Programmation *Socket*

« La notion de socket a été introduite dans les distributions de Berkeley (un fameux système de type UNIX, dont beaucoup de distributions actuelles utilisent des morceaux de code), c'est la raison pour laquelle on parle parfois de sockets BSD (Berkeley Software Distribution).

Il s'agit d'un modèle permettant la communication inter processus (IPC - Inter Process Communication) afin de permettre à divers processus de communiquer aussi bien sur une même machine qu'à travers un réseau TCP/IP. »
Wikipedia

Manuel du programmeur réseau

- Les pages man sous Unix/Linux :
  - **socket(7)** : interface de programmation des *sockets*
  - **packet(7)** : interface par paquet au niveau périphérique
  - **raw(7)** : sockets brutes (*raw*) IPv4 sous Linux
  - **ip(7)** : implémentation Linux du protocole IPv4
  - **udp(7)** : protocole UDP pour IPv4
  - **tcp(7)** : protocole TCP

Remarque : xxx(7) → man 7 xxx

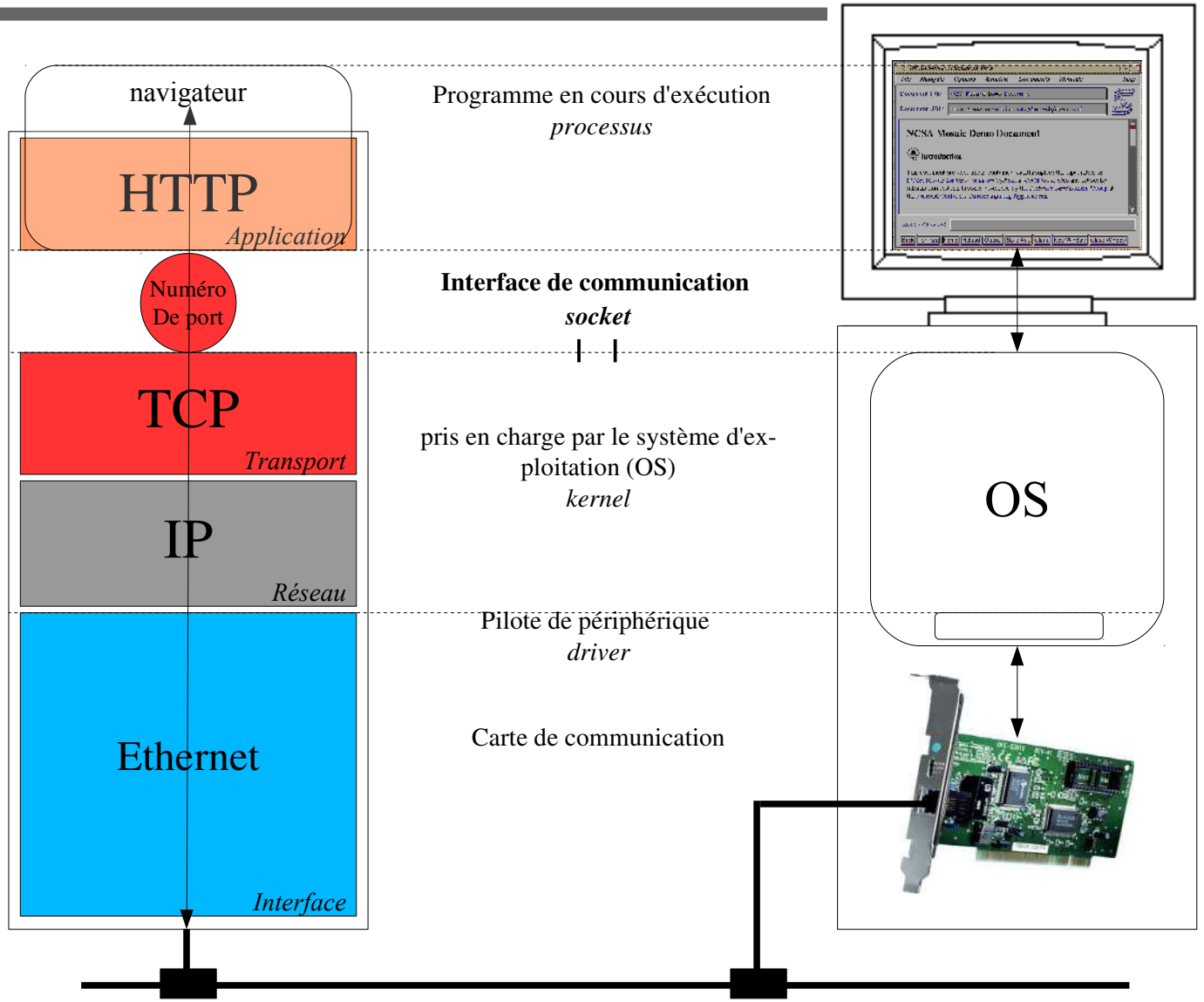
- Le service en ligne MSDN pour Windows :
  - Windows Socket 2 :
    http://msdn.microsoft.com/en-us/library/ms740673(VS.85).aspx
  - Les fonctions Socket :
    http://msdn.microsoft.com/en-us/library/ms741394(VS.85).aspx

Présentation

- Intégration d'IP dans Unix BSD (1981)
- Interface de programmation **«socket»** de Berkeley (1982) : la plus utilisée et intégrée dans le noyau.
- ***Socket :*** mécanisme de communication bidirectionelle entre processus
- Il existe d'autres interfaces : *Remote Procedure Call* (**RPC**), *Transport Layer Interface* (OSI), ...

Modèle de référence

- Un modèle de référence est utilisé pour décrire la structure et le fonctionnement des communications réseaux
- Le modèle DoD (*Departement of Defense*) ou « TCP/IP » est composé de 4 couches
- L'interface *socket* se place au-dessus de la couche Transport (et des services forunies par l'OS).

