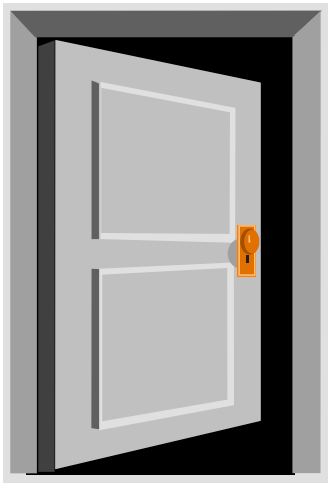
PROGRAMMATION RESEAUX

« API SOCKET UDP ET TCP (en langage C) »

SUPPORT DE COURS

Christophe CHASSOT
INSA Toulouse / LAAS-CNRS
e-mail : chassot@insa-toulouse.fr



3MIC / IMACS 2024/25
INSA Toulouse

Plan du cours

1. Introduction 3

1.1. Services de Transport dans l'Internet 3

1.2. Fonctions des protocoles TCP et UDP 5

1.3. L'API socket 6

2. Définition et adresse d'un socket 8

2.1. Définition d'un socket 8

2.2. Adressage d'un socket 11

3. Manipulation des socket du domaine Internet 13

3.1. Création d'un socket 13

3.2. Destruction d'un socket 14

3.3. Adressage d'un socket 15

4. Communication par socket UDP 28

4.1. Principe général 28

4.2. Primitives d'émission et de réception 29

5. Communication par socket TCP 33

5.1. Principe général 33

5.2. Phase d'établissement de connexion 35

5.3. Phase de transfert de données 44

5.4. Phase de fermeture d'une connexion 48

6. Fichiers à inclure en entête des programmes 49

* *

*

1. Introduction

1.1. Services de Transport dans l'Internet

- **2 principaux types de service (offerts par UDP et TCP)**

– assurant chacun un transfert de données

- de **processus applicatif** à **processus applicatif**
- **bidirectionnel**, i.e. : communication possible dans les deux sens

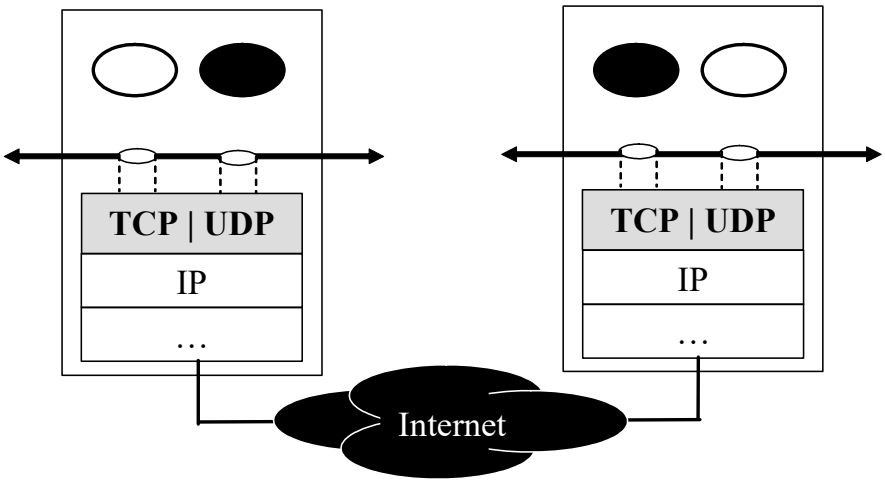
– mais garantissant une QoS différente selon qu'ils sont :

- **avec connexion → TCP**
  - établissement d'une connexion entre processus applicatifs distants avant échange des données, et :

garantie de QoS = ordre total + fiabilité totale

- **sans connexion→ UDP**
  - échange hors connexion des données entre proc. applicatifs, et :

garantie de QoS = rien : ni ordre, ni fiabilité



- **2 modes de transfert d'une donnée applicative D**

– **le mode message** (ou **datagramme**) → **UDP**

- du point de vue du service de Transport :

  - D est considérée comme un message indépendant des messages déjà soumis ou à venir

↓

la limite entre 2 données soumises consécutivement doit être préservée lors de leur délivrance à l'user récepteur

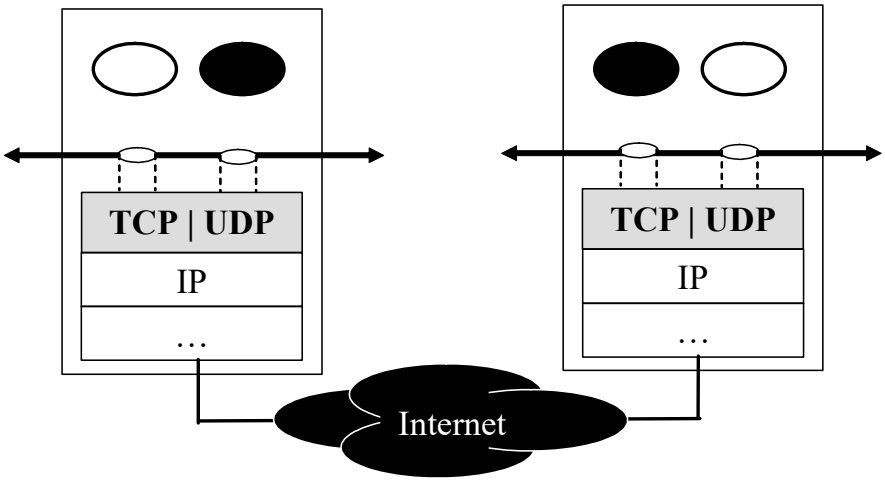
– **le mode flux d'octets → TCP**

- du point de vue du service de Transport :

  - D est considérée comme faisant partie d'un unique flux d'octets

↓

la limite entre 2 données soumises consécutivement n'a pas à être préservée lors de leur délivrance à l'user récepteur



1.2. Fonctions des protocoles TCP et UDP

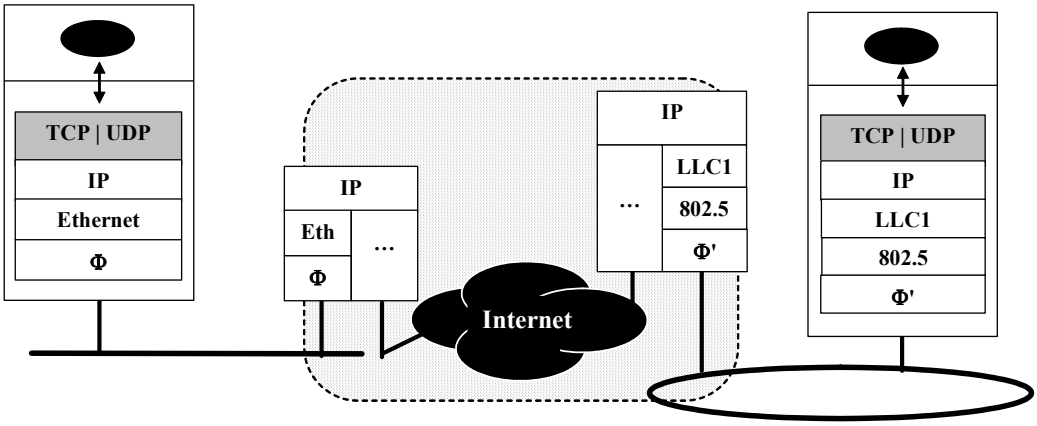
- **Caractéristiques du service sous-jacent (IP)**

– service (de bout en bout) sans connexion (mode datagramme)

→ pertes et/ou déséquencements possibles des PDU Transport

– un seul point d'accès par protocole de Transport

– pas de limite sur la taille des PDU Transport



- **Fonctions attendues d'un protocole de Transport**

– **garantissant ordre et fiabilité → TCP**

- MUX/DMUX des canaux TCP sur canal IP
- contrôle des pertes :
  - reprise des pertes + contrôle de flux → voir 4ème année
  - contrôle de congestion (des routeurs) → voir 4ème année
- contrôle de l'ordre : intégré dans la reprise des pertes
- + mise en relation des applications avant le début des échanges

