Chap1. Communication

On établit une fonction
envoyer permet l'envoi d'une chaîne de caractère d'un programme à un autre.
Elle doit prendre en paramètre des données (chaine de caractères).
recevoir permet la réception de la chaîne.
De la même manière elle doit délivrer des données au programme récepteur.

Le programme Emetteur **alloue un tableau de caractères pour stocker une chaîne** qu'il lit ensuite au clavier et utilise la fonction envoyer pour envoyer cette chaîne au programme Recepteur.
Le programme Recepteur déclare une **variable de type pointeur sur caractère** dans laquelle il reçoit la chaîne à l'aide de la fonction recevoir puis l'affiche.

Emetteur

```
char chaine_envoyer[100];

/* Saisie de la chaine */
printf("Donnez une chaine: ");
scanf("%s", chaine_envoyer);

/* Envoi de la chaine */
envoyer(chaine_envoyer);
```

Recepteur

```
char *chaine_recevoir;

/* Reception du message */
chaine_recevoir = recevoir();

/* Affichage */
printf("Message recu: %s",
  chaine_recevoir);
```

Ici on voit que l'envoi de la chaîne se fait de
l'émetteur au récepteur.Ici on voit que l'envoi de la chaîne se fait de l'émetteur au récepteur.
Lequel doit être lancé en premier pour que ça marche ?
Ces problèmes de synchronisation est réglé par les fonctions elles-mêmes.
La fonction recevoir est généralement bloquante (programme Recepteur est en attente tant qu'aucune donnée n'est reçue) donc un processus est endormi en attente de l'arrivée d'un message.
La fonction envoyer n'est pas bloquante sans se soucier de l'arrivée du message.
C'est le système et protocoles de communication qui s'occupent d'acheminer le message.
Il faut que le programme Recepteur soit en exécution à cet instant sinon le système voyant qu'aucun programme n'attend ce message le supprime.

Le programme Recepteur n'a pas besoin d'être au moment de l'émission en attente dans la fonction recevoir car le système crée une file d'attente de message pour le programme ce qui permet de mémoriser les messages tant que le récepteur ne fait pas appel à la fonction de réception.

Il faut donc lancer le Recepteur avant l'Emetteur.

1.2 Synchronisation de la communication

Il faut savoir si les fonctions sont bloquantes et dans quel ordre elles sont appelées.

1.2.1 Les Problèmes de synchronisation:

- En émission il faut s'assurer que l'autre est prêt à recevoir (exécute et que le système a créé la file d'attente).
- En émission s'assurer que l'autre va recevoir un jour ce message, il faut que le récepteur appel la fonction de réception sinon on sature la file d'attente du récepteur.

- En réception il faut s'assurer que l'autre envoie un message qui correspond à cette attente.
- Il faut s'assurer que les 2 programmes ne font pas appel simultanément à la fonction de réception sinon inter-blocage (deadlock) car chacun attend que l'autre lui envoie un message.

Tout ça est à vérifier au moment du développement et non à l'exécution. En effet certaines parties du code sont utilisées sous condition et ne sont pas toujours testées complètement par une exécution. Pour éviter ces problèmes lors du développement d'un application 2 approches sont possibles:

-Soit les parties sont développées par la même personne il faut mettre en relation chaque envoi avec une réception.

-Soit le code est fait pour communiquer avec une application existante sans que vous ayez accès à son code alors il est nécessaire de connaître l'ordre et le type des appels à effectuer.

Il n'existe pas encore d'environnements permettant de prouver de manière formelle l'exactitude des échanges entre 2 programmes. Seule une aide par l'étude d'automates par exemple. Le domaine de de la vérification de protocole est un domaine de recherche actif.

1.2.2 Echange désynchronisé

Il est possible de réaliser un envoi de plusieurs messages successifs où l'appel à la fonction d'envoi n'est pas obligatoirement suivi d'une réception aucun souci avec la file d'attente pour mémoriser. Avantages : des programmes indépendants et ne bloque pas l'émetteur. Mais si l'émetteur souhaite savoir quand son message est traité par le récepteur ce schéma ne le permet pas.

1.2.3 Echange synchronisé

Offre plus de garantie comme valider la réception d'un message par accusé de réception ou d'une réponse. On aura 2 appels successif dans chaque programme envoyer-recevoie (Emetteur) et recevoir-envoyer (Recepteur). Successif veut dire qu'il n'y a pas d'autre appel à une fonction de communication entre mais n'empêche pas l'appel d'autres instructions.
On a un point de rendez-vous, synchronisation, utilisé sur des programmes communicants travaillent sur la même tâche. Mode application client/serveur (interroge).
Certaine interfaces ou bibliothèques implémentent un échange synchrone sous la forme d'une seule fonction envoyer_et_recevoir. C'est aussi le schéma de communication qui sert de base à tous les appels de procédure à distance (RPC, Remote Procedure Call) traiter au chapitre Java RMI.
Donc des garanties mais pénalise les performances.

1.2.4 Autres échanges

D'autres schéma de construction de communication comme les échanges croisé pour une synchronisation moins fortes (appel chacun fait appel une fois envoyer et une fois recevoir)

- Plusieurs envois successifs de la part d'un programme avant réception et acquittement global de la suite de message: point (acquittement du récepteur pour un ensemble de messages améliore l'indépendance.
- Appels synchrones imbriqué (callback) le programme appelée réalise un échange synchrone imbriqué vers l'appelant avant de répondre: double point de synchronisation et l'appel imbriqué.
- Appels croisés permet une exécution en parallèle des 2 programmes avec une synchronisation faible car chacun émet dès qu'il le peut et traite le message de l'autre ensuite à l'opposé des échanges synchrones répétés.

Ces schémas sont liés aux propriétés associées aux fonctions de communication.

1.3 Problématiques de communication

Avant on traité la communication entre deux programmes. Mais on se trouve dans la cas entre un nombre quelconque de programmes avec d'autres problématiques. (même langage, le correspondant est-il à mon écoute …)

1.3.1 Identification

Dans l'interface d'avant rien n'est fait pour savoir quel est le programme avec lequel nous communiquons et envoyer et recevoir sont uniquement entre nos deux programmes.
Pour communiquer de manière plus générale, il est souhaitable de pouvoir s'adresser à un programme en précisant le destinataire d'un programme.
Indentification de chaque destinataire d'un programme (nom de machine (IP) et un port (nom du service))

1.3.2 Connexion

Mode connecté comme le téléphone composer le numéro et une fois la connexion les échanges se font dans les deux sens et uniquement entre les postes connectés.
Mode non-connecté comme le courrier postal ou électronique on peut émettre de n'importe quelle boîte mais sur chacun des messages, on doit préciser le destinataire.

1.3.3 Protocole

Une fois les programmes prêts à échanger des messages (connaissent l'adresse de leur correspondant, la communication peut avoir lieu. Mais elle doit respecter des règles strictes données échangées et synchronisation. C'est règle appelée le protocole. C'est une mise en œuvre d'un échange. Il peut être de bas niveau comme de haut (nos programmes) niveau applicatif.

1.3.4 Représentation et structuration des données

On échange des données dans la mémoire de l'ordinateur elle est sous forme binaire. Mais elle peut avoir une signification différente suivant les ordinateurs ou le type de donnée (ASCII, octet pour un entier). On a 2 types de codages de données en mémoire quand il s'agit d'octets, double-octets(16 bits), quadruple octet ou plus. Ces codages enregistrent soit les bits ou octets de poids fort en premier (MSB most signifiant bit/byte first big indian) ou de poids faible (LSB least signifiant bit/byte first little indian). Comme les PC (intel, amd) LSB et sun MSB.
De même les normes de codages JPEG MSB, GIF LSB.
Des programmes avec des langages différents peuvent communiquer et pour garantir lrinteropérabilité des programmes on peut utiliser des langages de représentation tels que XDR (external data representation) ou ASN.1 (abstract syntax norm 1). Représente des données structuré, (structures, tableau, chaine) il peut s'agir d'un flus donc pas de séparateur entre deux échanges (audio, vidéo) paquet regroupées à l'arrivée car impossible réellement de circuler un flux continu de données. Sinon des messages aucun regroupement à l'arrivée

1.3.5 Garanties

Il n'est pas possible de répondre Est-ce que mon correspondant s'exécute correctement ? et nos échanges sont basés sur la communication. Et de plus le temps que la réponse arrive l'émetteur de cette réponse peut avoir un problème. Prévoir l'indisponibilté.