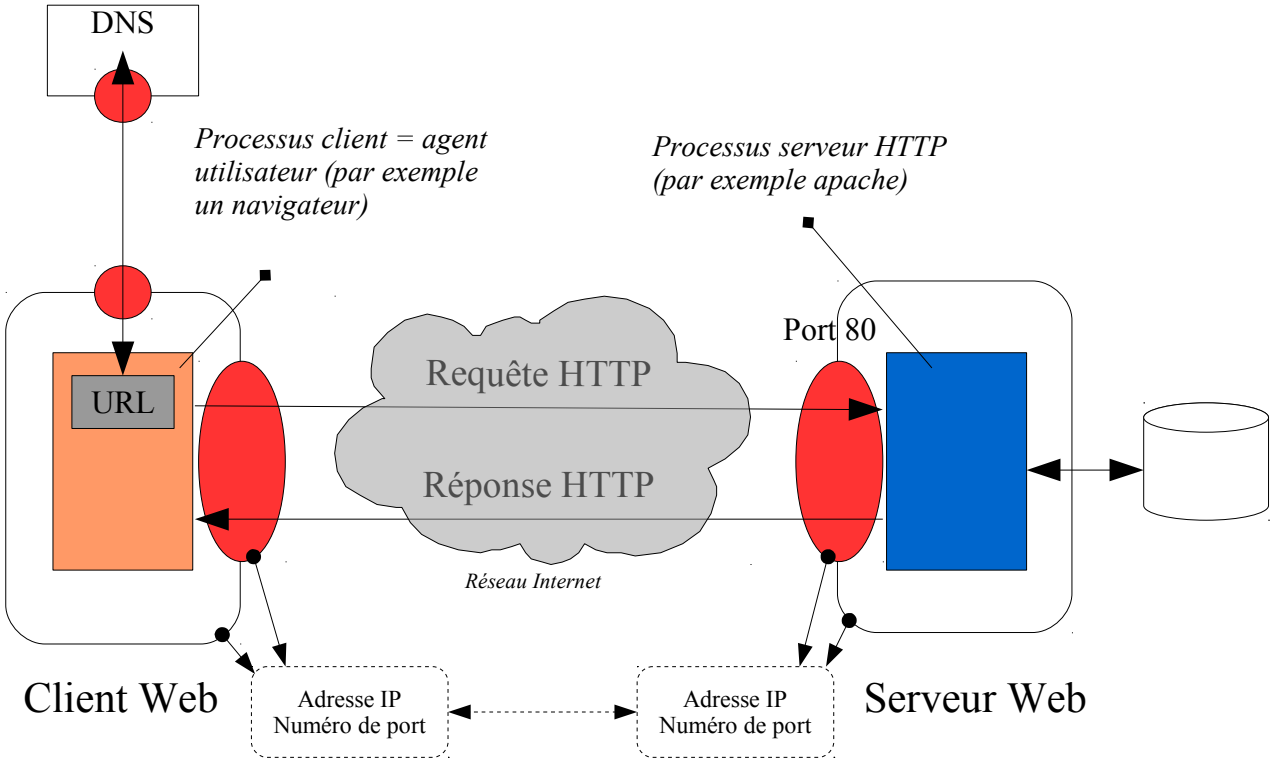


Notions

- L'utilisation de l'interface *socket* nécessite de connaître :
  - L'architecture client/serveur
  - L'adressage IP
  - Les numéros de port
  - Notions d'API et de programmation en langage C
  - Les protocoles TCP et UDP
  - Les modes connecté et non connecté

Architecture Client/Serveur

- Serveur : offre un service (en attente)
- Client : demandeur d'un service
- La communication s'initie TOUJOURS à la demande du client.



Notions d'API

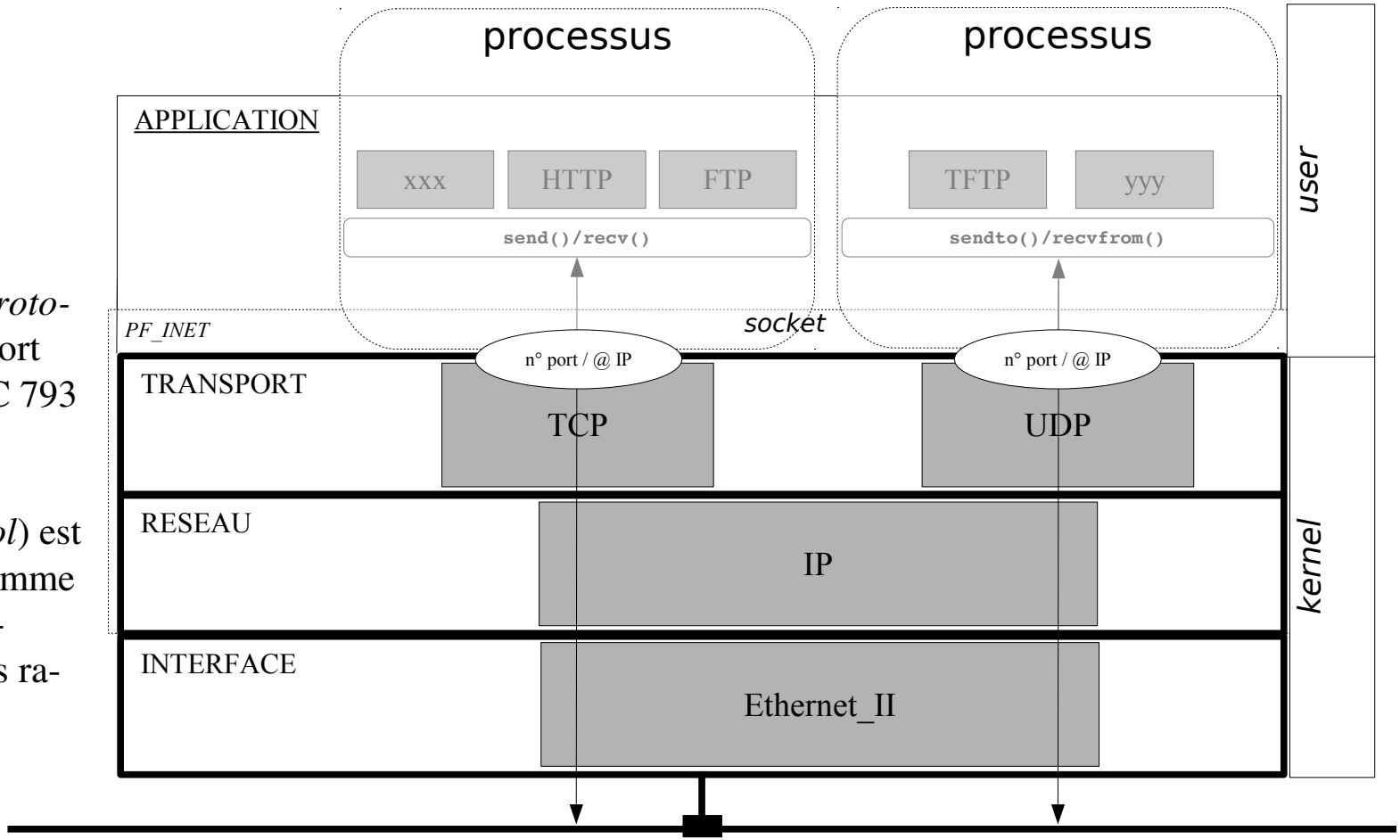
- Les services offerts par une couche constituent l'interface à la couche supérieure.
- Cette interface est implémentée sous forme de bibliothèque de fonctions soit une **API** (*Application Program Interface*).
- Le développeur utilisera donc concrètement une interface pour programmer une application TCP/IP grâce par exemple :
  - à l'API Socket BSD sous Unix/Linux ou
  - à l'API WinSocket sous Windows
- Une ***socket*** est un point de communication par lequel un processus peut émettre et recevoir des informations. Ce point de communication devra être relié à une adresse IP et un numéro de port.
- Une ***socket*** est communément représentée comme un point d'entrée initial au niveau TRANSPORT du modèle à couches dans la pile de protocole.