– **ne garantissant rien (ni ordre ni fiabilité) → UDP**

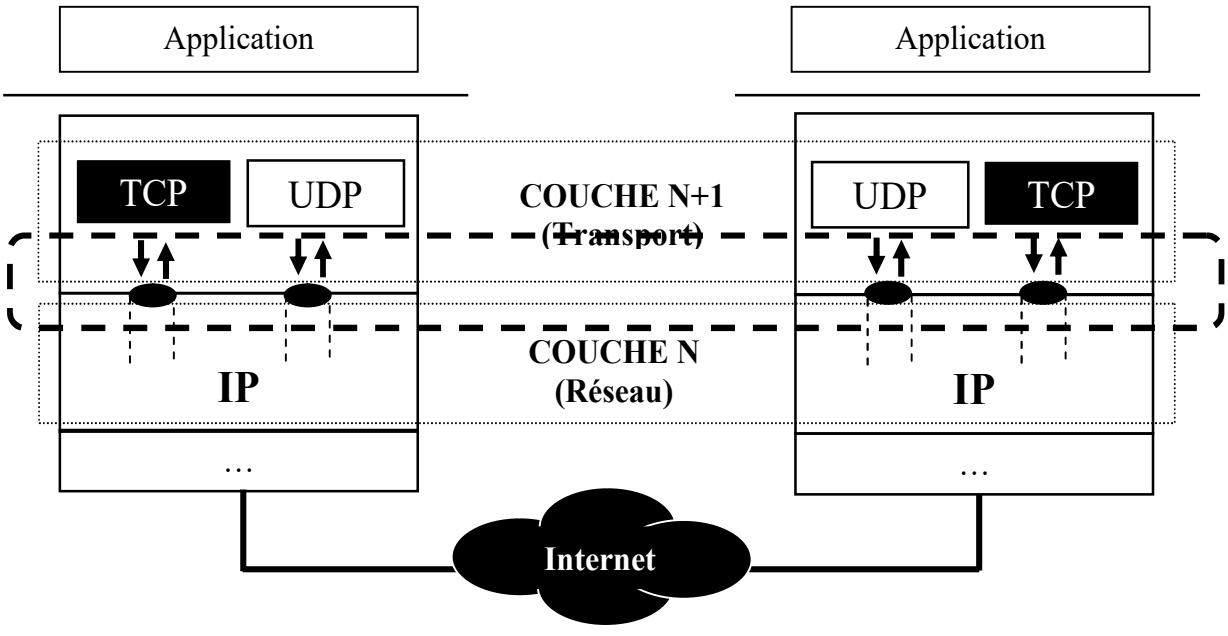
- MUX/DMUX des canaux UDP sur canal IP

1.3. L'API socket

- **A tout service N est associée une « interface de service »**
  - permettant au(x) protocole(s) P_i de niveau N+1
    - d'accéder au(x) service(s) disponible(s) au niveau N
- **Conceptuellement**
  - Une interface de service de niveau N est caractérisée par :
    - un ou plusieurs « **points d'accès au service** »

      SAP (N) : Service Acces Point (N)

      - représentant chacun l'extrémité d'un canal
      - identifiés par une « **adresse de SAP** »
    - un ensemble de « **primitives de service** »
      - et de **règles d'enchaînement de ces primitives**
  - **Ex**



- **Parmi ces interfaces**

– les interfaces de programmation applicative

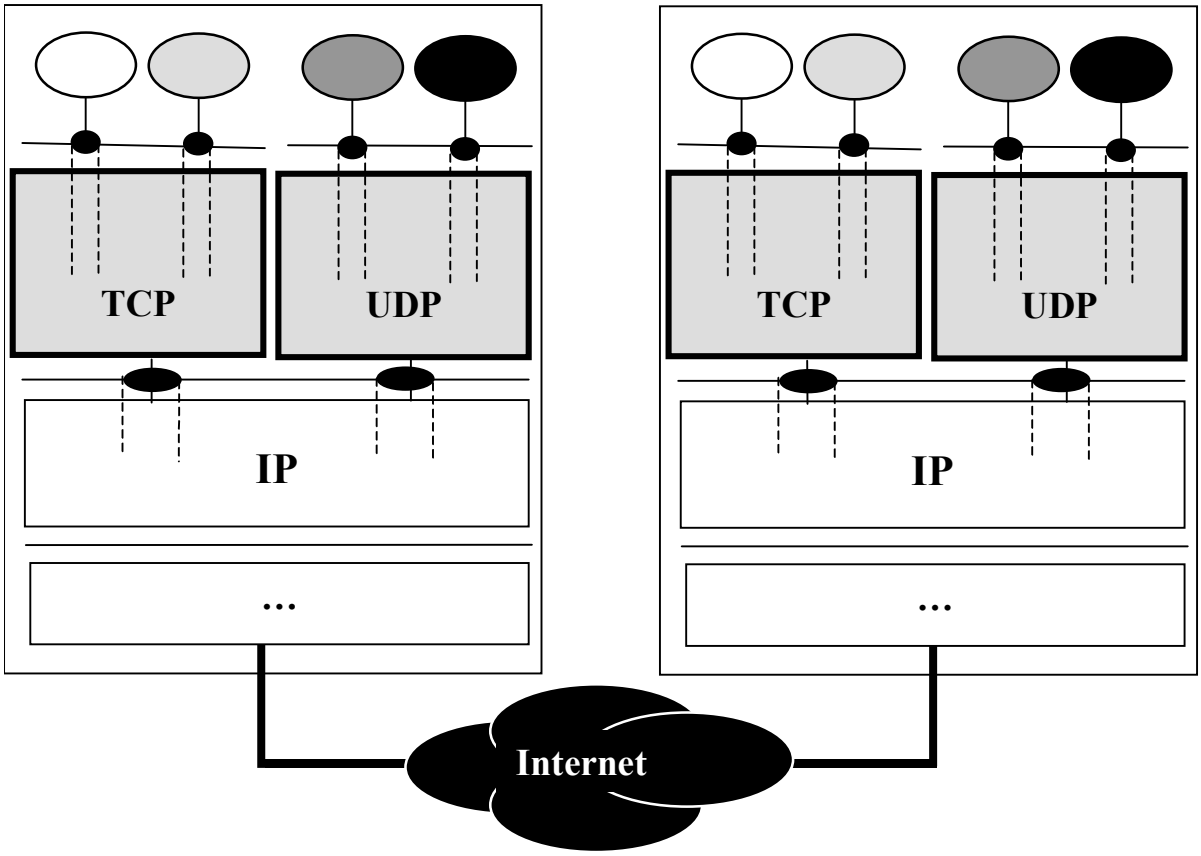
⇔ **« API : Application Programing Interface »**

- librairies dont dispose le programmeur d'une application pour émettre/recevoir des données
- permettent de réclamer la création d'un **canal dédié** aux échanges de données de l'application, puis de l'utiliser

– 2 niveaux d'API

- API de niveau Liaison → réseau purement local
- **API de niveau Transport** → réseau commuté ou Internet

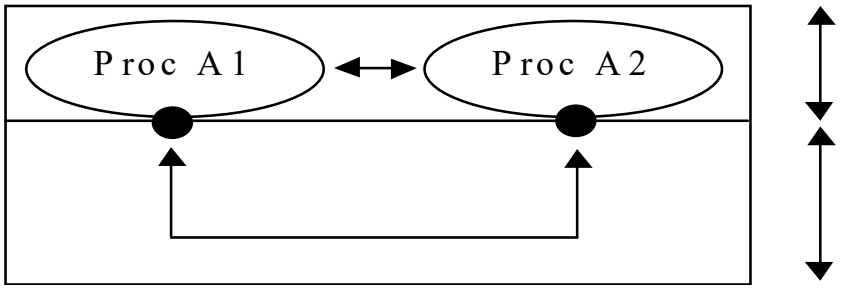
la + utilisée (de base) : « **l'API socket** »



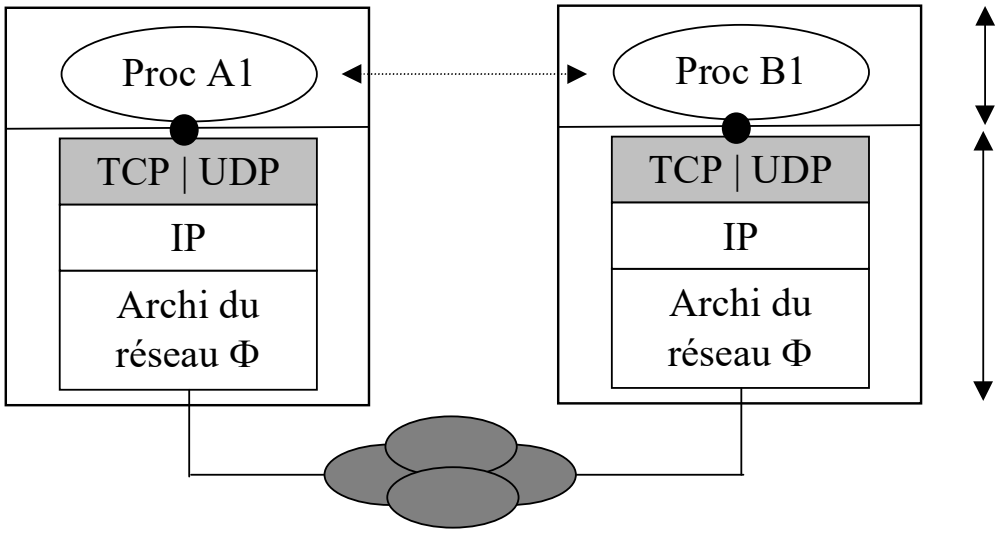
2. Définition et adresse d'un socket

2.1. Définition d'un socket

- **Socket**
  - Point d'accès à un service de transfert de données par lequel un processus peut émettre ou recevoir des données
- **Pour que 2 processus communiquent**
  - Nécessité de créer au moins un socket par processus
  - 2 cas de figure :
    - **Communication « intra machine »**



