Ch03 - Architectures Internet - Les sockets

- **Origine, présentation et conception des sockets**
- **Les types de sockets**
- **Exemple de serveur et de client UDP**
- **Modèles mémoires différents et échanges de données**

Origine, présentation et conception des sockets

Les sockets sont apparus avec BSD 4.2 et se sont stabilisés avec BSD 4.3. Leur utilité a fait qu'ils sont maintenant disponible sur tous les systèmes unix et sur de nombreux autres systèmes.

Ce sont en effet, le support privilégié de la plupart des communications entre process sur l'Internet.

C'est une interface de communication générale permettant de créer des applications distribuées.

Elle crèe un canal unique bidirectionnel (full duplex), contrairement aux "pipes" unix qui ne fonctionnent que dans un seul sens.

Les sockets couvrent deux domaines et sont de deux types différents.

Domaines :

- domaine AF_UNIX : permet à des process de la même machine de communiquer entre eux ;
- domaine AF_INET : permet à des process situés sur deux machines différentes d'un même internet de communiquer. Ceci suppose que les deux machines communiquent par la suite de protocoles réseau TCP/IP.

Un domaine défini donc l'ensemble des autres sockets avec lesquels un process donné pourra communiquer.

Types :

le type du socket défini les propriétés de la communication. Les deux types principaux sont :

- le type datagramme, permettant l'échange de messages complets et structurés, sans négociation préalable (communication sans connexion) ;
- le type connecté, permettant l'échange de flux de données sur un circuit virtuel préalablement établi.

Objectifs de conception de l'interface socket

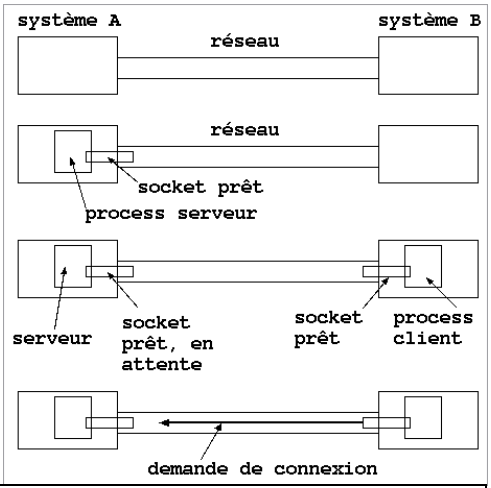
Les sockets ont été conçus pour permettre à des process de communiquer à travers un réseau tout en respectant :

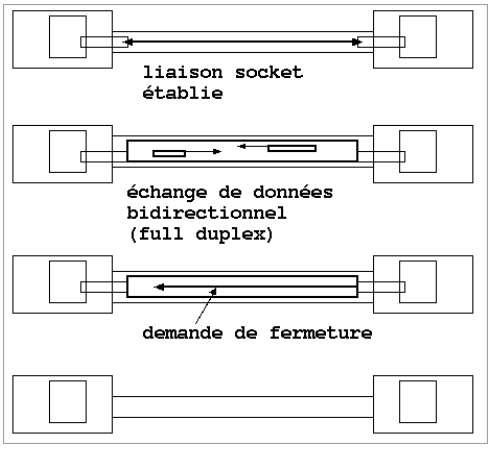
- **transparence :** que la communication soit identique si les process sont sur la même machine ou non ;
- **efficacité :** implantés comme bibliothèque système incluse dans le noyau, ils ont accès direct au driver de l'interface réseau évitant ainsi une double recopie du buffer de données et ils travaillent comme un appel système depuis le process utilisateur, dans le contexte de celui-ci, sans obliger à un changement de contexte comme cela aurait été le cas s'ils avaient été implantés sous forme de process indépendant ;
- **compatibilité :** sockets vus des process comme descripteurs de fichiers, => permet opérations habituelles de redirection E/S (dup()) ;

Mode client-serveur et établissement d'une communication

La caractéristique la plus fréquente des connexions en mode "client-serveur" est d'être asymétrique :

- **d'abord**, un process **serveur** se prépare ; il se mets en écoute sur un **port** de communication. Ce n'est encore qu'une "moitié" de la liaison. C'est le robinet sur lequel le tuyau n'a pas encore été branché.
- **ensuite**, un process **client** va envoyer à ce port serveur déja existant une demande de connexion.
- **alors**, la connexion est établie et les données peuvent circuler dans les deux sens.





Les types de sockets

Les deux types principaux de sockets correspondent aux protocoles **sans connexion** et aux protocoles **orientés connexion** :

- protocoles **sans connexion** : ("connectionless") il y a échange de messages autonomes, complets et structurés ; chaque envoi doit contenir le destinataire ; ceci peut se comparer à l'envoi de lettres par la poste.
- protocoles **orientés connexion** : ("connection oriented") après établissement de la connexion (circuit virtuel), tous les envois sont implicitement pour le même destinatire ; ceci peut se comparer au téléphone.

1. **sockets SOCK_DGRAM** : envoi de datagrammes (messages), mode non connecté ; protocole **UDP** au-dessus de **IP** :
   - messages de taille bornée,
   - préserve les frontières de message,
   - pas de garantie de remise ("best effort"),
   - pas de garantie de remise dans le même ordre que l'envoi ;
2. **sockets SOCK_STREAM** : flot de données, mode connecté, protocole **TCP** au-dessus de **IP** :
   - transfert fiab le (pas de perte ni d'altération de données),
   - données délivrées dans l'ordre d'envoi,
   - pas de duplication,
   - supporte la notion de messages urgents ("out-of-band") pouvant être lus avant les données "normales" ;

3. **sockets SOCK_RAW** : permet d'accéder au protocole de plus bas niveau (IP) pour construire d'autres protocoles. Réservé au mode superviseur.
4. **sockets SOCK_SEQPACKET** : paquets avec préservation des frontières, délivrés de façon fiab le et en séquence (dans l'ordre). Cité pour la complétude, était utilisé dans le protocole Decnet.