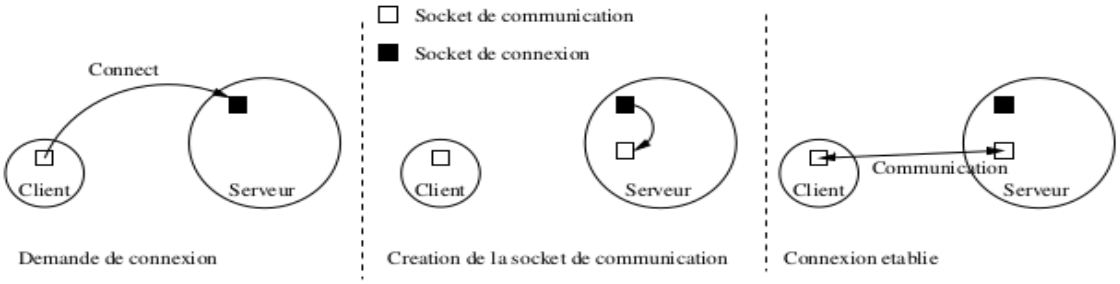
Rappels C
chaîne sous forme de caractère (type char) termine \0 elle est identifié à partir de l'adresse du premier élément de la suite. Une chaine est déclarée comme étant de type char * ou peut être stockée dans un tableau.

Fonctions: strlen ou strcopy et fonction de lecture et affichage utlise le caractère de fin de chaîne comme repère alors attention.

Chap.2 Premiers pas avec des sockets

Comment utiliser les sockets pour mettre en place une communication entre des programmes sur la base d'une interface de programmation simplifiée pour ce chapitre.

Une socket est une prise de communication depuis un processus vers l'extérieur et un autre possédant lui-même une socket. Un programme peut créer plusieurs sockets pour communiquer avec plusieurs programmes. Une socket est identifiée à partir d'un descripteur qui
Est une valeur entière qui par abus de langage assimilé à la socket.
La création d'une socket dépend du mode de communication qu'on souhaite utiliser la socket.



Mode connecté: une analogie du téléphone peut être faite même si c'est une relation asymétrique type client/serveur où on introduit la socket de connexion uniquement pour recevoir des demandes de connexion et le système crée une nouvelle socket sur le serveur pour communiquer cela permet au serveur de traiter simultanément plusieurs clients et de recevoir les requêtes de clients différents sur des socket différentes.

Mode non-connecté: une analogie du courrier la socket boite aux lettres

L'identifiant: une socket est identifié de façon unique nomMachine.nomdedomaine(IP)+numéro de port(nom de service) (entre 1024-65536 codé sur 16bits non signée)

Création et fermeture (simplification fonctions non standard suffixe _EAD)

Inclure d'abord #include "fonctionsSocket.h"

Création en mode non-connecté

Prototype: int socketUDP_EAD(unsigned short port); //envoi une valeur entière (descripteur) c'est la référence qu'à chaque fois que nous devons utiliser la socket / si elle ne peut être créée renvoi -1
Exemple crée une socket et mémorise son descripteur dans desc_sock:

```c
int main (int argc , char *argv[]) {
    int desc_sock; //descripteur de socket
    desc_sock = socketClient_EAD("smith", 2609);

    if ( desc_sock < 0) {
        printf("Erreur creation socket\n");
    }
return 0;
}
```

Socket directement utilisable.

Création en mode connecté

Nous devons savoir en plus le rôle du programme client/serveur car il ne travaille pas de la même façon.

Prototypes:

```c
int socketServeur_EAD(unsigned short port);
int socketClient_EAD(const char *host, unsigned short port);

int accept(int sock_desc, struct sockaddr *, socklen_t);
```

Principe générale :

1. Le serveur crée une socket destinée à recevoir les demandes de connexion avec la fonction **socketServeur_EAD**. Non bloquante elle retourne descripteur ou entier négatif si erreur.
2. Ensuite, le serveur se met en attente d'une connexion avec la fonction **accept**.
3. Enfin, le client demande à se connecter au serveur à l'aide de la fonction **socketClient_EAD**.

Sinon le reste nous procédons de la même manière que non-connecté pour associer un numéro de port à la socket. **SocketServeur_EAD** elle rend un descripteur de socket utilisé dans **accept** pour attendre les demandes de connexion de la part du ou des clients. Cette structure complexe permet à plusieurs clients de se connecter avec un seul serveur. Ainsi le serveur attend toujours sur la même identification de socket (port+machine), connue de l'ensemble de ses clients et traite les demandes de connexions successives.

La fonction **socketServeur_EAD** non bloquante, elle retourne dès que la socket est créée ou une valeur négative si une en cas de problème de création. La fonction **accept** est bloquante et ne sort que lorsque le serveur a reçu une demande de connexion de la part d'un client. En retour elle rend le descripteur de socket qui a été créé pour la connexion avec le client ou –1 en cas d'erreur.

*Les deux derniers paramètres de **accept** servent à recevoir une identification de la socket client*.

Exemple de création de socket desc_sock_conn pour recevoir les demandes de connexion puis la socket desc_sock_trans connectée avec un client: **serveur**

```c
int main (int argc , char *argv[]) {
    // descripteur de la socket de connexion
        //reçoit les demandes de connexion
    int desc_sock_conn;
    //descripteur de la socket de transmission
       //connectée auvec un client
    int desc_sock_trans;

    desc_sock_conn = socketServeur_EAD (2609);
    if (desc_sock_conn < 0) {
        printf("Erreur creation socket \n");
        return 1;
    }

    desc_sock_trans = accept ( desc_sock_conn , NULL, NULL);
    if (desc_sock_trans == -1) {
        printf("Erreur connexion socket \n");
        return 2;
    }
return 0;
}
```

Le **client** utilise la fonction socketClient_EAD pour créer sa socket les paramètres machine et port de la socket serveur auquel il souhaite se connecter. Cette fonction est bloquante tant que la connexion avec le serveur ne peut être obtenue ou timeout dépassée.
Exemple création de la socket du client.

```c
int main (int argc , char *argv[]) {

    int desc_sock; //descripteur de socket

    desc_sock = socketClient_EAD("smith", 2609);
    if ( desc_sock < 0) {
        printf("Erreur creation socket\n");
    }
return 0;
}
```

Fermeture des connexions et des sockets

Lorsque l'échange client/serveur en mode connecté se termine on ferme la connexion avec le prototype: **int shutdown(int desc_sock, int how);**
how quel mode, est fermée la connexion
soit de rendre la connexion unidirectionnelle (mode 0 et 1)
soit de couper celle-ci. Attention shutdown pour le mode non-connecté est une erreur.
Pour terminer proprement quel que soit le mode de communication utilisé il faut penser à ferme les sockets qui ont été ouvert avec le prototype: **int close (int desc_sock);**
descripteur qu'on veut fermer sinon retourne –1 en cas d'erreur. Le fait d'appeler la fonction

remarque: **shutdown** ne ferme pas la socket mais seulement la connexion. Mais si ni l'une ni l'autre est appelées avant la fin du programme le système les ferment pour terminer proprement le programme mais il est plus correct de le faire soi-même dès que possible pour éviter de consommer inutilement des ressources.