L'interface *socket*

- En programmation, on utilisera l'interface *socket* pour réaliser des applications réseaux :

TCP (*Transmission Control Protocol*) est un protocole de transport fiable, en mode connecté (RFC 793).

UDP (*User Datagram Protocol*) est un protocole souvent décrit comme étant non-fiable, en mode non-connecté (RFC 768), mais plus rapide que TCP.



Les numéros de port

- Un numéro de port sert à identifier l'application (un processus) en cours de communication par l'intermédiaire de son protocole de couche application (associé au service utilisé, exemple : 80 pour HTTP).
- Pour chaque port, un numéro lui est attribué (codé sur 16 bits), ce qui implique qu'il existe un maximum de 65 536 ports (2^16) par ordinateur (et par protocoles TCP et UDP).
- L'attribution des ports est faite par le système d'exploitation, sur demande d'une application. Cette dernière peut demander à ce que le système d'exploitation lui attribue n'importe quel port, à condition qu'il ne soit pas déjà attribué.
- Lorsqu'un processus client veut dialoguer avec un processus serveur, il a besoin de connaître le port écouté par ce dernier. Les ports utilisés par les services devant être connus par les clients, les principaux types de services utilisent des ports qui sont dits réservés. Une liste des ports attribués est disponible dans le fichier **/etc/services** sous Unix/Linux.

Caractéristiques des sockets

- Les *sockets* compatibles BSD représentent une **interface uniforme** entre le processus utilisateur (*user*) et les piles de protocoles réseau dans le noyau (*kernel*) de l'OS.
- Les *sockets* sont groupées en :
  - **familles de protocoles** (comme **PF_INET**, PF_IPX, PF_PACKET, ...) et en
  - **types de sockets** (comme **SOCK_STREAM**, **SOCK_DGRAM**, SOCK_RAW, ...)
- Donc :
  - Une *socket* appartient à une famille.
  - Il existe plusieurs types de sockets.
  - Chaque famille possède son adressage.

Pour plus de détails : man 7 socket et man 2 socket

Les fonctions de l'interface socket (I)

- Ces fonctions (primitives) servent à un processus utilisateur pour envoyer ou recevoir des paquets et pour faire d'autres opérations sur les *sockets*.
- Les appels indispensables :
  - **socket(2)** : crée une *socket*,
  - **connect(2)** : connecte une *socket* à une adresse de socket distante,
  - **bind(2)** : attache une *socket* à une adresse locale,
  - **listen(2)** : indique à la *socket* que de nouvelles connexions doivent y être acceptées,
  - **accept(2)** : fournit une nouvelle *socket* avec la nouvelle connexion entrante.
  - **send(2)**, **sendto(2)**, et sendmsg(2) : envoient des données sur une *socket*,
  - **recv(2)**, **recvfrom(2)**, recvmsg(2) : reçoivent des données depuis une *socket*.
  - **poll(2)** et **select(2)** : attendent que des données arrivent ou que l'émission soit possible.
  - **write(2)** et **read(2)** : les opérations d'E/S standards peuvent servir à écrire ou lire des données sur une *socket* en mode connecté
  - **close(2)** : sert à fermer une *socket*
  - **shutdown(2)** : ferme un sens de communication d'une *socket full-duplex* connectée.

Les fonctions de l'interface socket (II)

- Les appels pouvant rendre des services :

  - **getsockname(2)**: renvoie l'adresse locale d'une *socket*
  - **getpeername(2)**: renvoie l'adresse de la *socket* distante.
  - **getsockopt(2)** et **setsockopt(2)** : servent à fixer ou lire des options du niveau *socket* ou protocole.
  - **ioctl(2)** : peut servir pour lire ou écrire d'autres options.
  - **fcntl(2)** : peut servir à manipuler le descripteur de la *socket* (mode non bloquant avec l'attribut O_NONBLOCK, mode asynchrone avec l'attribut FASYNC, ...)

- Et aussi : **gethostbyname(), gethostbyaddr(), getservbyname(), getprotobyname(), getnameinfo(), inet_aton(), inet_addr(), inet_network(), inet_ntoa(), inet_makeaddr(), inet_lnaof(), inet_netof(), ...**

L'adressage par famille

- L'adressage dépend de la famille.

Exemple : IPv4 (Famille PF_INET) → adresse IPv4 et numéro de port

- On part d'une **structure d'adresse générique** :

```
struct sockaddr
{
  unsigned short int sa_family;     //au choix
  unsigned char sa_data[14];        //en fonction de la famille
};
```

- Puis, on utilise une **structure compatible par famille**.

