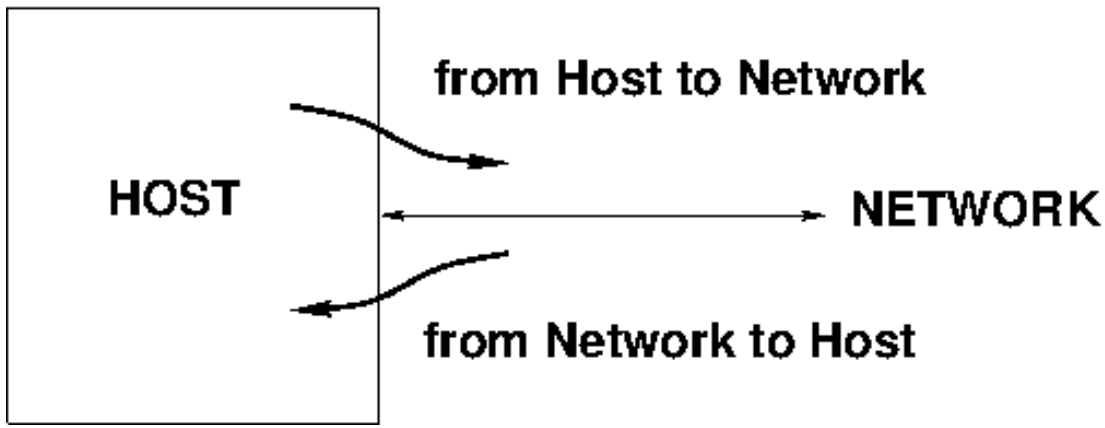
L'écriture est "inversée".

Le problème du format de représentation d'un nombre se pose pour tout nombre occupant plus d'un octet.

Notamment, au niveau réseau :

- Numéro de port
- Adresse Internet

L'idée, pour faire communiquer des machines utilisant potentiellement des standards de représentation des nombres différents, est de définir un `standard réseau` :



Les machines désireuses d'utiliser le réseau devront se conformer à ce standard de fait !

Fonctions d'arrangement

Pour réaliser cela, les API pévoient **sur chaque système des fonctions d'accés au standard réseau**.

Elles se déclinent en :

- ✔ **hton** lorsque les **données vont du host vers le network**,
- ✔ **ntoh** lorsque les **données vont du network au host**,
- ✔ **s** lorsqu'il s'agit de short (numéro de port)
- ✔ **l** lorsqu'il s'agit de long (adresse IP)

Ce qui donne les prototypes :

```
1  u_long htonl(u_long)  /* Pour les adresses Internet */
2  u_long ntohl(u_long)
3
4  u_short htons(u_short)  /* Pour les numeros de ports */
5  u_short ntohs(u_short)
```

Implémentations

Le **standard réseau c'est Big Endian** : "Most Significant Byte first".

Mais c'est **Intel qui s'impose au niveau des** μP :-)

- On the i80x86 the host byte order is Least Significant Byte first(Little Endian), whereas the network byte order, as used on the Internet, is Most Significant Byte first (Big Endian). (cf $< asm/byteorder.h >$)
- Sur "Sun Microsystems" (Big Endian) les fonctions sont des macros nulles ! (cf$< netinet/in.h >$).

Conclusion :

Le standard réseau est différent de celui de la majorité des machines (fixes) en Intel !

Primitives de manipulation

- ✔ Voir en détail les différents concepts et fonctions de l'API Socket.
- ✔ Ca y est . . . on code !

File Descriptor

Les E/S "fichiers" de bas niveau (UNIX) utilisent le concept de **file descriptor** :

```
1    int cc, fd, nbytes;
2    char buf[100];
3
4    nbytes = sizeof buf;
5    fd = open("tmp.txt", O_RDWR);
6    cc = read(fd, buf, nbytes);
7    cc = write(fd, buf, nbytes);
```

- ➢ Ils sont généralement les résultats des appels `open` et `create` .
  On obtient ainsi un flôt d'octets.
- ➢ Dans ce flôt, on peut lire (via read(...)) et écrire (via write(...)).

L'API socket utlise aussi cette approche d'un "descripteur" . . . **de sockets**.

Socket Descriptor

L'appel système `socket(...)` renvoie une valeur entière, qui est un descripteur de socket (**socket descriptor** ou sockfd).

```
1  #include <sys/types.h>
2  #include <sys/socket.h>
3  int socket(int domain, int type, int protocol);
```

Les paramètres font référence au choix d'un "sémantique utilisateur" (cf page 47) :

① **domain** (AF :"address family") specifie un domaine de communication à l'intérieur duquel la communication prend place : ceci sélectionne la famille de procotoles qui va être utilisée.

Ces familles sont définies dans `sys/socket.h` . Sous Ubuntu, on peut les voir dans le fichier : `/usr/include/x86_64-linux-gnu/bits/socket.h`

- ➤ Les domaines courants/possibles : AF_UNIX et AF_INET.

② **type** indiqué spécifie la semantique de la communication :

- ➤ Les types courants sont SOCK_STREAM, SOCK_DGRAM, SOCK_RAW, ...

③ **protocol** vaut typiquement 0 **ce qui laisse le système choisir** !

- ➤ On peut aussi spécifier certains protocoles **spécialisés** (cf /etc/protocols).

Suppression d'une socket : close()

Lorsque vous avez fini d'utiliser un socket, on peut **fermer et rendre au système son descripteur** avec la fonction close :

```
1 #include <sys/types.h>
2 #include <sys/socket.h>
3 int close(int fd);
```

Comme pour les fichiers, si des données sont en attente, la fonction close essaye d'achever l'émission.

- On peut contrôler cette phase avec un timeout (option de socket : SO_LINGER).

Au niveau du système, comme pour les fichiers, **une socket est supprimée à la réalisation effective de la fermeture du dernier descripteur permettant d'y accéder**.

Attachement d'une adresse : bind()

Aprés sa création, à la différence d'un fichier, la socket n'est effectivement connue que du processus !

➟ Le rôle de la primitive `bind(...)` est de lui **attacher** une adresse/nom à usage du monde extérieur.

```
1  #include <sys/types.h>
2  #include <sys/socket.h>
3
4  int bind (int sockfd,
5            struct sockaddr *myadr, int addrlen);
```

Les fonctions de manipulations de socket ont été écrites pour des sockets dont l'adresse peut être de type Unix ou Internet ou ...

➤ Elle sont donc définies, notamment au niveau de leurs paramètres formels en utilisant une structure d'adresse générique : **sockaddr**

➤ Le paramètre **addrlen** est supposé être la taille en octet de l'adresse effective.