La communication (envoi et recevoir)

La communication socket se fait à base d'échanges de données grâce à des foncions d'envoi et de réception. Les fonctions d'envoi prennent en paramètre les données à envoyer sous la forme d'un buffer: une zone de mémoire caractérisée par son adresse de début et sa taille.
Le système copie le contenu de cette zone dans le message à envoyer. De la même manière le **récepteur** reçoit les données du message dans une zone mémoire qui devra être pré-alloué avant de demander à recevoir pour que le système dispose dans votre programme d'une zone de mémoire pour recopier les données celle-ci n'étant pas allouée par le système. Suivant le mode de communication associé à la socket différentes fonctions seront utilisées.

Mode non connecté

La socket de l'émetteur n'est pas reliée à celle du récepteur il est nécessaire de préciser le destinataire à chaque envoi.
Prototype de la fonction d'envoi **sendto**:

```
size_t sendto(
    int desc_sock,     //descripteur socket
    const void *buf,   //pointeur sur zone de memoire
    size_t len,        //taille de la zone de memoire
    int flags,         //options d'envoi
    const struct sockaddr *to, //id socket destinatrice
    socklen_t addrlen //taille de l'adresse
);
```

Desc_sock est le descripteur de la socket obtenu à sa création.
Les deux autres définissent les données à envoyer : Comme les données sont envoyées sous la forme de buffer identifié à partir d'un pointeur **buf** et de type **void** (pointeur) et de la taille **len** de type size-t.
flags correspond à des propriétés de notre envoi (prioritaire, limité etc) toujours 0 plus simple.
Les deux derniers servent à passer l'identification de la socket destinatrice. L'adresse est donnée par un pointeur **to** de type structaddr * et sa taille **addrlen** de type socklen_t. Pour simplifier on utilise la fonction **socketAddr_EAD** qui rend directement l'identification à partir des informations dont nous disposons (nom+port) socket et la taille est données par **taillAddr_EAD**.

Cette fonction sendto est bloquante tant que les données ne sont pas parties de la machine. Il est donc possible de réutiliser le buffeur dès que le programme est sorti de l'appel. A son retour la fonction donne le nombre d'octets envoyés ou –1 elle s'est mal déroulée.

Création de la socket

```
int main (int argc , char *argv[]) {
    int desc_sock; //descripteur de socket
    desc_sock = socketClient_EAD("smith", 2609);

    if ( desc_sock < 0) {
        printf("Erreur creation socket\n");
    }
return 0;
}
```

Exemple envoi d'une donnée contenue dans une variable entière

```c
int main (int argc , char *argv[]) {
    int desc_sock; //descripteur de socket
    int val=42;
    int envoyes;
    //creation de la socket
    . . .
    //envoi des donnees
    envoyes = sendto ( desc_sock, &val, sizeof (in), 0,
                       socketAddr_EAD("smith", 2610), tailleAddr_EAD());

    if (envoyes == -1) {
          printf("Erreur a l'envoi \n");
    }

    close (desc_sock);
return 0;
}
```

Ici le buffer de donnée est constitué d'une seule donnée, la valeur entière, le buffeur est donc identifié par l'adresse en mémoire de la variable et la taille de la variable.

Pour recevoir un message nous utilisons la fonction **recvfrom** dont le prototype est le suivant:

```c
ssize_t recvfrom (
    int desc_sock,          //descripteur socket
    void *buf,              //pointeur sur zone de memoire
    size_t len,             //taille de la zone de memoire
    int flags,              //options de reception
    struct sockaddr *from, //id envoyeur
    socklen_t *addrlen      //taille de l'adresse
);
```

desc_sock le descripteur de la socket qui est utilisée pour recevoir.
Les deux autres décrivent le buffeur utilisé pour recevoir les données
flags donne des options de réception à 0 pour une réception standard
Les deux derniers permettent de recevoir d'identification de la socket émettrice à NULL pour la simplification.. La fonction rend –1 en cas de problème.
Elle est bloquante tant que rien n'est reçu et retourne le résultat dès la réception d'un premier message même si le buffeur prévu n'est pour recevoir le message n'est pas rempli.
En retour la fonction rend alors la taille du message reçu. Donc si le récepteur ne connait pas à priori la taille du message à recevoir il doit allouer avant l'appel à la fonction un buffeur de taille suffisante pour recevoir différents messages mais il peut traiter le message même s'il ne remplit pas entièrement le buffer.
Exemple de réception de la donnée envoyée précédemment

```c
int main (int argc , char *argv[]) {
    int desc_sock; //descripteur de socket

    int val;
    int recus ;

    //creation de la socket
    . . .
    //reception des donnees
    recus = recvfrom (desc_sock, &val, sizeof(int), 0, NULL, NULL);

    if(recus == -1) {
       printf ("Erreur a la reception\n");
    }

    printf ( "J'ai recu la valeur : %d\n", val );

    close (desc_sock);
    return 0;
}
```

Le mode connecté

On relie temporairement 2 descripteurs de socket et il n'est plus nécessaire de préciser l'adresse à chaque échange.

En utilisant la socket connecté les messages sont envoyés grâce à la fonction **send** dont le prototype:

```c
ssize_t send(
    int desc_sock,   //descripteur socket
    const void *buf, //pointeur sur zone de memoire
    size_t len,      //taille de la zone de memoire
    int flags,       //options d'envoi
);
```

C'est une version réduite de sendto dans laquelle il n'est pas nécessaire de préciser le destinataire.

La fonction est bloquante tant que le message n'est pas parti de la machine locale.

Cela garantit que le buffeur du message peut réutiliser sans risque après l'appel à la fonction. Le retour de la fonction correspond au nombre d'octets envoyés ou un code d'erreur -1.

Création de la socket

```c
int main (int argc , char *argv[]) {
    // descripteur de la socket de connexion
        //reçoit les demandes de connexion
    int desc_sock_conn;
    //descripteur de la socket de transmission
       //connectée auvec un client
    int desc_sock_trans;

    desc_sock_conn = socketServeur_EAD (2609);
    if (desc_sock_conn < 0) {
        printf("Erreur creation socket \n");
        return 1;
    }

    desc_sock_trans = accept ( desc_sock_conn , NULL, NULL);
    if (desc_sock_trans == -1) {
        printf("Erreur connexion socket \n");
        return 2;
    }
return 0;
}
```

Exemple d'utilisation une fois la connexion établie pour l'envoi d'une chaîne de caractères :

```c
int main (int argc , char *argv[]) {
    int desc_sock; //descripteur de socket
    int *chaine =  "Hello world";
    int envoyes ;

    //creation de la socket
    . . .
    //reception des donnees
    envoyes = send(desc_sock, chaine, 12, 0);

    if(envoyes == -1) {
       printf ("Erreur a l'envoi\n");
    }

    close (desc_sock);
    return 0;
}
```

Ici la chaîne de caractères est données à partir de l'adresse en mémoire chaine du premier caractère de cette chaîne et de la taille de la chaîne. Note elle n'est pas que de 11 caractères+ \0 qui est également transmis pour conserver à la chaîne ses propriétés.

Les messages sont reçus grâce à la fonction **recv** dont le prototype:

```c
ssize_t recv(
int desc_sock, //descripteur socket
void *buf, //pointeur sur zone de memoire
size_t len, //taille de la zone de memoire
int flags, //options de reception
);
```

Version réduite de la fonction recvfrom.
Cette fonction est bloquante tant que nous n'avons pas reçu de message mais elle n'attend pas d'avoir reçu la taille spécifiée dans len pour terminer son attente.
Le retour est le nombre d'octets reçu par la fonction ou –1 en cas d'erreur.
Exemple de réception de la chaîne de caractères envoyés avec la fonction

```c
int main (int argc , char *argv[]) {
     int desc_sock; //descripteur de socket
     int *buf[20];
     int recus;
     //creation de la socket
     . . .
     //reception des donnees
     recus = recv(desc_sock, buf, sizeof(buf), 0);

     if(recus == -1) {
        printf ("Erreur a la reception\n");
     }
     printf ("J'ai recu : %s\n" buf);
     close (desc_sock);
     return 0;
}
```

On illustre aussi le fait que les fonctions de réception n'attendent pas d'avoir reçu autant de caractère que demande len. Ici on nous demande à recevoir 20 octets alors que nous n'en envoyons que 12 (12 caractères Hello world donc len correspond à la taille maximum.

