# Le modèle client serveur

## L'interface des sockets

1. INTRODUCTION ..... 2
2. LES SCHEMAS DE PRINCIPE ..... 3
3. PRESENTATION DE TCP/IP ..... 4
4. CREER UN SOCKET : `SOCKET()` ..... 5
5. INITIALISER UN SOCKET : `BIND()` ..... 6
6. INITIALISER UNE FILE D'ATTENTE : `LISTEN()` ..... 7
7. ATTENDRE UNE COMMUNICATION : `ACCEPT()` ..... 7
8. CONNECTER UN CLIENT AU SERVEUR : `CONNECT()` ..... 8
9. RECEPTION DE DONNEES : `READ()` ..... 9
10. EMISSION DE DONNEES : `WRITE()` ..... 9
11. CLORE UNE COMMUNICATION : CLOSE() ..... 10
12. VOUS AVEZ DIT DNS? ..... 10
12.1 Recherche de machine par nom : gethostbyname() ..... 11
12.2 Recherche de machine par adresse : `gethostbyaddr()` ..... 11
13. ADRESSES MACHINES, ADRESSES RESEAUX? ..... 12
13.1 Définitions ..... 12
13.2 Conversion d'adresses : `htons()` -- `htonl()` -- `ntohs()` -- `ntohl()` 12
13.3 Les adresses IP : `inet_addr()` -- `inet_ntoa()` ..... 13
14. CONCLUSION ..... 14

## *1. Introduction*

Le concept de socket est assez difficile à appréhender. C'est BSD qui l'a choisi pour accomplir les communications inter-processus (IPC). Cela veut dire qu'un socket est utilisé pour permettre aux processus de communiquer entre eux de la même manière que le téléphone nous permet de communiquer entre nous. L'analogie entre le concept de socket et le téléphone est assez significative. Elle sera utilisé pour éclaircir la notion de socket.

Ainsi pour recevoir des appels téléphoniques, vous devez d'abord installer le téléphone chez vous. De la même façon, vous commencez par créer un socket avant d'attendre des demandes de communications. La commande `socket()` est alors utilisée pour créer un nouveau socket (que nous désignerons *master*).

Seulement il faut créer un socket avec les bonnes options. Vous devez spécifier les type d'adressage du socket, le numéro de port et surtout le type de socket que vous voulez utiliser. Nous allons détailler ces points un peu plus tard.

Ensuite, de la même façon qu'on vous attribue un numéro de téléphone pour recevoir vos appels, on doit spécifier au socket une adresse à laquelle il doit recevoir les messages qui lui sont destinés. Ceci est réalisé par la fonction `bind()` qui associe un numéro au socket.

Les sockets du type que nous allons utiliser ( `SOCK_STREAM`) ont la possibilité de mettre les requêtes de communication dans une file d'attente, de la même façon que vous pouvez recevoir un appel pendant une conversation téléphonique. C'est la fonction `listen()` qui permet de définir la longueur de la queue d'attente. Il n'est pas indispensable d'utiliser cette fonction mais c'est plutôt une bonne habitude que de ne pas l'oublier.

L'étape suivante est d'attendre les demandes de communication. C'est le rûle de la fonction `accept()`. Cette fonction retourne un nouveau socket (désigné *slave*) qui est connecté à l'appelant. Le socket initial (*master*) peut alors se remettre à attendre les demandes de communication. C'est pour cette raison qu'on exécute un `fork()` à chaque demande de connexion dans le cas du modèle à accès concurrents. Dans le cas d'un serveur à accès unique, on ne se préoccupe plus du socket principal.

On sait maintenant comment créer un socket qui reçoit des demandes de communications, mais comment l'appeler?

Pour téléphoner vous devez d'abord avoir le numéro. Le rôle de la fonction `connect()` est de connecter un socket à un autre socket qui est en attente de demande de communication. Maintenant qu'une connexion est établie entre deux sockets, la conversation peut enfin commencer. C'est le rôle des fonctions `read()` et `write()`.

Enfin, à la fin d'une communication on doit raccrocher le téléphone ou fermer le socket correspondant. C'est la fonction `close()`.

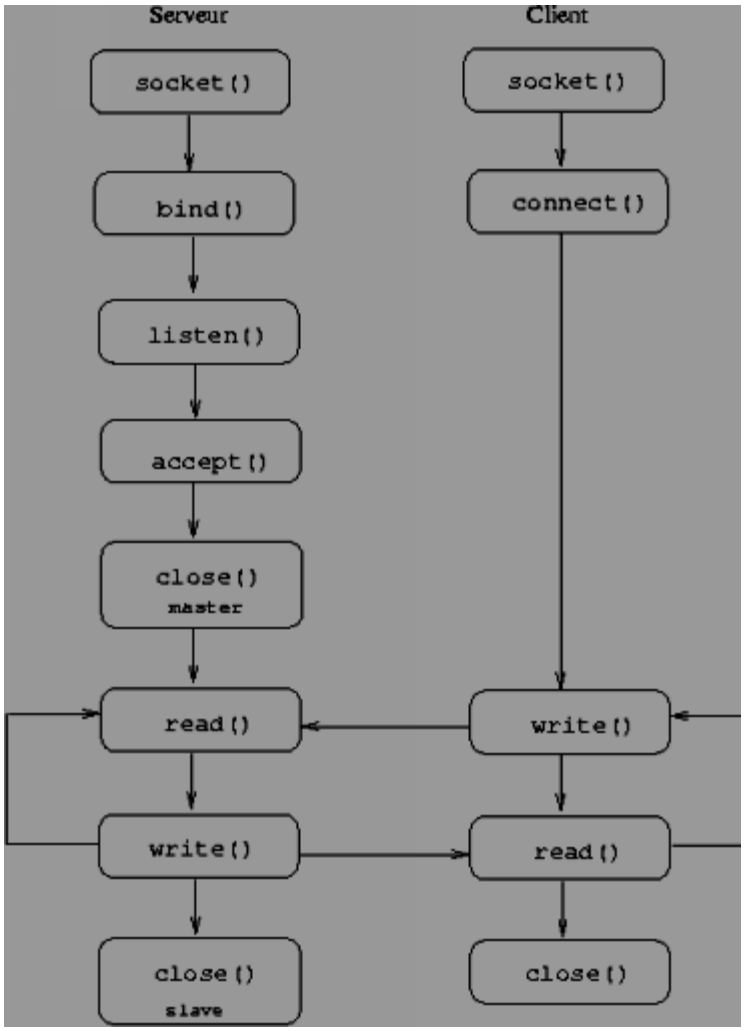
Il faut aussi noter qu'il existe deux types de connexion:

- Le mode connecté correspond au protocole `TCP`. Lorsque deux sockets sont en mode connecté, ce que l'on envoie à l'une est forcement reçu par l'autre. Le protocole se charge de vérifier l'intégrité de la communication et de renvoyer les données perdues.
- Le mode déconnecté correspond au protocole `UDP`. Ce protocole ne tient pas compte de la qualité de la liaison. C'est au programmeur de vérifier qu'aucune information n'a été altérée pendant la transmission

Dans la suite, nous ne nous intéresserons pas au protocole `UDP`. Nous citerons seulement les primitives qui permettent de l'utiliser.

## *2. Le schéma de principe*

- Le modèle à accès uniques : le serveur attend une requête de connexion. Dès qu'un client se connecte, le serveur ferme le socket principal et ne s'intéresse qu'au socket connecté au client. Donc, avec ce modèle, on ne traite qu'un client à la fois. La figure schématise un tel serveur.



### *3. Présentation de TCP/IP*

TCP/IP est sûrement le moyen de communication entre machines le plus utilisé dans le monde. Nous ne rentrerons pas dans les subtilités de ses protocoles. Nous donnerons seulement quelques notions pour que le lecteur puisse imager les propos qui suivront.

TCP/IP est un ensemble de protocoles organisés en couche. Les protocoles définissent un ensemble de commandes quune machine doit envoyer à une autre pour qu'elles se comprennent. En fait les protocoles supposent l'existence d'un moyen de communication fiable entre les deux machines.

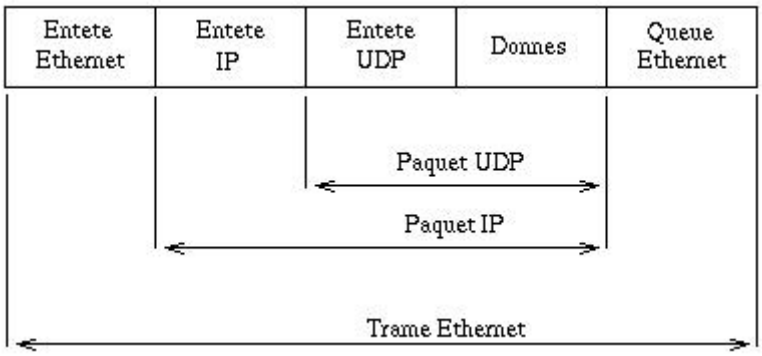
Pour envoyer un segment de données à une machine distante, il faut d'abord découper le segment en morceaux plus petits : des paquets. C'est la couche TCP ou *Transmission Control Protocol*, qui de chaque côté du fil découpe et recolle les morceaux.

D'autre part, les différents paquets ne sont pas obligés de se suivre et peuvent employer des voies différentes pour arriver à destination. C'est le protocole IP ou *Internet Protocol*, qui assure le voyage ou *routage* des différents paquets.

L'émission et la réception des paquets eux-mêmes sont gérées par le protocole UDP ou `User Datagram Protocol`. Ce protocole ne vérifie pas l'intégrité des données. Mais TCP, au recollage, fait des tests de validité et demande la retransmission des paquets douteux.

Enfin, le protocole ethernet permet de faire communiquer les machines entre elles au plus bas niveau : le matériel.

L'emploi de tous ces protocoles implique l'habillage ou *encapsulation* des données pour que l'information voyage sans se perdre dans le réseau des réseaux.




Nous voyons donc sur le schéma que seule une partie des informations des paquets est disponible à lutilisateur. Le reste étant réservé aux différents protocoles.

### *4. Créer un socket : `socket()`*

**Synoptique**

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

int socket(family, type, protocole)
int family;    /* détermine la famille du socket*/
int type;      /* détermine le type du socket*/
int protocole; /* option sur le mode de transfert*/
```

**Description**

Les tableaux et résument l'utilisation des variables `family` et `type`.

| `family` | détermine la famille du socket |
|---|---|
| `AF_UNIX` | protocoles internes UNIX |
| `AF_INET` | protocoles internet |
| `AF_NS` | protocoles Xerox NS |
| `AF_IMPLINK` | protocoles spéciaux |

| `type` | détermine le type du socket |
|---|---|
| `SOCK_STREAM` | utilisé en mode connecté (TCP) |
| `SOCK_DTGRAM` | utilisé en mode déconnecté (UDP) |
| `SOCK_RAW` | ?? |
| `SOCK_SEQPACKET>` | ?? |

L'argument `protocole` est généralement mis à `0` sauf dans certaines applications. Cependant, d'une façon générale on peut lui donner une autre valeur. Le tableau suivant résume l'utilisation de la variable `protocole`.

| `protocole` | | `type` |
|---|---|---|
| `0` | dans tous les cas | tous |
| `IPPROTO_UDP` | protocole UDP | `SOCK_DTGRAM` |
| `IPPROTO_TCP` | protocole TCP | `SOCK_STREAM` |
| `IPPROTO_ICMP` | protocole ICMP | `SOCK_RAW` |
| `IPPROTO_RAW` | protocole RAW | `SOCK_RAW` |

L'appel système `socket()` retourne un entier dont la fonction est similaire à celle d'un descripteur de flux. Nous appellerons cet entier un descripteur de socket : `sockf}`.

**Exemple**

Création d'un socket pour communiquer en mode connecté via le réseau Internet.