    - **Communication « inter machines » (via un réseau)**

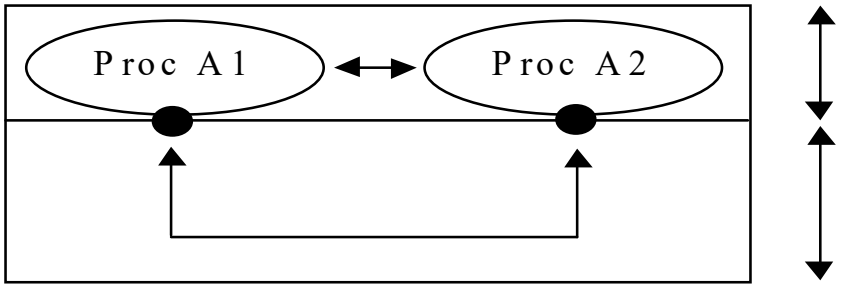


- **Paramètres à préciser lors de la création d'un socket**

  – **1er paramètre = <u>domaine</u>** d'utilisation du socket

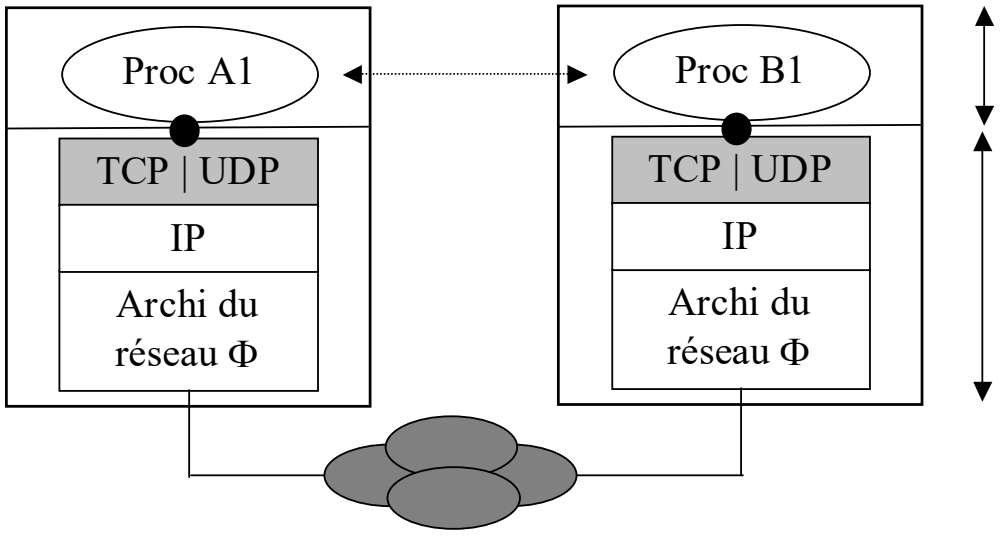
  - si communication intra-machine

    - **domaine** = Unix (AF_UNIX), …



  - si communication inter-machines

    - **domaine** = Internet (AF_INET), DECnet, AppleTalk, …



  – **2ème paramètre = <u>adresse</u>** du socket dans son domaine

  - nécessaire pour que le socket puisse être identifié de l'extérieur

    - cf. section 2.2 : Adressage d'un socket

– **3ème paramètre** = **<u>type d'utilisation</u>** du socket

- **en mode non connecté** (ou « *datagram* »)
  - pas d'établissement de connexion
  - aucune garantie d'ordre ni de fiabilité
  - préservation de la limite des messages
- **en mode connecté** (ou « *stream* »)
  - établissement d'une connexion entre processus communicants avant échange des données
  - garantie d'ordre et de fiabilité
  - pas de préservation de la limite des messages

➢ **Rmq**

▪ Pour être utilisé en mode non connecté

– un socket doit être déclaré de type **SOCK_DGRAM**

- lors de sa création (∀ son domaine)

▪ Pour être utilisé en mode connecté

– un socket doit être déclaré de type **SOCK_STREAM**

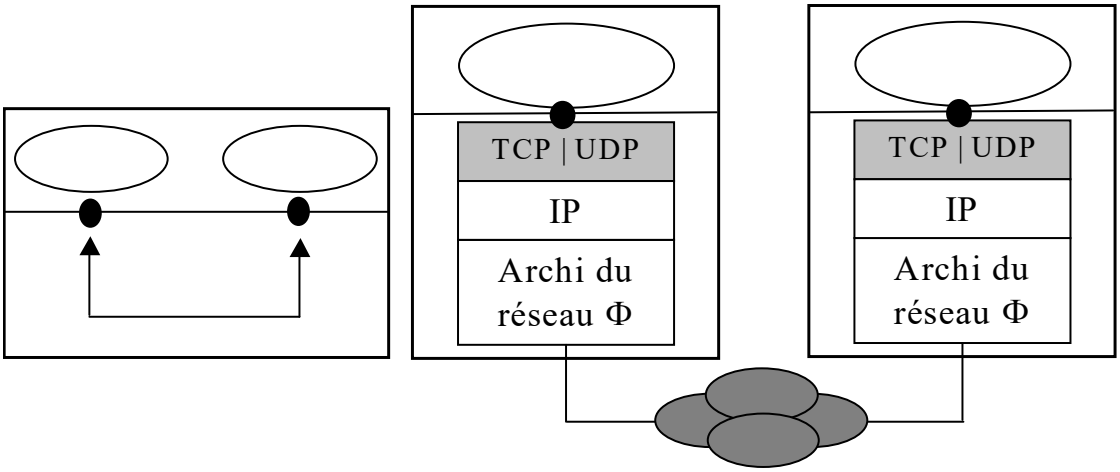
- lors de sa création (∀ son domaine)

▪ 2 processus ne peuvent communiquer que s'ils possèdent des sockets du **même domaine** et fonctionnant selon le **même mode**

2.2. Adressage d'un socket

- **Pour qu'un socket puisse être désigné par un processus distant**
  - Obligation pour le processus créateur du socket de lui attribuer :
    - une représentation externe (« **adresse** ») utilisable de l'extérieur

      - plusieurs domaines de com. → plusieurs formats d'adresse
        - adresse Unix, adresse Internet, …



- **Adressage d'un socket dans le domaine Unix**
  - Dans le domaine Unix :
    - l'@ d'un socket est composée d'un **nom de fichier**
  - Structure de l'@ d'un socket du domaine Unix

```
struct sockaddr_un {
    short sun_family ;      /* = AF_UNIX pour le domaine Unix */
    char sun_path[108] ;    /* = nom de fichier */
} ;
```

- **Adressage d'un socket dans le domaine Internet**

– Dans le domaine Internet, l'@ d'un socket est composée

- d'une **@ IP** = celle de la machine sur laquelle s'exécute le proc.
- d'un **numéro de port =** entier codé sur 16 bits

  - attribué soit par le processus créateur du socket, soit par l'OS

  - permettant de distinguer le socket parmi tous ceux déjà crées sur la machine considérée ⇔ « adresse de processus »

– Structure de l'@ d'un socket du domaine Internet

```
struct sockaddr_in {
 short sin_family ;   /* = AF_INET pour le domaine Internet*/
 u_short sin_port ;            /* = numéro de port */
 struct in_addr sin_addr ;     /* « comporte » l'@ IP */
 char sin_zero[8] ;            /* à zéro */
} ;
```

avec :

```
struct in_addr {
 union{
        struct {…} ;
        struct {…} ;
        u_long S_addr ;     /* = @ IP */
 } S_un ;
#define s_addr S_un.S_addr
} ;
```

3. Manipulation des socket du domaine Internet

3.1. Création d'un socket

- **Pour communiquer avec un processus distant :**
  - un processus doit tout d'abord créer un socket local par appel à la primitive **socket()**
- **Syntaxe**

