Les sockets

B. Dupouy

Plan

Bibliographie

• "Internetworking with TCP/IP vol. III" par Comer/Stevens, *Prentice-Hall*
• "L'informatique répartie sous Unix" par Gabassi et Dupouy, *Eyrolles*

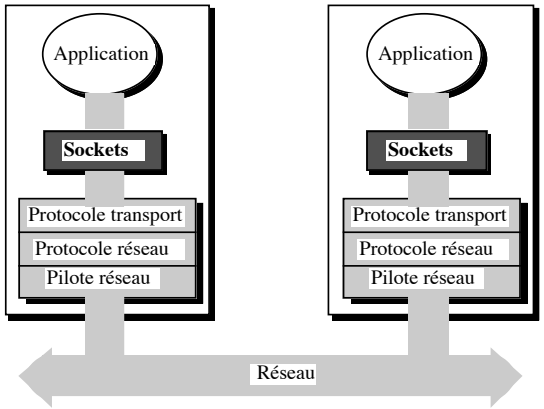
Plan

+ 1. Introduction
  2. Mode connecté
  3. Mode datagramme
  4. Fonctions associées
  5. API Java

Développés à l'université de Berkeley, les sockets sont une interface de programmation (ou API, Application Program Interface) entre les applications et les couches réseau.
Il s'agit d'une interface souple, d'assez bas niveau (couches 4 et 3).

Le terme socket désigne à la fois une bibliothèque d'interface avec le réseau et l'extrémité d'un canal de communication bidirectionnel via lequel un processus peut émettre et recevoir des données.
Sous UNIX, ce point de communication est représenté par un variable entière (`int`), manipulée comme un descripteur de fichier.

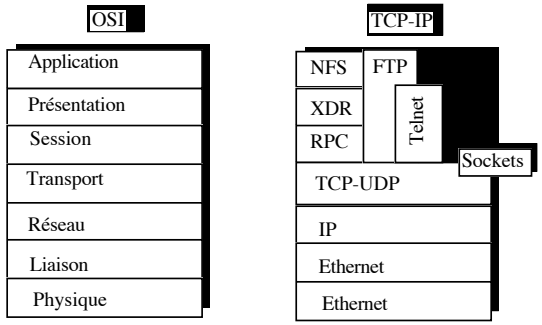
Modèle de la communication :



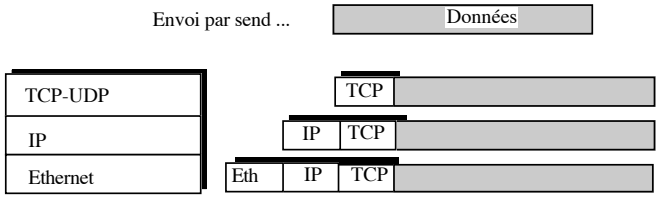
Pour chaque connexion, la bibliothèque des sockets gère un tampon en émission et un tampon en réception.

Rappels réseau

• Différences OSI et TCP-IP

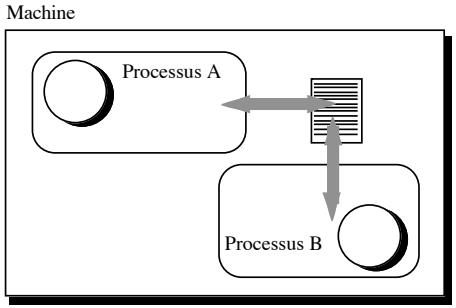


Structure des trames:

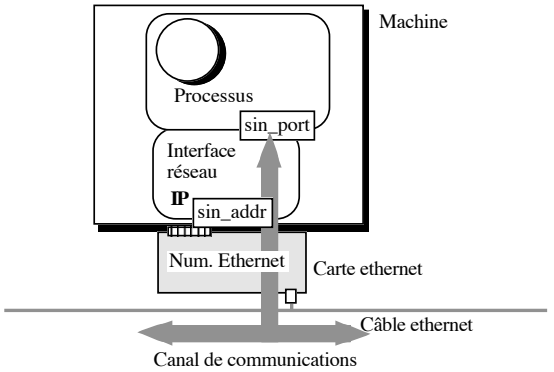


Les domaines AF_UNIX et AF_INET

Dans AF_UNIX, le socket est un simple fichier :



Dans AF_INET, le socket est un port vers la couche 4 du protocole Internet :