Exemple : Famille PF_INET dans **<netinet/in.h>**

```
struct in_addr { unsigned int s_addr; };//adresse Ipv4 (32 bits)
struct sockaddr_in
{
  unsigned short int sin_family;    //PF_INET
  unsigned short int sin_port;      //numéro de port
  struct in_addr sin_addr;          //adresse IPv4
  unsigned char sin_zero[8];        //ajustement compatible
};
```

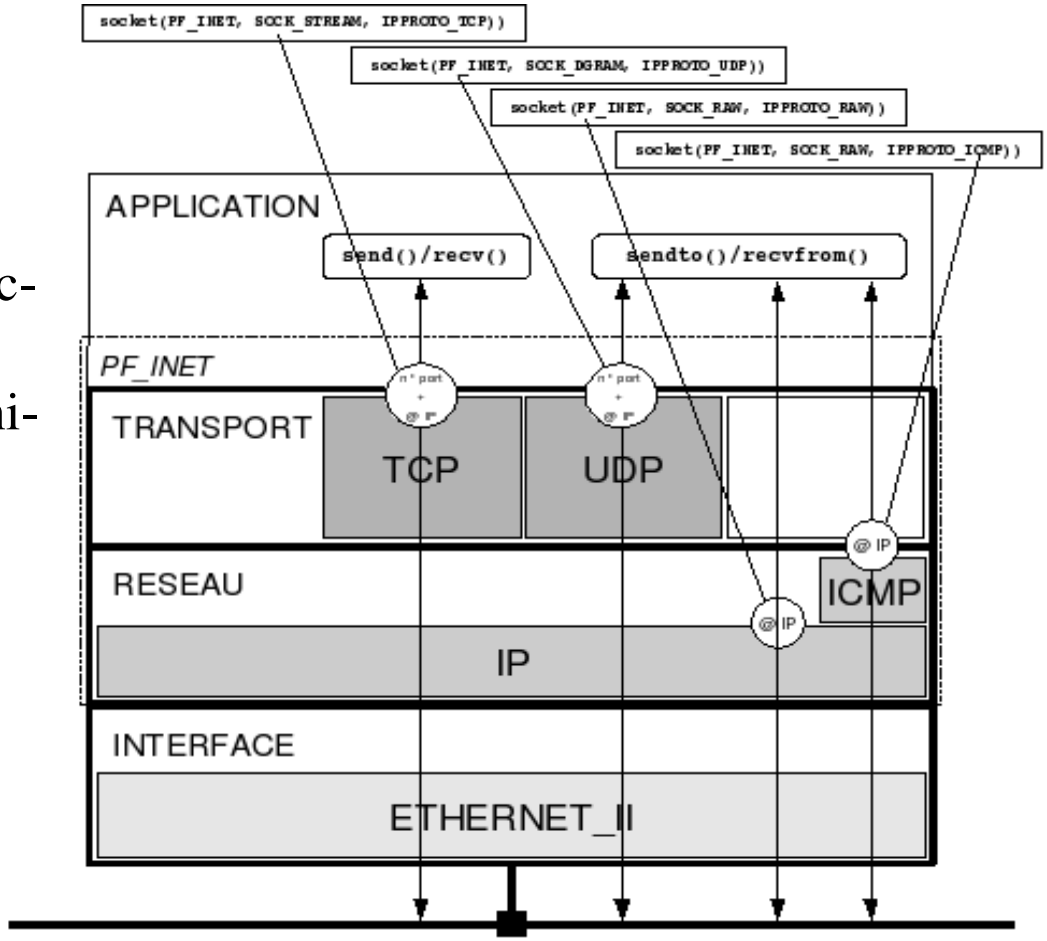
Les types de *socket*

- Le type de socket précise le mode d'utilisation dans la famille choisie.

- Pour PF_INET, on aura le choix entre :
  - **SOCK_STREAM** (mode connecté, par défaut TCP),
  - **SOCK_DGRAM** (mode non connecté, par défaut UDP),
  - **SOCK_RAW** (accès direct au protocole, IP, ICMP, ...).

Point de communication

- Il regroupe les informations nécessaires à la communication entre deux processus utilisant le réseau pour dialoguer :
  - le domaine de communication : sélectionne la famille d'adresse
  - le type de communication (mode connecté, mode non connecté, ...)
  - le protocole associé au type de communication (TCP, UDP, ...)

