Programmation réseau

Cyril Rabat
cyril.rabat@univ-reims.fr

Licence 3 Informatique - Info0601 - Systèmes d'exploitation - concepts avancés

2021-2022

Cours n°10

Programmation réseau en C : modes connecté et non connecté
Surveillance de descripteurs de fichier

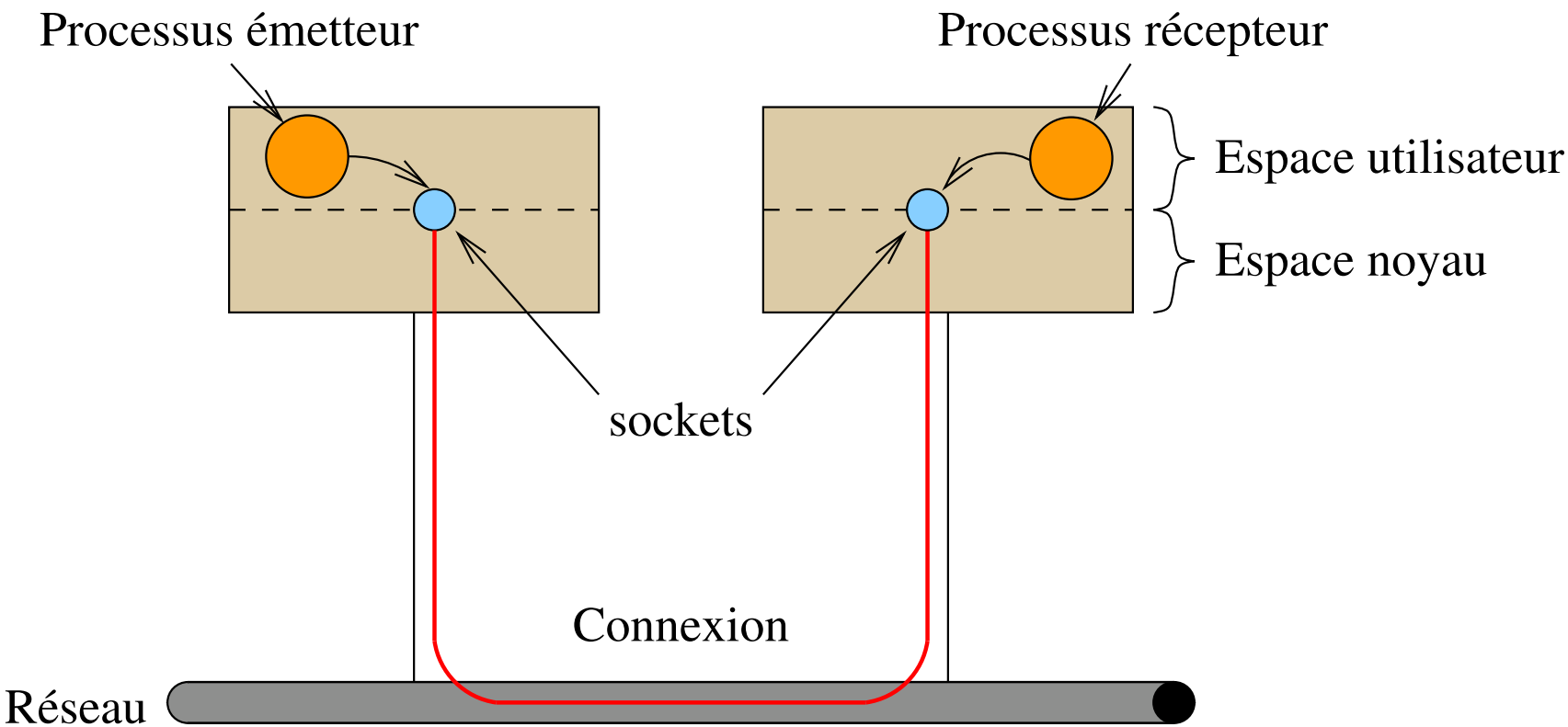
Version 25 février 2022

Table des matières

Les *sockets*

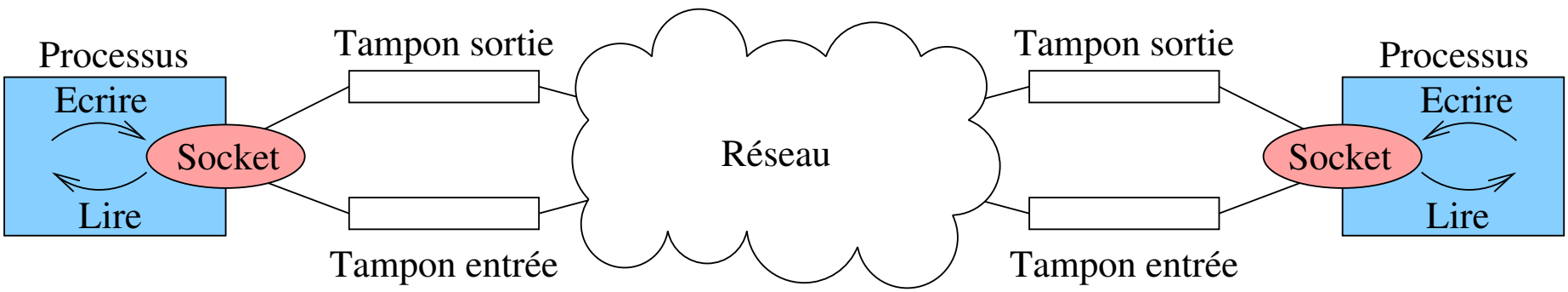
- Introduites sur les systèmes BSD
- Similaires (dans le fonctionnement) aux tubes :