Gestion de la généricité de bind()

Lorsque l'on veut utiliser ces fonctions, on doit leur fournir des adresses en paramètres dont le type n'est pas générique.

- **sockaddr_un** . . . dans le domaine AF_UNIX,
- **sockaddr_in** . . . dans le domaine AF_INET, . . .

Pour être utilisable quelque soit le type de l'adresse, la fonction bind() a été écrite en utilisant un pointeur générique :

- **Il faut donc réaliser une coercition** (i.e. cast) vers la structure **sockaddr** générique !

Par exemple, lors de l'appel avec une adresse du domaine UNIX) :

```
1    struct sockaddr_un adresse; /* Domaine UNIX */
2
3    bind(sd,
4         (struct sockaddr *)&adresse, sizeof(adresse))
```

Notion de bind() implicite

Si une socket n'est pas explicitement attachée à une adresse par la fonction `bind()`, la primitive `sendto()` réalisera un **attachement implicite** pour les sockets UDP.

Pour les sockets TCP, ceci est réalisé par la primitive `connect()`.

De ce fait c'est le système qui choisit un port . . . éphémère ?

Pour connaître les choix opérés par le système, on utilisera aposteriori la fonction `getsockname()` : cf exemple suivant . . .

Dans cet exemple, il s'agit de **factoriser les phases de création d'une socket** INET au travers de l'élaboration d'une fonction : `create_socket`

```
1   /** Fichier : mysocket.c */
2   #include "mysocket.h"
3   int create_socket(int type, int port, struct sockaddr_in *p_adress){
4     /* Fonction de creation et d'attachement d'une socket INTERNET.
5        type : de la socket (SOCK_STREAM ou SOCK_DGRAM)
6        port : le numero de port peut etre choisi lors du bind (si 0)
7        p_adress : REND l'adresse effective choisie lors du bind par
8                   exemple du fait du choix de l'adresse machine IADDR_ANY
9        Return : le descripteur de socket ... si OK ? */
10    static struct sockaddr_in adresse;
11    int desc; /* descripteur de la socket */
12    socklen_t  longueur = sizeof(struct sockaddr_in); /* taille de l'adresse */
13    /* 1) Creation de la socket */
14    if ((desc=socket(AF_INET, type, 0)) == -1){
15      perror("Creation de socket impossible");
16      return -1;
17    }
18    /* 2) Preparation de l'adresse d'attachement */
19    adresse.sin_family=AF_INET;
20    adresse.sin_addr.s_addr = htonl(INADDR_ANY);
21    adresse.sin_port = htons(port); /* Numero de port en format reseau */
22   /* 3) Demande d'attachement de la socket */
23    if (bind(desc,(struct sockaddr *) (&adresse),longueur) == -1){
24     perror("Attachement de la socket impossible");
25     close(desc); /* => fermeture de la socket */
26     return(-1);
27    }
28    /* 4) Recuperation de l'adresse effective d'attachement : p_address. */
29    if (p_adress != NULL)
30      getsockname(desc, (struct sockaddr *) p_adress, &longueur);
31    return desc;
32  }
```

```
1   /** Fichier : mysocket.h */
2   #ifndef MYSOCKETH
3   #define MYSOCKETH
4
5   #include <stdio.h>
6   #include <stdlib.h>
7   #include<unistd.h>
8   #include <sys/types.h>
9   #include <sys/socket.h>
10  #include <netinet/in.h>
11
12  int create_socket(int type, int port, struct sockaddr_in *p_adress);
13  #endif
```

```c
/** Fichier : main_sockinet.c */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include "mysocket.h"
#include <netinet/in.h>
#include <arpa/inet.h>

/*
   Tester la creation de sockets
*/
int main(int argc, char *argv[]){
  struct sockaddr_in adresse_udp, adresse_tcp;
  int port_udp, port_tcp;
  int socket_udp, socket_tcp;

  /* Cree une socket de type SOCK_DGRAM sur port 2467 */
  port_udp = 2467;
  if((socket_udp = create_socket(SOCK_DGRAM, port_udp,
                  &adresse_udp))!=-1){
    printf("Socket UDP attachee a l'interface : %s\n", inet_ntoa(adresse_udp.sin_addr));
    printf("Socket UDP attachee au port : %d\n", ntohs(adresse_udp.sin_port));
  }
  /* Cree une socket de type SOCK_STREAM sur port quelconque */
  port_tcp = 0;
  if((socket_tcp = create_socket(SOCK_STREAM, port_tcp,
                  &adresse_tcp))!=-1){
    printf("Socket TCP attachee a l'interface : %s\n", inet_ntoa(adresse_tcp.sin_addr));
    printf("Socket TCP attachee au port : %d\n", ntohs(adresse_tcp.sin_port));
  }

  sleep(60);
}
```

En Python :

Les principes restent !

```
#!/usr/bin/env python
'''
Creation d'une socket en python
'''
import socket

#Creation du point de communication
s = socket.socket(socket.AF_INET,  #Family/Domaine = Internet
                  socket.SOCK_DGRAM) #Type = Datagram

#Internet + Datagram donc : Protocol UDP

#Attachement : INADDR_ANY et port automatique
s.bind(("", 0))
print "socket for sending :", s.getsockname()
#Close UDP socket
s.close()
```

En Java :

```
package netclient;

import java.net.DatagramSocket;
import java.net.InetAddress;

public class Client {

    public static void main(String[] args) throws Exception {

      // création d'une socket liée au port 7777
      DatagramSocket socket = new DatagramSocket(7777);

      //----- close UDP socket -----
      socket.close();
    }
}
```

Services et Adresse IP

Avant d'envoyer ou de se connecter à une machine distante, il y a souvent un travail préparatoire à effectuer, pour mettre en place les éléments du dialogue :

- Adresse IP de la machine distante,
- Numéro de port d'un service,
- ...

Il y a quelques fonctions de bibliothèque particulièrement utiles à la manipulation des adresses et des services ...

Problématique

Communiquer suppose de **connaître l'adresse IP et/ou le service (numéro de port)** que l'on veut atteindre.

Dans les deux cas (IP/port), on a vu qu'il existait une approche **nominative** de ces paramètres.

- Adresse IP : Utilisation des serveurs de noms (DNS) pour les machines
- Service : Fichier `/etc/services`

Comme on peut difficilement demander à un programmeur de connaître les translations :

- entre tous les noms de machines et leurs numéros IP
- entre tous les services et leurs numéros de ports

Il existe des fonctions que l'on peut utiliser pour obtenir dynamiquement ces informations.