Caractéristiques d'un socket

Vu "de l'intérieur" du process utilisateur, un socket est un descripteur de fichiers. On peut ainsi rediriger vers un socket les E/S standard d'un programme développé dans un cadre local.

Une autre conséquence est que les sockets sont hérités par le fils lors d'un fork.

Un socket est créé par la primitive **sd = socket()**. La valeur de retour étant le descripteur sur lequel on va faire les opérations de lecture et d'écriture.

La différence avec un fichier est qu'il est ensuite possible (et nécessaire !) **d'attacher** au socket une adresse du domaine auquel il appartient. Ceci se fait avec la primitive **bind()**. Cette adresse est communément appelée **numéro de port**.

Création et attachement d'un socket

```
NAME
       socket - create an endpoint for communication
SYNOPSIS
       #include <sys/types.h>
       #include <sys/socket.h>

       int socket(int domain, int type, int protocol);
```

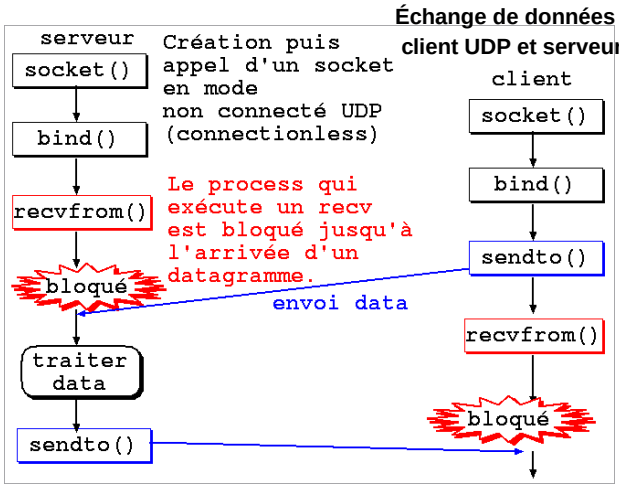
Le troisième paramètre est souvent donné à 0, car souvent il n'y a qu'un seul type de protocole disponible (par exemple pour un SOCK_DGRAM dans le domaine AF_INET, il n'y a que UDP, et pour SOCK_STREAM que TCP).


```
Exemple:   sd = socket (AF_INET, SOCK_DGRAM, 0);
     ou:   sd = socket (AF_INET, SOCK_STREAM, 0);
```

Attachement (bind)

```
$ man 2 bind
NAME
     bind - bind a name to a socket
SYNOPSIS
     #include <sys/types.h>
     #include <sys/socket.h>
     int bind(int sockfd, struct sockaddr *my_addr,
              socklen_t addrlen);
(addrlen est la longueur de la structure my_addr)
```


Échange de données entre client UDP et serveur UDP




```
$ man 2 send
$ man 2 recv

SEND(2)          Linux Programmer's Manual          SEND(2)
NAME
     send, sendto, sendmsg - send a message from a socket
SYNOPSIS
   #include <sys/types.h>
   #include <sys/socket.h>
   ssize_t send(int s, const void *msg, size_t len,
                int flags);
   ssize_t  sendto(int s, const void *msg, size_t len,
     int flags, const struct sockaddr *to, socklen_t tolen);
   ssize_t sendmsg(int s, const struct msghdr *msg,
                   int flags);
```

```
DESCRIPTION
    Send, sendto, and sendmsg are used to transmit a message
    to another socket. Send may be used only when the socket
    is in a connected state, while sendto and sendmsg may be
    used at any time.

    The address of the target is given by to with tolen
    specifying  its size. The length of the message is given
    by len. If the message is too long to pass atomically
    through the underlying protocol, the  error EMSGSIZE is
    returned, and the message is not transmitted.
```

L'émetteur peut être soit le client, soit le serveur. La seule contrainte est qu'ils doivent être programmés pour qu'il y ait autant de recv() exécutés par l'un que de send() exécutés par l'autre.


```
Exemples :
        sendto(sa, bufa, nbufa, flag, &to, lento)
        recvfrom( sb, bufb, nbufb, flag, &from, lenfrom)
ou:
        send(sa, bufa, nbufa, flag)
        recv(sb, bufb, nbufb, flag)
```

Remarque : le "man" précide que "Send may be used only when the socket is in a connected state". Or, on est ici dans un contexte UDP, c'est-à-dire sans connexion. Qu'en est-il ?

En fait il s'agit d'une facilité de programmation : il est possible, afin d'éviter de remettre dans chaque appel "sendto()" l'adresse du destinataire, si c'est toujours le même, de stocker cette adresse dans la structure socket. On peut alors utiliser simplement send() vers ce

destinataire implicite. Il s'agit d'une **pseudo-**connexion dont l'autre entité n'a pas connaissance. C'est juste une facilité de programmation locale.

On peut ainsi faire un send() vers un process qui lui-même est uniquement programmé avec des recvfrom(), et inversement.

Flags :

- **MSG_OOB** send or recv Out-of-Band data
- **MSG_PEEK** permet de "regarder" les data à lire sans que le système ne les enlève du buffer d'arrivée.