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

  - **Signification des arguments**
    - domaine : AF_INET
    - type : SOCK_DGRAM ou SOCK_STREAM
    - protocole : 0 = protocole par défaut
      IPPROTO_UDP si type = SOCK_DGRAM
      IPPROTO_TCP si type = SOCK_STREAM
  - **Valeur retournée**
    - représentation interne (à l'OS) du socket si OK
      ⇔ descripteur de fichier
    - (-1) si erreur

3.2. Destruction d'un socket

- Destruction d'un socket **local** par appel à la primitive *close()*
- **Syntaxe**

```
int close(sock)
        int sock ;
```

  - **Signification des arguments**
    - sock : représentation interne du socket
  - **Valeur retournée**
    - (-1) si erreur

➢ **Ex**

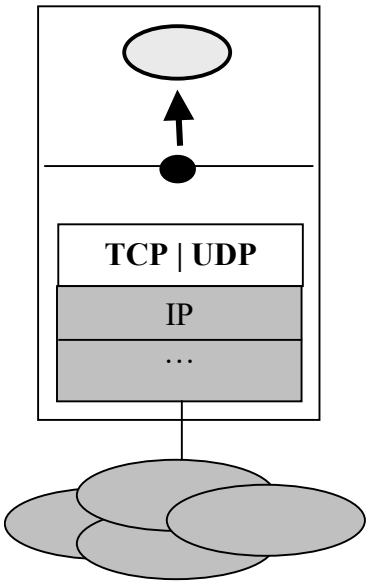
```
/* partie déclaration */
  int sock ;
/* création puis destruction d'un socket UDP */
  if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
  { printf("échec de création du socket\n") ;
    exit(1) ; }
  …
  if (close(sock) = = -1)
  { printf("échec de destruction du socket\n") ;
    exit(1) ; }
```

3.3. Adressage d'un socket

- **Une fois créé, un socket possède une représentation interne**
  - entier retourné par la primitive socket()
- **Pb** : cette valeur n'est connue que du processus créateur du socket

  ➔ Pour que d'autres processus puissent communiquer avec le processus père du socket :

  une adresse (externe) doit être attribuée au socket !

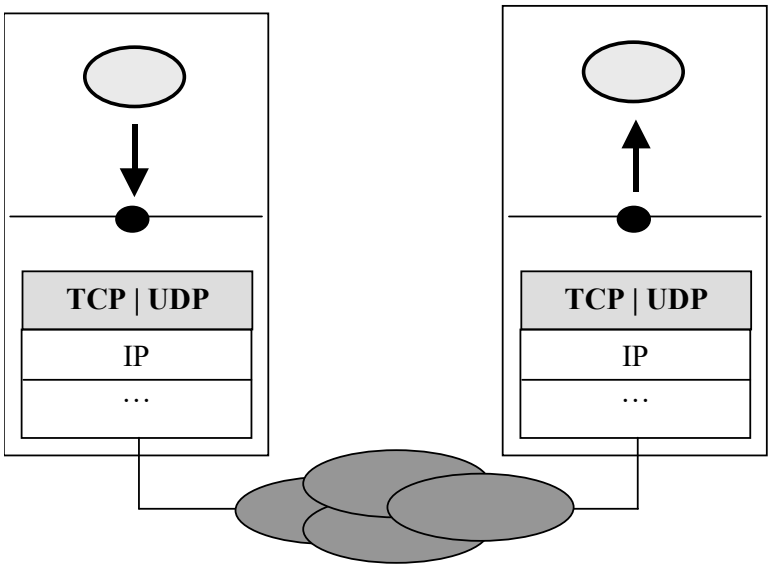


- **Deux phases dans cette procédure**

  1] Construction de l'adresse

  2] Association de l'adresse du socket à sa représentation interne au moyen de la primitive **bind()**

➢ Rmq importantes



- Si l'on se trouve dans le processus **RECEPTEUR** (d'une requête de connexion si TCP, ou d'un message si UDP) :

– **obligation de construire l'adresse du <u>socket local</u>**

– **… puis d'associer, <u>ensuite</u>, cette adresse au socket**

- Si l'on se trouve dans le processus **EMETTEUR** :

– **obligation de fournir l'adresse du <u>socket distant</u> au système de com. sous-jacent** pour pourvoir s'adresser à lui**, MAIS** :

- non obligation de construire l'adresse du socket local
- dans ce cas :
  - l'adresse du socket local sera créée et associée au socket par l'OS lors de l'exécution :
    - de la primitive de demande d'établissement de connexion si utilisation de TCP
    - de la primitive d'émission si utilisation de UDP

1] Construction d'une @ de socket (local ou distant)

- **@ d'un socket = variable du type « sockaddr_in »**

– à laquelle il s'agit d'affecter :

- l'@ IP de la machine sur laquelle s'exécute le proc père du socket
- un numéro de port

– **Rappel**

```
struct sockaddr_in {
  short sin_family ;    /* = AF_INET pour le domaine Internet*/
  u_short sin_port ;            /* = numéro de port */
  struct in_addr sin_addr ;     /* « comporte » l'@ IP */
  char sin_zero[8] ;            /* à zéro */
} ;
```

avec :

```
struct in_addr {
  union{
        struct {…} ;
        struct {…} ;
        u_long S_addr ;     /* = @ IP */
  } S_un ;
#define s_addr S_un.S_addr
} ;
```

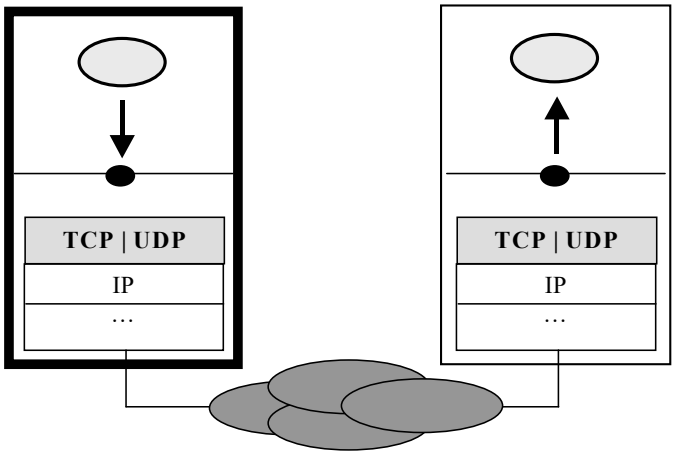
- **Soit *adr* une variable de type sockaddr_in**

⇒ Champs à mettre à jour :

- adr.sin_family → AF_INET dans le domaine Internet
- **adr.sin_port**
- **adr.sin_addr.s_addr**

a] Construction d'une @ de socket DISTANT (PAR UN EMETTEUR)

- « Emetteur » au sens { émetteur d'une requête de connexion si TCP / émetteur du premier message échangé si UDP



- **1 seule façon de mettre à jour le champ adr.sin_port**

– **explicitement → ex** : adr.sin_port = 9000

- **1 seule façon de mettre à jour le champ adr.sin_addr.s_addr**

– **explicitement**

- au moyen (le plus fréquemment) de la primitive **gethostbyname**()
  - primitive qui permet de récupérer l'@IP d'une machine
    - à partir de son nom logique

- **La primitive gethostbyname()**

– **Syntaxe**

```
struct hostent *gethostbyname(nom_station)
        char *nom_station ;
```

- **Signification de l'argument**
  - nom_station : nom de la station dont on cherche l'adresse IP
- **Valeur retournée**
  - pointeur sur une structure de type « hostent »
  - NULL en cas d'erreur

```
struct hostent
{
    char *h_name ; /* nom de la station */
    char **h_alias ; /* nom des alias */
    int h_addrtype ; /* classe A, B ou C */
    int h_length ; /* longueur du champ h_addr_list */
    char **h_addr_list ; /* liste d'adresse IP */
  # define h_addr h_addr_list[0] ; /* 1ère @ de la liste */
}
```