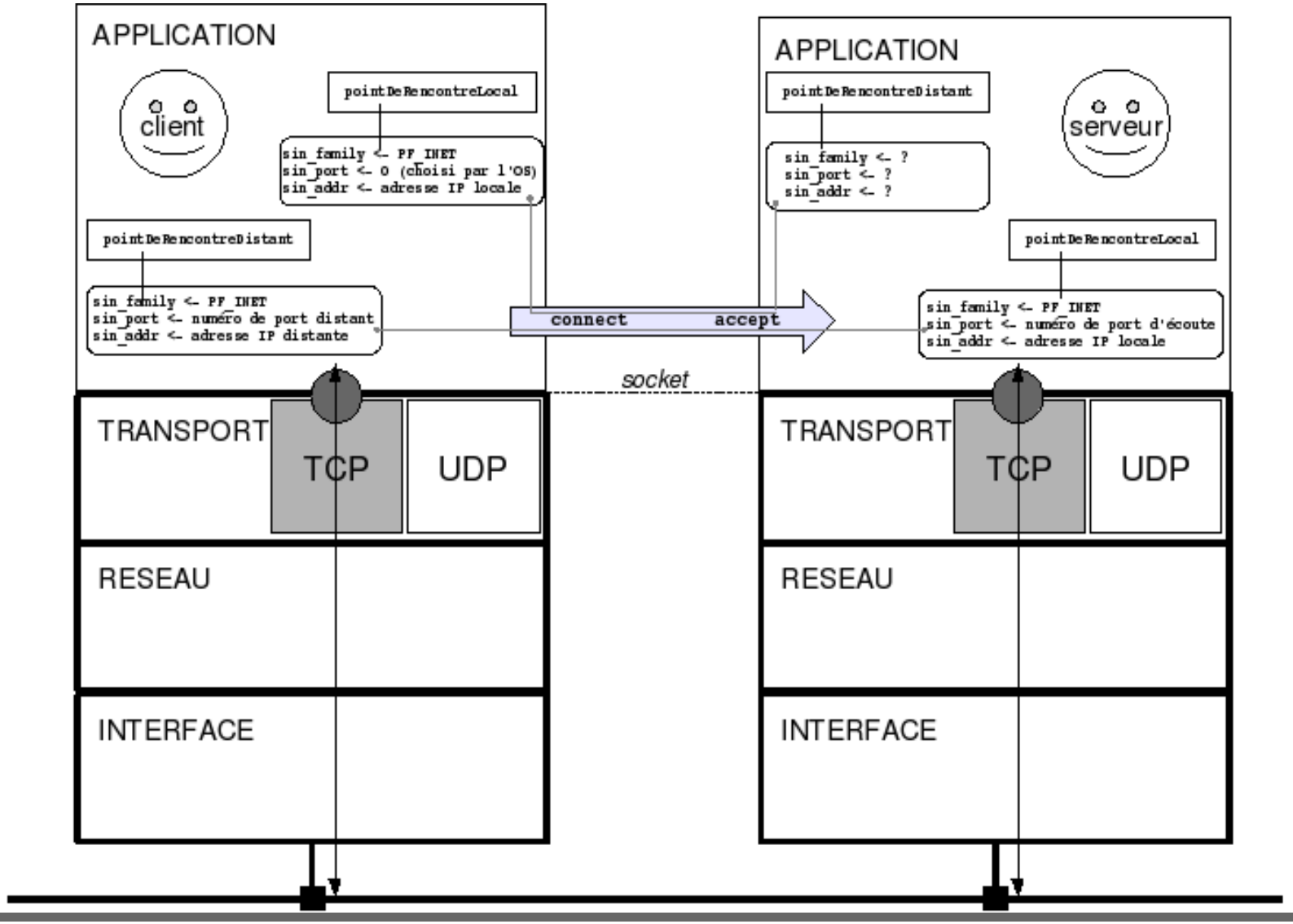


Point de rencontre

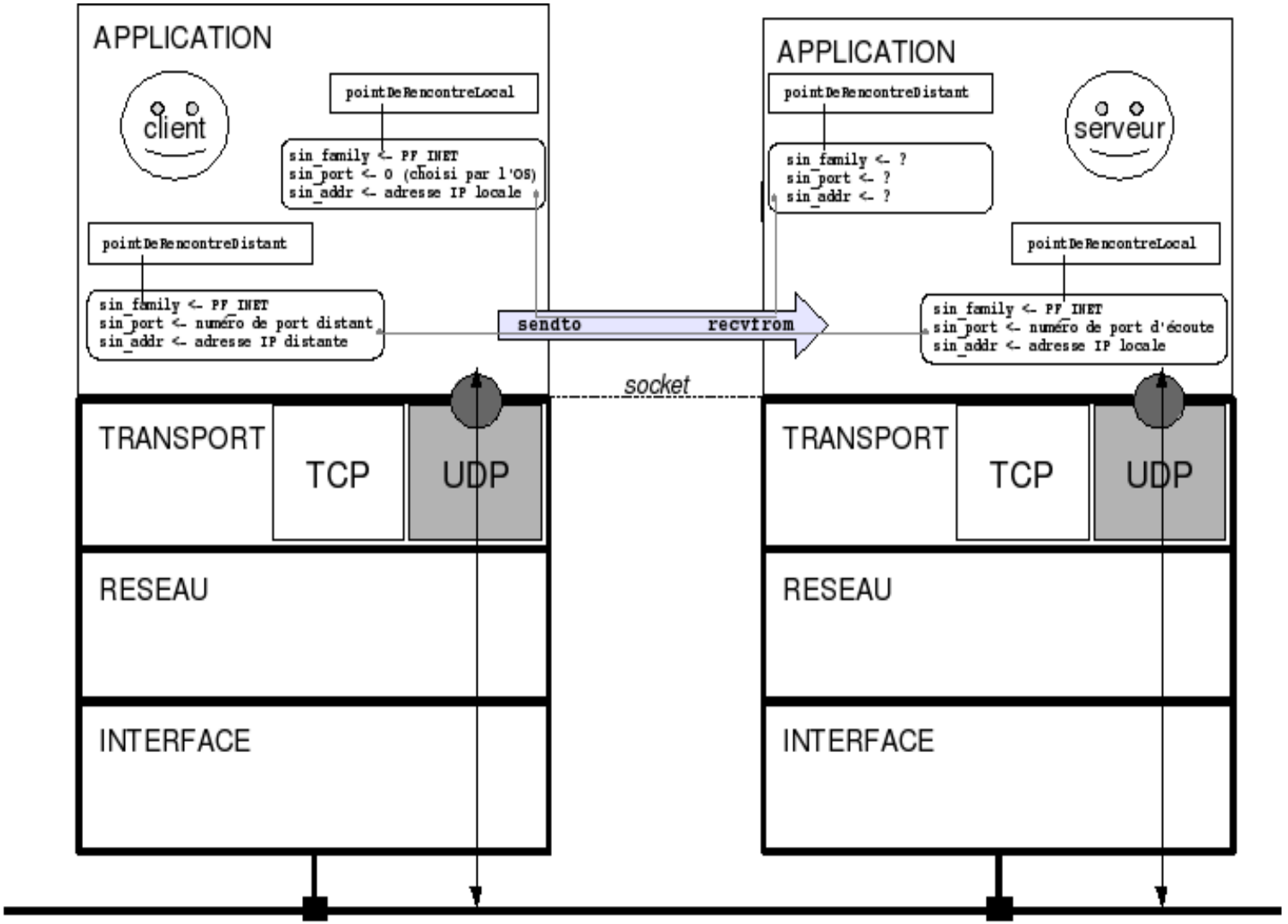
- Il regroupe les informations nécessaires à la reconnaissance mutuelle des deux **processus** qui dialoguent.
- Par exemple pour une communication Ipv4 (famille PF_INET), chaque processus définit ses points de rencontre local et distant :
  - Local : adresse IP source et numéro de port source
  - Distant : adresse IP destination et numéro de port destination

Remarque : le dialogue entre deux processus revient à mettre en relation **deux points de rencontre (local → distant et distant → local)**.

Point de rencontre (TCP)



Point de rencontre (UDP)



Création d'une socket

- On utilise la fonction **socket()** qui **retourne un descripteur de socket** (de type ***int***) en fonction des paramètres : famille (ou domaine), type (ou mode) et protocole.