Résolution des adresses IP

① `gethostbyname()` : permet de récupérer l'adresse IP d'une machine dont on connaît le nom.

② `gethostbyaddr()` : permet de récupérer le nom d'une machine dont on connaît l'adresse IP (sous forme binaire/entier).

Ces deux fonctions ("deprecated") retournent un pointeur sur un objet dont la structure est :

```
1  struct hostent {
2      char    *h_name;        /* official name of host */
3      char    **h_aliases;    /* list of alias : NULL terminated */
4      int     h_addrtype;     /* host address type : AF_INET */
5      int     h_length;       /* length of address */
6      char    **h_addr_list; /* list of addresses : NULL terminated
7  };
8  /* for backward compatibility */
9  #define h_addr  h_addr_list[0]
```

Structure hostent

```
1  struct hostent {
2    char    *h_name;        /* official name of host */
3    char    **h_aliases;    /* list of alias : NULL terminated */
4    int     h_addrtype;     /* host address type : AF_INET */
5    int     h_length;       /* length of address */
6    char    **h_addr_list; /* list of addresses : NULL terminated
7  };
8  /* for backward compatibility */
9  #define h_addr  h_addr_list[0]
```

Cette structure contient les informations définissant un "hôte".

- ✔ Le nom officiel de la machine
- ✔ Une liste de noms d'alias (i.e. nom alternatifs)
- ✔ Un entier codant le type d'adresse utilisé par l'hôte
- ✔ La longueur de l'adresse de l'hôte
- ✔ Un tableau/list de pointeurs vers des adresses de l'hôte (au format/endianess réseau et NULL terminated).

gethostbyname() : "deprecated" mais si fréquente ...

```c
/** Fichier : gethostbyname.c */
#include <stdio.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int main(int argc, char *argv[]){ /* On veut l'adresse IP d'un hostname */
  char hostname[]= "www.hotmail.com";  /*"sesame-mips.unice.fr", "www.free.fr" */
  struct hostent *host;              /* Représentation d'un hote */
  struct in_addr ip_addr;            /* Représentation d'une internet address */
  host = gethostbyname(hostname); /* printf("Requete DNS ...\n"); */
  if(host == NULL) {
    printf("Resolution DNS : %s\n", hstrerror(h_errno));
    return -1;
  }
  else { /* Analyse de la structure retournée */
    printf("Hostname   : %s \n", host->h_name);
    printf("Alias      :  \n");      /* Print alias names list */
    printf("\t %d : %s \n", 0, host->h_aliases[0]);
    if (host->h_aliases[1] == NULL)
      printf("\tNULL\n");
    else
      printf("\t %d : %s \n", 1, host->h_aliases[1]);

    /* Ce code est intentionnellement incomplet et meriterait de belles iterations ! */
    printf("IP     : \n");      /* Print IP list ... ou du moins la premiere */
    ip_addr.s_addr =  *( (unsigned long *) host->h_addr_list[0]) ;
    printf("\t  %d : %s\n", 0 ,  inet_ntoa(ip_addr));  /* Look at inet_ntoa() ! */
    if (host->h_addr_list[1] == NULL)
      printf("\tNULL\n");
  }
}
```

gethostbyaddr() : "deprecated" mais si fréquente ...

```c
/** Fichier : gethostbyadd.c */
#include <stdio.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>

int main(int argc, char *argv[]){ /* On veut le hostname d'une adresse */
  struct hostent *s;

  char ip_str[] = "134.59.2.254";
  unsigned long ip = inet_addr(ip_str); /* IP with network byte order */
  printf("%s => %lx / %x\n", ip_str, ip, ntohl(ip)); /* See  endianess*/

  struct in_addr ipa;
  ipa.s_addr = ip; /* Define the IP address */

  /* Then get the name using a reverse dns */
  s = gethostbyaddr((char *)&ipa, sizeof(struct in_addr), AF_INET);
  if ( !s ) { /* Fail */
    fprintf(stderr,"Error: %s\n", hstrerror(h_errno));
  }
  else { /* Success */
    fprintf(stderr,"Name : %s\n", s->h_name); /* Official name */
  }
}
```

Résolution des services

① `getservbyname()`

② `getservbyport()`

A partir d'un numéro de port ou d'un nom de service fourni en paramètre, **ces fonctions retournent un pointeur sur un objet de type**

`struct servent`

renseigné à partir des champs du fichier **/etc/services**.

```
1  #include <netdb.h>
2
3  struct servent {
4    char     *s_name;          /* official service name */
5    char     **s_aliases;      /* alias list */
6    int      s_port;             /* port number */
7    char     *s_proto;         /* protocol to use */
8  };
```

Il s'agit d'une façon "pratique"/dynamique d'utiliser le fichier **/etc/services**.

getservbyname()

```c
/** Fichier : getservbyname.c */
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>

int main(int argc, char *argv[]){
   struct servent *serv;
   if (argc < 3) {
      puts("Incorrect parameters. Use:");
      puts("    ./a.out service-name protocol-name");
      puts(" \n Ex :   ./a.out ssh tcp");
      return EXIT_FAILURE;
   }
   /* getservbyname() - opens the etc.services file and returns the */
   /* values for the requested service and protocol.                */

   serv = getservbyname(argv[1], argv[2]);
   if (serv == NULL) {
      printf("Service %s not found for protocol %s\n", argv[1], argv[2]);
      return EXIT_FAILURE;
   }
   /* Print it. */
   printf("Name: %-15s  Port: %5d    Protocol: %-6s\n",
            serv->s_name,ntohs(serv->s_port),serv->s_proto);
   return EXIT_SUCCESS;
}
```

getservbyport()

```c
/** Fichier : getservbyport.c */
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
/* Suivre les appels systeme avec strace ... */
int main(int argc, char *argv[]){
   struct servent *serv;
   if (argc < 3) {
      puts("Incorrect parameters. Use:");
      puts("    ./a.out port-number protocol-name");
      puts(" \n Ex :   ./a.out 22 tcp");
      return EXIT_FAILURE;
   }
   /* getservbyport() - opens the etc.services file and returns the */
   /* values for the requested service and protocol.                */

   serv = getservbyport(htons(atoi(argv[1])), argv[2]);
   if (serv == NULL) {
      printf("Service %s not found for protocol %s\n", argv[1], argv[2]);
      return EXIT_FAILURE;
   }
   /* Print it. */
   printf("Name: %-15s  Port: %5d    Protocol: %-6s\n",
            serv->s_name,ntohs(serv->s_port),serv->s_proto);
   return EXIT_SUCCESS;
}
```

getnameinfo() /getaddrinfo()

Les fonctions précédentes **ne sont pas compatibles** avec l'occurrence de IPv6 .