Remarque : il n'y a aucune relation entre les rôles client/serveur et les rôles émetteur/récepteur. Les rôle client/serveur détermine avant l'établissement de la connexion socketClient/socketServeur. Est une fois la connexion elle s'efface au profit de émetteur/récepteur de send/recv.

Echange des données

Pour bien maîtriser les envois et la réception sur les sockets il faut se rappeler quelques informations *Transmettre* des données l'*envoi se fait à partir de l'adresse en mémoire et la taille des données* alors que la r*éception se fait sur la base d'un buffeur alloué par le programme* avant la réception
Les types de base en C sont mémorisé dans des variables donc on utilise l'adresse de la variable (**&nom_variable)** pointeur de **début+taille** obtenue avec la fonction sizeof taille du message.
Et pour la réception l'adresse de **&nom_variable** et la **taille avec sizeof.** Cela marche pour l'envoi de type structure de données sauf s'il contiennent des pointeurs, tableau, chaîne de caractère.
Pour les **tableaux c'est une réservation de zone de mémoire qui est la taille du tableau** et à l'initialisation de la variable identifiant le tableau avec l'adresse de cette zone de mémoire.
La taille du tableau est obtenue avec sizeof et la définition d'un tableau est un moyen aisé pour définir un buffeur de réception d'une taille voulue.

Pour les **chaînes de caractères (zone mémoire et char * initialisé à l'adresse de début de cette zone et un caractère \0 pour la fin**. Pour calculer la taille on ne peut pas utiliser sizeof car cela donne la taille associée au type de la donnée or la type de la donnée est char * qui vaut 4octets pour sizeof donc la **taille doit être avec la fonction strlen+\0.**
Pour la réception on peut comme dans le mode connecté utilisé un tableau de caractères mais la difficulté est que la taille d'une chaîne n'est pas limitée. Il est difficile de savoir combien de caractères réserver dans le tableau qui doit être d'une taille fixe. La meilleure solution on établit une convention entre l'émetteur et récepteur pour garantir le nombre de caractère envoyé ne dépasse pas la capacité de réception mais ce n'est pas toujours possible il faut avoir recours à autres choses..

Pour les pointeurs nous déconseillons de les passer dans un message car elle correspond à une adresse dans une machine locale mais elle ne correspond pas à la machine de réception.

Chap.4 Les socket des Experts

L'API Application Programming Interface (interface de programmation) du protocole TCP/IP repose sur l'utilisation de sockets de communication et permet l'utilisation au niveau applicatif des fonctionnalités du protocole. Mode connecté repose sur TCP et non-connecté sur UDP.

Création des sockets

On travaille sur les réseaux internet TCP/IP définie à partir du nom de domaine **AF_INET** si nous utilisons d'autres protocoles la valeur peut être trouvée dans le fichier **<sys/socket.h>**.
Le type de protocole définit un mode d'utilisation spécifique de la famille de protocole qui permettent de définir des propriétés des communications avec d'autres mode qui permettent d'envoyer en mode datagramme ou message sur des sockets. Mais on reste sur les modes vus comme en mode connecté on utilisera les socket de type **SOCK_STREAM** et non_connecté **SOCK_DGRAM**.

Création d'une socket: **int socket (int domain, int type, int protocol);**

domain sera AF_INET , type sera SOCK_STREAM ou SOCK_DGRAM, protocol vaut 0.
La fonction renvoie un descripteur de socket sinon –1 en cas d'erreur.
Le résultat est la création de la socket à laquelle aucun nom n'est associé simplement nous avons la prise à laquelle **nous devons attribuer une identification**.

Le point de vue de programmation on voit le descripteur comme un fichier c'est une entrée dans la table des descripteurs de fichier du processus qui exécute le programme.

Correspond à un numéro alloué dans la table des descripteurs. Cette table est limitée nous aurons un nombre limité de sockets créées à un instant donné dans notre programme d'où l'intérêt de fermer celles dont nous ne nous servons plus. Les fonctions de fichier sont donc disponibles, par contre une socket n'est créée/ouverte qu'avec une fonction spécifique.
Pour utiliser cette fonction on inclut les fichiers définissant les types et valeur associés aux sockets:

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

Association d'une adresse

La socket qu'on a créé constitue le combiné téléphonique ou la boîte aux lettres (la prise). Il faut définir l'identification de la socket appelé adresse en communication socket.
Elle est définie dans une structure **struct sockaddr**
Associer une adresse socket à l'aide de la fonction **bind** voici le prototype:

```
int bind( int desc_sock, const struct sockaddr  *addr, socklen_t addrlen);
```

Desc-sock le descripteur de la socket, addr un pointeur sur l'adresse qui doit être associé à la socket, addrlen la taille de la structure d'adresse parce que les adresses n'ont pas la même taille dans tous les domaines de communication possible d'utiliser sizeof(addr).
La valeur de retour est un code d'erreur en cas de problème -1 sinon 0

Remarque : l'interface des sockets a été conçue pour supporter plusieurs protocoles. L'adresse que nous souhaitons associer à notre socket traduit en général la position dans le réseau de la socket. Cette adresse est dépendante de la famille de protocole utilisée. Or **bin est générique** elle est utilisée quel que soit le protocole de communication : l'adresse est passée sous la forme d'un pointeur, qui référence l'emplacement en mémoire de l'adresse quelle que soit sa taille et sa structure en accédant directement à la zone de mémoire correspondante.
Ce type est générique d'adresse **struct sockaddr** est défini pour résoudre les problèmes de type.
Le pointeur donné par le programmeur est transtypé en struct sockaddr * pour obtenir la correspondance des types.
La fonction bind peut ensuite déterminer quel est le type d'adresse utilisé grâce au premier champ de la structure d'adresse. Avec comme premier champ un entier de 2 octets short qui contient un identificateur de la famille du protocole de communication qui est dans notre cas AF_INET car c'est des adresses Internet.

Les adresses : l'API classique

Dans le domaine de communication AF_INET on utilise des adresses décrites par struct sockeaddr_in. Les adresses AF_INET permettent d'accéder à toutes les machines d'un domaine internet dedans (AF_INET) il y a 2 identificateurs machine (IP) et (numéro du port) socket sur la machine.

Contenu d'une adresse AF_INET:

```
struct sockaddr_in  {
    sa_family_t  sin _family; /*famille d'adresses: AF_INET */
    in_port_t    sin _ port;
    /* port dans l'ordre des octets reseau*/
    struct in_addr sin_addr; /*adresse Internet*/
};
```

Cette structure contient des champs de sécurité, d'aligner la taille de toutes les adresse réseaux. Donc avant d'initialiser cette structure il est nécessaire de la mettre complètement à 0 de la manière suivante :

```
struct sockaddr_in addr;
 memset(&addr, 0, sizeof(addr));
```

Pour initialiser une adresse AF_INET il faut initialiser un numéro IP et port.
Le champ **sin_addr** initialise l'IP de la machine sur laquelle se trouve la socket
Le champ **sin_port** avec le port qui doit être associé à la socket.
Dans le cas d'une adresse destinée à être associée à une socket (utilisation de bind) il n'est pas toujours nécessaire d'initialiser ce champ sin_addr avec l'adresse d'une machine. Le champ peut être initialisé à **INADDR_ANY** dans ce cas c'est l'adresse de la machine courante (au moment de l'exécution) qui est utilisé ceci permet d'écrire des programmes qui attribuent bien une adresse locale à la socket sans avoir à connaitre le numéro IP de la machine au moment où on écrit le programme. De la même manière avec le port avec **sin_port** à 0 dans ce cas le numéro de port est attribué par le système en fonction des numéros déjà utilisés par les autres sockets.

