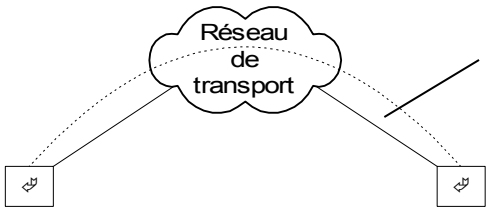
IUT 'A' Paul SABATIER
Dpt Informatique
S4

M4102C : Programmation répartie

TD1: l'API Socket

1- Concepts de base

On veut disposer d'un environnement commun de communication inter-processus.



Les processus communiquent entre eux.

Notion de domaine :
- architecture de protocole utilisée (pile incluant Transport et les couches basses),
- organisation du réseau de Transport.

2 utilisations immédiates :
- communication sous UNIX
- communication sous TCP/IP

Sous Windows : winsock 2.0 :
- API
- SPI, ATM, ...

Définition d'un Socket

C'est une extrémité d'un canal de communication entre deux processus qui permet une communication bidirectionnelle point à point : notion de **prise**.
Une prise **Socket** est définie localement à l'aide d'un triplet (domaine, type, protocole) :

* domaine :
  caractérise l'espace de communication (Unix, Internet, Decnet, Appletalk…).
  - Unix : nom = chemin d'accès à un fichier.
  - Internet : nom = adresse IP du site + numéro de port.

* type :
  définit le service de communication.
  - mode connecté : communication fiable, pas de duplication, pas de perte, ...
  - mode sans connexion : pas de côntrole, …

* protocole :
  fichier **/etc/protocols** (ensemble des protocoles utilisés).

2- Services et protocoles du domaine Internet

Il existe en C deux API pour manipuler les adresses réseaux et l'accès au DNS :

- l'ancienne API, représentée par la fonction **gethostbyname**, qui ne supporte que IPv4
- la nouvelle API, représentée par **getaddrinfo**, qui supporte IPv4 et IPv6 de façon uniforme permettant à un même programme de fonctionner selon ces deux modes sans modification majeure.

Nous étudierons ici la nouvelle API qui est décrite dans le RFC 3493.
Une machine est caractérisée par une adresse IP ou un nom. Les services considérés comme publics sont listés dans le fichier **/etc/services** avec le format suivant :

```
nom service  n° port / nom protocole [alias]
……
ftp-data 20/tcp
ftp 21/tcp
telnet 23/tcp
```

La fonction **getaddrinfo()** permet d'obtenir toutes informations réseau en faisant appel au DNS si nécessaire. La fonction **getnameinfo**() effectue le travail inverse. A partir d'une adresse IP/numéro de port, elle permet d'obtenir les noms de machine et de service.
Ces deux fonctions sont définies dans **<netdb.h>**.

```
int getaddrinfo(const char *hostname, const char *servname, const struct addrinfo *hints,
                struct addrinfo **res);
```

hostname = le nom de la machine ou son adresse IP à résoudre. Si hostname est nul, il s'agit de la machine locale.
servname = le nom ou le numéro de port. Si servname = 0 dans le cas d'un serveur, le système choisira dynamique un port > 1024.
hints = permet de contrôler le traitement effectué par la fonction getaddrinfo.
res = la valeur retournée est 0 en cas de succès, et -1 en cas d'erreur.

Pour décoder le code d'erreur, il faut utiliser la fonction suivante :

```
const char *gai_strerror(int errcode);
```

Dans cette nouvelle, API, l'ensemble des adresses d'un hôte est représenté par une liste chaînée de structures de type **struct addrinfo** :

```
struct addrinfo {
   int ai_flags; /* drapeau en entrée : AI_PASSIVE si appel dans un serveur */
   int ai_family; /* famille de protocole pour la socket: AF_INET (IPv4) AF_INET6 ou
                     AF_UNSPEC  (v4 ou v6) */
   int ai_socktype;    /* type de socket: SOCK_STREAM (TCP), SOCK_DGRAM (UDP) ou
                          SOCK_RAW */
  int ai_protocol;      /* protocole pour la socket : 0 (un seul protocole existe en pratique pour une
                           prise socket type */
   socklen_t ai_addrlen; /* longueur de l'adresse socket */
   struct sockaddr ai_addr; /* un pointeur sur une adresse socket (struct sockaddr_in ou
                               sockaddr_in6) */
   char ai_canonname; /* nom canonique */
   struct addrinfo ai_next; /* pointeur sur l'élément suivant dans la liste */
};
```