➤ D'autres fonctions ont été définies pour effectuer le travail de recherche de nom et d'adresse.

Il s'agit de :

① `getaddrinfo()` : Etant donnés, un noeud et un service (qui identifie un hôte Internet et un service), la fonction retourne une ou plusieurs structures addrinfo.

On peut ainsi obtenir l'équivalent de `gethostbyname` , de `getservbyname` et de `getservbyport` .

② `getnameinfo()` : Fonction inverse, elle permet de convertir une adresse de socket en un hôte et un service.

On peut ainsi obtenir l'équivalent de `gethostbyaddr` .

getnameinfo()

```c
/** Fichier getnameinfo.c
 * Here, getnameinfo() is used for reverse lookup
 * i.e. to get hostname from the IP address
*/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
int main(int argc, char *argv[]){
  struct sockaddr_in sa;          /* input IP address */
  socklen_t len = sizeof(sa);
  char *addstr =  "134.59.2.13"; /* "127.0.0.1" */
  sa.sin_family = AF_INET;
  /* from text/string to binary/integer for IPv4 AND IPv6 */
  inet_pton(AF_INET, addstr, &sa.sin_addr);

  char node[NI_MAXHOST]; /* Hostname ... TO FILL from sockaddr "sa" ! */
  int res = getnameinfo((struct sockaddr*)&sa, len,
            node, sizeof(node),
            NULL, 0, NI_NAMEREQD);
  if (res){ /* printf("could not resolve hostname"); */
    printf("%s\n", gai_strerror(res));
    exit(1);
  }
  else
    printf("%s\n", node); /* hostname */
  return 0;
}
```

getaddrinfo() I

```c
/*
 * getaddrinfo.c - Simple example of using getaddrinfo(3) function.
 *  voir l'utilisation de getnameinfo en fin de fonction .../
 * Michal Ludvig <michal@logix.cz> (c) 2002, 2003
 * http://www.logix.cz/michal/devel/
 * License: public domain.
 */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>

int lookup_host (const char *host, const char *serv) {

  struct addrinfo hints;
  struct addrinfo *ailist, *aip; /*List of struct addrinfo : returned
  by getaddrinfo */

  int errcode;
  char addrstr[100];
  socklen_t len;
  void *ptr;

  printf ("Host: %s\n", host);

  /*The hints argument points to an addrinfo structure that specifies
    criteria for selecting the socket address structures returned in
    the list pointed to by ailist */
  memset (&hints, 0, sizeof (hints));
  hints.ai_family = PF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;
```

getaddrinfo() II

```
35      hints.ai_flags |= AI_CANONNAME;
36
37      /*Given node and service, which identify an Internet host and a
38        service, getaddrinfo() returns one or more addrinfo structures,
39        each of which contains an Internet address */
40      errcode = getaddrinfo (host, serv, &hints, &ailist);
41      if (errcode != 0){
42        printf("%s\n", gai_strerror(errcode));
43        return -1;
44      }
45
46      /* ailist contains list of struct addrinfo  */
47      for (aip = ailist; aip != NULL; aip = aip->ai_next){
48        inet_ntop (aip->ai_family, aip->ai_addr->sa_data, addrstr, 100);
49        switch (aip->ai_family){
50        case AF_INET:
51          ptr = &((struct sockaddr_in *) aip->ai_addr)->sin_addr;
52          len =  sizeof(struct sockaddr_in);
53          break;
54        case AF_INET6:
55          ptr = &((struct sockaddr_in6 *) aip->ai_addr)->sin6_addr;
56          len =  sizeof(struct sockaddr_in6);
57          break;
58        }
59        /* ptr contient l'adresse d'une struct addr (in ou in6) */
60        inet_ntop (aip->ai_family, ptr, addrstr, 100);
61        printf ("IPv%d address: %s (eq. gethostbyname of %s)\n",
62            aip->ai_family == PF_INET6 ? 6 : 4,
63            addrstr,
64            aip->ai_canonname);
65
66        char node[NI_MAXHOST]; /* Reverse Resolve Hostname from address */
67        if (getnameinfo(aip->ai_addr, len,
68                node, sizeof(node), NULL, 0, NI_NAMEREQD) == 0)
```

getaddrinfo() III

```
69        printf("%s\n", node);
70      }
71      return 0;
72    }
73
74    /*--------------------------------------*/
75
76    int main (int argc, char *argv[]){
77      const char* host = "sesame-mips.unice.fr";
78      const char* serv = "ssh";
79
80      /* return lookup_host (host, NULL); */
81      return lookup_host (host, serv);
82    }
```

Utilisation de getaddrinfo() I

Résolution de nom :

```c
/*  http://www.binarytides.com/hostname-to-ip-address-c-sockets-linux */
#include<stdio.h> //printf
#include<string.h> //memset
#include<stdlib.h> //for exit(0);
#include<sys/socket.h>
#include<errno.h> //For errno - the error number
#include<netdb.h> //hostent
#include<arpa/inet.h>
/* Get ip from domain name ...  */
int hostname_to_ip(char *hostname , char *ip){
  int sockfd;
  struct addrinfo hints, *servinfo, *p;
  struct sockaddr_in *h;
  int rv;

  memset(&hints, 0, sizeof hints);
  hints.ai_family = AF_UNSPEC; /* use AF_INET6 to force IPv6 */
  hints.ai_socktype = SOCK_STREAM;
  if ( (rv = getaddrinfo(hostname , NULL , &hints , &servinfo)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
    return 1;
  }

  // loop through all the results and connect to the first we can
  for(p = servinfo; p != NULL; p = p->ai_next) {
      h = (struct sockaddr_in *) p->ai_addr;
      strcpy(ip , inet_ntoa( h->sin_addr ) );
    }
  freeaddrinfo(servinfo); // all done with this structure
  return 0;
}

```

Utilisation de getaddrinfo() II

```
33  /* ============================================================*/
34
35  int main(int argc , char *argv[]){
36    char *hostname = argv[1];
37    char ip[100];
38    if(argc <2){
39      printf("Please␣provide␣a␣hostname␣to␣resolve");
40      exit(1);
41    }
42    hostname_to_ip(hostname , ip);
43    printf("%s␣resolved␣to␣%s" , hostname , ip);
44    printf("\n");
45  }
```