Exemple de serveur et de client UDP

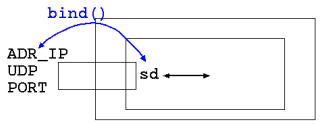
```
        Un serveur UDP  et  un client UDP
(1)  le serveur se met en attente sur un port UDP,
(2)  le client crèe son socket local,
(3)  le client envoie un message sur le host
     et le port UDP du serveur,
(4)  à réception du message, le serveur récupère
     l'adresse (host,port) de l'émetteur et traite
     le message, puis envoie une réponse.
```



```c
/* udpser.c  */                          /* serveur UDP */
/* UTC UV SR03 - (c) Michel.Vayssade@utc.fr
linux:   gcc -o udpser udpser.c
solaris: gcc -o udpser udpser.c -lsocket -lnsl */

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

#define PORT 4677 /* port du serveur */
int sd,i,n;
int *fromlen;
struct sockaddr_in mon_s;
struct sockaddr_in son_s;

typedef struct messages {
  char mtext[50];
} message;
message *mess;

void affiche_errno() { printf("\nerrno=%d",errno); }
```

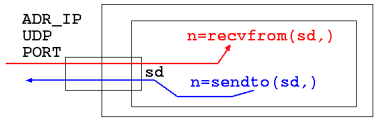
21

```c
main(){                                          /* serveur UDP */

sd=socket(PF_INET,SOCK_DGRAM ,0);

if(sd==-1)
   perror("udpser:err socket"),affiche_errno(),exit(1);

bzero(&mon_s,sizeof(mon_s));

/* accept input from any interface */
mon_s.sin_addr.s_addr = INADDR_ANY;
mon_s.sin_family = PF_INET; /* sur IP */
mon_s.sin_port = htons(PORT); /* port du serveur */

n=bind(sd,(struct sockaddr*)&mon_s, sizeof(mon_s));

if(n==-1) perror("udpser:err bind"),affiche_errno(),exit(1);
mess=(message*)malloc(sizeof(message));
```



22

```c
                                        /* serveur UDP */
fromlen=(int*)malloc(sizeof(int));
*fromlen=sizeof(son_s);

n=recvfrom(sd,mess,sizeof(message),0,
   (struct sockaddr*)&son_s,fromlen);

if (n==-1)
    perror("udpser:err recvfrom"),affiche_errno(),exit(1);
printf("Message recu= %s\n",mess->mtext);

strcpy(mess->mtext,"serveur repond et envoie un message");
printf("serveur envoi message\n");

n=sendto(sd,mess,sizeof(message),0,
        (struct sockaddr*)&son_s,sizeof(son_s));

if (n==-1)
    perror("udpser: err sendto "),affiche_errno(),exit(1);
}
```

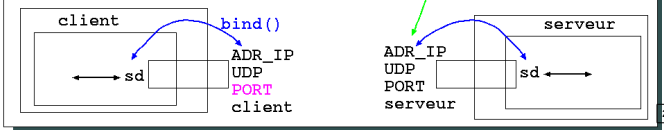


23

```c
/* udpcli.c   */                          /* client UDP */
/* UTC UV SR03 - (c) Michel.Vayssade@utc.fr
 linux:   gcc -o udpcli udpcli.c
 solaris: gcc -o udpcli udpcli.c -lsocket -lnsl */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <errno.h>

#define PORT 4677 /* port du serveur */
int sd,i,n;
int *fromlen;
struct sockaddr_in mon_s;
struct sockaddr_in son_s;
struct hostent *hs;

typedef struct messages {
  char mtext[50];
} message;
message *mess;

void affiche_errno() { printf("\nerrno=%d",errno); }
```

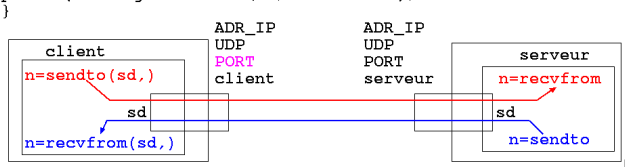
24

```
main(){                                          /* client UDP */
hs=gethostbyname("localhost");

sd=socket(PF_INET,SOCK_DGRAM ,0);

if(sd==-1)perror("udpcli:err socket"),affiche_errno(),exit(1);
bzero(&son_s,sizeof(son_s));    /* adr IP serveur */
bcopy(hs->h_addr,&son_s.sin_addr,hs->h_length);
son_s.sin_family = PF_INET;  /* sur IP */
son_s.sin_port = htons(PORT); /* port du serveur */
bzero(&mon_s,sizeof(mon_s));
mon_s.sin_addr.s_addr = INADDR_ANY;
mon_s.sin_family = PF_INET;  /* sur IP */
/* port client choisi par le système : */
mon_s.sin_port = htons(0);
/* si on impose le même port que pour le serveur,
   et que l'on est sur la même machine => déjà utilisé */

n=bind(sd,(struct sockaddr*)&mon_s,sizeof(mon_s));
if (n==-1) perror("udpcli:err bind"),affiche_errno(),exit(1);
```