Plan

1. Introduction
+ 2. Mode connecté
3. Mode datagramme
4. Fonctions associées

Communications en mode connecté

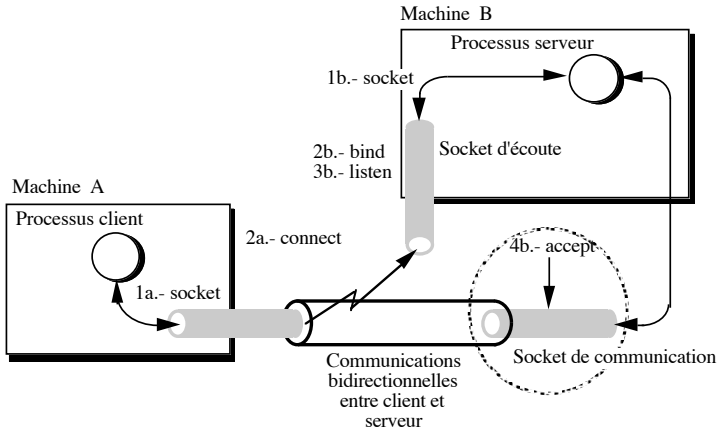
• Déroulement de la communication

L'appelant (ou client) :
- créée une socket ;
- se connecte au serveur en donnant l'adresse Internet du serveur et le numéro de port du service. Cette connexion attribue automatiquement un numéro de port au client ;
- lit ou écrit sur la socket ;
- ferme la socket.

L'appelé (ou serveur) :
- créée une socket ;
- associe une adresse socket (adresse Internet et numéro de port) au service : "binding" ;
- se met en attente des connexions entrantes ;
- pour chaque connexion entrante :
  - "accepte" la connexion (une nouvelle socket est créée);
  - lit ou écrit sur la nouvelle socket ;
  - ferme la nouvelle socket.

Mode connecté

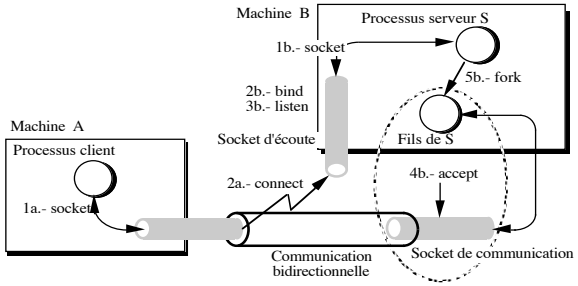
Schéma fonctionnel de la communication de base



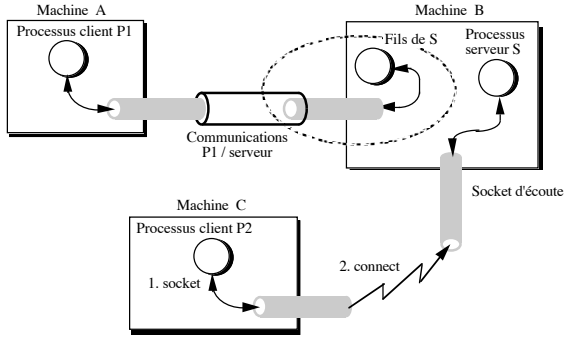
Mode connecté

Fonctionnement d'un serveur efficace

- Une communication...



- ... et deux communications



A la création de la socket, on précise dans quel *domaine* on travaille : UNIX (PF_UNIX), TCP/IP (PF_INET), ...

On précise également le protocole de communication :

- SOCK_DGRAM : envoi de messages sous forme de datagrammes. Il peut y avoir perte et déséquencement, citons UDP dans le domaine PF_INET ;
- SOCK_STREAM : envoi de flots d'octets. Le protocole de communication utilisé est en mode connecté, comme TCP dans le domaine PF_INET ;
- SOCK_RAW : il permet l'accès à des protocoles de plus bas niveau, comme IP dans le domaine PF_INET, ou d'implanter de nouveaux protocoles.

Si on utilise les sockets au-dessus d'UDP, la taille d'un échange est limitée à quelques kilooctets : de deux Ko à huit Ko selon les systèmes.

Le domaine PF_UNIX :
On peut utiliser l'interface socket pour faire communiquer deux processus se trouvant sur une **même** machine. On précise alors que l'on travaille dans le domaine PF_UNIX. Les appels sockets pour le domaine PF_UNIX sont les mêmes que pour le domaine PF_INET ; seules changent les structures associées aux adresses.

Format d'une adresse dans le domaine PF_INET :

```
struct sockaddr_in
     {
     short sin_family;          /*PF_INET*/
     u_short sin_port;          /*NUMERO DE PORT*/
     struct in_addr sin_addr;   /*ADRESSE MACHINE SUR 32 BITS*/
     char sin_zero[8];          /*inutilise*/
     };
struct in_addr
     {
     u_long s_addr;
     };
```

bind :
Si on donne la valeur zéro pour le numéro de port, le système attribue un numéro libre, qu'on peut retrouver grâce à la primitive *getsockname().* Cette opération, impérative du côté serveur, n'est pas nécessaire du côté client, en effet une adresse est automatiquement attribuée au moment de la connexion avec le serveur.