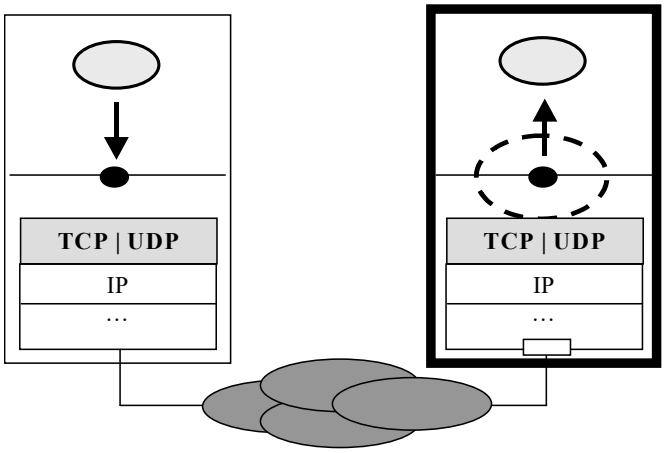
➢ **Ex**

- Construire l'adresse d'un socket auquel on souhaite s'adresser
  - **Hyp** : @IP$_{dest}$ = « dumas.insa-tlse.fr » ; n° port$_{dest}$ = 9000

```
/* déclaration */
    …
    struct hostent *hp ;
    struct sockaddr_in adr_distant;
/* main */
    …
/* affectation domaine et n° de port */
    /* reset de la structure */
    memset((char *)& adr_distant, 0, sizeof(adr_distant)) ;
    adr_distant.sin_family = AF_INET ; /* domaine Internet*/
    adr_distant.sin_port = 9000 ; /* n° de port */
/* affectation @_IP */
    if ((hp = gethostbyname("dumas.insa-tlse.fr")) = = NULL)
    {     printf("erreur gethostbyname\n") ;
          exit(1) ; }
    memcpy( (char*)&(adr_distant.sin_addr.s_addr),
                hp->h_addr,
                hp->h_length ) ;
    …
```

b] Construction d'une @ de socket LOCAL A UN RECEPTEUR

- « Récepteur » au sens { récepteur d'une requête de connexion si TCP
  ou récepteur du 1er message échangé si UDP



- **2 façons de mettre à jour le champ adr.sin_port**

1) soit **explicitement ⇔ cas le + fréquent !**

- **ex** : adr.sin_port = 9000

2) soit **implicitement** en lui donnant la valeur 0 : adr.sin_port = 0

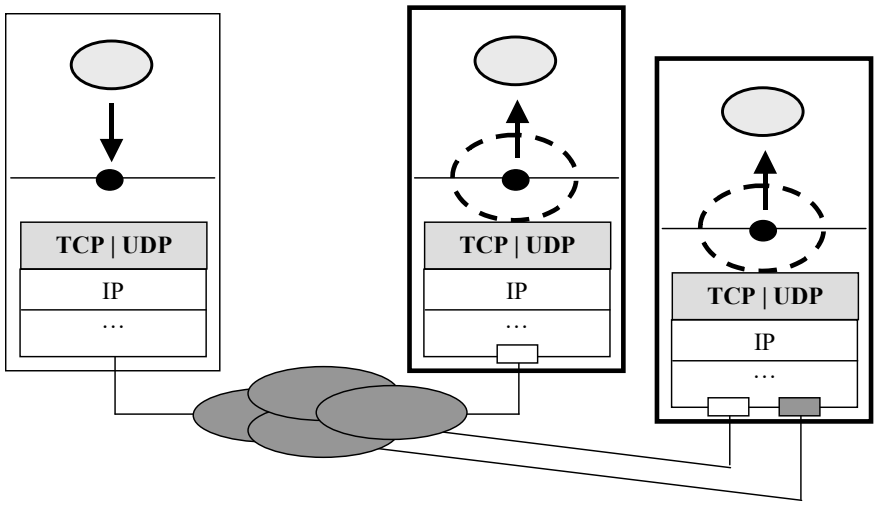
- Dans ce cas :
  - l'OS fournit alors la première valeur disponible
    - lors de l'association de l'adresse au socket (*bind*)
- Intérêt :
  - Etre certain d'utiliser un n° de port disponible
  - Besoin en conséquence = ??
    - réponse via la primitive getsockname()

- **La primitive getsockname()**
  - permet de récupérer l'adresse d'un socket **local**
  - **Syntaxe**

```
int getsockname(sock, padr, plg_adr)
    int sock ;
    struct sockaddr *padr ;
    int *plg_adr ;
```

- **Signification des arguments**
  - sock : représentation interne du socket
  - padr : adresse mémoire à laquelle l'@ du socket sera stockée en sortie de la fonction
  - plg_adr : @ mémoire à laquelle la longueur de la structure pointée par padr sera stockée en sortie de la fonct.
    - longueur max à l'appel, longueur effective en sortie !
- **Valeur retournée**
  - 0 si succès
  - (-1) si erreur

b] Construction d'une @ de socket LOCAL A UN RECEPTEUR (suite)



- **2 façons de mettre à jour le champ adr.sin_addr.s_addr**

1) Soit en lui donnant la valeur **INADDR_ANY ⇔ cas le + fréquent !**

- Revient à affecter au socket n'importe laquelle de toutes les @ IP de la machine ; en d'autres termes :

  - si la machine possède plusieurs interfaces Φ (donc plusieurs @ IP)
    - seront acceptées sur le socket toutes les données
      - **à destination de l'une quelconque des @ IP de la machine**
        - sous réserve bien sur que le n° de port soit OK
  - si la machine ne possède qu'une seule @ IP
    - qui peut le plus pouvant le moins …

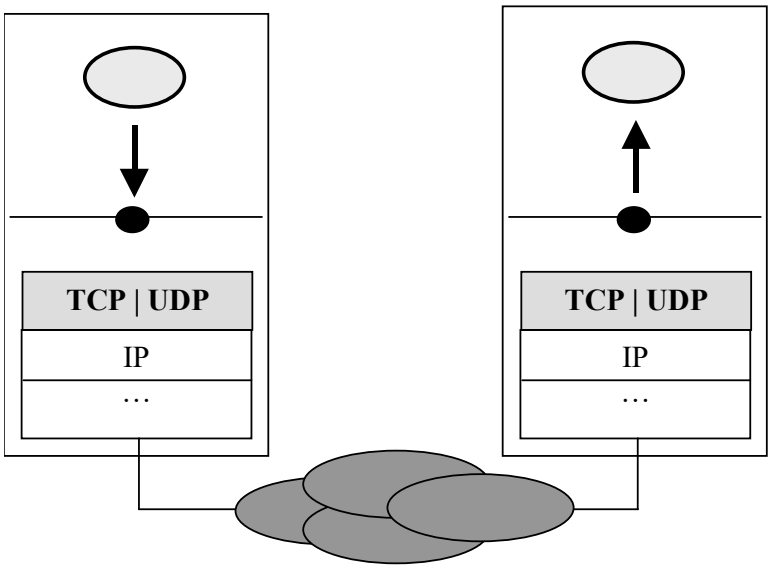
2) Soit explicitement au moyen de la primitive **gethostbyname**()

- Intérêt : uniquement si la machine possède plusieurs interfaces Φ

  - ne seront acceptées sur le socket que les données
    - **à destination de l'@ IP explicitement mentionnée**

➢ **Ex**

- Construire l'adresse d'un socket local (à un récepteur)
  - **Hyp** : @ IP_{local} = « dumas.insa-tlse.fr » ; n° $port_{local}$ = 9000




```
/* déclaration */
    …
    struct sockaddr_in adr_local;
/* main */
    …
/* affectation domaine, n° de port et adresse IP */
    /* reset de la structure */
    memset((char *)& adr_local, 0, sizeof(adr_local)) ;

    adr_local.sin_family = AF_INET ; /* domaine Internet */
    adr_local.sin_port = 9000 ; /* n° de port */
    adr_local.sin_addr.s_addr = INADDR_ANY
```

2] Association d'une adresse à un socket LOCAL A UN RECEPTEUR

- **Une fois un socket créé et son adresse construite**

– nécessité d'associer cette @ à la représentation interne du socket

pour pouvoir RECEVOIR
un message (UDP) ou une demande de connexion (TCP)

– cette association se fait par appel à la primitive **bind()**

- **Syntaxe**

```
int bind(sock, padr, lg_adr)
    int sock ;
    struct sockaddr *padr ;
    int lg_adr ;
```

– **Signification des arguments**

- sock : représentation interne du socket
- padr : pointeur sur la structure contenant l'@ du socket
- lg_adr : taille de la structure pointée par padr

– **Valeur retournée**

- 0 si succès
- (-1) si erreur

➢ **Ex d'association d'une @ à un socket <u>forcément</u> LOCAL**

…

```
int sock ; /* représentation interne du socket local */
struct sockaddr_in adr_local ; /* adresse du socket local */
int lg_adr_local = sizeof(adr_local) ;

/* création du socket */
if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
{     printf("échec de création du socket\n") ;
      exit(1) ; }

/* construction de l'@ du socket avec n° de port = 9000 et   */
/* @ IP = celle de la machine sur laquelle s'exécute le proc */
memset((char*)&adr_local, 0, sizeof(adr_local)) ; /* reset */
adr_local.sin_family = AF_INET ;
adr_local.sin_port = 9000 ;
adr_local.sin_addr.s_addr = INADDR_ANY ;

/* association @ socket ↔ représentation interne */
if (bind (sock, (struct sockaddr *)&adr_local, lg_adr_local) == -1)
{     printf("échec du bind\n") ;
      exit(1) ; }
```