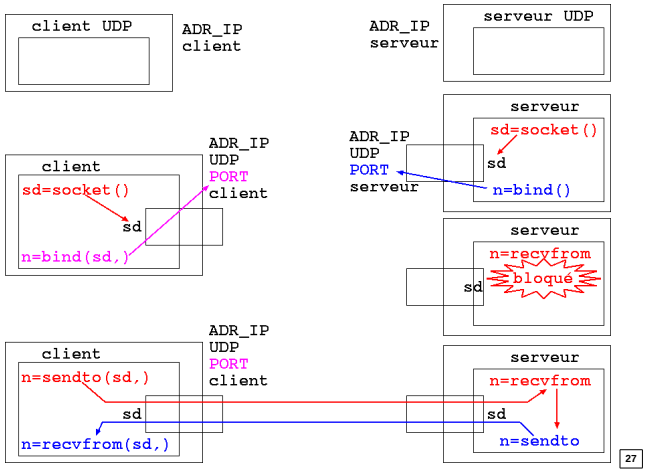


25

```
mess=(message*)malloc(sizeof(message));      /* client UDP */
strcpy(mess->mtext,"j'envoie un message");

n=sendto(sd,mess,sizeof(message),0,
        (struct sockaddr*)&son_s,sizeof(mon_s));

if(n==-1)perror("udpcli:err sendto"),affiche_errno(),exit(1);
fromlen=(int*)malloc(sizeof(int));
*fromlen=sizeof(mon_s);
printf("lire message\n");

n=recvfrom(sd,mess,sizeof(message),0,
          (struct sockaddr*)&mon_s,fromlen);

if (n==-1)
    perror("udpcli: err recvfrom "),affiche_errno(),exit(1);
printf("Message recu= %s\n",mess->mtext);
}
```

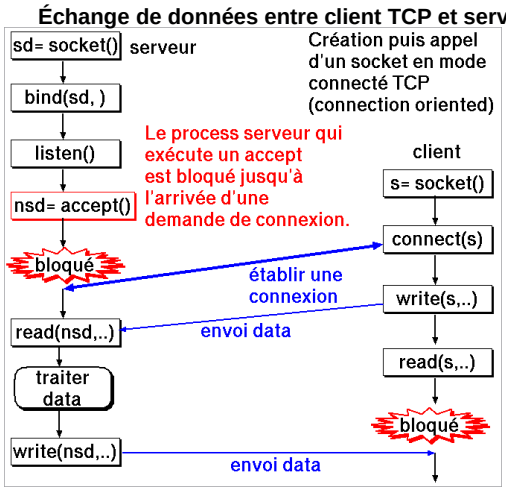


26



27

Échange de données entre client TCP et serveur TCP



28

Utilisation :

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

sd = socket(AF_INET, SOCK_STREAM, 0);
|              |          |
descripteur    domaine    type
=-1 si erreur

#define PORT 0x1234
#include <netinet/in.h>   in: domaine internet
struct sockaddr_in mon_soc;
bzero(&mon_soc, sizeof(mon_soc)); /* init mon_soc */
mon_soc.sin_family = AF_INET;
```

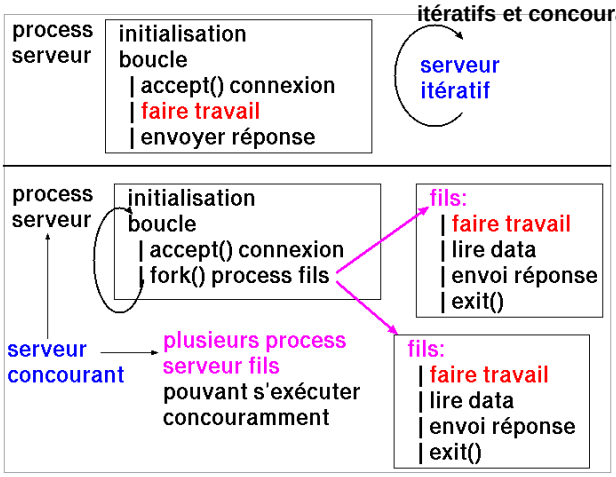


```
mon_soc.sin_port = htons(PORT);

bind(sd, &mon_soc, sizeof(mon_soc));
/* bind() donne un identifiant (le numéro de port
   dans le domaine AF_INET) au socket pour que les
   clients puissent lui envoyer une demande de
   connexion en utilisant cet identifiant */

listen(sd, 5);
/* listen indique au système que l'on va se mettre
   en attente de demandes de connexion */

newsd = accept(sd, 0, 0);
/* on se met en attente d'arrivées de demandes de
```



```
connexion de clients qui exécutent l'appel connect()
Cet appel est BLOQUANT. On en ressort seulement si
un client appelle le serveur.
accept() renvoie un nouveau descripteur sur lequel
se feront tous les échanges de données avec ce client.
Le descripteur initial est réservé à l'écoute des
demandes de connexion.  */
```


Différents types de serveurs TCP : itératifs et concourants



32

– **itératifs :** si le travail à faire est petit et prévisible, cette méthode est plus rapide et moins coûteuse.
– **concourants :** si le travail à chaque connexion est plus long ou non prévisible, il y a un risque de perte de demandes de connexion dans le mode itératif. On crée alors un process fils pour faire le travail et le serveur principal reboucle aussitôt sur le accept().

Exemple de serveur et de client TCP

```
      Un serveur TCP  et  un client TCP
(1)  le serveur se met en attente sur un port TCP
(2)  le client se connecte au serveur
(3)  le serveur accepte la connexion sur un nouveau socket newsd
(4)  client boucle sur envoi "n" messages de tailles 8, 16,...
(5)  le serveur boucle sur la lecture du socket,
     pour récupérer des messages complets, de 8,16,32, il doit
     implanter une boucle (while(attendu)) pour chaque message
     car TCP mets tous les messages à la file, sans frontière.
```