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

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

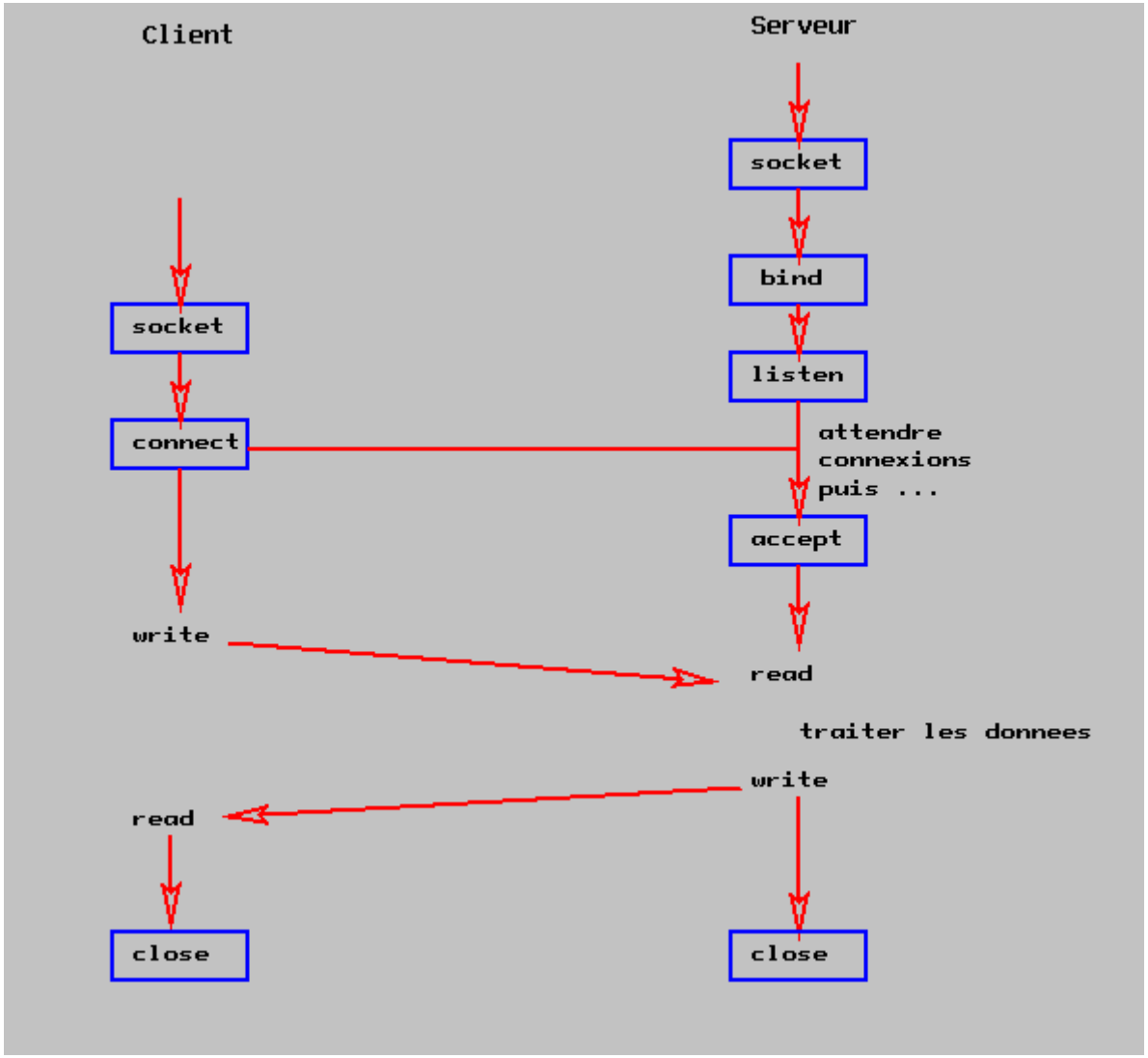
- Paramètres :
  - **domain** : **PF_INET** soit IPv4 Protocoles Internet voir ip(7)
  - **type** : **SOCK_STREAM** (mode connecté par défaut TCP), **SOCK_DGRAM** (mode non connecté par défaut UDP), **SOCK_RAW** (accès direct au protocole).
  - **protocol** : à **0** pour le protocole par défaut ou un protocole compatible au type

Création d'une socket (exemple)

- Exemples :

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

int socket_tcp, socket_udp, socket_icmp; //descripteurs de sockets

socket_tcp = socket(PF_INET, SOCK_STREAM, 0); //par défaut TCP

socket_udp = socket(PF_INET, SOCK_DGRAM, 0); //par défaut UDP

socket_icmp = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP); //on
choisit ICMP
```

Le mode connecté (SOCK_STREAM)



Le mode connecté (client TCP)

- Créer le point de communication : **socket()**

```
int socketDialogue = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
```

- Déterminer le point de rencontre **distant** (le serveur) : remplir une structure **sockaddr_in**

```
struct sockaddr_in pointDeRencontreDistant;
socklen_t longueurAdresse;

longueurAdresse = sizeof(pointDeRencontreDistant);
memset(&pointDeRencontreDistant, 0x00, longueurAdresse);
pointDeRencontreDistant.sin_family = PF_INET;
pointDeRencontreDistant.sin_port = htons(IPPORT_USERRESERVED);
inet_aton("192.168.52.83", &pointDeRencontreDistant.sin_addr);
```

- Se connecter au point de rencontre distant (le serveur) : **connect()**

```
connect(socketDialogue, (struct sockaddr *)&pointDeRencontreDistant,
longueurAdresse);
```

- Echanger des données avec le point de rencontre distant (le serveur) : **send()/recv()** ou **read()/write()** en (définissant et) utilisant un protocole et une procédure d'échange communs

- Fermer la connexion et le point de communication : **close()** ou **shutdown()**

```
close(socketDialogue);
```

Le mode connecté (serveur TCP)

- Créer le point de communication : **socket()**

```
int socketEcoute = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
```

- Renseigner le point de rencontre local en écoute : remplir une structure **sockaddr_in**

```
struct sockaddr_in pointDeRencontreLocal;
socklen_t longueurAdresse;

longueurAdresse = sizeof(pointDeRencontreLocal);
memset(&pointDeRencontreLocal, 0x00, longueurAdresse);
pointDeRencontreLocal.sin_family = PF_INET;
pointDeRencontreLocal.sin_port = htons(IPPORT_USERRESERVED);
pointDeRencontreLocal.sin_addr.s_addr = htonl(INADDR_ANY);
```

- Attacher (lier) le point de rencontre local à la *socket* d'écoute : **bind()**