La liste chaînée obtenue par **getaddrinfo** doit être libérée à l'aide de la fonction :

```
void freeaddrinfo(struct addrinfo *res).
```

La fonction **getnameinfo**() est la réciproque de **getaddrinfo**() : elle convertit une adresse de socket en un hôte et un service correspondants, de façon indépendante du protocole.

int **getnameinfo**(const struct sockaddr **sa*, socklen_t *salen*, char **host*, size_t *hostlen*, char **serv*, size_t *servlen*, int *flags*);

Le code suivant essaie d'obtenir le nom de l'hôte ainsi que le nom du service sous forme numérique, et ce, pour une adresse de socket donnée. Nulle référence à une quelconque famille d'adresse n'est codée en dur.

```
struct sockaddr_storage *sa; /* input */
socklent_t len; /* input */
char hbuf[NI_MAXHOST], sbuf[NI_MAXSERV];

if (getnameinfo((struct sockaddr_storage *)sa, len, hbuf, sizeof(hbuf), sbuf, sizeof(sbuf),
                    NI_NUMERICHOST | NI_NUMERICSERV) == 0)
    printf("host=%s, serv=%s\n", hbuf, sbuf);
```

La version suivante vérifie si l'adresse de la socket peut se voir associer un nom.

```
struct sockaddr_storage *sa; /* input */
socklent_t len; /* input */
char hbuf[NI_MAXHOST];
if (getnameinfo((struct sockaddr_storage *)sa, len, hbuf, sizeof(hbuf), NULL, 0, NI_NAMEREQD))
                printf("could not resolve hostname");
else printf("host=%s\n", hbuf);
```

3. Les primitives associées aux Sockets

3.1- Les étapes d'une communication

Les primitives utilisent le SGF UNIX.
La prise **Socket** est représentée par un **descripteur de fichier** identifié par un numéro interne (comme pour les fichiers ordinaires) et une **structure socket**.
Les étapes d'une communication sont les suivantes :

① Création d'un Socket + demande de connexion lorsque le service est en mode connecté.
② Emission (écriture) et réception (lecture) de données.
③ Déconnexion si le service est en mode connecté et destruction du Socket.