Voici différentes manières d'initialiser le numéro IP et port

Les numéros Internet

La description de ce numéro varie mais il est toujours composé de 4 octets
Implémentations anciennes :

```
struct in_addr {
    union {
        struct { u_char s_b1, s_b2, s_b3, s_b4;} S_un_b;
        struct { u_short s_w1, s_w2; } S_un_w;
        u_long S_addr;
    } S_un;
} ;
#define s_addr S_un . S_addr
```

Version récentes de Linux

```
typedef uint32_t in_addr_t;
struct in_addr
{
    in_addr_t s_addr;
};
```

:

Les noms de machine

Ce numéro IP n'est pas facile à mémoriser, il est possible grâce au DNS de donner un nom aux machines et établir une correspondance entre les noms et les numéros. Pour obtenir l'adresse AF_INET d'une machine on utilise la fonction **gethostname**. ;Permet d'obtenir des informations sur une machine à partir de son nom.
Son prototype : struct hostent ***gethostname**(const char *name);

Le paramètre **name** est une chaine de caractère le nom de la machine dont on cherche l'adresse.
La structure struct hostent

```
#include <netdb.h>
struct hostent {
char *name; // nom officiel de l'hote
char **h_aliases; // liste d'alias
int h_addrtype; // type d'adresse de l'hote
int h_length ; //longueur de l'adresse
char **h_addr_list; //liste d'adresse
}
#define h_addr h_addr_list [0] //pour compatibilité
```

h_name= name
Cette fonction retourne une structure de données de type Struct hostent donne les information du point de vue du DNS.

A l'appel le système recherche dans ses tables locales puis dans le DNS les informations liées à la machine suivant sa la politique du système dans le fichier /etc/nsswitch.conf

Struct hostent comprend :

h_name nom principal de la machine

h_aliases une liste d'alias; une machine peut avoir plusieurs noms comme lifc et lifc.univ-fcomte.fr il y a un nom principale et les autres nom son mémorisés dans le champ des alias.

h_addrtype le type des adresses réseaux pour nous c'est IP associées à la machine mais la fonction peut fonctionner avec plusieurs protocoles ce champs désigne comment traiter les adresses réseaux qui sont mémorisées dans le champ h_addr_list

h_length longueur d'une adresse réseau comme bind les adresses réseaux sont gérées à partir d'un pointeur sur le début de la zone et la taille occupée par l'adresse réseau.

H_addr_list liste des adresses réseaux de la machine de la même manière qu'une machine peut avoir plusieurs noms elle peut avoir plusieurs adresses réseau.

- L'adresse principale est en premier accessible de **h_addr**. On remarque que le tableau obtenu n'est pas constitué de pointeurs sur un type d'adresse réseau génériques mais plutôt sur des caractères. Cette définition était classique lorsque l'interface socket a été définie, la définition char * était utilisée pour les pointeurs génériques. Il sera donc nécessaire de transtyper (caster) le pointeur donné **par h_addr** en un pointeur sur une structure **struct in_addr**.

La structure **struct hostent** est allouée par le système et la fonction retourne un pointeur sur la structure associée. Il n'est pas nécessaire d'allouer une structure avant l'appel à la fonction mais il faut déclarer le pointeur qui sert en retour de la fonction.

gethostbyname. Permet d'obtenir des informations sur une machine **à partir de son nom**
Encore une fois cette structure n'est pas triviale à comprendre pour ce qui ne maitrise pas le C. pour cette raison on donne la formule magique qui permet d'obtenir le numéro IP d'une machine :

```
struct hostent *host; //description de la machine serveur
struct sockaddr_in addr; //adresse de la scoket du serveur

// Recherche de l'adresse de la machine
host = gethostbyname (nom_machine);
  if ( host == NULL) {
     //traitement de l'erreur
  }
//Recopie de l'adresse IP
addr.sin_addr.s_addr = ( ( struct in_addr *) ( host->h_addr_list[0]) )->s_addr;
```

Il existe la fonction réciproque **gethostbyaddr** qui permet d'**obtenir le nom** d'une machine **à partir de son adresse**.

Certaine fois il n'est pas possible de connaître le nom de la machine sur laquelle s'exécute le programme, ni de l'obtenir par un passage de paramètre ou à partir d'un fichier de configuration.
Pour obtenir le nom de la machine locale (pour obtenir ensuite son adresse ou faire un bind sur une socket), on peut utiliser la fonction **gethostname**, définie dans l'en-tête **unistd.g** et dont le prototype est :

```
int gethostname (char *name, size_t len );
```

Les ports

Pour localiser une socket sur une machine le système utilise un identificateur local appelé port.
Le numéro de port associé à une socket est codé sur 2 octets.
Les ports inférieurs à 1024 sont réservés aux processus superviseurs (root).

2 possibilités pour obtenir un numéro de port: le fixer par le programme ou en demander l'attribution au système.

Pour faire communiquer deux programmes il faut qu'un des programmes donne une adresse (un port à sa socket) et d'autre part que l'autre programme connaisse cette adresse donc ce numéro de port pour contacter destinataire.
(de même pour une conversation téléphonique il faut avoir le combiné avec un numéro et un interlocuteur qui connaisse le numéro à appeler)

Pour **fixer un numéro de port**, il suffit de positionner une valeur dans le champs sin_port avant de faire appel à la fonction bind. Si le programme est appelé à s'exécuter avec des droits étendus (superviseur), le numéro de port peut être choisi de manière quelconque, sinon pour le mode utilisateur il doit être supérieur à 1024.
Le problème lié au choix du port il ne faut pas qu'il soit associé à une socket ce qui génère une erreur de la fonction bind. Généralement ce n'est pas le cas pour les numéros de port utilisateurs sauf si des applications particulières s'exécutent sur la machine.

attribuer dynamiquement un numéro de port le champ **sin_port** est initialisé à 0 avant l'appel à la fonction **bind** et pour connaître ce port attribué par le système il faut demander au système qu'elle est l'adresse associé à la socket après l'appel à la fonction bind et en extraire le numéro du port
Vois la fonction qui extrait le numéro de port :

```
int getsockname (
    int desc_sock,
    struct sockaddr *addr,
    socklen_t *addrlen
);
```

Au retour de l'appel, la structure **addr** est initialisée avec l'adresse de la socket, y compris le numéro de port attribué. (-1 si erreur)

Une fois le numéro de port connu il faut le faire parvenir aux interlocuteurs de notre programme. On peut afficher le numéro de port à l'écran. L'utilisateur lit la valeur et la donne à un autre programme et l'autre programme peut obtenir cette valeur par saisie, lecture dans un fichier, etc.

Attention, le mode de codage sur Internet est de type MSB alors que sur la machine PC est LSB. Donc dans le cas d'un entier sur 2 octets (numéro de port) le codage est réalisé dans l'ordre inverse. De la même manière les ordinateurs utilisés dans la communication n'utilisent pas forcément le même codage.

On peut utiliser les fonctions de conversion de l'ordre des octets

```
uint16_t  htons (uint16_t  hostshort);
uint16_t  ntohs (uint16_t  netshort);
```

La fonction **htons** (**host to network**) convertit un numéro de port depuis une représentation machine vers la représentation IP.
La fonction **ntohs** (**network to host**) convertit un numéro de port depuis une IP vers une machine.

Ces fonctions sont dépendantes de la machine sur laquelle elles s'exécutent. Elles peuvent ne rien faire su certaines machines (MSB) mais sur d'autres machines (LSB) elles inversent l'ordre des octets.

Sur des machines de type PC il est impératif de les utiliser même si le programme peut marcher sans y faire appel dans la mesure où en utilisant le même codage faux sur deux machines qui communiquent la valeur est bien la même. Ceci peut générer des erreurs dans la mesure où la valeur donnée à la socket ne correspond pas à la valeur attendue.
Un numéro de port ne peut pas être affiché avec un simple format %d puisque le type du port est unsigned short. Il faut donc utiliser le format d'affichage %hu.