```
/*  tcpser.c  */                          /* serveur TCP */
/* UTC UV SR03 - (c) Michel.Vayssade@utc.fr
 linux:       gcc -o tcpser tcpser.c
- créer socket SOCK_STREAM
- init structure sockaddr
- bind socket to PORT
- listen socket: accepte demandes connexions
- accept une connexion demandee par un client
- boucle:  . lire données sur le socket
           . IF valeur[0]=-1 fin de boucle, FIN programme
           . afficher qques valeurs lues sur le socket    */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <signal.h>
#include <errno.h>
#define NBENREG  10  /* transferer NBENREG messages */
#define ENREGSIZE 8  /* de longueurs 8,16,24,32,40,..*/
#define END_OF_SEND -1
#define SLEEPTIME 0
#define PORT  0x2001
union type_uni { char txt[ENREGSIZE*NBENREG];
   double valeur[NBENREG];  } msg_src;
static int nbErreurTotal = 0 , recu , attendu , fois ;
void affiche_errno() { printf("\nerrno=%d",errno); }
```



```
main() {                                        /* serveur TCP */
        struct  sockaddr_in    sin;
        int     sd, i, j, newsd, count, nbInfo , d1=0 ;
        printf("\nser - init socket");
/*- socket SOCK_STREAM -----------------------------------*/
        if ((sd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
                perror("\nser: error init socket"),exit(1);
/*- init structure sockaddr ------------------------------*/
        bzero(&sin, sizeof(sin));
        sin.sin_family = AF_INET;
        sin.sin_port    = htons(PORT);
/*- bind socket to PORT ----------------------------------*/
        if(bind(sd, (struct sockaddr*)&sin,sizeof(sin))==-1)
                perror("ser: error bind"), exit(1);
/*- listen socket: accepte demandes connexions ------------*/
        if (listen(sd, 5) == -1)
                perror("ser: error listen"), exit(1);
        printf("\nAttachement reussi- Attente connexion.\n");
/*- accept une connexion demandee par un client -----*/
        if ((newsd = accept(sd, 0, 0)) == -1)
                perror("ser: error accept"), exit(1);
        printf("\nConnexion etablie\n");
/*- raz buffer lecture ----------------------------------*/
        for ( i=0 ; i<NBENREG ; i++)
                msg_src.valeur[i] = 0;
```



```
/*- BOUCLE - lecture messages -------- serveur TCP -------*/
        for ( nbInfo=1 ; nbInfo<= NBENREG+5 ; nbInfo++)
        {
        sleep(1);
        attendu = nbInfo*ENREGSIZE; recu = 0; fois = 0;
        while (attendu) {
        count = recv(newsd, &msg_src.txt[recu],attendu,0);
        if ((int)msg_src.valeur[0] == END_OF_SEND)
        {       printf("\nRecu END_OF_SEND.");
        printf("\n mess coupes= %6d. d1= %5d. Exit\n",
               nbErreurTotal,d1);
               exit(0); }
        if (count==-1) perror("ser: error recv"),exit(1);
        recu = recu + count; attendu= attendu - count;
        fois = fois +1;
        }
            if (fois=1) d1= nbInfo; else nbErreurTotal++ ;
printf(\
"\nmsg # %4d,errno=%5d,recu= %4d,pre=%6d der=%6d en %3d fois",
nbInfo,errno,recu,
(int)msg_src.valeur[0],(int)msg_src.valeur[nbInfo-1],fois);
        }
printf("\n Fin bcle lecture. Mess coupes=%6d. d1=%5d. Exit\n",
  nbErreurTotal,d1);
}
```



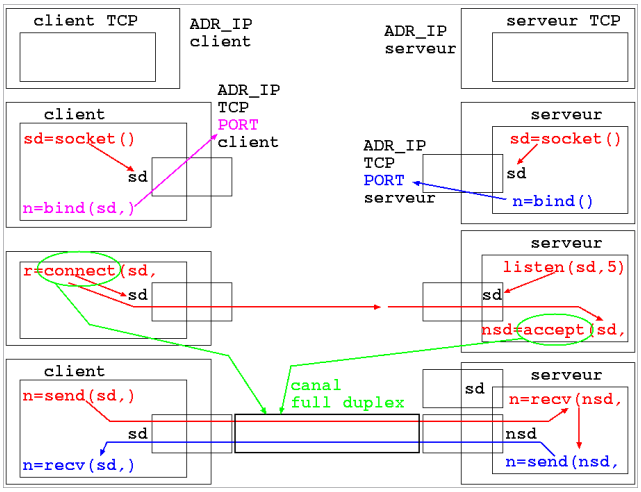
```
/*  tcpcli.c  */                                /* client TCP */
............mêmes déclarations que serveur
main() {
      struct   sockaddr_in      sin;
      struct   hostent          *hp;
      int      sd, i, j, nbInfo, result, count;