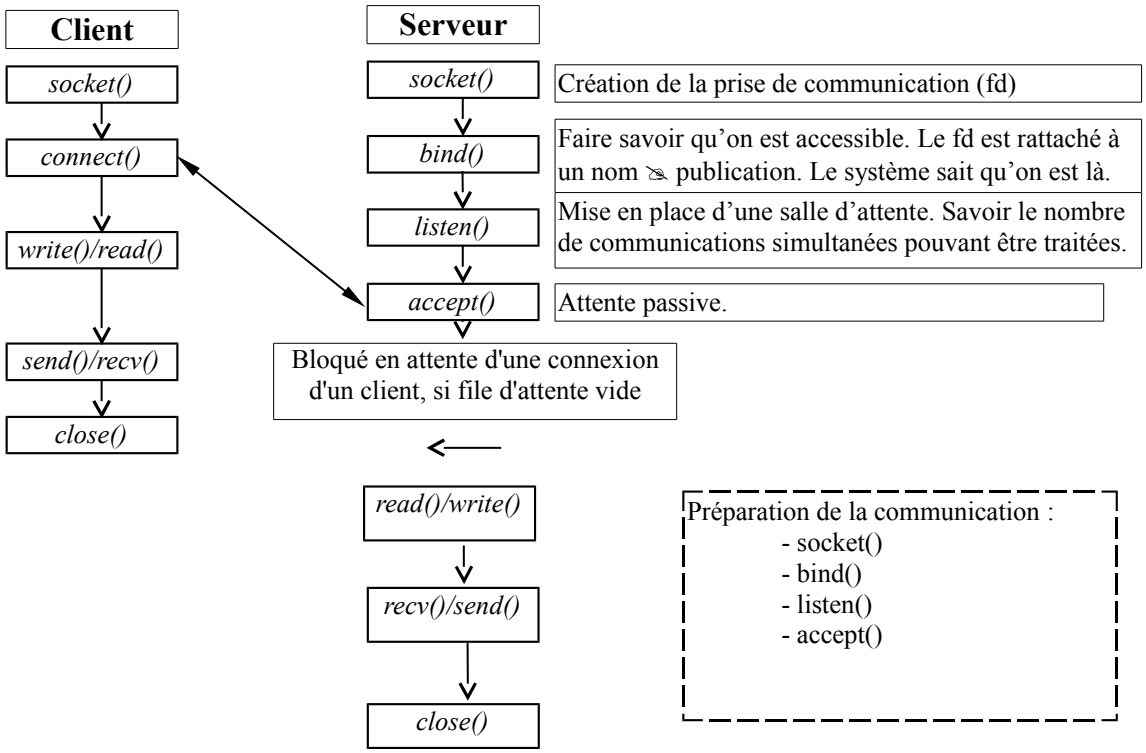
3.2- Fichiers d'en-tête

Ils sont associés au concept (quelque soit le domaine dans lequel on travaille).

<sys/types.h>	types basiques pour tous les appels système
<sys/socket.h>	constantes identifiant les domaines, types et protocoles réseau + prototypes des fonctions de l'API socket
<netinet/in.h>	structures propres au **domaine Internet**
<netdb.h>	structures de données utilisées par les fonctions de gestion des BD réseau
<arpa/inet.h>	prototypes de fonctions de conversions d'@IP

3.3- Services de communication en mode connecté

La communication entre deux processus se fait selon le schéma **client/serveur**.
Dans le **mode connecté**, le client fait une ouverture active (côté initiateur) alors que le serveur fait une ouverture passive (côté récipiendaire).
La connexion est établie de la manière suivante entre les deux processus :



3.4- Nom externe d'une prise socket

Le **nom externe** d'une prise socket (appelé aussi **adresse** de la socket) est décrit par la structure générique **sockaddr** qui provient de l'ancienne API.
La structure réellement utilisée pour stocker les informations réseau dépend du type de protocole réseau (sockaddr_in pour IPv4 et sockaddr_in6 pour IPv6).
La nouvelle API permet de ne pas manipuler directement cette structure afin que le code C soit indépendant de la version IP utilisée.
Par contre, la structure **sockaddr_in6** a une longueur de **28** octets, et est donc plus grande que le type générique **struct sockaddr**. Afin de faciliter la tâche des développeurs, une nouvelle structure de données, **struct sockaddr_storage**, a été définie.
Celle-ci est de taille suffisante afin de pouvoir prendre en compte tous les protocoles supportés et alignée de telle sorte que les conversions de type entre pointeurs vers les structures de données d'adresse des protocoles supportés et pointeurs vers elle-même n'engendrent pas de problèmes d'alignement.
Il faut donc définir des variables de type **struct sockaddr_storage** et faire de la conversion de type pour garder cette indépendance vis à vis de la version d'IP.

3.5- Mode connecté : primitives socket

Création d'un Socket :

int **socket** (int *domaine*, int *type*, int *protocole*) ;

domaine	= AF_INET	/* AF = Address Form */
type	= SOCK_STREAM	/* type flot */
protocole	= 0	/* automatique */

retourne : le descripteur *fdSock* du Socket ou **-1** en cas d'erreur.

Publication d'une Socket :

int **bind** (int *fdSock*, struct sockaddr * *adrSock*, int *lgNom*) ;

fdSock	= descripteur du Socket retourné parla primitive **socket**
adrSock	= pointeur sur la structure contenant le *nom externe de la prise Socket locale à publier* **(paramètre en entrée)**
lgNom	= *longueur* du nom externe pour le domaine Internet *lgNom* = **sizeof** (struct sockaddr_in)

retourne : **0** si la publication a réussi ou **-1** en cas d'erreur.