Les services

Les sockets sont souvent utilisés pour mettre en place des applications accessible à travers le réseau, nous parlons de service. Un service est un serveurs web, service http, de messagerie etc. Des noms de services sont définis pour éviter d'avoir à mémoriser le numéro de port. Le numéro associé au service est défini dans /etc/services ou sur des bases de données définies sur le réseau.

Un protocole d'accès peut aussi être défini pour un service (certain service sont accessibles qu'avec UDP)
Le numéro de port associé à un service est obtenu avec la fonction **getservbyname** similaire à *gethostbyname*
Le prototype :

```
#include <netdb.h>
struct  servent  *getservbyname (const char *name, const char *proto);
```

Les chaîne de caractères service et proto donnent le service recherché et le protocole utilisé par le service.
La structure se servent :

```
struct  servent {
    char *s_name; //nom officiel  service
    char **s_aliases; //liste d'alias
    int s_port; //numéro de port
    char *s_proto; //protocole a utiliser
}
```

De manière similaire avec le nom et le numéro IP de la machine, cette fonction retourne un pointeur sur une structure **struct servent**, alloué par le système. Nous n'aurons qu'à déclarer une variable du type pointeur sur cette structure dans le programme pour lui affecter le retour de la fonction. Les paramètres de la structure :
Le nom du service et la liste des alias qui peuvent être associés, le numéro de port utilisé par le service, le nom du protocole utilisé pour y accéder.
Noter : ici le numéro de port est défini par un int alors qu'il est défini par un u_short dans l'adresse de la socket, difficile de trouver une explication à cette définition.
Heureusement les numéros de port sont bien définis en entier de deux octets dans les structures protocole IP

Synthèse

La manière de créer une Socket pour remplacer les fonctions socketUDP, socketAddr, socketServeur, socketClient utilisées dans le chapitre précédent. C'est Souvent la même chose.

Pour créer une socket identifiée il est nécessaire de suivre les étapes suivantes :
1 Créer la **socket** avec la fonction socket
2 Initialiser une structure de données correspondant à l'adresse qui doit être donnée à la socket. Il faut obtenir l'entier qui correspond à l'adresse IP de la machine et il faut fixer le numéro de port.
3 Attribuer l'identification à la socket avec la fonction **bind**.

La création de sockets identifiées se fait de la même manière que la socket soit en mode connecté ou mode non-connecté

Exemple création d'une socket identifié en **mode non-connecté** sur le port 2609 en respectant le codage Internet pour le numéro de port :

```c
int main (int argc , char *argv []) {
    int sock; //descripteur socket
    int err; //code erreur
    struct sockaddr_in addr; //adresse de la socket

    //Creation de la socket, protocole UDP
    sock = socket (AF_INET, SOCK_DGRAM, 0);
    if (sock < 0) {
       printf ( "Erreur creation de socket \n");
       return(-1);
    }
    //Initialisation de l'adresse de la socket
    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_port = htons (2609);
    addr.sin_addr . s_addr = INADDR_ANY;

    //Attribution de l'adresse à la socket
    err = bind (sock, ( struct sockaddr *)&addr , sizeof(addr));
    if (err<0) {
       printf ("Erreur sur le bind ");
       return(-2);
    }
    return 0 ;
}
```

Le numéro IP associé à la socket n'a pas besoin d'être précisé car il s'agit de la machine locale. Par contre si je désire une connexion en **mode connecté** ou envoyer une donnée en mode non-connecté le champ s_addr doit être correctement initialisé.

L'exemple montre l'initialisation d'une adresse avec le port 2610 et la machine smith

```c
int main (int argc , char *argv []) {
    int sock; //descripteur socket
    int err; //code erreur
    struct sockaddr_in addr; //adresse de la socket
    struct  hostent  *host;

    //Creation de la socket, protocole UDP
    sock = socket (AF_INET, SOCK_DGRAM, 0);
    if (sock < 0) {
       printf ( "Erreur creation de socket \n");
       return(-1);
    }
    //Initialisation de l'adresse de la socket
    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_port = htons (2609);

    //Recherche de l'adresse de la machine
    host = gethostbyname("smith");
    if(host ==NULL) {
        printf("Erreur gethostbyname \n");
        return(-2);
    }

    //Recopie de l'adresse IP
    addr.sin_addr.s_addr = ((struct in_addr *)  (host->h_addr_list[0]))->s_addr;

    //*****Suite du programme******
    //........

    return 0 ;
}
```

Dans le cas où la communication se fait en mode non-connecté, la création d'une socket identifié est suffisante. Pour le mode connecté nous avons en plus à établir la connexion.

Etablir une connexion

Dans le mode connecté on relie temporairement 2 descripteurs de socket pour ne plus préciser l'adresse à chaque échange. Cela implique une petite gymnastique préliminaire pour établir la connexion : ce qui était fait automatiquement avec les fonctions **socketClient** et **socketServeur** doit maintenant être réalisé explicitement.

Une fois une socket créée, le client se limite à demander sa connexion au serveur grâce à la fonction **connect** dont le prototype est le suivant :

```c
int connect(
    int desc_sock,
    const  struct  sockaddr  *addr,
    socklet_t  addrlen
);
```

L'appel de cette fonction suppose que le serveur ait donné une adresse à sa socket au moment où le client exécute cette fonction.

Par contre, il n'est pas indispensable que le client associe une adresse à la sienne pour la connecter avec celle du serveur.

A l'appel de la fonction, le système envoie une demande de connexion au serveur. Cet appel est bloquant tant que la connexion n'a pas été acceptée ou refusée. (erreur retourne code erreur)

La fonction socketClient contient dont principalement une création de socket, l'initialisation d'une structure d'adresse identifiant le serveur et l'appel de la fonction de connexion.

Du côté serveur et donc du côté de la fonction socketServeur , nous avons vu qu'il devait disposer d'une socket de connexion pour ne pas mélanger les demandes de connexion et les données envoyées par les clients. Ceci est réalisé en créant une socket par la procédure normale puis en la déclarant comme dédiée aux demandes de connexion grâce à la fonction **listen** dont le prototype est :

```
int listen(
    int desc_cock, //descripteur socket a transformer
    int backlog //nb max demandes en attente
);
```

Cette fonction est appliquée à la socket principale du serveur, créée précédemment avec la fonction socket.Cette fonction a pour effet de transformer cette socket en socket de connexion et de créer, associée à la socket, une file d'attente de demandes de connexion.

La taille de cette file d'attente est fixée par **backlog**, de taille maximale huit ce qui devrait permettre de limiter les demandes de connexion en attente.

Exemple la valeur de backlog est à 2 et que 3 clients font un appel connect sans que le serveur ne fasse appel à accept alors le dernier client devrait recevoir une erreur.Attention ceci ne marche pas sur tous les systèmes comme Linux car cette valeur n'est pas prise en compte.L'appel à listen est non bloquant il rend -1 en cas d'erreur. Une fois l'appel listen réalisé sur une socket il ne faut plus l'utiliser pour recevoir des données

La fonction socketServeur contient principalement une création de socket, l'initialisation d'une structure d'adresse qui est associée à la socket par la fonction **bind** et l'appel à la fonction **listen** sur la socket pour la dédier aux connexion.

Nous avons vu qu'ensuite, le serveur se met en attente de demandes de connexion grâce à la fonction **accept** dont le prototype est :

```
int accept(
    int desc_sock_conn, //socket connexion
    struct sockaddr *addr, //socker demandant la connexion
    socklen_t *addrlen //pointeur taille de l'adresse
);
```

La fonction **accept** a pour effet de créer une nouvelle socket dont le descripteur est donné en valeur de retour.Cette socket est connectée avec celle qui a demandé la connexion.C'est donc cette nouvelle socket qui sert pour échanger des données avec le client.

La socket **desc_sock_conn** reste affectée à la réception de demande de connexion, en attente d'une demande de connexion.