      errno = -1;
/*- lire adresse destinataire dans "/etc/hosts" ----------*/
      if ((hp = gethostbyname("localhost")) == 0)
          perror("cli: gethostbyname"), exit(1);
/*- init structure sockaddr ------------------------------*/
      bzero(&sin, sizeof(sin));
      bcopy(hp->h_addr, &sin.sin_addr, hp->h_length);
      sin.sin_family = hp->h_addrtype;
      sin.sin_port   = htons(PORT);
/*- ouvrir socket ----------------------------------------*/
      sd = socket(AF_INET, SOCK_STREAM, 0);
      affiche_errno();
      if (sd == -1) perror("cli: socket"), exit(1);
/*- connect sur PORT ------------------------------------*/
      result=connect(sd,(struct sockaddr*)&sin,sizeof(sin));
      affiche_errno();
      if (result == -1) perror("cli: connect"), exit(1);
      printf("\nConnexion etablie. Envoi messages.");
```



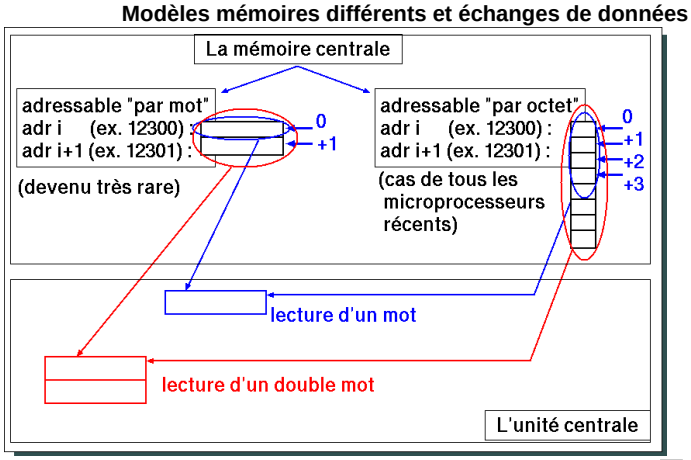
```
                                        /* client TCP */
/*- BOUCLE envoi successif d'une table de 8,16,.. octets -*/
     for ( nbInfo=1 ; nbInfo<=NBENREG ; nbInfo++)
     {
        init_table(nbInfo);
        count = send(sd, msg_src.txt,nbInfo*ENREGSIZE, 0);
        if (count == -1) perror("cli: send"), exit(1);
        printf("\nmsg data # %4d errno=%4d envoi %5d
        octets pre=%6d der=%6d",nbInfo,errno,
        nbInfo*ENREGSIZE,(int)msg_src.valeur[0],
        (int)msg_src.valeur[nbInfo-1]);
     }
/*- Envoi message terminaison -----------------------------*/
     nbInfo = 1;
     msg_src.valeur[0] = END_OF_SEND;
     for ( i=1 ; i<=3 ; i++)
     {  count = send(sd, msg_src.txt,nbInfo*ENREGSIZE,0);
        if (count==-1) perror("cli:send termin."),exit(1);
        printf("\nmsg fin # %4d errno=%4d envoi%5d
        octets pre=%6d der=%6d",i,errno,nbInfo*ENREGSIZE,
        (int)msg_src.valeur[0],(int)msg_src.valeur[nbInfo-1]);
     }
/*- Fermer connexion -------------------------------------*/
     close(sd); printf("\nExit.\n"); exit(0);
}
```



```
$ gcc -o tcpcli tcpcli.c
$ gcc -o tcpser tcpser.c

$ ./tcpser                                (1)
                                           |
ser - init socket                          |
errno=-1                                   |
Attachmt reussi- Attente connexion        (2)
                                           |
errno=-1                                   |
Connexion etablie                          | (4)------\  (6) réception
                                           |  |       |  |
msg #    1, errno= -1, recu=   8,pre=      |  | 1 der= |  | 1 en 1 fois
msg #    2, errno= -1, recu=  16,pre=      |  | 2 der= |  | 1 en 1 fois
............                               |  |        |  |
msg #   10, errno= -1, recu=  80,pre=      |  |10 der= |  | 9 en 1 fois
Recu END_OF_SEND.                          |  |        |  |
 mess coupes=      0. d1=    10.Exit       |  |        |  |
-----------------------------------------------------------------
Dans une autre fenêtre :                   |  |        |  |
$ ./tcpcli                                 |  |        |  |
                                           |  |        |  |
errno=0                                    v  |        |  |
Connexion etablie. Envoi messages.        (3)-/      (5)-/ envoi