  ↪ Communication entre plusieurs processus
  ↪ Lectures destructrices
- Associées à un descripteur de fichier :
  ↪ Entrée dans la table des fichiers ouverts
  ↪ *i-node* qui ne pointe vers aucune donnée
- La différence :
  - Peuvent être utilisées pour communiquer dans les deux sens
  - Les processus peuvent se trouver sur des machines distantes

Les *sockets* et le système d'exploitation (1/2)

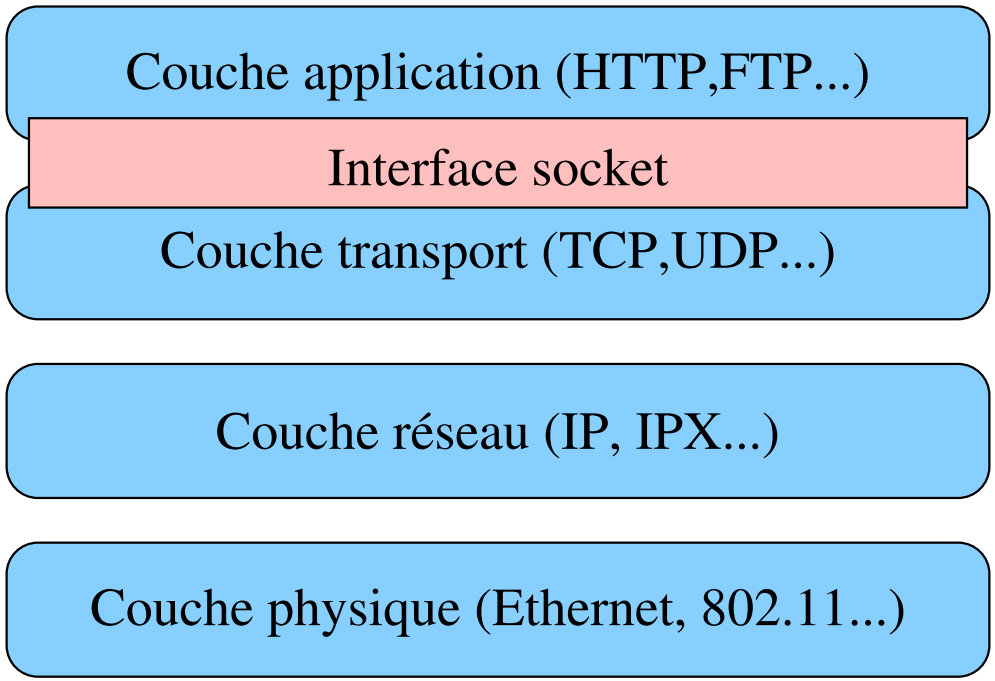


Les *sockets* et le système d'exploitation (2/2)