```
int sockfd; /* descripteur de socket*/

...

if ((sockfd=socket(AF_INET,SOCK_STREAM,0))==-1)
{
```

```
        perror("Oops! Impossible de créer un socket");

        exit(1);

}
```

### *5. Initialiser un socket : `bind()`*

#### Synoptique

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

int bind(sockfd, myaddr, addrlen)
int             sockfd;  /* descripteur de socket*/
struct sockaddr *myaddr; /* structure d'adresse*/
int             addrlen; /* taille de la structure*/
```

avec:

```
struct sockaddr {
        unsigned short  sa_family;
        char            sa_data[41];
};
```

#### Description

L'appel à `bind()` se fait dans trois cas:

- Le serveur enregistre sa propre adresse auprès du système pour que tous les messages qui lui sont destinés lui parviennent.
- Un client enregistre une adresse spécifique pour lui même.
- Un client sans connexion doit s'assurer que le système lui a affecté une unique adresse que ses correspondants utiliseront afin de lui envoyer des message.

`bind()` renvoie `0` si tout se passe bien et `-1` en cas de problème.

Nous n'utiliserons que le premier point dans le cadre de ce projet.

Pour cela, il est décrit dans `<netinet/in.h>` deux structures simplifiant l'utilisation de `bind()`.

```
struct sockaddr_in { /* équivalent de struct sockaddr*/
        short          sin_family;  /* voir socket()*/
        unsigned short sin_port;    /* le port au format réseau*/
        struct in_addr sin_addr;    /* adresse*/
        char           sin_zero[8]; /* inutilisé*/
};

struct in_addr {
        unsigned long  s_addr;      /* adresse au format réseau */
};
```

Une simple conversion de cast transforme une `struct sockaddr` en une `struct sockaddr_in`.

#### Exemple

On suppose que `sockfd` est le descripteur créé par l'exemple de `socket()`. Il est donc de type `SOCK_STREAM` et de famille `AF_INE`T. Pour initialiser ce socket :

```
struct sockaddr_in sockadr          /* structure d'adresse associée*/
                                    /* au socket sockfd*/
...
sockadr.sin_family=AF_INET;         /* comme a l'appel socket()*/
sockadr.sin_port=htons(1976);       /* port sur lequel fonctionne le*/
```

```
                                    /* serveur. Nous verrons plus*/
                                    /* tard l'utilisation de htons()*/
sockadr.sin_addr.s_addr=INADDR_ANY; /*informonsle systeme qu'un*/
                                    /* serveur attend des connexions*/
if (bind(sockfd,(struct sockaddr*)&sockadr,sizeof(sockaddr_in))==-1) {
        perror("Oops! Impossible d'initialiser la socket");
        exit(1);
}
```

**Note**

Les format réseaux seront décrit un peu plus tard. Sachez que la primitive `htons()` est dans cette exemple, le moyen simple de faire la conversion entre le format interne de la machine hûte et le réseau. `INADDR_ANY` est décrit dans `<netinet/in.h>`. Cette constante informe le système qu'un serveur tourne sur la machine et que toute connexion sur le port concerné doit être lui transmis.

### *6. Initialiser une file d'attente : `listen()`*

**Synoptique**

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

int listen(sockfd, backlog)
int sockfd;  /* descripteur de socket*/
int backlog; /* longueur de la file d'attente*/
```

**Description**

L'appel à `listen()` initialise la queue d'attente pour les requêtes de connexion non encore traitées. L'argument `backlog` spécifie la longueur de cette file d'attente. Il semble que le nombre de place de cette file soit limité de façon implicite à `20` sur la plupart des systèmes. Cela signifie, que le serveur pourra assurer le traitement de `20` requêtes intervenant exactement au même instant.

`listen()` renvoie `-1` en cas d'echec ou `0` si tout va bien.

**Exemple**

On aura pour notre serveur les lignes suivantes : (on n'a évidemment pas d'équivalent pour le client)

```
if (listen(sockfd,5)<0) {
         perror("Oops! Impossible de creer la file d'attente");
         exit(1);
}
```

**Note**

Il faut bien faire la différence entre file d'attente et nombre de clients connectés au serveur. En effet, un client fait d'abord une requête de connexion auprès du serveur. Il rentre de ce fait dans la file d'attente. Dès qu'il obtient la connexion, il sort de la file d'attente mais reste connecté au serveur.

### *7. Attendre une communication : `accept()`*

**Synoptique**

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

int accept(sockfd, peer, addrlen)
int             sockfd;   /* descripteur de socket*/
```

```
struct sockaddr *peer;    /* struct d'adresse*/
int             *addrlen; /* longueur de cette structure*/
```

**Description**

L'argument `sockfd` désigne le descripteur de socket sur lequel seront attendues les connections.

Lorsqu'une requête arrive, le système enregistre l'adresse du client dans la structure `*peer` et la longueur de l'adresse dans `*addrlen`. Il crée alors un nouveau socket connecté avec la destination spécifié par le client, et renvoie à l'appelant un descripteur de socket. Les échanges futurs avec ce client se feront donc par l'intermédiaire de ce nouveau socket.

Le socket `sockfd` n'a donc pas de destination et reste ouvert pour accepter de nouvelles demandes de communications.

Tant qu'il n'y a pas de connexion, le serveur reste bloqué sur l'appel à `accept()`}.