…

➢ **Rmq IMPORTANTE !**

- Plusieurs des primitives utilisées par la suite (incluant le bind)

– ne manipulent pas explicitement les structures sockaddr_in, sockaddr_un, etc …

– … mais la structure générique :

```
struct sockaddr {
        u_short sa_family ;     /* domaine */
        char sa_data[14] ;      /* référence */
} ;
```

↓

- **Démarche à adopter quand on communique dans un domaine quelconque (D) :**

– déclarer les variables d'adresse **du même type** que celles manipulées dans le domaine D

⇔ **struct sockaddr_in si D = Internet !**

– affecter les champs de ces variables lors de la construction de l'adresse des socket (local et/ou distant)

– **PUIS** :

- passer en paramètre de la fonction : **l'adresse** de la variable (&)
- et (éventuellement) forcer au type sockaddr la variable passée en paramètre (struct sockaddr *)

4. Communication par socket UDP

4.1. Principe général

- **Utilisation de socket de type SOCK_DGRAM**
- **Pour émettre un message M, un processus doit :**
  - créer un socket local
  - construire l'adresse du socket **auquel il souhaite s'adresser**
  - faire appel à la primitive **sendto()** en spécifiant en particulier :
    - l'adresse du socket destinataire
    - l'adresse de la zone mémoire à laquelle se trouve stockée le message à émettre
    - la longueur de ce message
- **Pour recevoir un message, un processus doit :**
  - créer un socket local
  - en construire une adresse
  - « binder » adresse et représentation interne du socket
  - faire appel à la primitive **recvfrom()** en spécifiant en particulier :
    - l'adresse de la zone mémoire à laquelle le système doit aller copier les données reçues
    - la taille (en octets) allouée à cette zone ⇔ **longueur maximale** des données que le processus souhaite lire
    - l'adresse de la zone mémoire à laquelle le système doit aller copier l'adresse du socket émetteur
    - la taille (en octets) allouée à cette dernière zone

4.2. Primitives d'émission et de réception

- **sendto() = primitive d'émission d'un message**

  – **Syntaxe**

```
int sendto(sock, pmesg, lg_mesg, option,
      padr_dest, lg_adr_dest)
      int sock ;
      char *pmesg;
      int lg_mesg ;
      int option ;
      struct sockaddr *padr_dest ;
      int lg_adr_dest ;
```

  - **Signification des arguments**

    - sock : représentation interne du socket local
    - pmesg : adresse du message à envoyer
    - lg_mesg : taille du message en octet
    - option : 0
    - padr_dest : pointeur sur la structure contenant l'@ du socket distant
    - lg_ adr_dest : longueur de la structure pointée par padr_dest

  - **Valeur retournée**

    - le nombre de caractères envoyés et (-1) si erreur

➢ Ex d'utilisation du sendto()

- Soit à émettre via UDP un message de 5 caractères (par ex : 5 « a ») à destination d'un socket de n° port = 9000 créé sur la machine « dumas.insa-toulouse.fr »

```
/* Déclaration */
int sock ;
struct sockaddr_in adr_distant ;
int lg_adr_distant = sizeof (adr_distant) ;
char M[5] = "aaaaa";
int lg_M = 5;
int lg_emis ;
…

/* Création du socket local */
   …

/* Construction de l'adresse du socket distant */
   …

/* Emission du message */

lg_emis = sendto(sock, M, lg_M, 0, (struct sockaddr*)&adr_distant,
lg_adr_distant)
```

- **Question subsidiaire :**

– Et si M est d'un autre type ? par ex de type *struct X*

lg_emis = sendto(sock, (char)&M, ...)*

➢ Rmq (rappel)

- Il n'est pas nécessaire d'associer (bind) une adresse externe à un socket pour pouvoir émettre des données

– alors que c'est **OBLIGATOIRE** en **réception**

- **Dans ce cas**

– l'@ IP de la machine source est récupérée par le système

– le numéro de port est attribué par le système

– l'OS réalise automatiquement le bind() lors du premier appel à la primitive sendto()

– Pour récupérer l'adresse d'un socket établie automatiquement par l'OS : appel à la primitive **getsockname**()

- **recvfrom() = primitive de réception d'un message**

  – **Syntaxe**

```
int recvfrom(sock, pmesg, lg_max, option,
                                    padr_em, plg_adr_em)
    int sock ;
    char *pmesg ;
    int lg_max ;
    int option ;
    struct sockaddr *padr_em ;
    int *plg_adr_em ;
```

- **Signification des arguments**

  - sock : représentation interne du socket local
  - pmesg : adresse mémoire à laquelle le message lu sera stocké en sortie de la fonction
  - lg_max : taille de l'espace réservé pour stocker le message reçu ⇔ **taille max du message à lire**
  - option : 0
  - padr_em : adresse mémoire à laquelle l'adresse du socket émetteur sera stockée en sortie de la fonction
  - plg_adr_em : @ à laquelle la longueur de la structure pointée par padr_em sera stockée en sortie de la fonction

  ➔ longueur max à l'appel, longueur effective en sortie !

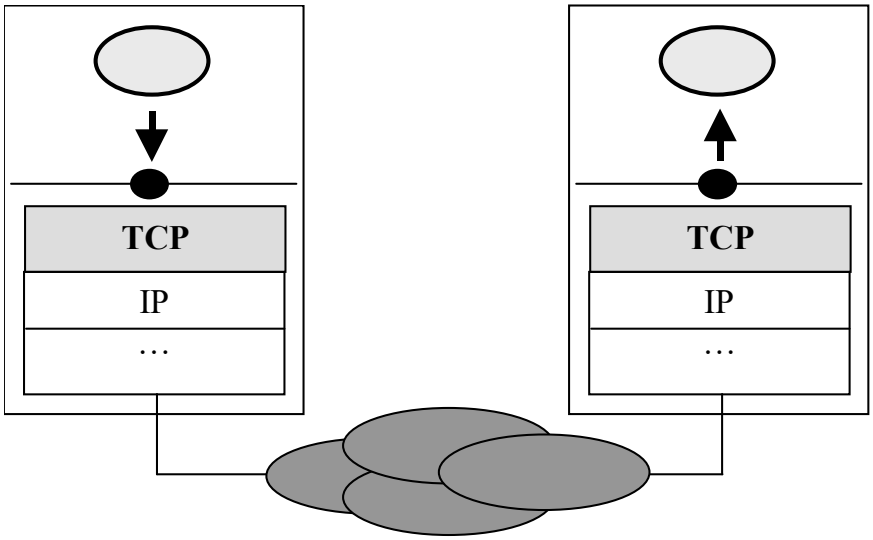
- **Valeur retournée**

  - nombre d'octets lus ; (-1) si erreur

5. Communication par socket TCP

5.1. Principe général

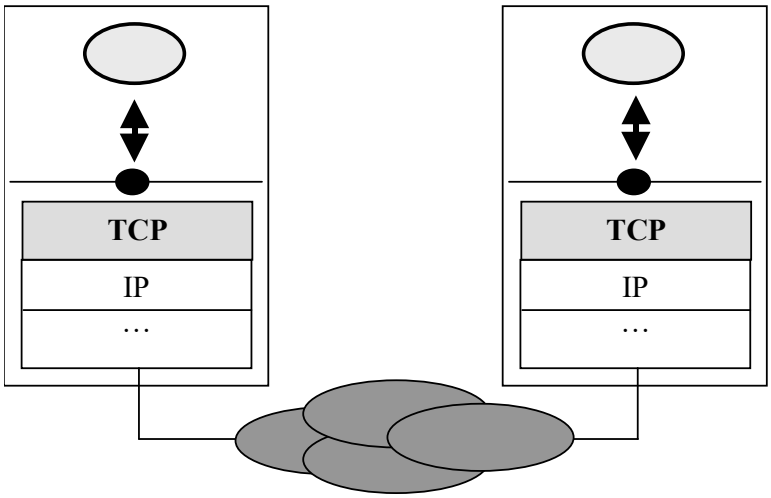
- **Utilisation de socket de type SOCK_STREAM**
- **Etablis[t] de la connexion selon un mode « client - serveur »**
  - **client** ⇔ processus qui effectue la demande de connexion à l'adresse d'un processus serveur
  - **serveur** ⇔ processus en état d'attente d'une (de) demande(s) de connexion en provenance d'un (ou plusieurs) client(s)



➢ **Rappel sur les caractéristiques de la communication**

- obligation d'établir une connexion entre les processus communicants avant le transfert des données
- garantie d'ordre et de fiabilité dans le transfert des données
- pas de préservation de la limite des messages