Création et nommage des sockets

• Creation

- Sock = socket (Domaine, Type, Protocole)

  Sockentrée dans la table des fichiers ouverts par le processus

Domaine	Type	Protocole
PF_UNIX	SOCK_DGRAM	
PF_INET	SOCK_DGRAM SOCK_STREAM SOCK_RAW	UDP TCP
Autres		

• Nommage

- bind (Sock, &Le_Sock, Taille)

  Sock numéro renvoyé par la primitive socket
  Le_Sock structure définissant le socket Sock
  Taille taille de cette structure

Domaine	Nom de la structure	Format

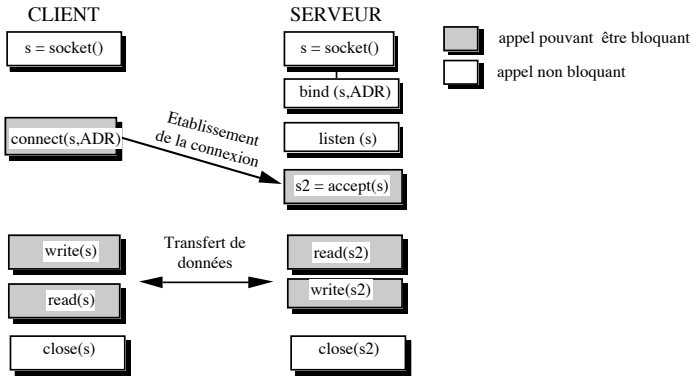
PF_UNIX	sockaddr_un
	définie dans :
	/usr/include/sys/un.h
		`struct sockaddr_un {`
		`  short sun_family;`
		`  char sun_path[14];`
		`};`

PF_INET	sockaddr_in
	definie dans :
	/usr/include/netinet/in.h
		`struct sockaddr_in {`
		`  short sin_family;`
		`  u_short sin_port;`
		`  struct in_addr sin_addr;`
		`  char sin_zero[8];`
		`};`

Autres		

Sockets

Communication en mode SOCK_STREAM.



Sur quel type d'événement se bloquent ces fonctions :

Côté client :

- `connect()` attend qu'un serveur effectue un `accept()` ;
- `write()` se bloque si le tampon d'émission est plein ;
- `read()` attend qu'un caractère au moins ai été reçu, à la suite d'une opération d'écriture effectuée par le serveur.

Côté serveur:

- `accept()` attend qu'un client effectue un `connect()` ;
- `read()` attend un caractère, émis par un client ;
- `write()` se bloque si le tampon d'émission est plein.

`accept()` retourne un nouveau descripteur de socket, qui sera utilisé pour l'échange de données avec le client. Il est ainsi possible au serveur de créer un processus fils qui se servira du nouveau descripteur créé, pendant que le processus père reprendra l'écoute (`accept()`) sur l'ancien descripteur.

Sockets

Communications en mode connecté

Mise en place du canal de communication

Côté client

- connect (Sock_Cli, &Le_Serveur, Taille_Serv)

  connexion sur le socket du serveur

Sock_Cli	numéro renvoyé par la fonction socket chez le client
Le_Serveur	structure définissant le nom du socket du serveur
Taille_Serv	taille de cette structure

Côté serveur

- listen (Sock_Serv, Nb_Clients)

  définition de la taille de la file d'attente sur un socket :g

Sock_Serv	numéro renvoyé par socket chez le serveur
Nb_Clients	taille de la file d'attente sur Sock_Serv

- **Sock_Comm** = accept (Sock_Serv, &Le_Client, &Taille_Cli)

  attente d'une demande de connexion (par connect) :

Sock_Comm	numéro de socket renvoyé par accept dès qu'il reçoit une demande de connexion
Sock_Serv	numéro renvoyé par socket chez le serveur
Le_Client	structure décrivant le nom du socket du client appelant
Taille_Cli	taille de cette structure

Sockets

Les appels `read()` et `write()` sont utilisables comme pour un descripteur de fichier.

Les appels `send()` et `recv()` ont un argument supplémentaire qui permet de mieux contrôler la communication, à utiliser si on a des problèmes avec `read` et/ou `write`.
Les appels `sendto()` et `recvfrom()` sont utilisés pour les sockets de type SOCK_DGRAM. Deux paramètres permettent de préciser l'adresse du site distant dans le cas de `sendto()` et de récupérer cette adresse dans le cas de `recvfrom()`.

Sockets

Mode connecté

• Emissions/Réceptions

- avec les fonctions classiques

- read/write (Sock, Mess, T_Mess)

Sock	numéro renvoyé par socket
Mess	adresse du début du message transmis
T_Mess	sa taille (en octets)

- avec des fonctions spécifiques

- send/recv (Sock, Mess, T_Mess, Option)

Sock	numéro renvoyé par socket
Mess	adresse du début du message transmis
T_Mess	sa taille (en octets)
Option	0, MSG_OOB (send/recv) ou MSG_PEEK (recv seulement)

• Remarques
une façon élégante de récupérer un caractère envoyé “OOB”, est de traiter le signal SIG_URG envoyé par le noyau lors de la réception de ce caractère

Plan

1. Introduction
2. Mode connecté
+ 3. Mode datagramme
4. Fonctions associées

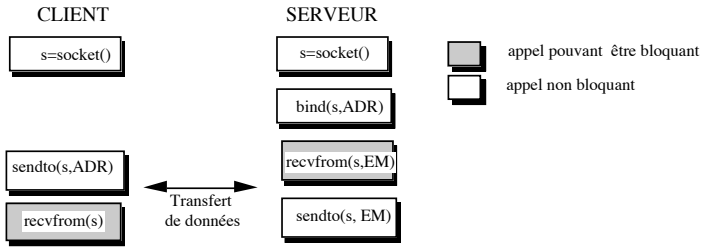
Communications en mode datagramme

• Déroulement de la communication

Client (appelant) :
- créée une socket ;
- lit ou écrit sur la socket ;