**Exemple**

Pour un serveur qui attend des connexions, on peut avoir:

```
struct sockaddr_in client_adr; /* structure d'adresse du client*/
...
if ((slave=accept(sockfd,(struct sockaddr*)&client_adr,sizeof(sockaddr_in)))<1) {
         perror("Oops! Impossible d'etablir la connexion");
         exit(1);
}
/*
 * on récupere un descripteur de socket 'slave' connecté sur le client
 * et aussi l'adresse du client, contenue dans 'client_adr'
 */
```

### *8. Connecter un client au serveur : `connect()`*

**Synoptique**

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

int connect(sockfd, servaddr, addrlen)
int             sockfd;    /* descripteur de socket*/
struct sockaddr *servaddr; /* adresse du serveur*/
int             addrlen;   /* longueur de la structure pécédente*/
```

**Description**

Un socket est initialement créé dans un état non connecté, ce qui signifie qu'il n'est associé à aucune destination éloignée. L'appel système `connect()` associe de façon permanente un socket à une destination éloigné et le place dans un état connecté.

`connect()` renvoie `-1` en cas d'echec et `0` si tout va bien.

**Exemple**

Dans notre client, on aura les lignes suivantes :

```
struct sockaddr_in serv_adr    /* structure d'adresse du serveur*/
...
serv_adr.sin_family=AF_INET;   /* comme a l'appel a socket()*/
serv_adr.sin_port=htons(1976); /* port sur lequel fonctionne*/
                               /* le serveur*/
```

```
serv_adr.sin_addr.s_addr=adrress_du_serveur_sur_32_bits_au_format_réseau;
if (connect(sockfd,(struct sockaddr*)&serv_adr,sizeof(sockaddr_in))<0) {
        perror("Oops! Impossible de connecter le serveur");
        exit(1);
}
```

**Note**

L'appel à `connect()` ne s'utilise qu'avec des sockets utilisables en mode connecté comme le type `SOCK_STREAM`.

### *9. Réception de données : `read()`*

**Synoptique**

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

int read(sockfd, buff, nbytes)
int  sockfd; /* descripteur de socket*/
char *buff;  /* zone de donnees tampon*/
int  nbyte;  /* taille de la zone de données*/
```

**Description**

De la même façon que l'on lit dans un fichier, on lit dans un socket. La seule différence majeure est que la connexion peut se terminer à tout moment. Il faut donc être attentif et vérifier régulièrement que client et serveur soient toujours dans l'état connecté.

`read()` renvoie `-1` en cas d'echec ou `0` si tout va bien.

**Exemple**

On suppose que `sockfd` soit un descripteur de socket dans l'état connecté:

```
char buff[256];
if ((length=read(sockfd,buf,sizeof(buff)-1))<0) {
                /* -1 pour pouvoir rajouter le '\0'*/
        perror("Connexion perdue");
        exit(1);
}
buff[length]=0; /* pour 'finir' la chaîne de caracteres*/
                /* et par sécurité*/
printf("Message recu:\n%s",buf);
```

**Note**

En cas de perte de connexion, on lira un `0` dans le descripteur de socket, puis des `EOF`.

Nous ne nous intéressons qu'à la fonction `read()`, sachez quand même qu'il existe d'autres fonctions comme `recv()` ou `readv()` qui permettent la gestion des données hors-bande (i.e. urgentes).

### *10. Emission de données : `write()`*

**Synoptique**

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

int send(sockfd, buff, nbytes)
int  sockfd; /* descripteur de socket*/
char *buff;  /* zone de donnees a envoyer*/
```

```
int  nbyte;  /* taille de la zone de données*/
```

**Description**

C'est la fonction symétrique de `read()`.

`write()` renvoie `-1` en cas d'echec et `0` si tout va bien.

**Exemple**

On suppose que `sockfd` est un descripteur de socket dans le mode connecté:

```
char buff[256];
buff="MESSAGE PIPO";
if (send(sockfd,buff,strlen(buff))<0) {
        perror("Connexion perdue");
        exit(1);
}
```

**Note**

Nous ne nous intéresserons qu'à la fonction **`write()`**, sachez quand même qu'il existe d'autres fonctions comme **`send()`** ou **`sendmsg()`** qui permettent la gestion des données hors-bande (i.e. urgentes).

### *11. Clore une communication : close()*

**Synoptique**

```
int close(sockfd)
int sockfd; /* descripteur de socket*/
```

**Description**

L'appel à `close()` est une manière élégante de clore une communication entre deux sockets. Cependant il faut noter que `close()` n'interrompt pas la communication. Le système essaye de transmettre les données sockées dans les buffer du coté serveur ou client.

`close()` renvoie `-1` en cas de problème ou `0` si tout va bien.

**Exemple**

On a `sockfd` un descripteur de socket:

```
close(sockfd); /* fin de connexion*/
```

**Note**

On a vu qu'un descripteur de socket est à peu de choses près assez semblable à un descripteur de flux. C'est pour cette raison qu'il n'existe pas de fonction du style `close_socket()`, mais une fonction commune à tous les descripteurs de flux: `close()`. Pour un meilleur contrôle de la fin de connexion, voir l'appel à `shutdown()`page.

### *12. Vous avez dit DNS?*

Un DNS ou *Domain Name Serveur*, est une méthode pour stocker l'adresse d'une machine, non pas par une suite de numéro, (par exemple `147.38.94.17`), mais avec une notation plus 'lisible' de la forme: machine.noeud.indicatif_de_pays (par exemple `morbier.imt-mrs.fr`). Pour convertir une adresse numérique en une adresse alphanumérique et vice-verça, deux primitives sont à la disposition du programmeur.

### 12.1 Recherche de machine par nom : gethostbyname()

#### Synoptique