- **Une fois la connexion établie**



– **Pour émettre un message M, il suffit à l'un des processus connectés :**

• de faire appel à l'une des primitives **send()** ou **write()**

- en spécifiant en particulier :

- l'adresse à laquelle l'entité TCP en charge de la connexion doit aller lire le message à émettre
- la longueur du message à émettre

– **Pour lire des données reçues par le service de Transport local, il suffit à l'un des processus connectés :**

• de faire appel à l'une des primitives **recv()** ou **read()**

- en spécifiant en particulier :

- l'adresse à laquelle l'entité TCP en charge de la connexion doit aller copier les données reçues
- la taille (en octets) allouée à cette zone ⇔ longueur max données que le processus souhaite lire

5.2. Phase d'établissement de connexion

- **Coté client (de la connexion) ⇔ processus appelant**

– Avant de pouvoir émettre ou recevoir des données, le processus appelant doit obligatoirement :

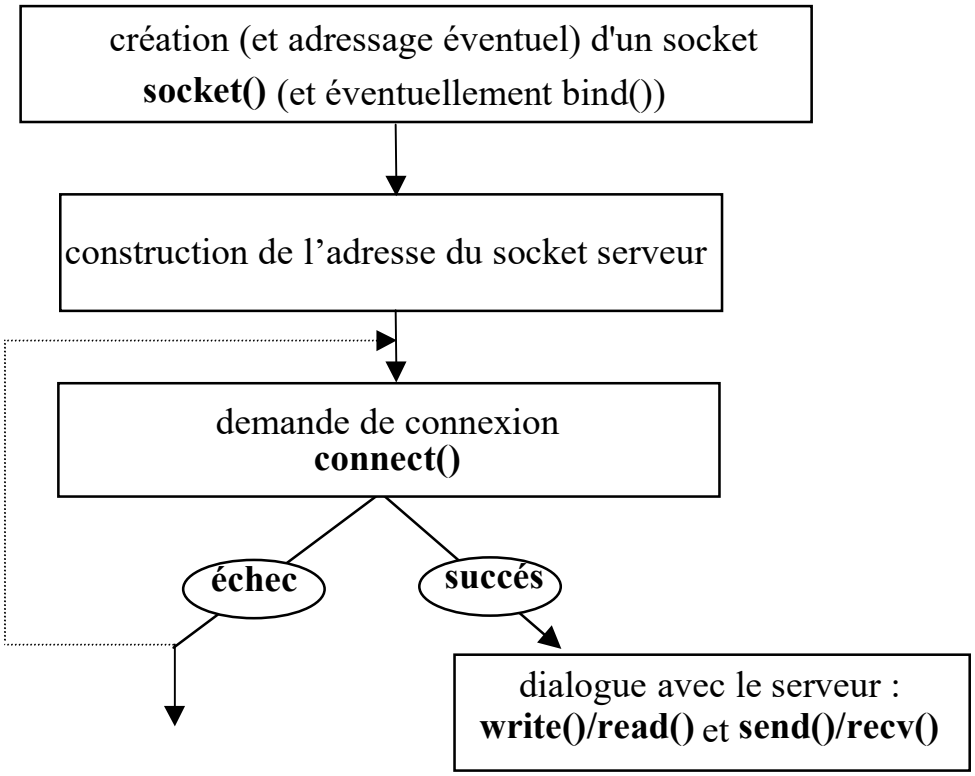
- créer un socket local
- construire l'adresse du socket **auquel il souhaite s'adresser**

➢ **Rmq**

- Il peut éventuellement construire l'adresse de son socket et binder cette adresse à la représentation interne du socket

– puis se connecter au socket serveur

- par appel à la primitive **connect**()



- **La primitive connect()**

⇔ primitive de demande de connexion

– **Syntaxe**

```
int connect(sock, padr_serv, lg_adr_serv)
    int sock ;
    struct sockaddr *padr_serv ;
    int lg_adr_serv ;
```

- **Signification des arguments**
  - sock : représentation interne du socket local
  - padr_serv : adresse mémoire à laquelle se trouve l'adresse du socket serveur
  - lg_adr_serv : longueur de la structure pointée par padr_serv
- **Valeur retournée**
  - 0 si succès
  - (-1) si échec de connexion

➢ **Rmq**

- L'appel à connect() n'est pas bloquant
- Rappel : pour récupérer l'@ IP de la station destinataire à partir de son nom logique ➔ utiliser la primitive gethostbyname()

- **Coté serveur de la connexion ⇔ processus appelé**

– Avant de pouvoir émettre ou recevoir des données, le processus appelé doit obligatoirement :

  - créer un socket local (*socket()*)
  - construire l'adresse **de ce socket**
  - associer adresse et représentation interne du socket (*bind()*)
  - puis :
    - se mettre en état d'acceptation de toute demande de connexion entrante par le biais de la primitive **accept()**
    - et lorsqu'une demande de connexion entrante arrive :
      - soit traiter lui même la requête
      - soit :
        - créer un proc. fils qui se chargera de traiter la requête
        - puis retourner en état d'attente d'une demande de connexion

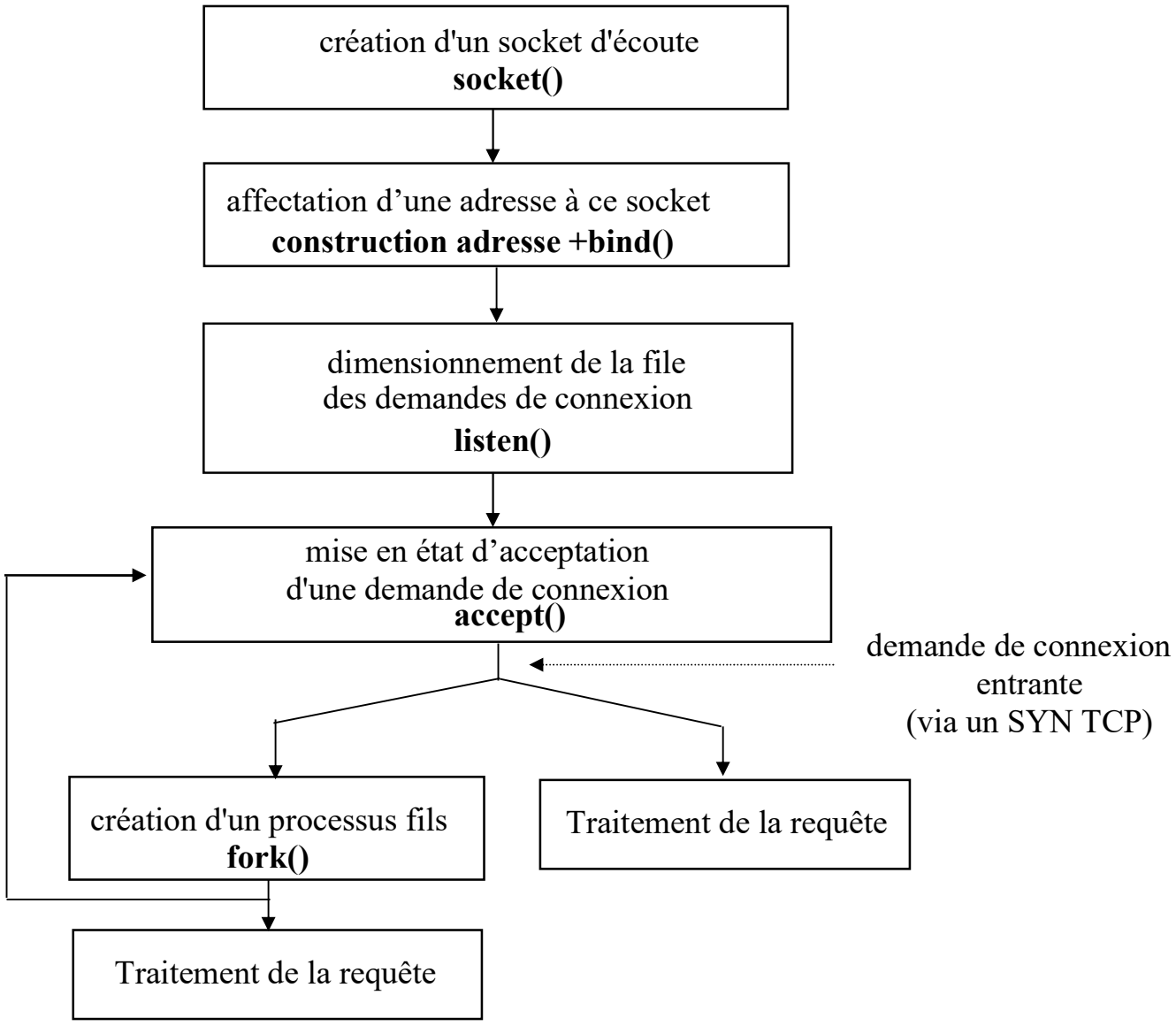
➢ **Rmq**

- En fait, une opération supplémentaire est à effectuer en amont du cinquième point :