- Pour chaque *socket* :
  ↪ Structures nécessaires gérées par le système
  ↪ Association de tampons d'entrée et de sortie



Les *sockets* dans le modèle TCP/IP



- **Remarque** : les *sockets* sous *Unix/Linux* permettent de forger des messages depuis les couches basses !

Création d'une *socket*

- API de programmation complète
- Création d'une *socket* : fonction `socket`
- Possibilité :
  ↪ De préciser le domaine (IPv4, IPv6, local...)
  ↪ Le type de communication (connecté ou non connecté)
  ↪ Le protocole utilisé (si plusieurs disponibles)
- Une *socket* est associée à une adresse :
  ↪ Par exemple : adresse IP + numéro de port TCP/UDP

La fonction `socket` (1/3)

- **En-tête de la fonction (S2)** :
  - `int socket(int domaine, int type, int protocole)`
  - *Inclusions* : `sys/socket.h` voire `sys/types.h`
- **Explications** : création d'une *socket* et retour du descripteur de fichier associé
- **Le domaine** :
  - Famille de protocoles à utiliser
  - Par exemple :
    - `AF_LOCAL` ou `AF_UNIX` : local à l'ordinateur
    - `AF_INET` : protocoles Internet IPv4
    - `AF_INET6` : protocoles Internet IPv6
    - `AF_PACKET` : interface paquet bas-niveau

La fonction `socket` (2/3)

- **Le type** :
  - Dépend du domaine spécifié
  - Fixe la sémantique des communications :
    - `SOCK_STREAM` : mode connecté, flux d'octets
    - `SOCK_DGRAM` : mode non connecté, transmissions par paquets
    - `SOCK_RAW` : mode bas niveau (nécessite des droits spécifiques)
- **Le protocole** :
  - Spécifie le protocole à utiliser dans la famille
  - Normalement, un seul protocole pour un type de *socket* et de domaine donné (consulter le fichier `/etc/protocols`) :
    ↪ Vaut généralement 0 (le système choisit)
  - Exemples : `IPPROTO_TCP`, `IPPROTO_UDP`

La fonction `socket` (3/3)

- **Retour et erreurs générées** :
  - Descripteur de fichier créé ou -1 en cas d'erreur
  - Quelques erreurs :
    - `EACCES` : famille et protocole non autorisés
    - `EINVAL` : famille ou protocole inconnus
    - `EPROTONOSUPPORT` : type de protocole non disponible ou non disponible dans la famille spécifiée
    - `EAFNOSUPPORT` : famille non supportée

Nommage de la *socket*

- Création de la *socket*, locale au processus :
  ↪ Connaissance locale (descripteur de fichier)
- Nécessaire de lui attribuer une adresse :
  ↪ Adresse IP + numéro de port pour TCP/IP ou UDP/IP
- Une fois nommée, elle est accessible de "l'extérieur" :
  - Dépend de l'adresse (*localhost* ou non)
  - Dépend du réseau (privé ou non)
  - Dépend du "système" : attention aux pare-feux logiciel !
  - Dépend de la "configuration" du réseau (pare-feux)
- Utilisation de la fonction `bind`

Si la *socket* n'est pas nommée avant l'envoi du premier message, elle est nommée par défaut : adresse de l'interface par défaut, numéro de port aléatoire

La fonction `bind` (1/2)

- **En-tête de la fonction (S2)** :
  - `int bind(int fd, struct sockaddr *adresse, socklen_t longueur)`
  - *Inclusions* : `sys/socket.h` voire `sys/types.h`
- **Explications** :
  - Nommage d'une *socket* :
    ↪ Attribution d'une adresse
- **Paramètres** :
  - `fd` : le descripteur de fichier correspondant à la *socket*
  - `adresse` : adresse à attribuer à la *socket*
  - `longueur` : longueur de l'adresse en octets