```
#include <netdb.h>

struct hostent gethostbyname(hostname)
char *hostname;              /* nom en toute lettre de l'hûte*/
```

avec

```
struct hostent{
        char *h_name;         /* nom officiel de la machine*/
        char *h_aliases;      /* liste des surnoms de la machine*/
        int  h_addrtype;      /* type d'adressage (ex.AF_INET)*/
        int  h_length;        /* longueur de l'adresse*/
        char **h_addr_list;   /* liste des adresses de la machine*/
};                            /* au format NetworkByteOrder*/

#define h_addr h_addr_list[0] /* c'est l'adresse principale*/
                              /* d'une machine*/
```

#### Description

En général on se sert de cette fonction pour obtenir l'adresse d'une machine dont on connait le nom.

#### Exemple

```
struct hostent *h;
char host[]="maroilles.imt-mrs.fr";
struct sockaddr_in addr;
...
if((h=gethostbyname(host))==NULL) {
        perror("Impossible d'obtenir l'adresse IP");
        exit(1);
}
printf("Nom de l'hote: %s\n",h->h_name);
bcopy((char*)&h.h_addr,(char *)&addr.sin_addr.s_addr,h.h_length);
/*
 * on a addr.sin_addr.s_addr qui pointe sur l'adresse
 * de maroilles.imt-mrs.fr
 *
 */
```

#### Note

`bcopy()` est décrit dans `<bstring.h>`. Cette appel copie une zone de mémoire vers une autre. Cependant `<bstring.h>` n'est pas implémenté sur tous les systèmes d'exploitation. Vous trouverez en annexes les *listings* des fonctions de cette bibliothèque.

### 12.2 Recherche de machine par adresse : `gethostbyaddr()`

#### Synoptique

```
#include <netdb.h>
struct hostent gethostbyaddr(addr, length, type)
char *addr;  /* adresse de l'hote au format "xxx.xxx.xx.xx"*/
int  lenght; /* longueur de cette adresse*/
int  type;   /* type d'adresse (AF_INET par exemple)*/
```

#### Description

En général on se sert de cet appel pour connaitre le nom d'une machine dont on connait l'addresse.

**Exemple**

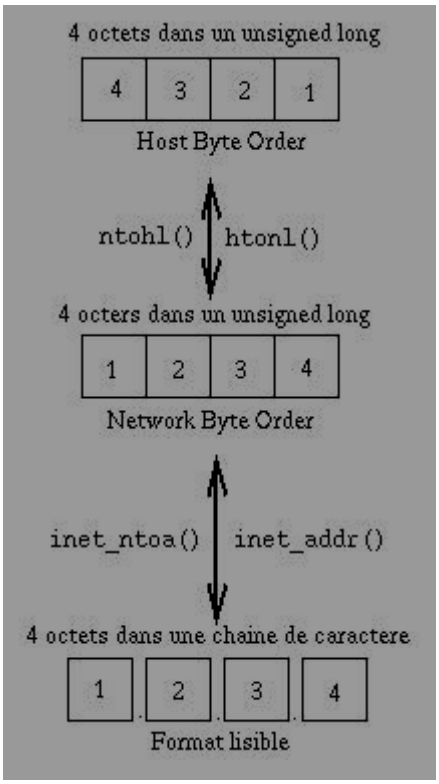
Supposons que l'on connaisse l'adresse IP d'une machine, ces quelques lignes permettent d'en connaitre le nom.

```
struct hostent client;
...
client=gethostbyname("147.94.38.17",sizeof("147.94.38.17"),AF_INET);
printf("La machine s'appelle %s\n",client.h_name);
```

## *13. Adresses machines, adresses réseaux?*

### 13.1 Définitions

Les adresses sont stockées par la machine sous forme d'entier. Malheureusement, toutes les machines ne traitent pas ces entiers de la même façon que le réseau. Ainsi sur une machine, l'adresse est stockée sous la forme du *Host Byte Order* alors que le réseau attend du *Network Byte Order*. En fait certaines machines travaillent en *Network Byte Order* par défaut. Mais la portabilité étant le maître mot en UNIX, l'utilisation des commandes suivantes est vivement recommandée.




La figure schématise les différents formats d'adresse et montre les domaines d'applications des fonctions que nous détaillons dans les paragraphes suivants.

**Note**

Il n'y a aucun rapport avec le DNS ici.

### 13.2 Conversion d'adresses : `htons()` -- `htonl()` -- `ntohs()` -- `ntohl()`

**Synoptique**

```
#include <sys/types.h>
#include <netinet/in.h>

short htons(short int arg) /* Host TO Network Short*/
```

```
long  htonl(long int arg)  /* Host TO Network Long*/
short ntohs(short int arg) /* Network TO Host Short*/
long  ntohl(long int arg)  /* Network TO Host Long*/
```

**Description**

Ces fonctions assurent la conversion entre le format *Network Byte Order* et le format *Host Byte Order*.

**Note**

`sin_family` ne doit pas être converti à la notation réseau, au contraire de `sin_addr` et `sin_port`. En effet `sin_family` est juste utilisée par le noyau et ne voyage pas dans le réseau.

**Exemple**

Voir l'appel à `bind()` page : on convertit le numéro de port au format réseau.

### 13.3 Les adresses IP : `inet_addr()` -- `inet_ntoa()`

**Synoptique**

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_addr(addr)
char *addr; /* adresse au format "xxx.xxx.xx.xx"*/
```

**Description**

Cette primitive convertit une chaîne de caratères en une adresse au format réseau (*Network Byte Order*). On s'en sert pour l'appel à `bind()`. On évite ainsi l'emploi de `htonl()` et `htons()`.