– dimensionner la file d'attente des demandes de connexion

  - par le biais de la primitive **listen()**

- **Schématiquement : coté serveur de la connexion**



➢ Rmq

- Dans le cas du fork : toutes les données en provenance de l'émetteur du connect() seront aiguillées vers le processus fils

- **La primitive listen()**

  – Permet au processus appelé (serveur) de dimensionner

    • la file d'attente des demandes de connexion entrantes

  – **Syntaxe**

```
int listen(sock, nb_max)
        int sock ;
        int nb_max ;
```

    • **Signification des arguments**

      - sock : représentation interne du socket local

      - nb_max : nb max de demandes de connexion en attente

        = taille de la file des demandes de connexion

    • **Valeur retournée**

      - 0 si succès

      - (-1) si erreur

- **La primitive accept()**

⇔ primitive d'acceptation d'une demande de connexion

– permet au processus appelé (serveur) de récupérer l'adresse du socket client lorsqu'une demande de connexion survient

– **Syntaxe**

```
int accept(sock, padr_client, plg_adr_client)
        int sock ;
        struct sockaddr *padr_client ;
        int *plg_adr_client;
```

- **Signification des arguments**

  - sock : représentation interne du socket local
  - padr_client : adresse mémoire à laquelle sera stockée l'@ du socket client en sortie de la fonction
  - plg_adr_client : adresse mémoire à laquelle la longueur de la structure pointée par padr_client sera stockée en sortie

  ➔ longueur max à l'appel, longueur effective en sortie

- **Valeur retournée**

  - représentation interne du socket <u>local</u> dédié à la connexion établie (⇔ un second socket !)

  ➔ qui sera ensuite utilisée dans les read()/write executés par le serveur !!

  - (-1) si erreur

➢ **Rmq**

- La primitive accept() est bloquante tant qu'aucune connexion n'a été établie

  – note : existence d'un mode non bloquant

- Une fois la primitive exécutée

  – padr_client pointe sur l'adresse du socket appelant (client)

  – plg_adr_client contient la taille réelle de cette adresse

- Après acceptation d'une connexion, le processus appelé (serveur) a deux possibilités :

  – soit **prendre à son compte la connexion**

    • toute autre demande de connexion ne sera prise en compte qu'après terminaison du traitement associé à la connexion

  – soit **sous-traiter le traitement de la connexion** à un proc fils

    • en utilisant par exemple la primitive fork()

➢ **Ex sans sous-traitance**

```
…
int sock, sock_bis ;
struct sockaddr_in adr_client ;    /* adresse du socket client */
int lg_adr_client = sizeof(adr_client) ;
int lg_rec ;              /* longueur du message reçu */
int max = 10 ;            /* nb de messages attendus */
int lg_max = 30 ;     /* taille max de chaque message */
…
sock = socket(AF_INET, SOCK_STREAM, 0) ;

/* construction de l'adresse locale + association */
…
bind(…) ;

listen(sock, 5) ;

if ((sock_bis = accept(   sock,
                          (struct sockaddr *)&adr_client,
                          &lg_adr_client)) == -1)
        {printf("échec du accept\n") ;
         exit(1) ;
}
for (i=0 ; i < max ; i ++) {
        if ((lg_rec = read(sock_bis, message, lg_max)) < 0))
                {printf("échec du read\n") ; exit(1) ;}
        afficher_message(message, lg_rec) ;
}
```

➢ **Ex avec sous-traitance**

```
…
int sock, sock_bis ;
struct sockaddr_in adr_client ; /* adresse du socket client */
int lg_adr_client = = sizeof(adr_client) ;
…
sock = socket(AF_INET, SOCK_STREAM, 0) ;
…
bind(…) ;
listen(sock, 5)
while (1) {
    if ((sock_bis = accept( sock, (struct sockaddr *)&adr_client,
                            &lg_adr_client)) = = -1) {
        printf("échec du accept\n") ; exit(1) ;}
    switch (fork()) {
        case - 1 : /* il y a une erreur */
            printf("erreur fork\n") ; exit(1) ;
        case 0 : /* on est dans le proc. fils */
            close(sock) ; /* fermeture socket du proc. père */
            for (i=0 ; i < max ; i ++) {
               if ((lg_rec = read(sock_bis, message, lg_max)) < 0)){
                  printf("échec du read\n") ; exit(1) ;}
               afficher_message(message, lg_rec) ; }
            exit(0) ;
        default : /* on est dans le proc. père */
            close(sock_bis) ; /* fermeture socket du proc. fils */
    }
}
```

5.3. Phase de transfert de données

- Une fois la connexion établie, processus client et serveur peuvent émettre ou recevoir des données

⇔ connexion bidirectionnelle

♦ Les primitives d'émission : write() et send()

- **La primitive write()** = primitive standard

– **Syntaxe**

```
int write(sock, pmesg, lg_mesg)
    int sock ;
    char *pmesg ;
    int lg_mesg ;
```

- **Signification des arguments**

  - sock : représentation interne du socket local
  - pmesg : adresse mémoire à laquelle se trouve le message à envoyer
  - lg_mesg : taille en octet du message à envoyer

- **Valeur retournée**

  - nombre de caractères envoyés
  - (–1) si erreur

- **La primitive send()**

– send() ⇔ write() avec une option supplémentaire :

  • le traitement des messages urgents

– **Syntaxe**

```
int send(sock, pmesg, lg_mesg, option)
      int sock ;
      char *pmesg ;
      int lg_mesg ;
      int option ;
```

  • **Signification des arguments**

    - sock : représentation interne du socket local
    - pmesg : adresse du message à envoyer
    - lg_mesg : taille du message en octet
    - option : 0 ou MSG_OOB ⇔ message urgent

  • **Valeur retournée**

    - nombre de caractères envoyés ; (-1) si erreur

➢ **Rmq**

- Le proc qui exécute un write() ou un send() se bloque lorsque :

– les buffers de réception associés au socket dest. sont pleins

– les buffers d'émission associés au socket local sont pleins

♦ Les primitives de réception : read() et recv()

▪ La primitive read() = primitive standard

– **Syntaxe**

```
int read(sock, pmesg, lg_max)
    int sock ;
    char *pmesg ;
    int lg_max ;
```

- **Signification des arguments**

  - sock : représentation interne du socket local
  - pmesg : adresse mémoire à laquelle le message reçu sera stocké en sortie de la fonction
  - lg_max : longueur de la zone allouée au stockage du message reçu ⇔ **nb max d'octets à lire**

- **Valeur retournée**

  - nombre d'octets lus
  - (-1) si erreur

- **La primitive recv()**

  – recv() ⇔ read() avec options supplémentaires :

    • Traitement des messages urgents

    • Consultation sans extraction

  – **Syntaxe**

```
int recv(sock, pmesg, lg_max, option)
    int sock ;
    char *pmesg ;
    int lg_max ;
    int option ;
```

    • **Signification des arguments**

      - sock : représentation interne du socket local

      - pmesg : adresse mémoire à laquelle le message reçu sera stocké en sortie de la fonction

      - lg_max : longueur de la zone allouée au stockage du message reçu ⇔ **nb max d'octets à lire**

      - option : 0 ou MSG_OOB ou MSG_PEEK ⇔ consultation sans extraction

    • **Valeur retournée**

      - nombre d'octets lus ; (-1) si erreur

5.4. Phase de fermeture d'une connexion

- **La primitive shutdown()**

– permet de fermer la connexion sans détruire les sockets

- dans un sens, dans l'autre ou dans les deux

– **Syntaxe**

```
int shutdown(sock, sens)
      int sock ;
      int sens ;
```

- **Signification des arguments**

  - sock : représentation interne du socket local

  - sens : 0 → le processus ne veut plus recevoir

    1 → le processus ne veut plus émettre

    2 → le processus ne veut plus ni recevoir ni émettre

- **Valeur retournée**

  - 0 si succès

  - (-1) si erreur

6. Fichiers à inclure en entête des programmes

- Fichiers contenant la définition de toutes les structures (struct sockaddr_in, …) et toutes les fonctions (socket(), …) utilisées

/* déclaration des types de base */

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

/* constantes relatives aux domaines, types et protocoles */

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

/* constantes et structures propres au domaine UNIX */

```
#include <sys/un.h>
```

/* constantes et structures propres au domaine INTERNET */

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

/* structures retournées par les fonctions de gestion de la base de données du réseau */

```
#include <netdb.h>
```

/* pour les entrées/sorties */

```
#include <stdio.h>
```