Pour que le serveur utilise un numéro de port fixe, on initialise le numéro de port de l'adresse de socket au moment de l'appel de la fonction **getaddrinfo**() en spécifiant l'argument **servname.**

Si cet argument est égal à **0**, le noyau, en effectuant le **bind**, allouera alors un numéro de port **non utilisé** (>1024).

On appellera ensuite la primitive **getsockname()** pour récupérer la structure **sockaddr** mise à jour, et la primitive **getnameinfo**() pour obtenir la valeur de ce numéro de port.

int **getsockname** (int *fdSock*, struct sockaddr * *adrSock*, int * *adrLgNom*) ;

fdSock	= descripteur du Socket retourné parla primitive **socket**
adrSock	= pointeur sur la structure contenant le *nom externe de la prise Socket* (**paramètre en sortie**)
adrLgNom	= pointeur sur la variable contenant la *longueur* du nom externe (**paramètre en mise-à jour)** avant l'appel il faut initialiser cette variable, par exemple : *lgNom* = **sizeof** (struct sockaddr_in) il faut passer en paramètre l'adresse de la variable (&*lgNom*) en sortie *lgNom* reçoit la longueur effective du nom externe de la prise socket

retourne : **-1** en cas d'erreur ou **0** sinon.

Etablissement de la connexion :

***Côté CLIENT, ouverture* Active**

int **connect** (int *fdSock*, struct sockaddr * *adrSockServ*, int *lgNom*) ;

fdSock = descripteur du Socket retourné parla primitive **socket**
adrSockServ = pointeur sur la structure contenant le *nom externe de la prise Socket du serveur auquel le Client veut se connecter*
lgNom = *longueur* du nom externe
Pour le domaine Internet *lgNom* = **sizeof** (struct sockaddr_in)
retourne : **0** si la connexion s'est correctement réalisée ou **-1** en cas d'échec.

Côté SERVEUR, ouverture* Passive *:

int **listen** (int *fdSock*, int *lgFile*) ;

fdSock = descripteur du Socket retourné parla primitive **socket**
lgFile = nombre de demandes de connexions à mémoriser
/* taille de la file d'attente (socket de rendez-vous) */

retourne : **-1** en cas d'erreur ou **0** sinon.

int **accept** (int *fdSock*, struct sockaddr * *adrSockCli*, int * *adrLgNom*) ;

fdSock = descripteur du Socket retourné parla primitive **socket**
adrSockCli = pointeur sur la structure contenant le *nom externe de la prise Socket du Client qui vient de se connecter* (**paramètre en sortie**)
adrLgNom = pointeur sur la variable contenant la *longueur* du nom externe (**paramètre en mise-à jour)**
avant l'appel il faut initialiser cette variable, par exemple :
lgNom = **sizeof** (struct sockaddr_in)
il faut passer en paramètre l'adresse de la variable (&*lgNom*)
en sortie *lgNom* reçoit la longueur effective du nom externe de la prise socket du Client

retourne : le **descripteur d'un nouveau socket** (socket de **communication**) créé par le système pour la communication avec le client (notion de multiplexage)
ou **-1** en cas d'erreur.

Communication :

int **read** (int *fdSock*, char **tampon*, int *nbOct*) ;

fdSock = descripteur du Socket retourné parla primitive **socket** (côté Client)
ou descripteur du Socket retourné parla primitive **accept** (côté Serveur)
tampon = adresse de la zone mémoire de réception
nbOct = taille de la zone mémoire de réception
retourne : le nombre d'octets effectivement lus (≤ *nbOct)*
ou **0** si la communication est coupée (socket distant fermé)
ou **-1** en cas d'erreur.