msg data #   1 errno=   0 envoi     8 octets pre=   1 der=   1
msg data #   2 errno=   0 envoi    16 octets pre=   2 der=   1
............
msg data #  10 errno=   0 envoi    80 octets pre=  10 der=   9
msg fin #    1 errno=   0 envoi     8 octets pre=  -1 der=  -1
msg fin #    2 errno=   0 envoi     8 octets pre=  -1 der=  -1
msg fin #    3 errno=   0 envoi     8 octets pre=  -1 der=  -1
Exit.
```



Modèles mémoires différents et échanges de données



Les UC sont construites pour être capables de manipuler plusieurs **types de données** (entiers, flottants,caractères) qui sont de tailles différentes :

– certaines sont **csur 32 ou 64 bits** (int, long, float, double),

– d'autres sont des **collections d'octets**.

Quand on doit ranger (lire) une donnée de 32 bits dans une mémoire organisée en octets, il faut couper cette donnée en 4 morceaux et ranger (lire) chaque morceau (chaque octet) dans un octet de mémoire.

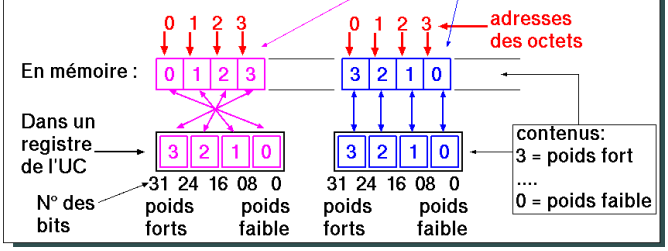
Mais par quel "bout" commencer ? Par le gros bout ou par le petit bout ?

Il n'y a aucune nécessité technique qui favorise l'un plutôt que l'autre.

Schémas de rangement d'une suite de 4 octets formant un mot de 32 bits

Il y a deux possibilités: - les octets de poids forts d'abord
- les octets de poids faibles d'abord

Soit : le "gros bout" d'abord ("big endian" first),
ou le "petit bout" d'abord ("little endian" first).



Comme les protagonistes des Voyages de Gulliver certains préfèrent commencer par le gros bout ("big endian") d'autres par le petit bout ("little endian").

Et, comme il y avait **deux choix** possibles, la moitié des constructeurs d'ordinateurs ont fait l'un des choix, et l'autre moitié ... l'autre choix !

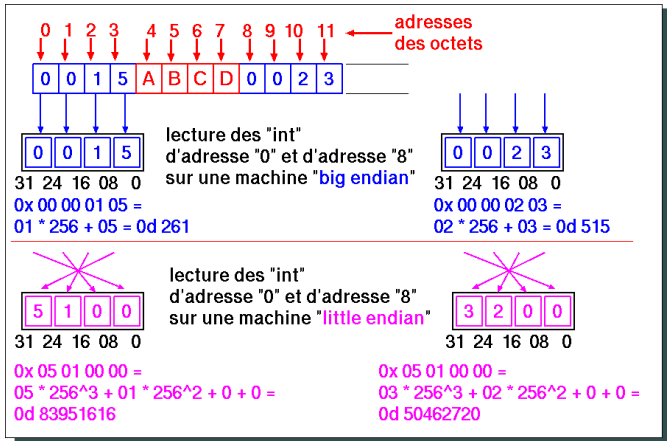
big endian : Sparc, MIPS (SGi), ...

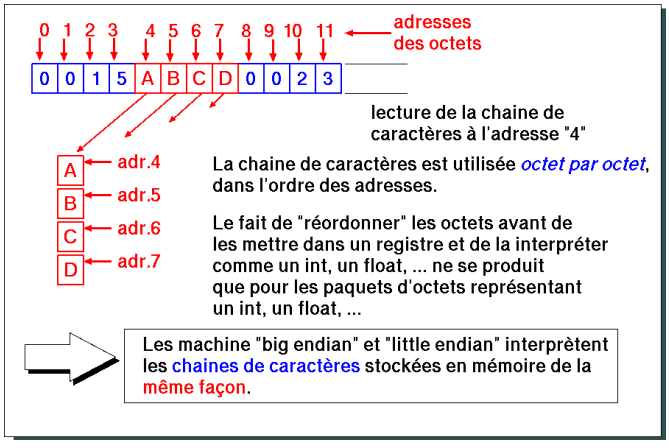
little endian : VAX, Motorola, Intel x86, ...

Remarque : certains processeurs modernes peuvent fonctionner selon les deux modes : Alpha, Power-PC, MIPS, ARM9, ...

Mais en quoi ceci nous concerne-t-il ?

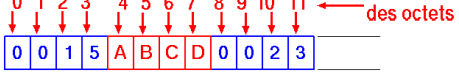
L'ordre des octets ("byte ordering") sur une machine concerne ***toutes*** les applications "réseau".





Si on veut avoir successivement en mémoire :
- un entier valant 261,
- une chaine de caractères contenant A,B,C,D,
- un entier valant 515,

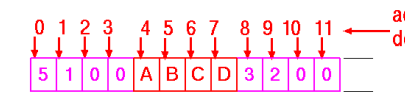
alors, sur une machine big-endian, la mémoire devra contenir :
dans l'ordre des adresses.

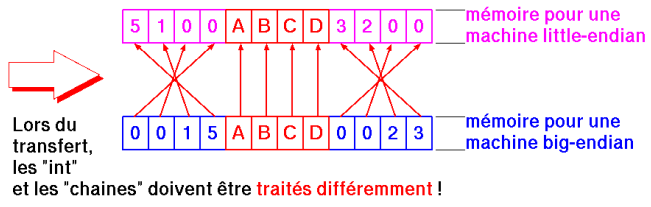


Si on transfère telle quelle cette portion de mémoire
dans la mémoire d'une machine little-endian,
alors, lors de l'interprétation de son contenu, on trouvera :
- un entier valant 83951616,
- une chaine de caractères contenant A,B,C,D,
- un entier valant 50462720.

Après transfert, valeurs fausses pour les entiers, mais juste pour les chaines de caractères.

Si on veut avoir une interprétation correcte,
[entier 261, chaine A,B,C,D, entier 515]
alors, sur une machine little-endian, la mémoire devra contenir :

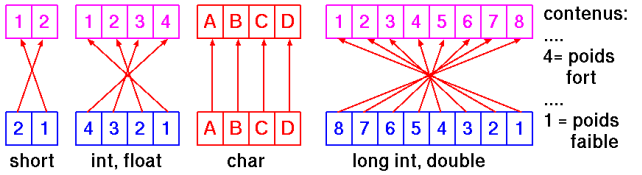




Lors du transfert, les "int" et les "chaines" doivent être traités différemment !

49

En fait, lors d'un transfert, c'est chaque type de donnée qui doit subir un traitement particulier, qui dépend de la nature de la machine de départ et de celle de la machine cible.

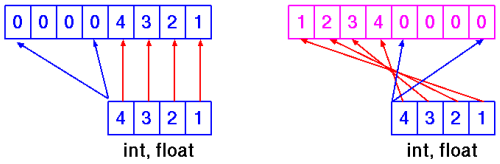


Le transfert est impossible si on ne connait pas le type de chaque donnée.
On ne peut pas prendre un bloc de mémoire et écrire une fonction de transfert générale.

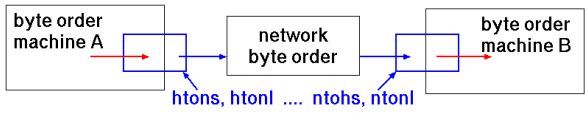
Pour chaque paquet de données, il faut indiquer à la fonction de transfert, la suite des types à transférer.

50

Les choses se compliquent encore un peu quand on transfère des données d'une machine 32 bits vers une machine 64 bits.



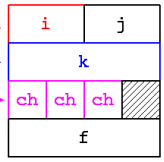
Pour l'utilisation des appels systèmes "réseaux" :



Pour les applications → Utilisation de la bibliothèque système "XDR"

51