Datagrammes/UDP

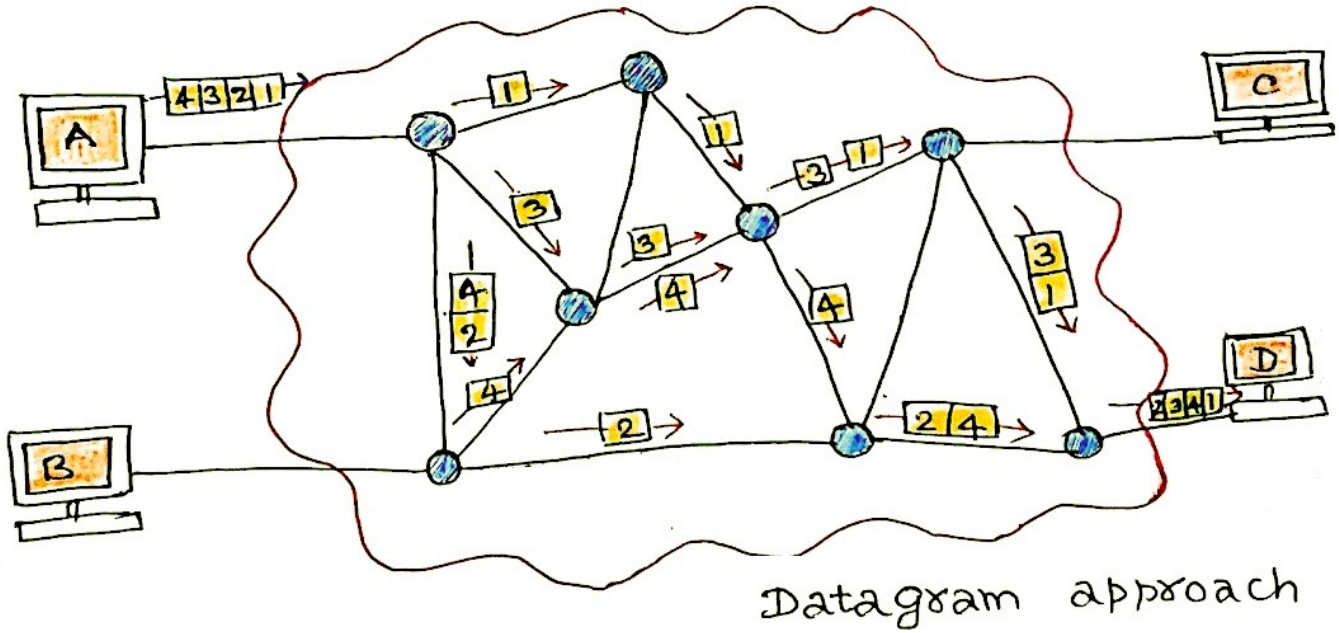
Cette partie est consacrée à l'étude des mécanismes relatifs à la communication de processus par datagrammes.

Sockets de type **SOCK_DGRAM** :

- La notion de **datagram** est associée à un service de type **connectionless**.
- Le protocole de mise en œuvre dans la famille Internet : UDP

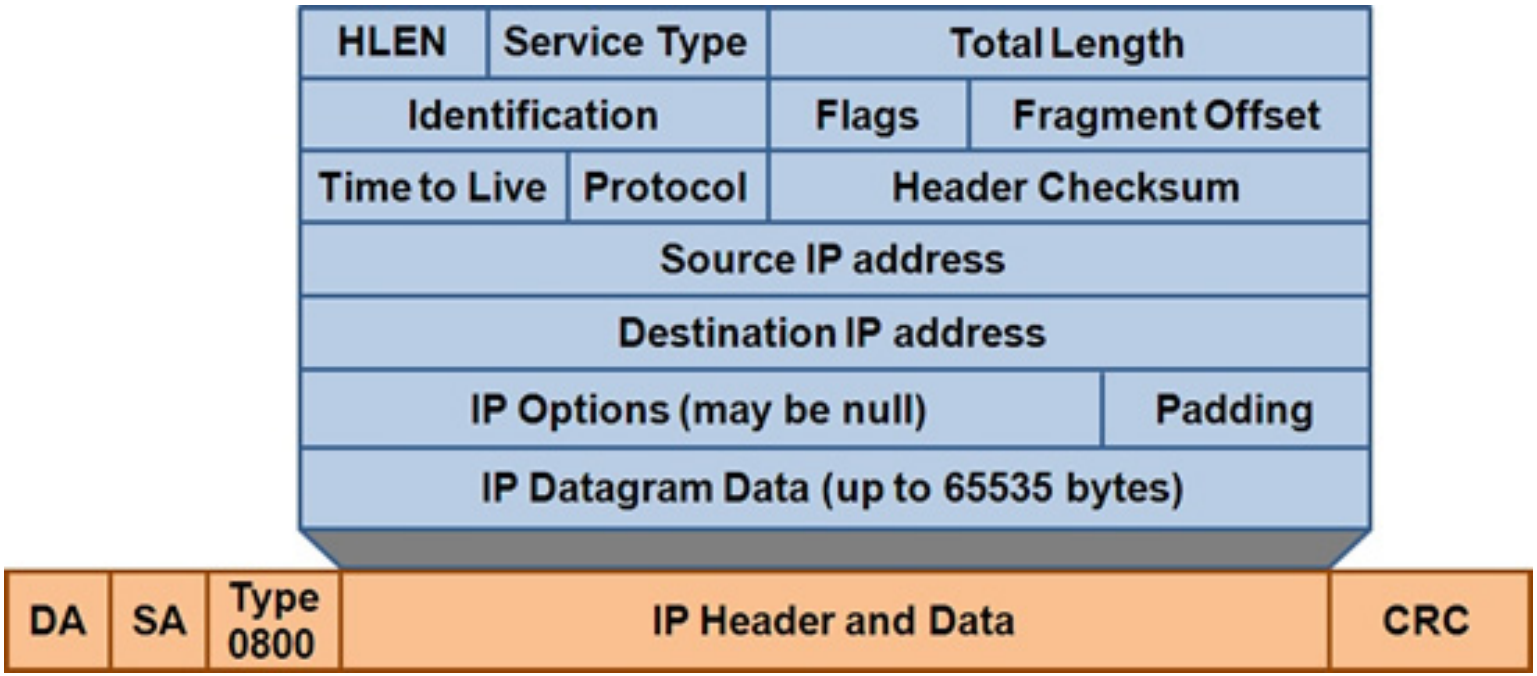
Caractéristiques des datagrammes : Rappel

La **commutation de paquets** s'appuie sur le même principe que celui utilisé pour la commutation de messages . . . **avec la différence qu'un message est divisé en une série de paquets** de longueurs finies (max 1000 à 2000 bits) que l'on appelle **segments** / **datagrams** .