Les derniers paramètres qu'on n'avait pas encore utilisés, permettent de recevoir l'adresse de l'émetteur dans une structure d'adresse, que nous devons pré-allouer. Cette structure est identifiée par le pointeur **addr** de type **struct sockaddr *** et un pointeur sur une variable de type **socklen_t**, contenant la taille **addrlen** de l'adresse pré-allouée.

En retour de la fonction cette variable contient la taille de l'adresse reçue, raison pour laquelle il est nécessaire de faire un passage de paramètre par adresse (au sens pointeur). Si nous n'avons pas besoin de cette adresse, il est possible de passer la valeur NULL pour les paramètres **addr** et **addrlen**.

Communication

Les principales difficultés dans l'utilisation de l'interface des sockets proviennent de la création, nous venons de les passer, et les fonctions que vous avez utilisées au chapitre précédent pour la communication avec les sockets sont déjà celles de l'interface, vous avez donc connaissance du principal.

On va revoir la partie communication dans le but de donner quelques détails complémentaires.

Donnée Echangées

La communication sockets ne se fait que sur la base d'échanges de données vues par les programmes sous la forme d'un buffer. Donc le message est constitué du contenu d'une zone mémoire, recopiée dans le message envoyé. De la même maire ce que reçoit le récepteur est recopié dans une zone de mémoire.

Les données échangées le sont sous leur forme brute, non typées. Par exemple si on a défini une structure dans votre programme émetteur et qu'il envoie au récepteur, celle-ci est reçue sans indication de typage sur les champs de la structure. En fait la structure est reçue telle qu'elle était mémorisée chez l'émetteur.

Ceci implique qu'une donnée émise avec un certain type peut être reçue avec un type différent sans erreur. Il est nécessaire d'être prudent sur le type de réception des messages. Nous verrons au chapitre client/serveur comment mettre en place les procédures nécessaires au typage des structures.

En ce qui concerne la réception, il est nécessaire de préparer une zone de mémoire pour la réception du message. Cette zone de mémoire est passée à la fonction de réception sous la forme d'un pointeur sur le début de la zone et de la taille de la zone. Pour garantir l'intégrité de vos données la fonction de réception ne recopie jamais en dehors de cette zone de mémoire. Cela signifie que si la taille des données arrivées dépasse cette zone, seule la partie qui peut être stockée dans la mémoire l'est, à partir du début des données. Le reste est laissé dans un tampon du système en attendant une demande de réception ultérieure. Ainsi si l'émetteur a envoyé une chaîne de cent caractères et que je ne demande qu'à en recevoir cinquante, seuls les cinquante premiers caractères de la chaîne sont reçus les autres restent en attente.

Attention nous avons vu que les chaîne de caractères sont en langage C codées sous la forme d'une suite de caractères terminée par un octet nul (point de fin de traitement). Dans notre exemple la chaîne de cent caractères sera suivi d'un 101^{ième} caractère, un octet nul. Donc si on reçoit que cinquante caractère l'octet nul n'en fera pas partie et nous ne pouvons utiliser les fonctions de traitement des chaînes de caractère sans risque d'erreur. On peut ajouter manuellement cette valeur.

Les données échangées sont transmises sous la forme d'un flux. C'est-à-dire qu'aucun marquer n'est positionné entre les différents envois et il n'est pas possible de déterminer la taille d'un envoi à partir du buffer de réception. Ainsi il est possible de recevoir deux envois successifs de cinquante octets aussi bien sous la forme d'un buffer de cent octets que de dix buffeur de dix octets. C'est à l'application de réaliser la délimitation des données si elle en a besoin. En utilisant le protocole de communication TCP, nous avons la garantie que l'ordre des données est préservé à la réception. Par contre en UDP cette garantie est perdue.

Mode non-connecté

Dans le chapitre précédent, nous avons évité au maximum l'utilisation des adresses dans la communication. Ainsi dans la fonction **sendto** et **recvfrom**, les champs d'adresse ont été remplis à partir de la fonction prédéfinie ou ignorés. Pour une utilisation avancée il est utile de savoir se servir de ces champs. Rappels

```
size_t sendto(
int desc_sock, //descripteur socket
const void *buf, //pointeur sur zone de memoire
size_t len, //taille de la zone de memoire
int flags, //options d'envoi
const struct sockaddr *to , //id socket destinatrice
socklen_t addrlen //taille de l'adresse
);
```

On peut voir que les paramètres 5 et 6 doivent être remplis avec une structure d'adresse équivalente à celle utilisée dans la fonction **bind** il s'agit de la structure d'adresse identifiant la socket qui doit recevoir le message.

Puis on a vu que la fonction est bloquante tant que les données ne sont pas parties de la machine, il faut préciser que si la taille du message est trop importante pour être envoyé en une seule fois la fonction rend une erreur.

```
ssize_t recvfrom (
int desc_sock, //descripteur socket
void *buf, //pointeur sur zone de memoire
size_t len, //taille de la zone de memoire
int flags, //options de reception
struct sockaddr *from, //id envoyeur
socklen_t *addrlen //taille de l'adresse
);
```

De même que dans la fonction **accept** les derniers paramètres permettent de recevoir l'adresse de la socket avec laquelle le serveur est connecté. Le mode de passage des paramètres est donc identique

et la taille de l'adresse doit être initialisée avant l'appel à la fonction. Si nous n'avons pas besoin de cette adresse il est possible de passer la valeur **NULL** pour les paramètres **from** et **addrlen**

```
int socketServeur_EAD(unsigned short port);
int socketClient_EAD( const char *host , unsigned short port);

int accept ( int sock_desc , struct sockaddr* , socklen_t);
```

```
int accept(
    int desc_sock_conn, //socket connexion
    struct sockaddr *addr, //socker demandant la connexion
    socklen_t *addrlen //pointeur taille de l'adresse
);
```

Remarque L'interface initiale des sockets qu'il n'est pas rare de rencontrer encore sur certains systèmes utilise des types **char *** à la place des types **void *** pour le pointeur sur les données et des types **int** à la place de **size_t**, **ssize_t** et **socklen_t.**

Mode connecté

Du fait que les adresses ne sont pas utilisées dans les échanges en mode connecté, nous avons utilisé les fonctions de l'interface dans le chapitre précédent. On peut ajouter que la fonction **send** peut rester bloquée tant que l'ensemble des données passées en paramètre ne sont pas envoyées, même si cela doit se faire en plusieurs fois. Ce n'est pas le cas de la fonction **sendto** qui rend une erreur si elle ne peut envoyer en une fois.

Généralement on choisit le mode connecté si l'accès au serveur demande plusieurs requêtes, peut être anonymes, doit être sûr, etc. Si l'accès aux serveur est ponctuel ou si on communique avec différents processus, il vaut mieux utiliser le mode non-connecté qui est plus rapide.

Remarques et mise en garde

- L'ensemble des fonctions de l'interface est constitué par des fonctions systèmes. Le déroulement de ces fonctions dépend donc des structures de données internes au système et il n'est pas possible de prédire leur bon déroulement. Par exemple, la fonction de création des sockets fait appel à des structures de données de taille fixe, ce qui fait que si d'autres programmes ont déjà consommé ces ressources, il ne sera pas possible de créer la socket. Il est indispensable de tenir compte de ce fait dans vos programmes et cela se traduit par le test systématiquement des retours des fonctions systèmes et par un traitement d'erreur qui tient compte de ce risque. L'appel à la fonction exit constitue un traitement sommaire mais efficace dans ce cas.
- Les fonctions **recfrom** et **accept** retournent l'adresse de l'émetteur il faut bien se souvenir que la taille de l'adresse est passée par un pointeur en paramètre et qu'elle doit être initialisé.