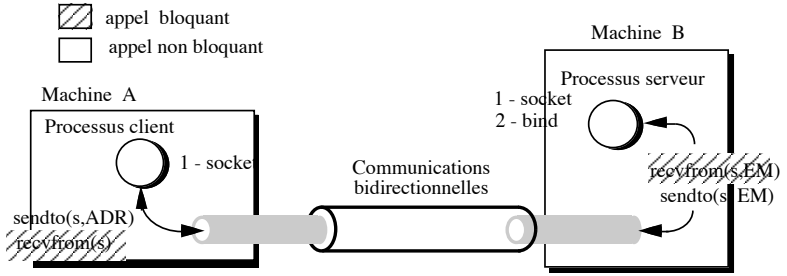
Serveur (appelé) :
- créée une socket ;
- associe une adresse socket au service : "binding" ;
- lit ou écrit sur la socket.

Communication en mode SOCK_DGRAM.



Communication de type UDP

• Schéma fonctionnel :



Communications en mode datagramme

Emission

- sendto (Sock, Mess, T_Mess, 0, &Le_Distant, Taille_Dist)

Reception

- recvfrom (Sock, Mess, T_Mess, Drapeaux, &Le_Distant, &Taille_Dist)

ici : Mess, T_Mess, Drapeaux, &Le_Distant, Taille_Dist
sont des paramétres mis à jour par **l'émetteur**.

Sock	numéro renvoyé par la fonction socket
Mess	adresse du début du message transmis ou reçu
T_Mess	sa taille (en octets)
Le_Distant	structure définissant le nom du socket distant
Taille_Dist	taille de cette structure
Drapeaux	indications sur la transmission

Remarques

- recvfrom permet de recevoir depuis n'importe quel émetteur, si bloquant
  la structure socket appelant n'est pas mise à jour
- sendto possible seulement vers un socket sur lequel le créateur a fait bind.

Exemple : client datagramme

```
struct sockaddr_in s_serveur;/* socket serveur */
struct hostent *m_cible;     /* machine cible */

char MESSAGE [] = "Coucou";

/*
 creation de la socket
 */
sock = socket(AF_INET, SOCK_DGRAM, 0);



/*
  Construction de l'adresse du serveur
 */
m_cible = gethostbyname("Machine_Serveur");
memcopy (&s_serveur.sin_addr, m_cible->h_addr,
          m_cible->h_length);
s_serveur.sin_family = AF_INET;
s_serveur.sin_port   = htons (7545);



/*
  Emission du message
 */
sendto (sock, MESSAGE, strlen(MESSAGE), 0,
         (struct sockaddr *)&s_serveur,
          sizeof(s_serveur)   )
```

Exemple : serveur datagramme

```
/*
 initilisation  la socket avec un numéro donné
 */
    Sock_Comm  = socket (AF_INET, SOCK_DGRAM, 0)
    ···
    Decrit_sock.sin_family        = AF_INET ;
    Decrit_sock.sin_port          = htons (7545);
    Decrit_sock.sin_addr.s_addr   = INADDR_ANY;
    ···
    bind (Sock_Comm, &(Decrit_sock), Taille)

/*
  Attente d'un message
 */
    recvfrom (socket, Tab, ···, ···, &Decrit_dist, ···);
    ···
```

Plan

1. Introduction
2. Mode connecté
3. Mode datagramme
+ 4. Fonctions associées

La fonction `close()` gère la fermeture de la connexion au niveau système (fichiers).

```
int close(sint ock)
        int sock; /*descripteur de socket*/
```

Le noyau essaie d'envoyer les données non encore émises avant de sortir du `close()`.

La fonction `shutdown()` permet de contrôler la fermeture de la connexion au niveau réseau.

int shutdown(int sock, int controle)

L'argument `controle` peut prendre les valeurs suivantes :

0 : on ne peut plus recevoir de données sur la socket ;
1 : on ne peut plus envoyer de données sur la socket ;
2 : on ne peut plus émettre ni recevoir de données sur la socket.