**Exemple**

On veut connecter sock sur l'adresse `147.94.38.17`, une solution pour initialiser l'adresse du socket :

```
struct sockaddr_in sock;
...
sock.sin_addr.s_addr=inet_addr("147.94.38.17");
...
```

**Synoptique**

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>

char *inet_ntoa(addr)
struct in_addr addr; /* adresse au format Network Byte Order*/
```

**Description**

Cette primitive a la fonction inverse de la précédente. Elle permet de rendre lisible une adresse au format réseau. Cette primitive se contente juste de convertir une adresse au format réseau au format dit IP. Elle ne fait en aucun cas appel au serveur de noms.

**Exemple**

Supposons que l'on veuille connaître la provenance d'un client. Le serveur pourra comporter les lignes suivantes.

```
struct sockaddr_in client; /* structure d'adresse d'un client */
...
printf("Connexion provenant du %s\n",inet_ntoa(client.sin_addr));
```

**Note**

L'appel à `htons()` ou `htonl()` est inutile si on utilise ces deux primitives.

## *14. Conclusion*

L'interface des socket permet donc de faire communiquer des processus tournants sur deux ou plusieurs machines distantes. On pourait même imaginer deux programmes tournants sur une même plateforme communiquer grâce aux sockets. Cependant, le type de communication que nous voulons établir entre les processus de la machine serveur ne sont pas aussi simples. Nous ne pourons donc pas nous contenter des sockets pour faire communiquer tous les processus. De plus, la découverte d'autres moyen d'intercommunication est assez importante en elle-même pour que l'on s'interresse à elle.

## 1) TDN°1 : Etude préliminaire : Les sockets

a) Lire le dossier fourni sur les sockets

b) complétez le document réponse N°1 en remplissant les cases vides par le nom des fonctions correspondantes.

c) L'adresse d'une communication possède plusieurs informations réunit dans une structure (chaque champ de la structure représente une information). Complétez le document réponse N°2 en indiquant dans le cadre vide les différentes informations présentent dans une adresse.

d) Combien d'adresses différentes vont être gérés lors d'une communication client serveur, donnez le nom utilisé pour chacune des adresses ou structures utilisées dans le polycopié.

e) remplir le document réponse N°3

# Document réponse N°1

| Action réalisée | Fonction langage C |
|---|---|
| Mise en place du matériel pour établir la communication  | |
| Spécification d'une adresse pour la réception des messages  | |
| Définition de la longueur de la file d'attente des messages  | |
| Attendre les demandes de communication  | |
| Demande de communication et connexion  | |
| Début de la conversation  | |
| Clore la communication  | |

## Document réponse N°2





## Document réponse N°3

| Fonctions | Description | Paramètres d'entrées | Paramètres de sorties |
|---|---|---|---|
| Socket() | | | |
| Bind() | | | |
| Listen() | | | |
| Accept() | | | |
| Read() | | | |

| | | | |
|---|---|---|---|
| **Write()** | | | |
| **Close()** | | | |
| **Connect()** | | | |

## TD N°2 Le programme client tcpsend

On demande lors de ce TD de mettre en place un programme client sur la cible OS9 afin qu'il interroge un programme serveur présent sur un ordinateur PC

Procédure à suivre pour la mise en œuvre du programme client

**1)** Créez sur microware HAWK un nouveau projet tcpsend en incluant le fichier source tcpsend.c et les librairies suivantes (en respectant l'ordre) :
C:\Mwos\OS9\68020\LIB\socklib.l;
C:\Mwos\OS9\68020\LIB\os_lib.l;
C:\Mwos\OS9\68020\LIB\netdb.l;
C:\Mwos\OS9\68020\LIB\sys_clib.l;
C:\Mwos\OS9\68020\LIB\sclib.l;
C:\Mwos\OS9\68020\LIB\clib.l

**2)** Chargez le fichier code machine dans la cible OS9 et complétez le fichier source tcpsend.c

**3)** Chargez un fichier texte dans la cible OS9 en utilisant ws--->FTP32
démarrez---> programmes----> wsftp32
hostname ms9-targetx.egc.es.tpn3
puis double cliquez sur whatsnew.txt

**4)** Dans le répertoire réseau MP12 lancez le programme serveur.exe et notez l'adresse IP du PC

**5)** Lancez telnet et la commande tcpsend 192.9.xxx.xxx whatsnew.txt
ou 192.9.xxx.xxx et l'adresse réseau du PC notée précédemment

**6)** observez avec le programme réseau le transfert du fichier

**7)** Questions

on pourra faire une analyse de trame avec le programme finisar surveyor web demo

**7.1)** Comment simplement en analysant le fichier source tcpsend, peut on affirmer que c'est un programme client?

**7.2)** Dans quel type de trame circule le texte <whatsnew.txt>

```
/* <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< tcpsend.c >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> */
/*
**      Syntax:    tcpsend <adresse IP > <file> programme client
**      Function: envoyer un fichier à travers un socket du type TCP programme client vers serveur
*/

/* Header Files */
#include <stdio.h>
#include <stdlib.h>
#include <types.h>
#include <errno.h>
#include <string.h>
#include <INET/socket.h>
#include <INET/in.h>
#include <INET/netdb.h>

/* Macro Definitions */

#define PORT_NUM  1200
#define INIT          77              /* commands */
#define DATA          78
#define END                   79

/* Type Definitions */
struct data {
        int code, count;
        char data[512];
};

/* Global Variables */

struct sockaddr_in ls_addr;/* paramètres de connexion vers le serveur */
char msgbuf[2048] = {0};

/* Function Prototypes */
void main(int argc, char* argv[], char* envp[]);

void main(int argc, char* argv[], char* envp[])
{
  int s;
        FILE *ifile;
        u_int32 n;
        u_int32 count = 10;
        struct hostent *host;

        /* initialisation des paramètres de connexion vers le serveur
        */
        /*************************************************************************************
        *************************************************************************************
        */
```