- Dans la mesure où les sockets sont identifiées dans le système comme des fichiers, il est possible d'utiliser certaines fonctions de manipulation des fichiers sur les sockets. Ainsi read fonctionne comme **recv** et **recfrom** avec 3 arguments et **write** fonctionne comme send avec 3 arguments. **Write** ne peut pas s'utiliser à la place de **sendto** car il faut pouvoir préciser le destinataire. Cette notation peut être rencontrée dans des programmes utilisant des sockets. Il est cependant recommandé d'utiliser les fonctions spécifiques pour plus de clarté.
- Les fonctions d'interface des sockets sont des fonctions systèmes. Elles retournent donc toutes un code d'erreur dans la mesure où nous avons aucune garantie de leur bon déroulement. Un retour d'erreur d'une fonction système est toujours (sous Unix) caractérisé par la valeur -1. Le code erreur spécifique est stockée dans la variable **errno**. Il est possible de traiter les erreur systèmes de trois manières différentes. Dans tous les cas il faut inclure le fichier système errno.h

1. `fprintf(stderr, "%d", errno)`
   Permet de faire afficher le code d'erreur puis aller de voir dans /usr/include/errno.h à quoi il correspond
2. `void perror(const char  *msg)`
   Affiche un message et la chaine donnée en paramètre en fonction de la valeur contenue dans errno
3. `char  *strerroe(int errno)`
   Rend une chaine qui contient un message d'erreur en fonction de errno

Autres fonctions

En cas de besoin on peut connaître l'adresse de la socket avec laquelle on est connecté grâce à la fonction **getpeername** dont le prototype est :

```
int getpeername (
    int desc_sock, //descritpeur socket connectee
    struct sockaddr *addr,
    socklen_t *addrlen
);
```

Au retour de la fonction les variables addr et addrlen contiennent respectivement l'adresse de la socket avec laquelle la socket identifiée par desc_sock est connectée et la longueur de cette adresse.

Socket non bloquante

Certains schémas de communication ont besoin de pouvoir tester si des données sont disponibles sur la socket sans bloquer lorsqu'il n'y en pas. C'est le cas d'un serveur qui veut traiter simultanément plusieurs clients : s'il demande à recevoir sur une socket avec la fonction recv et que le client qui y est connecté n'a rien envoyé alors le programme va être bloqué tant que ce client n'envoie rien. Cela peut être le cas lorsqu'un récepteur reçoit une suite de données dont in ne connaît pas la taille à chaque réception, il ne sait pas s'il reste des données à lire dans la socket.

Solution il est possible de rendre une socket non bloquante c'est-à-dire que les opérations effectuées dessus ne bloquent plus le processus.

Les appels bloquants aux socket peuvent être évités en utilisant la fonction **ioctl** avec le flag **FIONBIO**. Cette fonction est une fonction de contrôle des entrées-sorties en général. Elle travaille sur les fichiers

Dans ce cas elle s'utiliser de la manière suivante :

```
int block ;
/*
* block = 1 , pour rendre non bloquante ,
* block = 0 , pour rendre bloquant e
*/
 err = ioctl (desc_sock , FIONBIO, &block);
```

Dans ce cas où la socket ne contient pas de données ni de demande de connexion ou lorsque le serveur n'accepte pas immédiatement une connexion, les appels aux fonctions bloquantes retournent une erreur :

- Pour **accept**, **send**, **recv** on a **err** égal à -1 et l'erreur **errno** est **EWOULDBLOCK** ;
- Pour **connect**, on a **err** égal à -1 et l'erreur **errno** est **EINPROGRESS**.

Exemple :

```
int block = 1 ;
 err = ioctl (desc_sock , FIONBIO, &block );
 err = recv ( ... );
  if ( ( err < 0) && ( errno == EWOULDBLOCK) ) {
      /* rien dans la socket */
  }
```

L'utilisation des sockets non bloquantes permet de se mettre en attente sur plusieurs sockets en même temps pas scrutation successives. Attention dans ce cas le processus reste actif et s'il passe son temps uniquement à scruter les sockets non-bloquantes, il consomme de la puissance CPU pour rien. Il est donc possible de mettre un processus en attente sur plusieurs sockets

Multiplexages des appels socket

Le multiplexage des appels socket consiste à mettre en attente un programme sur plusieurs sockets simultanément et de savoir, après l'attente, celle qui est prête à envoyer ou à recevoir.

Encore une fois, cette fonction a été conçue pour les fichiers mais peut être utilisée sur les sockets puisque leur interface de programmation est intégrée à celle des fichiers.

L'utilisation de la fonction **select** suppose la constitution d'ensembles de descripteurs de fichiers : un pour les descripteurs en lecture, un pour les descripteurs en écriture et un pour les descripteurs d'exception.

Les descripteurs de lecture sont ceux sur lesquels nous appelons des fonctions de lecture : **read** pour les fichiers, **recv** ou **recvfrom** pour les sockets.

Les descripteurs d'écriture sont ceux sur lesquels nous appelons des fonctions d'écriture : **write** pour les fichiers, **send** ou **sendto** pour les sockets.

Les descripteurs d'exception sont les sorties d'erreur par exemple mais nous ne les utiliserons pas avec les sockets.

Le type associé aux ensemble de descripteurs est **fd_set**

Pour constituer les ensembles des descripteurs on utilise les fonctions suivantes :

```
int fd ;
fd_set fdset;

FD_ZERO (&fdset)
//initialise l'ensemble

FD_SET (fd, &fdset)
//met le descripteur fd dans l'ensemble fdset

FD_CLR (fd, &fdset)
// supprime le descripteur  fd de l'ensemble fdset

FD_ISSET (fd, &fdset)
//teste si le descripteur est pret apres un appel a select
```

Il faut noter que les ensembles de descripteurs sont passés par adresse car ils sont modifiés par la fonction. L'utilisation de la fonction **FD_ZERO** est recommandée avant chaque utilisation et avant chaque réutilisation de l'ensemble.

Il n'est pas nécessaire d'avoir un ensemble de chaque type pour utiliser la fonction select. Une valeur NULL peut-être passée en paramètre pour les ensembles que nous ne souhaitons pas tester.

En fait un ensemble de descripteur peut être de taille différente. Cette taille dépend du nombre de descripteurs de fichiers attribué par processus, ce qui est défini par un paramétrage du système d'exploitation. Pour cette raison le premier paramètre de la fonction **select** est la taille de ces ensembles. Cette taille n'est donc pas liée au nombre de descripteur que vous avez positionné dans votre ensemble mais à la taille du type **fd_set**. Elle est donnée par la constante **FD_SETSIZE**.

Cette constante sera donc systématiquement passée en premier paramètre de **select** :

```
int select(
    int nfds, // = FD_SETSIZE
    fd_set *readfds , //ensemble des desc lecture
    fd_set *writefds , //ensemble des desc ecriture
    fd_set *except fds , //ensemble des desc exception
    struct timeval *timeout //remps d'attente
);
```

L'utilisation de cette fonction suppose les inclusions suivantes

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

En fonction du code erreur on peut savoir par quoi l'attente de select a été interrompue.

Si la valeur de retour vaut 0 elle est sortie par timeout, -1 indique une erreur, et n>0 le nombre de descripteurs prêts.

En sortie de l'appel à select, si au moins l'un des descripteurs est prêt (valeur retour positive) alors seuls les descripteurs prêts sont maintenus dans les ensembles passés en paramètre.

L'utilisation de la fonction FD_ISSET permet alors en testant successivement tous les descripteurs initialement positionnés de savoir lequel est prêt.

Puisque la fonction select modifie le contenu des ensembles de descripteurs, il est nécessaire de réinitialiser ces ensembles avant chaque nouvel appel à la fonction.

La structure **timeout** de type **struct timeval** donne le temps maximum d'attente dans le select :

```
struct timeval{
    long tv_sec ; //secondes
    long tv_usec ; //microseconsde
};
```

Si le paramètre **timeout** à la valeur **NULL**, la fonction **select** n'a pas de délai de garde. Si la structure **struct timeval** est initialisée à 0, il n'y a pas d'attente mais les descripteurs sont quand même testés.