Les "datagrams" sont transmis indépendamment d'un système à l'autre !

Cette indépendance suppose que **chaque message contient toutes les informations nécessaires à sa livraison** :



- Notamment les adresses de l'émetteur et de la destination,
- . . . etc

Petite question : où sont les numéros des ports sur cette figure ?

SOCK_DGRAM + AF_INET = UDP

Si la sémantique utilisateur choisie est SOCK_DGRAM et que le domaine utilisé est AF_INET, la communication s'appuie sur le protocole UDP.

Conséquences "fonctionnelles :"

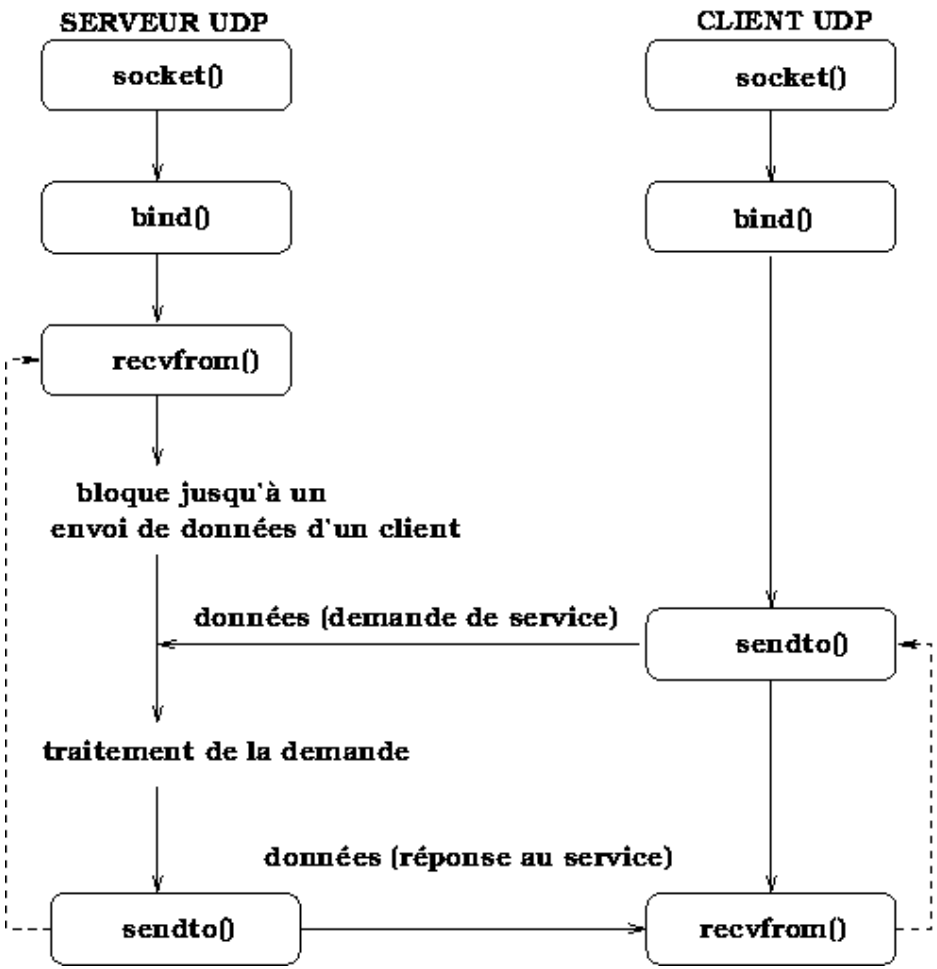
- Les datagrammes peuvent se "perdre"(i.e ne pas arriver).
- Il n'y a pas de service permettant à l'émetteur de savoir si son message est arrivé à destination.
- Dans le cas d'envois successifs de messages à un même destinaire, l'emetteur n'est pas assuré que ses messages seront délivrés dans le bon ordre.

C'est donc à l'application ("end to end") de mettre en place ces mécanismes si elle souhaite en disposer (délais de garde, acquittements, réémissions, ...).

- **C'est à la fois une faiblesse mais aussi une force !**
  On peut développer sa propre QOS ?
- Mais dans ce qui suit, on ne se préoccupe pas de gagner en QOS.

Organisation d'un client-serveur UDP

Des processus souhaitant communiquer vont donc réaliser, selon les circonstances, un certain nombre des opérations suivantes :



① demander la création d'une socket,

② demander éventuellement l'attachement à un port (sinon c'est automatique),

③ construire l'adresse de son interlocuteur,

④ procéder à des émissions et des réceptions de messages.

On remarque une **grande symétrie entre le client et le serveur** :

➢ Seuls les échanges sont dans un ordre différent.

Les données échangées

Au niveau de l'API, les données échangées entre client et serveur prennent la forme **d'un tableau d'octets.**

C'est à l'utilisateur de l'API de prévoir ce que contiennent ces octets.

① Par exemple, si le serveur ne propose qu'**un seul service** qui est : "incrémenter de un, une valeur entière fournie dans la requête."

Cela signifie que la donnée échangée est un nombre :

le client demande "3 " et le serveur répondra "4"

② Si maintenant, le serveur peut **effectuer différentes opérations** arithmétiques.

➢ **Il faut alors que les données échangées contiennent par exemple la nature de l'opération souhaitée et les arguments sur lesquels elle va portée** :

le client demande "* 3 4" et le serveur répondra "12"

On voit là se mettre en place **un protocole** : la donnée contient une information "composite"/structurée !

Serveur Itératif ou Concurrent ?

Un serveur peut être écrit de manière itérative ou concurrente.

① **Serveur itératif** :

Dans le premier cas, il traite les requêtes "séquentiellement" :

- ➠ Il reçoit une requête et la traite,
- ➠ et il recommence ...

Le serveur n'instancie pas de "main d'œuvre" / "sous traitance" (fork, thread, ...).

- ✔ C'est un mode, une architecture, bien adapté aux services qui peuvent s'executer rapidement et correspondent à un schéma question/réponse.
- ✔ C'est le cas de la plupart des serveurs développés avec des sockets du type SOCK_DGRAM.

Serveur Itératif ou Concurrent ?