```
        ls_addr.sin_family = host->h_addrtype;
        ls_addr.sin_port =  htons(_____________________________);

        /*****************************************************************************************
        *****************************************************************************************
        */

        /* vérification du format de l'adresse IP du serveur
        tapez en ligne de commande mode MSDOS
        tcpsend 192.299.9.3 whatsnew.txt
        mise au bon format pour les paramètres de connexion au serveur
        structure struct sockaddr_in ls_addr;
        */
        /*****************************************************************************************
        *****************************************************************************************
        */

        if ((host = gethostbyname(argv[1])) != NULL) {
                memcpy(&ls_addr.sin_addr.s_addr, host->h_addr, host->h_length);
        } else {
                ls_addr.sin_addr.s_addr = inet_addr(_____________________);
        }

        /*****************************************************************************************
        *****************************************************************************************
        */
        /* ouverture du fichier
        tapez en ligne de commande mode MSDOS
        tcpsend 192.299.9.3 whatsnew.txt
        si erreur à l'ouverture ecrire
        can't open file
        */
        /*****************************************************************************************
        *****************************************************************************************
        */
                if ((ifile = fopen(argv[2],"r")) ==-1) {
                printf("can't open file '%s'\n", _____________________________);                exit(0);
        }

        /*****************************************************************************************
        *****************************************************************************************
        */

while (1) {

        /* Mise en place de la connexion
        s descripteur de socket
        family=AF_INET
```

```
        type=SOCK_STREAM
        protocole=0
        /*si impossibilité d'ouvrir le socket affichez à l'écran can't open socket
        fermez le fichiez et sortir du programme*/
        */
        /*****************************************************************************************
        *****************************************************************************************
        */

                if ((s = socket(_____________________________________)) == -1) {
                        printf("can't open socket\n");
                        close(s);
                        exit(0);
                }

        /* Connecter un client(le programme suivant) au serveur les paramètres de connexion sont contenus dans la
struture ls_addr
        s descripteur de socket
        si impossible de se connecteur au serveur affichez    "connect failed"
        attente de la connexion en imprimant le nombre de tentative de connexionsi connexion impossible après
l'attente affichez      fermez le socket et fermez le fichier
        */
        /*****************************************************************************************
        *****************************************************************************************
        */
                if (connect(s, (struct sockaddr*)&ls_addr, sizeof(ls_addr)) == -1) {
                        printf("connect failed");
                        close(s);
                        fclose(ifile);
                        exit(0);
                }
                break;
        }

        /*****************************************************************************************
        *****************************************************************************************
        */
        printf("Connection established\nSending file '%s'...", *argv); fflush(stdout);

        count = sizeof(msgbuf);
        if ((errno = fread(_____________________________________________)) >count) {
                printf("lecture impossible du fichier \n");
                close(s);
                fclose(ifile);
                exit(0);}
  else{
                if ((errno = write(_____________________________________)) == -1) {
                                printf("socket write error\n");
                                        close(s);
                                        fclose(ifile);
```

```
                                        exit(errno);}
        count = sizeof(msgbuf);
        }
        if (errno ==-1){
                printf("read error on file\n");
                exit(0);
        }

        close(s);
        fclose(ifile);
        printf("sent %u bytes\n", count);
        exit(0);
}
        /*********************************************************/
```

On donne ci-dessous le programme partiel du serveur de la cible OS9 que l'on trouve sur le réseau sous le nom de tcp_recv, complémentez ce programme afin qu'il affiche l'adresse IP de la cible OS9. Compilez et testez ce programme (on pourra utiliser le programme windows client.exe pour le tester)

```
/* <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< tcprecv.c >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> */
/*      Syntax:    tcprecv <file>
**      Function:  receives a file across a stream (TCP) socket
*/

/* Header Files */

#include <stdio.h>
#include <stdlib.h>
#include <types.h>
#include <errno.h>
#include <string.h>
#include <signal.h>
#include "pit.h"

#include <INET/socket.h>
#include <INET/in.h>
#include <INET/netdb.h>

/* Macro Definitions */

#define PORT_NUM   1200

#define INIT        77          /* commands */
#define DATA        78
#define END         79

/* Global Variables */
char msgbuf[20480] = {0};
```

```
char *buff;
int sx;
_PIT_REG  *pit;

void main(int argc, char* argv[])
{
        unsigned char tab[12]="bonjour";
        unsigned char c;
        unsigned char reste=1;
        unsigned char quotient;
        int s;
        int i;
        int size;
        int totbytes = 0;
        u_int32 count = 1;
        struct sockaddr_in ls_addr;
        struct sockaddr_in to;
        setbuf(stdout,NULL);
        setbuf(stdin,NULL);

        /* création du socket pour la connexion avec les clients
        */
        /*****************************************************************************************
        */
        if ((sx = socket(_____________________________________)) == -1) {
                printf("can't open socket\n");
          close(_____________________);
                exit(errno);
        }

        /* initialisation des paramètres de connexion du serveur
        */
        /*****************************************************************************************
        */
        ls_addr.sin_family = AF_INET;
        ls_addr.sin_port = ____________________________________
        ls_addr.sin_addr.s_addr = 0;

        /* initialisation du socket
        */
        /*****************************************************************************************
        */
        if (bind(sx,(struct sockaddr*)&ls_addr,sizeof ls_addr)== -1) {
                printf("can't bind socket\n");
          close(sx);
          exit(errno);
        }

        /* initialisation de la file d'attente une connexion possible
        */
```

```
/*****************************************************************************************
*****************************************************************************************
*/
if (listen(_____________________________) < 0) {
        printf("tcp_listen - failed!\n");
        close(sx);
        exit(errno);
}
        size = sizeof(struct sockaddr_in);