```
bind(socketEcoute,(struct sockaddr *)&pointDeRencontreLocal,longueurAdresse);
```

- Attendre les connexions sur la *socket* d'écoute : **listen()**

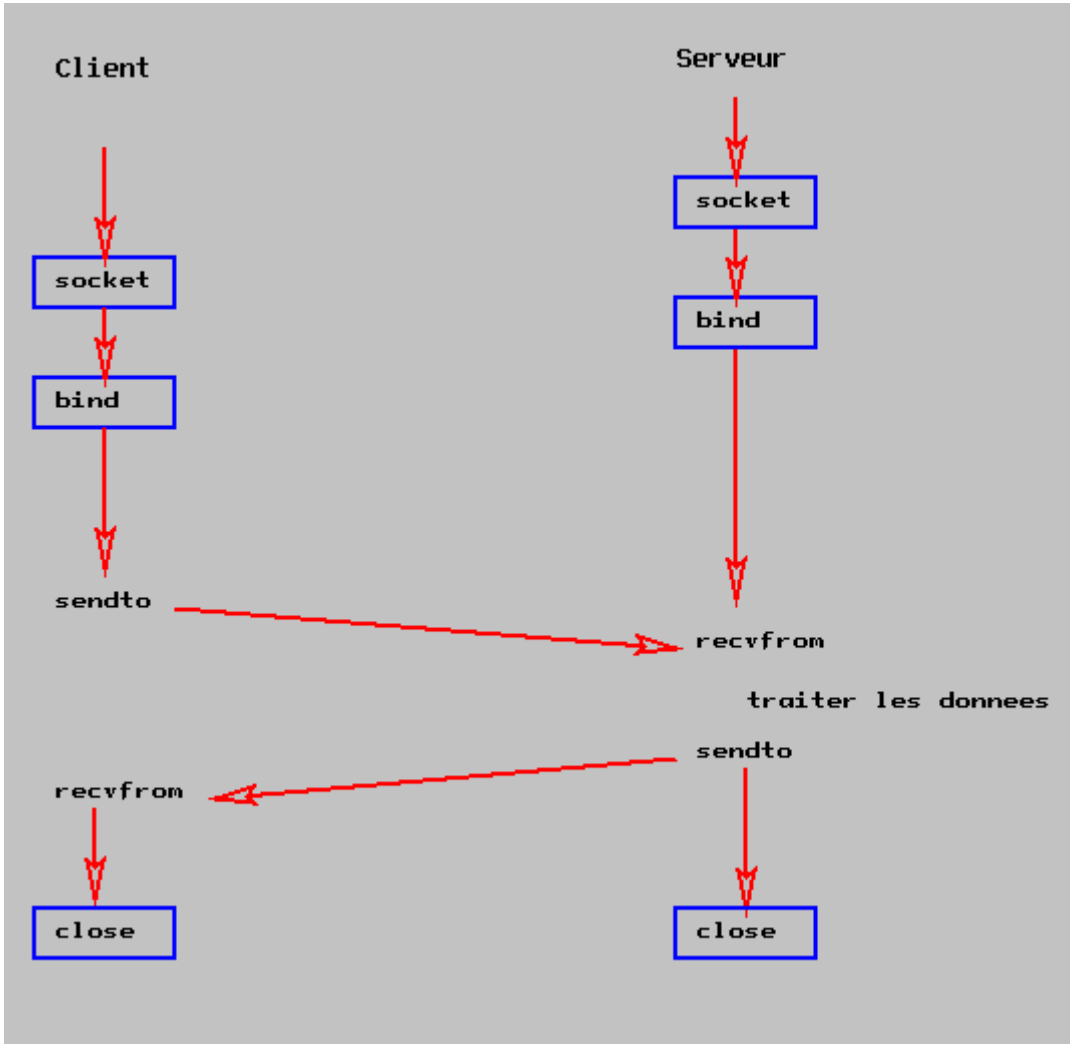
```
listen(socketEcoute, 5);
```

- Accepter une connexion : **accept()**

```
struct sockaddr_in pointDeRencontreDistant;
socketDialogue = accept(socketEcoute, (struct sockaddr *)&pointDeRencontre-
Distant, &longueurAdresse);
```

- Echanger des données avec le point de rencontre distant (le client) sur la socket de dialogue, ...

Le mode non connecté (SOCK_DGRAM)



Le mode non connecté (client UDP)

- Créer le point de communication : **socket()**

```
socketUDP = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);
```

- Renseigner le point de rencontre **local** : remplir une structure **sockaddr_in**

```
struct sockaddr_in pointDeRencontreLocal;
socklen_t longueurAdresse longueurAdresse = sizeof(pointDeRencontreLocal);
memset(&pointDeRencontreLocal, 0x00, longueurAdresse);
pointDeRencontreLocal.sin_family = PF_INET;
pointDeRencontreLocal.sin_port = 0;
pointDeRencontreLocal.sin_addr.s_addr = htonl(INADDR_ANY);
```

- Attacher (lier) le point de rencontre local à la *socket* d'écoute : **bind()**

```
bind(socketUDP, (struct sockaddr *)&pointDeRencontreLocal, longueurAdresse);
```

- Déterminer le point de rencontre **distant** (le serveur) : remplir une structure **sockaddr_in**

```
struct sockaddr_in pointDeRencontreDistant;
socklen_t longueurAdresse longueurAdresse = sizeof(pointDeRencontreDistant);
memset(&pointDeRencontreDistant, 0x00, longueurAdresse);
pointDeRencontreDistant.sin_family = PF_INET;
pointDeRencontreDistant.sin_port = htons(IPPORT_USERRESERVED);
inet_aton("192.168.52.83", &pointDeRencontreDistant.sin_addr);
```

- Echanger des données avec le serveur : **sendto()/recvfrom()**

- Fermer le point de communication : **close()**

```
close(socketUDP);
```

Le mode non connecté (serveur UDP)

- Créer le point de communication : **socket()**

```
socketUDP = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);
```

- Renseigner le point de rencontre **local** : remplir une structure **sockaddr_in**

```
struct sockaddr_in pointDeRencontreLocal;
socklen_t longueurAdresse;

longueurAdresse = sizeof(pointDeRencontreLocal);
memset(&pointDeRencontreLocal, 0x00, longueurAdresse);
pointDeRencontreLocal.sin_family = PF_INET;
pointDeRencontreLocal.sin_port = htons(IPPORT_USERRESERVED);
pointDeRencontreLocal.sin_addr.s_addr = htonl(INADDR_ANY);
```

- Attacher (lier) le point de rencontre local à la *socket* d'écoute : **bind()**

```
bind(socketUDP, (struct sockaddr *)&pointDeRencontreLocal, longueurAdresse);
```

- Echanger des données avec le client : **sendto()/recvfrom()** en (définissant et) utilisant un protocole et une procédure d'échange communs

- Fermer le point de communication : **close()**