Terminaison

Fin de communication

shutdown (Sock, Direction)

Sock	numéro renvoyé par socket
Direction	0 -> fin d'émission 1 -> fin de réception 2 -> plus d'émissions ni de réceptions

Fermeture de socket

close (Sock)

Sock	numéro renvoyé par socket

Multiplexage des entrées/sorties : `select`

```
Nb_rep = select (Nb_Fic,Masque_R,Masque_W,Masque_E,Delai)

   Nb_rep     nombre de ressources ayant répondu
   Nb_Fic     nombre de ressources sur lesquels on attend
   Masque_R   les ressources sur lesquels on attend une
          ecriture depuis l'extérieur
   Masque_W   les ressources sur lesquels on attend une
          lecture
   Masque_E   les ressources sur lesquels on attend un
              événement
   Delai      temporisation
```

Remarques :

- select est une primitive Unix classique,
- select n'est absolument PAS spécifique aux sockets,
- sur Unix BSD, un socket est vu comme un fichier, alors on utilise souvent select avec les sockets (cf. inetd)
- pour positionner les masques, utiliser :

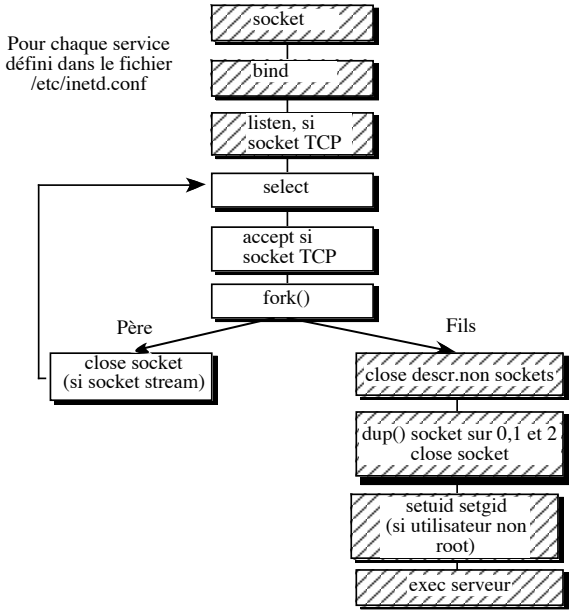
```
FD_ZERO (&Masque_R);
FD_SET (0, &Masque_R);
FD_SET (sock, &Masque_R);

select (FD_SETSIZE, &Masque_R, ..., ... , ...);
```

Pour faire démarrer un serveur par `inetd`, il faut effectuer les opérations suivantes :

- retirer du programme les primitives : socket(), bind(), listen(), accept()
- appliquer les opérations de lecture et d'écriture aux descripteurs 0 et 1 (à la place du descripteur de socket)
- sortir du programme par exit() lorsque le service est terminé ;
- configuer le service dans les fichiers /etc/services et /etc/inetd.conf ;
- relancer le démon inetd.

Le serveur `inetd`

- `inetd` trouve la liste les serveurs à lancer dans le fichier `/etc/inetd.conf`
- Il attend sur un `select()` :



Bibliotheque d'utilitaires (1)

Informations sur les machines

- gethostbyaddr (Adresse, Taille, Type),
  char *Adresse;
  int Taille, Type;

- gethostbyname (Nom),
  char *Nom;

- gethostent (),

Remarques :

- ces fonctions renvoient un pointeur sur un objet du type struct hostent,
- cette structure est initialisée d'après /etc/hosts.

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

Quelques protocoles et informations sur les machines

Exemple de quelques protocoles

```
#define IPPROTO_IP      0       /* dummy for IP */
#define IPPROTO_ICMP    1       /*control mess proto */
#define IPPROTO_TCP     6       /* tcp */
#define IPPROTO_RAW     255 /* raw IP packet */
#define IPPROTO_MAX     256
```

Extrait de /etc/services

```
ftp-data20/tcp
ftp     21/tcp
telnet      23/tcp
smtp        25/tcp      mail
time        37/tcp      timserver
name        42/udp      nameserver
sunrpc      111/udp
sunrpc      111/tcp
#
# UNIX specific services
#
# these are NOT officially assigned
#
exec        512/tcp
login       513/tcp
shell       514/tcp     cmd     # no passwords used
printer 515/tcp     spooler     # line printer spooler
who     513/udp     whod
#
```

Extrait de dmesg

```
Ethernet address = 8:0:20:9:ac:55
le0 at SBus slot 0 0xc00000 pri 5
```

Bibliotheque d'utilitaires (2)

Informations sur les services

- getservbyport(Port, Proto)
  int Port; char *Proto;
- getservbyname(Nom, Proto)
  char *Nom, *Proto;
- getservent (),

Remarques :

- ces fonctions renvoient un pointeur sur un objet du type struct servent,
- cette structure est initialisée d'après /etc/services.

```
struct    servent {
      char *s_name;      /* official name of service */
      char **s_aliases;  /* alias list */
      int  s_port;       /* port service resides at */
      char *s_proto;     /* protocol to use */
};
```

Attribution des numéros de ports

Un certain nombre de numéros sont réservés (*well known services*) : ceux situés dans la plage 1 à 1023 utilisables par les services standards TCP/IP.