② **Serveur concurrent (. . . parallèle)** :

Dans ce mode, le **serveur instancie des processus (fils) pour traiter les requêtes**.

- ✔ Ce second mode est plus adapté au mode connecté que l'on rencontre avec les sockets de type SOCK_STREAM.

  On gère des flux, pas des **salves**, comme avec les sockets du type SOCK_DGRAM.

- ✔ Dans ce cas, plusieurs processus de service travaillent de manière concurrente (voir parallèle sur les nouvelles machines multiprocesseurs ...)

Envoyer un message : sendto(...)

```
1   #include <sys/types.h>
2   #include <sys/socket.h>
3
4   int sendto (
5       int sockfd,     /* SD de la socket locale */
6       char *buff,     /* Message */
7       int nbytes,
8       int flags,      /* 0 */
9       struct sockaddr *to, /* Adresse du destinataire */
10      int toaddrlen );
```

Un appel à la fonction `sendto( ... )` correspond à la **demande d'envoi**, via la socket identifiée localement par le descripteur **sockfd**, du message **buff** de longueur **nbytes** à la destination de la socket dont l'adresse est pointée par **to** et est de longueur **toaddrlen**.

- ✔ La **valeur de retour** est, en cas de réussite, le nombre de caractères effectivement envoyés.

La primitive sendto(...)

Parmi les **erreurs éventuelles** générées par la fonction, précisons que :

① l'erreur EMSGSIZE correspond au fait que **le message est trop long** pour permettre la garantie d'un envoi atomique (un paquet) au protocole de niveau inférieur.

② EWOULDBLOCK correspond, en mode non–bloquant, à la **saturation des tampons d'émission** qui entraînerait en mode bloquant la suspension (temporaire) du processus émetteur.

③ Dans le cas du domaine AF_UNIX : le système est en mesure de vérifier si la référence destinatrice existe (si elle n'existe pas, une erreur ENOENT est renvoyée) et, si elle existe, c'est qu'il y a actuellement une socket qui lui est associée (dans le cas contraire l'erreur ECONNREFUSED est signalée).

Dans le cas du domaine AF_INET : plus aucune erreur sur le site distant n'est signalée.

La primitive sendmsg(...)

Juste pour la citer !

```
1       #include <sys/types.h>
2       #include <sys/socket.h>
3       int sendmsg (
4           int s,
5           const struct msghdr *msg ,
6           unsigned int flags);
```

Cette primitive permet d'**envoyer un message dont le contenu est constitué de fragments non contigus en mémoire**.

- ➢ Les différents fragments constituant le message sont rassemblés avant envoi.

Dans le domaine AF_UNIX, la primitive permet par ailleurs l'envoi de données interprétées : ainsi il est possible de transmettre à un processus un descripteur, lui permettant d'en obtenir un synonyme (c'est à dire un descripteur pointant sur la même entrée dans la table des fichiers ouverts que le descripteur transmis)

Recevoir un message : recvfrom(...)

```
1  #include <sys/types.h>
2  #include <sys/socket.h>
3  int recvfrom (
4       int sockfd, /* SD de la socket locale */
5       char *buff, /* Contenu DGRAM recu */
6       int nbytes,
7       int flags, /* 0 ou MSG_PEEK */
8       struct sockaddr *from, /* Adresse de l'expediteur */
9       int *fromaddrlen);      /* du DGRAM recu           */
```

Un appel à cette primitive **permet de lire sur la socket identifiée** par le descripteur **sockfd** un message **buff** de longueur **nbytes**.

- Le message extrait est mis dans le tampon de réception de la socket.
- Dans le cas où le message reçu est de longueur supérieure à l'espace réservé (le **sendto(..)** utilise un buffer de taille différente), le message sera tronqué.

  Le programmeur doit mettre en place un protocole pour éviter cela !

- Si le pointeur **from** est non nul, il est rempli avec l'adresse de la source du message.

  Idem pour **fromaddrlen** qui, avant l'appel contenait la taille de l'adresse, recoit la taille exacte de l'adresse **from**.

  Ces informations sont extraites du datagramme reçu !

- Le paramètre **flags** peut avoir la valeur MSG_PEEK et dans ce cas, s'il y a un message dans le tampon de lecture de la socket, il est recopié à l'adresse de **buff**, mais il n'est pas extrait.

  Le prochain appel à la primitive portera exactement sur le même message.

- **La primitive est bloquante, sauf demande contraire**, si le tampon est vide.

  . . . enfin ça reste à voir . . . cela peut être un point de portabilité à vérifier.

- Elle **renvoie le nombre de caractères recopiés** dans le buffer.

- En cas d'erreur, elle renvoie la valeur -1 (0 dans le cas d'une lecture non–bloquante avec **flags** O_NDELAY).

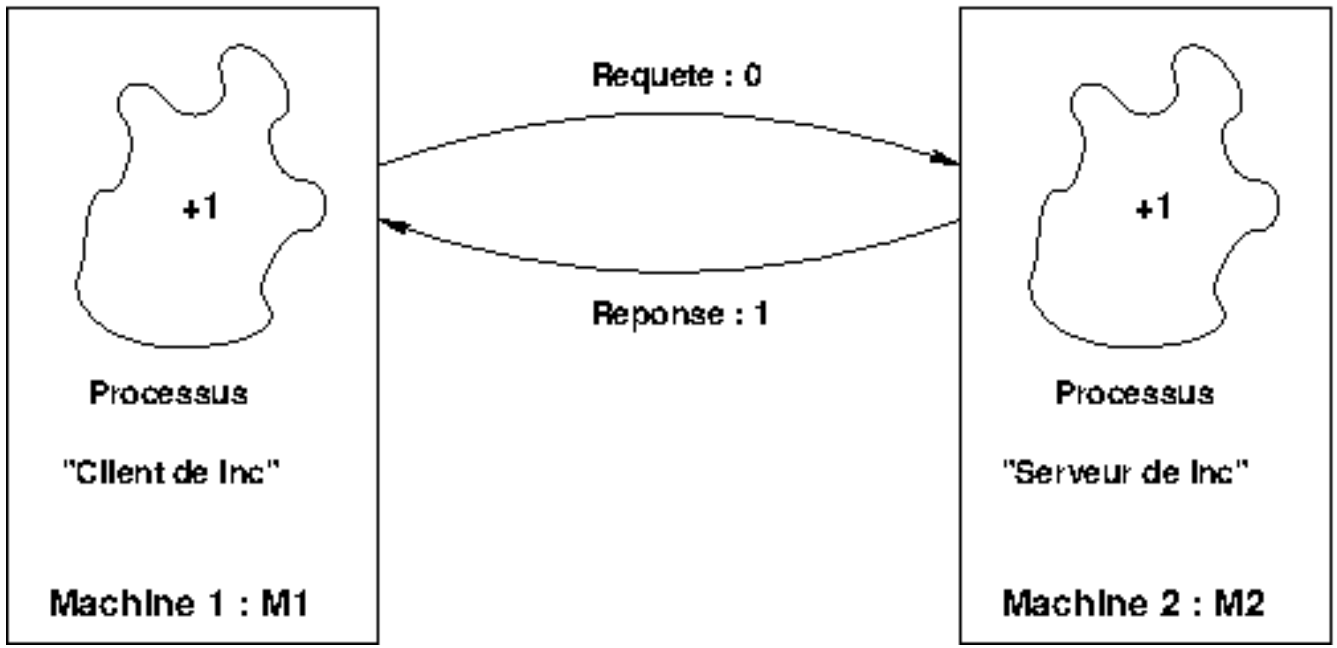
La primitive recvmsg()