```
close(socketUDP);
```

Serveur : accepter les connexions

- Plusieurs situations de départ pour **accept**er des connexions :
  - une **seule *socket* d'écoute en mode bloquant** (cas classique)
  - **plusieurs sockets d'écoute** : il est important de ne pas rester bloqué en attente avec l'appel **`accept`** sur une seule d'entre elles. Il est alors possible d'utiliser l'appel système **`select`** ou **`poll`** sur l'ensemble des sockets pour déterminer la disponibilité des données à lire, et d'effectuer ensuite l'appel **`accept`** sur celles qui ont effectivement reçu des demandes de connexion.

- *Remarques :*
  - Il est également possible de demander à une socket d'envoyer le signal **`SIGIO`** lorsqu'une activité la concernant se produit.
  - On peut déléguer l'attente de connexion à un service standard, par exemple **`inetd`** qui assure l'écoute pour tous les serveurs qui lui sont déclarés

Serveur : traiter les demandes

- Plusieurs situations de traitement :
  - **traitement séquentiel** : ceci a un sens si le traitement est rapide, car pendant ce temps la file d'attente des demandes de connexions peut s'allonger ...
  - **traitement parallèle ou multi-tâche** : c'est la méthode classique, on se dépêche de créer un processus fils (appel **`fork()`**) qui traite lui-même l'échange. Le processus père lui continue l'attente des demandes de connexion.

- *Remarques :*
  - Lors du **`fork()`**, les descripteurs du processus père sont dupliqués : on se retrouve donc avec 4 *sockets* (2 pour le père et 2 pour le fils) et il faut donc fermer les *sockets* inutiles (la *socket* d'écoute pour le fils et la *socket* d'échange pour le père)
  - Le temps de création des processus fils peut devenir une charge importante pour le système. Dans ce cas, on peut envisager les solutions suivantes : création anticipée des processus, utilisation des **threads**

Serveur : multi-clients (exemple)

```
while(1)
{
 /* on attend les demandes de connexion des clients (appel bloquant) */
 socketDialogue = accept(socketEcoute,(struct sockaddr *)&,&longueurAdresse);

 /* Création du processus de dialogue */
 switch(fork())
 {
  case -1 :       perror("fork:"); exit(-1);
  case 0  : /* processus fils */
  /* le fils n'utilise pas la socket d'écoute */
  close(socketEcoute);
  /* dialogue avec le client sur la socket de dialogue */
  /* ... */
  /* à la fin du dialogue, on ferme la socket de dialogue */
  close(socketDialogue); exit(0); /* fin du processus fils */
  default :       /* processus pere */
  /* le pere n'utilise pas la socket de dialogue */
  close(socketDialogue);
  /* le pere se replace en attente des demandes de connexion */
 }
}
```

Endianness

- En informatique, certaines données telles que les nombres entiers peuvent être représentées sur plusieurs octets. L'ordre dans lequel ces octets sont organisés en mémoire ou dans une communication est appelé *endianness* (mot anglais traduit par « boutisme »).
- De la même manière que certains langages humains s'écrivent de gauche à droite, et d'autres s'écrivent de droite à gauche, il existe une alternative majeure à l'organisation des octets représentant une donnée : l'orientation ***big-endian*** et l'orientation ***little-endian***.
- Les termes *big-endian* et *little-endian* ont été empruntés aux Voyages de Gulliver de Jonathan Swift, roman dans lequel deux clans de Lilliputiens se font la guerre à cause de la manière différente qu'ils ont de casser les œufs à la coque : par le gros ou le petit bout.

Bid-endian et Little-endian

- ***Big endian*** (unité l'octet) : Quand certains ordinateurs enregistrent un entier sur 32 bits en mémoire, par exemple 0xA0B70708 en notation hexadécimale, ils l'enregistrent dans des octets dans l'ordre qui suit : A0 B7 07 08 (dans l'ordre, octect de poids le plus fort vers l'octet de poids le plus faible).
  - Les architectures qui respectent cette règle sont dites *big-endian* (ou mot de poids fort en tête), par exemple les processeurs Motorola 68000, les SPARC (Sun Microsystems) ou encore les protocoles internet TCP/IP.
- ***Little endian*** (unité l'octet) : Les autres ordinateurs enregistrent 0xA0B70708 dans l'ordre suivant : 08 07 B7 A0, c'est-à-dire avec l'octet de poids le plus faible en premier.
  - De telles architectures sont dites *little-endian* ou mot de poids faible en tête (par exemple, les processeurs x86, qui se trouvent dans les PC).

Les macros de conversion

- La représentation des données pouvant être différente selon les machines (*host*), on utilise des macros de conversions afin de respecter la représentation des données au niveau réseau (*network*) dans les en-têtes des différents protocoles utilisés :

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

unsigned long int htonl(unsigned long int hostlong); // host to network (long)
unsigned short int htons(unsigned short int hostshort); // host to network (short)
unsigned long int ntohl(unsigned long int netlong); // network to host (long)
unsigned short int ntohs(unsigned short int netshort); // network to host (short)
```

- Les protocoles de la couche Application privilégie l'échange de données sous forme ASCII (caractère sur 8 bits) car il n'y pas d'inversion

- Pour l'échange de données numériques (types int, float, ...), on pourra alors utilisers un protocole spécifique comme XDR (*eXternal Data Representation*)

Fonctions de service

- Obtenir des informations (adresse IP, nom) sur la machine (structure **`hostent`**) : **`gethostbyname(), gethostbyaddr()`**
- Obtenir le numéro de port à partir du nom de service (structure **`servent`**) : **`getservbyname()`**
- Obtenir le numéro de protocole à partir du nom (structure **`protoent`**) : **`getprotobyname()`**
- Obtenir des informations sur la *socket* : **`getsockname(), getpeername()`**
- Traduction d'adresses et de services réseaux : **`getnameinfo(), getaddrinfo()`**
- Routines de manipulation d'adresses Internet : **`inet_aton(), inet_addr()`**
- Obtenir le message d'une erreur système : **`perror(), strerror()`**

Bibliographie

- "TCP/IP sous Linux" de JF Bouchaudy - Formation Tsoft © Ed. Eyrolles
- "TCP/IP Administration de réseau" de Craig Hunt © Ed. O'Reilly
- "Les protocoles TCP/IP et Internet" d'Eric Lapaille © NetLine 1999
- "Webmaster in a nutshell" © Ed. O'Reilly
- "Technique des réseaux locaux sous Unix" de L. Toutain © Ed. Hermes
- "Pratique des réseaux locaux d'entreprise" de JL Montagnier © Ed. Eyrolles
- "Transmission et Réseaux" de S. Lohier et D. Present © ED. DUNOD
- Les sites www.frameip.com, fr.wikipedia.org, www.w3.org, etc ...