```
/* wbl.c */
/* gcc -o wbl wbl.c */
#include <stdio.h>
/* #include <sys/file.h> define O_CREAT et O_RDWR */
#include <string.h> /* pour memcpy */
                    /* pour solaris */
#include <sys/fcntl.h> /* define O_CREAT et O_RDWR */

typedef struct {
   short i;
   short j;
   int k;
   char ch[3];
   int f;
} biglit;
extern int errno;
```



52

```
main()
{  biglit test;
   char *name = "wbl.dat";
   int fd,i;

   test.i = 10;
   test.j = 10*256+1;
   test.k = 10*256*256+11*256+12;
   test.ch[0]= 'A'; test.ch[1]= 'B'; test.ch[2]= 'C';
   test.f = 11*256*256+12*256+13;

   if ((fd = open(name,O_CREAT|O_RDWR,0600)) <=0)
   { printf("system err.in open, errno= %d\n",errno);
     exit(0); }

   i = write (fd,&test,sizeof(test));
   printf("écrit %d\n",i);
   if (i==-1)  {
     printf("system err.in write test, errno= %d\n",errno);
     perror("erreur="); }
   close (fd);
}
```

53

```
linux : Intel x86 32 bits, little endian
sunserv : Sparc 32 bits, big endian

[vayssade@linux endian]$ od -b wbl.dat
0000000 012 000 001 012 014 013 012 000 \
        101 102 103 277 015 014 013 000

lo33 sunserv:~/mv/endian> od -b wbl.dat
0000000 000 012 012 001 000 012 013 014 \
        101 102 103 004 000 013 014 015

1ère moitié  test.k = 10*256*256+11*256+12
linux   012 000 001 012 014 013 012 000   → 0*256*256*256
sunserv 000 012 012 001 000 012 013 014

2ème moitié  A B C
linux   101 102 103 277 015 014 013 000
sunserv 101 102 103 004 000 013 014 015
```

54

Ch03 - Les sockets. Exemples de questions

➤ En quelques mots, que sont les "sockets" et à quoi servent-ils ?
Réponse : diapo 2.

➤ Quels sont les deux domaines d'action des sockets ?
Réponse : diapo 3.

➤ Quels sont les deux types de sockets ?
Réponse : diapo 4.

➤ Décrire (un schéma serait bienvenu), le processus d'établissement d'une liaison par socket.
Réponse : diapos 6 à 8.

➤ Quels sont les caractéristiques des deux types principaux de sockets (leur méthodes d'échange de données) ?

Réponse : diapo 10.

➤ Quels sont les deux premiers appels systèmes dans l'établissement d'une liaison par socket ?
Réponse : diapos 12 à 14 : sd= socket() et bind(sd,..).

➤ Quels sont les appels systèmes d'échange de données sur un socket datagramme ?
Réponse : diapo 15 à 19 : sendto(), sendmsg(), recvfrom(), et, sous certaines conditions, send() et recv().

➤ Quel est le rôle exact de l'appel système bind() ?
Réponse : diapos 22, 25 et 27 : associer le descripteur "sd" renvoyé par sd=socket() au triplet (adr_ip, protocole, Port).

➤ Quel est le rôle de l'appel système listen() ? Dans quel cas doit-on l'utiliser ?

Réponse : diapos 28 à 30.

➤ Quel est l'appel système qui met un socket "connecté" en attente des demandes de connexion ? Et que renvoi cet appel ?

Réponse : diapos 30 et 31 : newsd= accept()

➤ Décrire le mode de fonctionnement des deux types de process serveurs de socket connecté ?

Réponse : diapo 32.

➤ Faite un schéma du process client et du process serveur connectés par un socket "stream", avec l'indication des directives d'échange de données et des descripteurs qu'elles utilisent.

Réponse : diapo 41.

➤ Soit le mot de 32 bits 0x01020304 (int ou long suivant les machines). Dessinez la position des *octets* composant cet entier dans

la mémoire d'une machine "big endian" et dans celle d'une machine "little endian".

Réponse : diapo 46 : Big= [01 | 02 | 03 | 04], Little= [04 | 03 | 02 | 01].

➤ Es-ce que c'est la machine (l'UC) ou bien la mémoire qui est "Big" ou "Little" endian ?

Réponse : diapo 49 : l'unité centrale.

➤ Quand on envoie un paquet de données d'une machine "Big endian" à une machine "Little endian" (ou l'inverse !), suffit-il d'inverser tous les octets 4 par 4 (avec htonl() par exemple) ?

Réponse : diapo 50 : Non ! !

➤ Si on a l'élément "char ch[3]" dans une structure de donnée, que font les compilateurs modernes ? Pourquoi ?

Réponse : diapo 52 : ils ajoutent un octet de remplissage (padding) parce que les accès à la mémoire "calés" sur des "frontières" de mot sont beaucoup plus performants (de 2 à 10 fois selon les machines).

Ch03 - Les sockets.

Fin du chapitre.

(diapo vide)