Un appel à cette fonction permet de récupérer, sous forme de fragments, un message reçu sur la socket spécifiée.

Sur un exemple : inc.c

Le client envoie un nombre au serveur —>

<— Le serveur répond ce nombre incrémenté de un



La structure du client-serveur est symétrique ... cela peut donc être "quasiment" le même programme : on commence facile . . .

Squelette d'un client-serveur UDP I

Il y a beaucoup de corrections à apporter à ce "squelette" préliminaire" pour avoir une solution convenable : votre **analyse doit être critique**.

```
1   /** Fichier : inc.c (Communication Sockets/UDP)
2    *   Les deux processus distants s'envoient un nombre qu'ils
3    *   incrementent successivement : L'un compte en pair, l'autre en impair ...  */
4   #include <stdio.h>
5   #include <string.h>
6   #include <stdlib.h>
7   #include <unistd.h>
8   #include <errno.h>
9   #include <sys/types.h>
10  #include <sys/socket.h>
11  #include <netinet/in.h>
12  #include <netdb.h>
13  #include <arpa/inet.h>
14
15  int main(int argc, char *argv[]){
16    int looptime = 0; /* Numero de la boucle */
17    socklen_t ls = sizeof(struct sockaddr_in); /* Taille des adresses */
18
19    /*------ Caracterisation de la socket d'émission ----------*/
20    int sd0;            /* Descripteur  */
21    int ps0 = 5001;     /* Port         */
22    struct sockaddr_in adr0, *padr0 = &adr0; /* Adresse  */
23
24    /*------ Caracterisation de la socket distante -------*/
25    struct sockaddr_in adr1,*padr1 = &adr1;  /* Adresse du destinataire */
26    struct hostent *hp1;        /* Adresse IP de la machine distante */
27
28    /*------ Buffers pour Messages ----------------------------------*/
29    char msg_in[3] = "0";      /* Message recu de "0" a "99" */
30    char msg_out[3] = "0";     /* Message a envoyer "0" a "99" */
```

Squelette d'un client-serveur UDP II

```
31
32      /* 0) Verifications de base : Syntaxe d'appel ======*/
33      if (argc != 2){
34        fprintf(stderr,"Syntaxe d'appel : a.out nom_du_host_peer \n");
35        exit(2);
36      }
37      /* 1) Preparation de  la socket d'émission ================*/
38      /* a) Creation : Domaine AF_INET, type DGRAM, proto. par defaut*/
39      if ((sd0=socket(AF_INET, SOCK_DGRAM, 0)) == -1)
40        perror("[SOCK_DGRAM, AF_INET, 0]");
41      else
42        printf("socket [SOCK_DGRAM, AF_INET, 0] creee\n");
43      /* b) Preparation de l'adresse d'attachement */
44      adr0.sin_family      = AF_INET;
45      adr0.sin_addr.s_addr = htonl(INADDR_ANY);  /* Format reseau */
46      adr0.sin_port        = htons(ps0);  /* Format reseau */
47      /* c) Demande d'attachement de la socket */
48      if(bind(sd0,(struct sockaddr *)(padr0),ls) == -1) {
49        perror("Attachement de la socket impossible");
50        close(sd0);  /* Fermeture de la socket                */
51        exit(2);       /* Le processus se termine anormalement.*/
52      }
53      /* d) Recuperation de l'adresse effective d'attachement. */
54      getsockname(sd0,(struct sockaddr *)padr0,&ls);
55
56      /* 2) Concernant l'adresse de la  socket de destination ======*/
57      /* a) A partir du nom du destinataire */
58      hp1=gethostbyname(argv[1]);
59      if(hp1 == NULL){
60        fprintf(stderr,"machine %s inconnue\n",argv[1]);
61        exit(2);
62      }
63      else{ /* Recuperation de l'adresse IP depuis la struct hostent */
64        memcpy(&adr1.sin_addr.s_addr, hp1->h_addr, hp1->h_length);
```

Squelette d'un client-serveur UDP III

```
65        adr1.sin_family = AF_INET;
66        adr1.sin_port   = htons(ps0); /* Meme port que sd0 : why not ? */
67        fprintf(stdout,"machine %s --> %s \n", hp1->h_name, inet_ntoa(adr1.sin_addr));
68      }
69
70      /* 3) Boucle emission-reception : A PARTICULARISER selon que l'on
71         est le serveur ou le client ... */
72      for(;;) {
73        int i;
74        struct sockaddr_in adr2, *padr2 = &adr2; /* Inutilise pour l'instant */
75        /* a) Emission */
76        printf("\n------------------\n\n\nEnvoi(%d) ... ", looptime);
77        if (sendto(sd0,msg_out, sizeof(msg_out), 0, (struct sockaddr *)padr1, ls) >0)
78          printf("termine : valeur = %s !\n",msg_out);
79        else
80          printf("inacheve : %s !\n",msg_out);
81        /* b) Reception */
82        printf("Attente de reception ... ");
83        if (recvfrom(sd0,msg_in, sizeof(msg_in), 0, (struct sockaddr *)NULL, NULL) == -1)
84          printf("inachevee : %s !\n",msg_in);
85        else  {
86          printf("terminee : valeur = %s !\n",msg_in);
87          /* c) Traitement : La reception est bonne, on fait evoluer i */
88          i = atoi(msg_in);
89          i = (i+1)%100;
90          sprintf(msg_out,"%d",i);
91        }
92        sleep(1); looptime++;
93      }
94    }
```