Quelques services standards

Les ports de numéro < 1024 réservés à root, les numéros > 5000 sont laissés pour les serveurs quelconques.

```
#define IPPORT_ECHO         7
#define IPPORT_DISCARD      9
#define IPPORT_SYSTAT       11
#define IPPORT_DAYTIME      13
#define IPPORT_NETSTAT      15
#define IPPORT_FTP          21
#define IPPORT_TELNET       23
#define IPPORT_SMTP         25
#define IPPORT_TIMESERVER   37
#define IPPORT_NAMESERVER   42
#define IPPORT_WHOIS    43
#define IPPORT_MTP          57

#define IPPORT_TFTP         69
#define IPPORT_FINGER       79

/*
 * UNIX TCP sockets
 */
#define IPPORT_LOGINSERVER  513
#define IPPORT_CMDSERVER514

/*
 * UNIX UDP sockets
 */
#define IPPORT_BIFFUDP      512
#define IPPORT_WHOSERVER513
#define IPPORT_ROUTESERVER  520 /* 520+1 also used */
```

Le numéro de port est codé sur deux octets et l'adresse Internet sur 4 octets. Il importe que le système client et le système serveur adoptent la même convention concernant l'ordre des octets transmis. Pour cela, des routines permettent de transformer les entiers courts ou longs en valeurs codées selon un format standard.

Les deux premières routines sont utilisées pour passer du format local au format réseau, et les deux dernières pour réaliser l'opération inverse.

```
u_long htonl (u_long hostlong)      /* host to network long */

u_short htons (u_short hostshort)   /* host to network short */

u_long ntohl (u_long hostlong)      /* network to host long */

u_short ntohs( u_short hostshort)   /* network to host short */
```

Opérations de conversion sur les adresses :
Deux routines permettent de passer d'une adresse Internet sous forme de caractères (telle qu'on la trouve dans le fichier `/etc/hosts`) à une adresse sous forme entière de quatre octets, et inversement.

```
u_long inet_addr(char *adresse)
char *inet_ntoa(struct in_addr inadresse)
```


Bibliotheque d'utilitaires (3)

Conversions de données

- net_long = htonl (host_long),
  u_long net_long, host_long;
- net_short = htons (host_short);
  u_short net_short, host_short;
- host_long = ntohl (net_long),
  u_long host_long, net_long;
- host_short = ntohs (net_short),
  u_short host_short, net_short;

Remarques :

- ces fonctions convertissent des informations sur 16 ou 32 bits en les passant du format réseau au format local et vice-versa,
- à utiliser dès que l'on emploie les fonctions du type gethost ou getserv,
- sur Sun 3, ce sont des macros NULL, voir <netinet/in.h>,

`getsockname` donne le numéro du port local, celui qui a été choisi par bind, ou attribué par le système, si on n'a pas fait bind (utile côté client ou pour le fils d'un serveur, par exemple)

`getpeername` donne le numéro du port distant, celui du partenaire avec qui on dialogue.

Pour construire l'adresse Internet à partir du nom d'une machine, on peut utiliser la fonction `gethostbyname()` :

```
struct hostent *gesthostbyname (char *hostname)
```

La fonction retourne un pointeur sur une structure `hostent` :

```
struct hostent {
     char *h_name;          /*NOM DE LA MACHINE*/
     char **h_aliases;      /*liste d'ALIAS*/
     int h_addrtype;        /*PF_INET*/
     int h_lenght;          /*4 octets*/
     char **h_addr_list; /*liste d'adresses Internet*/
};
```

Cette fonction trouve l'adresse dans le fichier `/etc/hosts`, ou bien utilise les services d'un serveur de noms. Un serveur de noms est un démon UNIX qui possède ou sait trouver le nom et l'adresse de toutes les machines du réseau. Les serveurs de noms les plus répandus sont le NIS (Network Information Service) et le BIND (Berkeley Internet Name Server), encore appelé DNS (Domain Name Server).

Exemple :

```
/* Affichage des adresses des machines dont les noms sont passés en
argument.*/
void main(int argc, char * argv [])
{
struct hostent *hostp;
long int i;

for (i = 1; i<argc; i++)
     {if ((hostp = (structhostent*)gethostbyname (argv[i])) == NULL)
          printf("Site %s inconnu\n", argv[i]);
      else
      printf("Site %s adresse : %s \n", hostp->h_name,
            inet_ntoa(*hostp->h_addr_list)  );
    }
}
```


Bibliotheque d'utilitaires (4)

Informations sur les ports

- getsockname (sock, &sa, &len) : renvoie le numéro de port local
- getpeername (sock, &sa, &len) : renvoie le numéro de port distant

Informations sur les machines

- gethostname (host, sizeof (host)),
- gethostbyname `(char *hostname)`
- gethostbyaddr `()`

Gestion des signaux

Trois signaux peuvent être envoyés pour les sockets :

`SIGIO`	Prévient qu'une socket est prête pour une entrée-sortie asynchrone. Le signal est envoyé au processus, ou au groupe de processus
`SIGURG`	Signifie que des données express sont arrivées sur une socket. Il est envoyé au processus ou au groupe de processus
`SIGPIPE`	Indique qu'il n'est plus possible d'écrire sur une socket. Il est envoyé au processus associé à la socket.

Gestion des erreurs

Les erreurs sont signalées de la manière suivante :

`read()`	retourne `zéro` si le processus distant est détruit ou `-1` si la liaison réseau est coupée ;
`write()`	provoque l'envoi du signal SIGPIPE si le processus distant est détruit ou si la liaison réseau est coupée.