Remarque : l'opération **read()** sur un socket est **bloquante**.

int **write** (int *fdSock*, char **tampon*, int *nbOct*) ;

fdSock = descripteur du Socket retourné parla primitive **socket** (côté Client)
ou descripteur du Socket retourné parla primitive **accept** (côté Serveur)
tampon = adresse de la zone mémoire d'émission
nbOct = nombre d'octets à émettre.
retourne : le nombre d'octets effectivement transmis ou **-1** en cas d'erreur.

int **recv** (int *fdSock*, char **tampon*, int *nbOct*, int *indics*) ;

int **send** (int *fdSock*, char **tampon*, int *nbOct*, int *indics*) ;

Ces primitives permettent de prendre en compte des **particularités** lors de la communication, ce que ne font pas les primitives **read()/write().**
Les indicateurs *indics* ci-dessous indiquent le service exigé au niveau du service de transport :

indics = **MSG_OOB** /* Message Out Of Band (urgence)*/
MSG_PEEK /* Lecture non destructive du message */
MSG_DONTROUTE /* Emission sans exploiter les possibilités de routage */

Fin de communication :

int **close** (int *fdSock*) ;

fdSock = descripteur du Socket pour lequel on ferme la communication
retourne : **-1** en cas d'erreur ou **0** sinon.

3.6- Fonctions de travail

Traitement de zones de mémoire

int **bzero** (char **ZONE*, int LG) ;/* Remet à zéro *LG* octets à partir de l'adresse *ZONE* */

int **bcopy** (char **SOURCE*, char **DESTINATION*, int *LG*) ;

/* Transfère *LG* octets de la zone *SOURCE* vers la zone *DESTINATION* */

int **bcmp** (char **ZONE1*, char **ZONE2*, int *LG*) ; /* Compare 2 zones mémoire et retourne 0 si égalité */

Fonctions de conversion

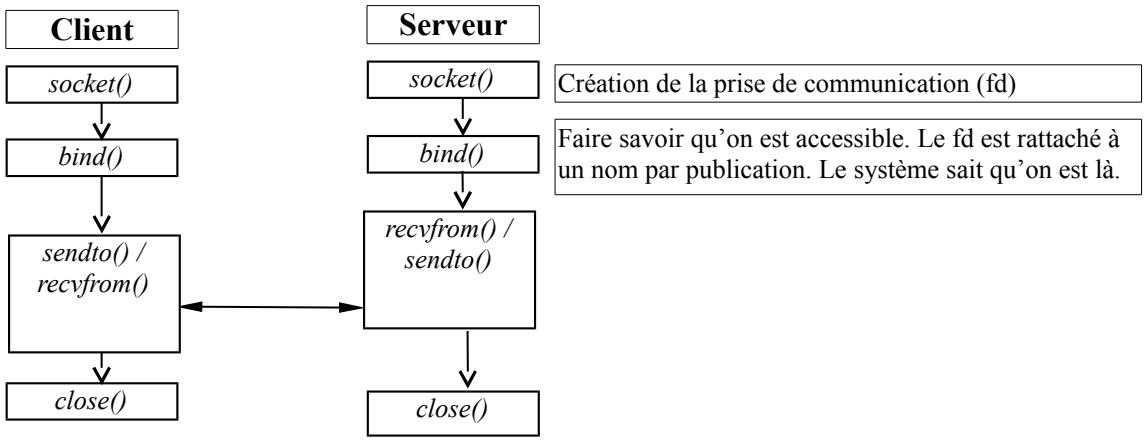
Pour communiquer entre systèmes utilisant des représentations internes différentes :
- Octet de poids fort en tête du mot mémoire (**Big endians**)
- Octet de poids faible en tête du mot mémoire (**Little endians**)

HOST TO NETWORK	NETWORK TO HOST
format machine → format réseau	format réseau → format machine
htons (short int)	**ntohs** (short int)
htonl (long int)	**ntohl** (long int

Ces fonctions ne sont utiles que si l'on manipule les champs des structures **sockaddr_in** ou **sockaddr_in6**.