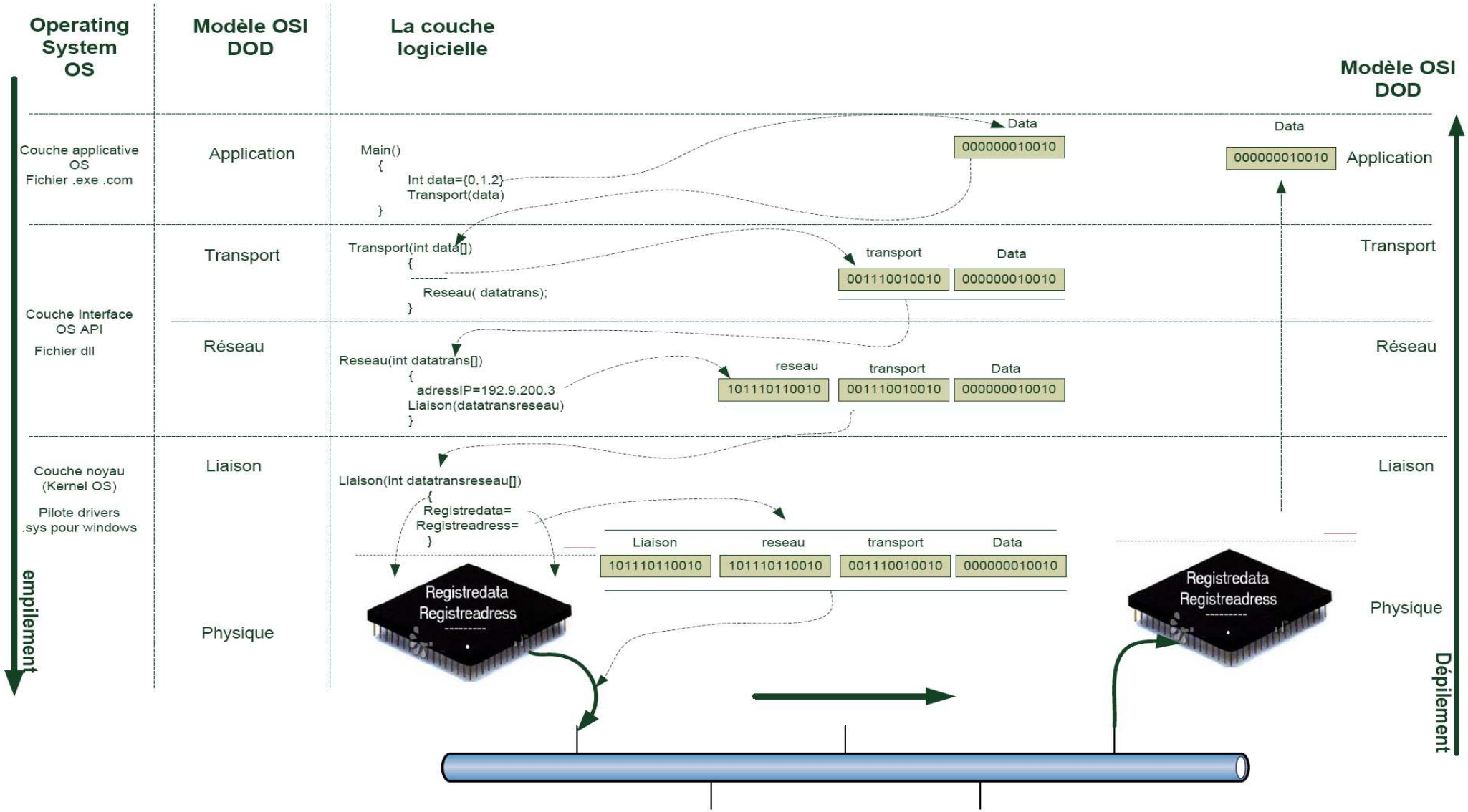
/* attendre la communication
*/
/*****************************************************************************************
*****************************************************************************************
*/
        if ((s = accept (sx,(struct sockaddr*)&to,&size)) < 0) {
        printf("can't accept\n");
        close(s);
        close(sx);
        exit(errno);
}
close(sx);
printf("connected to %s port %d\n",
        inet_ntoa(to.sin_addr), to.sin_port);

while (count) {
        count = (sizeof(msgbuf))-1;

/* Lecture des données issues du client
*/
/*****************************************************************************************
*/

                if ((errno =read(_____________________________________)) <0) {
                printf( "can't recv (cnt=%d)\n");
                close(s);
                exit(errno);
        } else if (count == 0) {
                break;
        } else {
                msgbuf[errno]=0;
                printf("message recu \n %s",msgbuf);
                write(_____________________________________________________);
                }
                totbytes += count;
        }
close(s);
printf("read %d bytes\n",totbytes);

exit(0);}
```


Operating System OS
Modèle OSI DOD
La couche logicielle
Modèle OSI DOD
Couche applicative OS
Fichier .exe .com
Application
Main()
{
Int data={0,1,2}
Transport(data)
}
Data
000000010010
Data
000000010010
Application
Transport
Transport(int data[])
{
Reseau( datatrans);
}
transport
Data
001110010010
000000010010
Transport
Couche Interface OS API
Fichier dll
Réseau
Reseau(int datatrans[])
{
adressIP=192.9.200.3
Liaison(datatransreseau)
}
reseau
transport
Data
101110110010
001110010010
000000010010
Réseau
Couche noyau (Kernel OS)
Pilote drivers .sys pour windows
Liaison
Liaison(int datatransreseau[])
{
Registredata=
Registreadress=
}
Liaison
reseau
transport
Data
101110110010
101110110010
001110010010
000000010010
Liaison
Registredata
Registreadress
Registredata
Registreadress
Physique
Physique
empilement
Dépilement