Il se peut que le programme client s'arrête de manière anormale. Le serveur peut contrôler la présence du client.de la manière suivante : il effectue périodiquement une écriture d'un octet sur une socket de contrôle. Ainsi, une erreur SIGPIPE sera détectée si le client n'existe plus, et le serveur s'arrêtera.

Il est possible d'émettre des données en diffusion pour les sockets en mode datagramme. Il faut positionner l'option SO_BROADCAST grâce à la primitive `setsockopt()`.

Traitements spéciaux

• setsockopt

```
setsockopt (sock, .., ...; SOL_SOCKET, SO_REUSEADDR,....)
```

• Quelques options intéressantes :

- permet la communication de groupe (multicast)

`SO_KEEPALIVE`	transmission périodique de messages de contrôle sur une socket en mode connecté. Si un des deux processus ne répond pas, la connexion est considérée comme rompue et l'erreur ETIMEDOUT est positionnée ;
`SO_RCVBUF, SO_SNDBUF`	précisent la taille des mémoires tampons TCP. Les performances peuvent être améliorées, en prenant des tampons de taille plus grande ;
`SO_REUSEADDR`	permet de réutiliser un numéro de port.

Opérations non bloquantes

La lecture sur la socket est bloquante tant qu'il n'y a rien dans le tampon réception (si la socket est déclarée non bloquante, la lecture retourne une erreur). Dès leur réception, les données sont envoyées à l'application (même si le nombre d'octets reçus est inférieur au nombre d'octets demandés par l'application).

L'écriture sur la socket est bloquante uniquement si le tampon émission est plein (si la socket a été déclarée non bloquante, l'écriture retourne une erreur). Les octets rangés dans le tampon sont émis sur le réseau tant qu'il n'est pas plein.

Les blocages peuvent être supprimés si on déclare les sockets non bloquantes grâce à la primitive `ioctl()` (flag FIOSNBIO) ou la primitive `fcntl()` (flag FNDELAY si le système est BSD et flag O_NDELAY si le système est System V).

Les fonctionnements sont alors les suivants :

`accept`	revient immédiatement si aucune demande de connexion n'est en cours avec l'erreur EWOULDBLOCK ;
`connect`	revient immédiatement si la connexion n'est pas possible, avec l'erreur EINPROGRESS ;

`recv`
`read`
`recvfrom`
	retourne -1 (FIOSNBIO) ou 0 (O_NDELAY) si aucune donnée n'est à lire ;

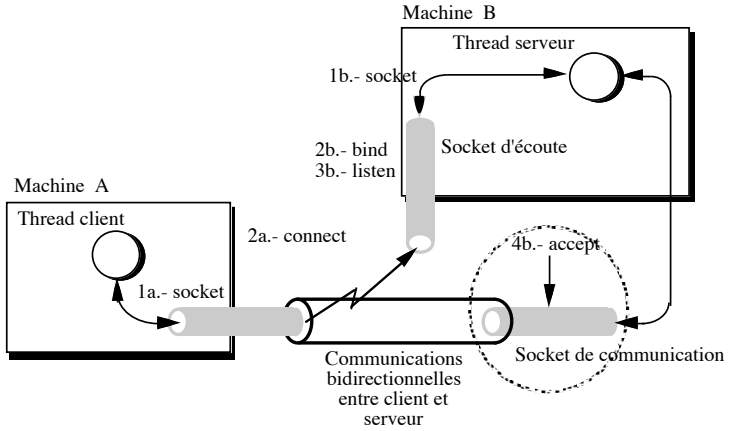
`send`
`write`
`sendto`
	retourne -1 (FIOSNBIO) ou 0 (O_NDELAY) si l'écriture est impossible(buffer d'émission plein).

On teste périodiquement la terminaison et on effectue un autre traitement tant que l'opération n'est pas terminée.

Plan

Cient-serveur TCP

• Rappel du schéma fonctionnel :



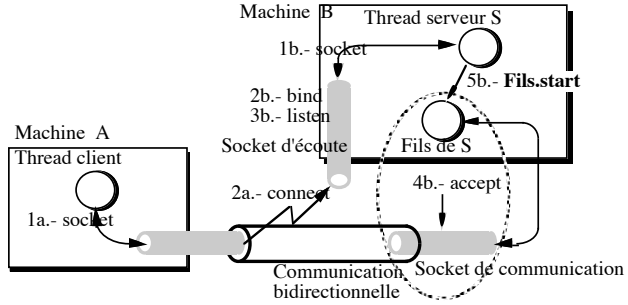
• Squelette du code java pour le serveur :

```
// creation d'un port d'ecoute
     ServerSocket  SockEcou =  new ServerSocket(Port);
     System.out.println  ("Serveur ecoute sur " + Port );
...
// Le serveur se bloque sur le port d'ecoute
// en sortie du accept, il recupere un port de
communication
       Socket SockComm = SockEcou.accept();
```