3.7- Services de communication en mode sans connexion

La communication entre deux processus se fait selon le schéma **client/serveur**.
Dans le **mode non connecté**, le client fait une ouverture active (côté initiateur) alors que le serveur fait une ouverture passive (côté récipiendaire).
La connexion est établie de la manière suivante entre les deux processus :



3.8- Nom externe d'une prise socket dans IPv4

Le **nom externe** d'une prise socket (appelé aussi **adresse** de la socket),est décrit par la structure **sockaddr_in** suivantet :

```
struct  sockaddr_in
{   short           sin_family ;     /* domaine : AF_INET */
    u_short         sin_port ;       /* numéro de port */
    struct in_addr  sin_addr ;       /* @ IP */
    char            sin_zero[8] ;    /* pour compléter à 16 octets : réservé au système */
} ;
```

Le champ **sin_addr** contenant une adresse IP, peut être manipulé comme un entier long 32 bits ou comme 2 entiers courts 16 bits ou comme 4 octets. Le format entier long 32 bits est celui qui est utilisé le plus utilisé d'où la structure **in_addr** :

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

3.9- Mode sans connexion : primitives de lecture/écriture

Communication en mode message :

int **recvfrom**(int sock, char* buffer, int tbuf, int attrb, struct sockaddr *addsrc, socklen_t *taille)

Lecture de données dans la socket **sock**, et stockage dans le tableau **buffer**
L'adresse de l'**émetteur** retournée dans **addsrc**

int **sendto**(int sock, char* buff, int tbuf, int flag, struct sockaddr *addrdst, socklen_t taille)

Envoi par la socket **sock** du contenu du tableau **buff** à l'adresse **addrdst**
Taille limitée à la taille d'un paquet
Champ flag :

MSG_OOB	: données hors bande (en urgence)
MSG_PEEK	: Lecture sans modification de la file d'attente

3.10- Options d'une socket : Lecture et Ecriture

Ces primitives permettent d'installer-modifier ou de récupérer les paramètres (caractéristiques) d'une prise *socket :*

int **getsockopt** (int sock, int couche, int cmd, void *val, socklen_t *taille)

Lecture des options d'une prise *socket*
couche : couche de protocole : SOL_SOCKET (socket-level option), IPPROTO_IP, IPPROTO_TCP),
cmd : commande utilisant le champ de données **val**

SO_TYPE	Type de socket
SO_RCVBUF	Taille du buffer de réception
SO_SNDBUF	Taille du buffer d'émission
SO_ERROR	Valeur d'erreur de la socket

int **setsockopt** (int sock, int couche, int cmd, void *val, socklen_t taille)

Modification des options d'une prise *socket*
cmd : commande utilisant le champ de données **val**

SO_BROADCAST	Autorisation de paquets broadcast
IP_ADD_MEMBERSHIP	Autorisation d'une requête multicast
IP_DROP_MEMBERSHIP	Résiliation de l'adhésion
SO_REUSEADDR	Autorisation de réutiliser une adresse déjà affectée

Retour pour les deux primitives :
0 : en cas de succès
-1 : en cas d'échec avec code d'erreur dans la variable **errno**

3.11- Services de communication en mode multicast

Adresses multicast : 224.0.0.0/4

TTL et paquets multicast

```
unsigned char ttl;
if (setsockopt(sock,IPPROTO_IP,IP_MULTICAST_TTL,&ttl,sizeof(ttl))==-1)
  {
    perror("setsockopt: IP_MULTICAST_TTL");
  }
```

L'émission d'un paquet en multidiffusion se fait en fonction du seuil relatif à la portée de ce paquet dans le réseau :

0	restricted to the same host
1	restricted to the same subnet
32	restricted to the same site
64	restricted to the same region
128	restricted to the same continent
255	unrestricted

Emission d'un paquet multicast

Appel à sendto() avec une adresse destination multicast

Réception d'un flux multicast

La réception d'un flux multicast nécessite l'adhésion à **un groupe multicast**.
L'appel à **setsockopt**() est alors nécessaire :

```
struct ip_mreq mreq;
if (setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq,sizeof(mreq))==-1)
  {
      perror("setsockopt: IP_ADD_MEMBERSHIP");
  }
```

La structure ***ip_mreq*** est définie de la manière suivante :

```
struct ip_mreq {
   struct in_addr imr_multiaddr;     /* groupe multicast à rejoindre */
   struct in_addr imr_interface;     /* interface à utiliser : INADDR_ANY
                                        pour ne pas se préoccuper de l'interface physique */
}
```