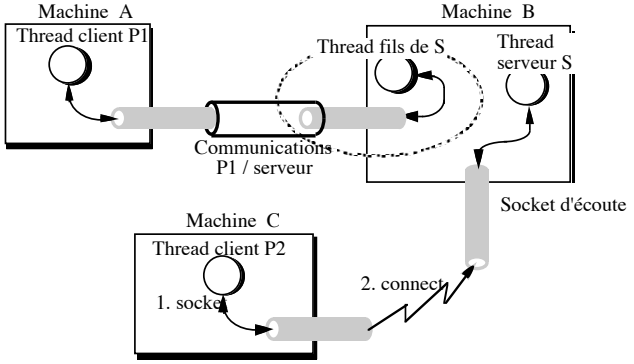
Rappel : Serveur TCP multithread

• Schéma fonctionnel :

Une communication...



... et deux communications



Serveur TCP Multithread en Java

• Canevas du code du serveur : attente sur le port d'écoute, gestion des communications avec les clients par des threads,

```
// creation d'un port d'ecoute
    ServerSocket  SockEcou =  new ServerSocket(Port);
    ...
// Le serveur se bloque sur le port d'ecoute
    Socket SockComm = SockEcou.accept();
    ...
// Un thread va gerer la comm. avec le client
    new Thread_Dialogue(SockComm).start();
    ...
```

• Canevas du code du thread de dialogue avec le client

```
class Thread_Dialogue extends Thread
{
    Socket Mon_Socket;
    public Thread_Dialogue (Socket Un_Socket)
      {
        this.Mon_Socket = Un_Socket;
      }
    public void run()
      {
       ··· = ··· ( Mon_Socket.getOutputStream() );
      }
}
```

Client TCP
En Java

• Client qui communique avec le serveur TCP précédent, :

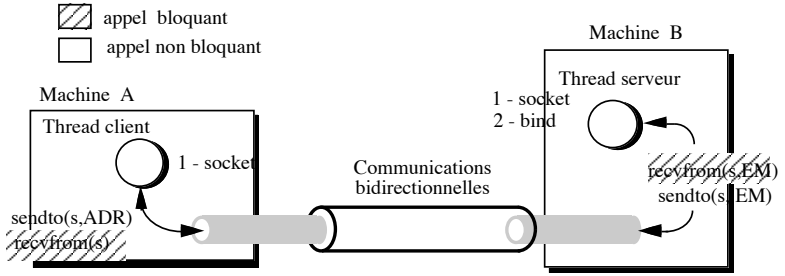
- il envoie au serveur une chaine de caracteres saisie au clavier.

- il attend ensuite une reponse qu'il affiche a l'ecran.

```
...
Socket    Mon_Socket;
String    Tampon;

Mon_Socket = new Socket("machine", Port);

// Ecrire sur le socket :
PrintWriter Sortie_TCP = new
           PrintWriter(Mon_Socket.getOutputStream());

// Lire sur le socket :
BufferedReader Entree_TCP = new BufferedReader (new
          InputStreamReader(Mon_Socket.getInputStream()));

// Entrer les donnees au clavier :
BufferedReader Entree_Clav = new BufferedReader(new

InputStreamReader(System.in));

// Arret de la communication par <CTRL D>
while(( Tampon = Entree_Clav.readLine() ) != null )
    {
     Sortie_TCP.println (Tampon);
     Sortie_TCP.flush ();
     System.out.println (Entree_TCP.readLine());
    }
```

Rappel : communication de type UDP

• Schéma fonctionnel :



Client en mode data gramme : exemple Java

• Squelette du code d'un client en mode UDP :

```
DatagramSocket        Socket_UDP;
DatagramPacket        Message;
byte[]                Tampon;
InetAddress           Adresse_IP;


// Initialisations
Tampon      = new byte[256];
Socket_UDP = new DatagramSocket();
Adresse_IP  = InetAddress.getByName( Nom_de_la_machine
);

// Emission
Message = new DatagramPacket
             (Tampon, Tampon.length, Adresse_IP,
Port);
Socket_UDP.send(Message);

// Reception
Message = new DatagramPacket(Tampon, Tampon.length);
Socket_UDP.receive(Message);

String Mess_Recu = new String(Message.getData());
System.out.println(" Recu : " + received);
```

Serveur en mode data gramme : exemple Java

• Squelette du code d'un serveur en mode UDP :

```
DatagramSocket      Socket_UDP;
DatagramPacket      Message = null;
byte[]              Tampon;
InetAddress         Adresse_IP;
int                 Port = 0;

// Initialisations
Socket_UDP = new DatagramSocket(Mon_Port);
Tampon     = new byte[256];
Message    = new DatagramPacket(Tampon,
Tampon.length);

// Attendre un message
Socket_UDP.receive(Message);

// Envoyer la reponse vers le client
Adresse_IP = Message.getAddress();
Port       = Message.getPort();
Message    = new DatagramPacket (Tampon,
Tampon.length,
                                 Adresse_IP, Port);
Socket_UDP.send(Message);
```