La fonction `bind` (2/2)

- La structure générique `sockaddr` pour spécifier l'adresse :

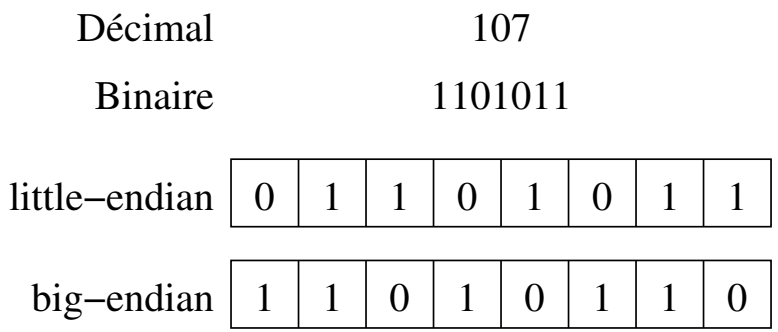
```
struct sockaddr {
  sa_family_t sa_family;   // Famille de protocoles
  char        sa_data[14]; // Adresse
}
```

Pas d'utilisation directe :
utilisation de structures spécifiques au domaine et à la famille

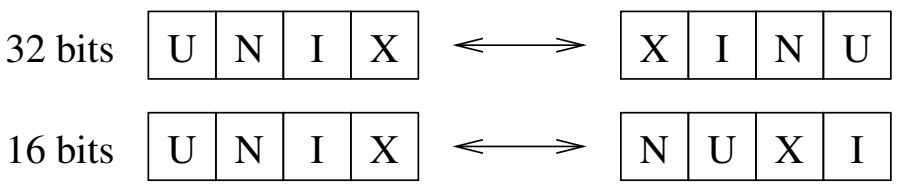
- **Retour et erreurs générées** :
  - 0 en cas de réussite ou -1 en cas d'erreur
  - Quelques erreurs :
    - `EACCES` : adresse protégée et l'utilisateur n'est pas *root*
    - `EADDRINUSE` : adresse déjà utilisée

Problématique de la représentation des données (rappels)

- Le petit-boutiste et le grand-boutiste
  ↪ De l'anglais *little-endian* et *big-endian*
- Représentation des binaires dans un ordre différent
- Arrangements des groupes d'octets différents (mots de 16/32/64 bits)
- *Intelx86* sont *little-endian*, *Motorola* sont *big-endian*
- *MacOS* est *big-endian*, *Windows* est *little-endian*



little-endian vs big-endian



Différents arrangements d'octets

Acheminer des données dans le réseau

- Représentation des données :
  ↪ Propre à la machine émettrice
- Gestion de la compatibilité :
  ↪ À l'application de le faire
- Problèmes :
  - L'adresse doit être lisible dans le réseau
  - Idem pour le numéro de port
- Utilisation de fonctions de conversion :
  - Conversion entier court/long vers format réseau
  - Conversion format réseau vers entier court/long
- Structures de différentes tailles :
  - Attention à l'initialisation
  - Solution : utiliser `memset`

Remplissage/effacement d'une zone mémoire

- **En-tête de la fonction (S3)** :
  - `void *memset(void *source, int octet, size_t nb)`
  - *Inclusions* : `string.h`
- **Paramètres** :
  - `source` : adresse de la zone mémoire
  - `octet` : valeur de remplissage
  - `nb` : nombre d'octets à remplir
- **Valeur retournée** :
  - Pointeur vers la zone mémoire (inutile ?)

Ancienne fonction `bzero` dépréciée

Les fonctions de conversion de format (S3)

- `uint16_t htons(uint16_t hote_court)` :
  ↪ *Convertit un entier court au format réseau.*
- `uint16_t ntohs(uint16_t reseau_court)` :
  ↪ *Convertit un entier court au format local.*
- `uint32_t htonl(uint32_t hote_long)` :
  ↪ *Convertit un entier long au format réseau.*
- `uint32_t ntohl(uint32_t reseau_long)` :
  ↪ *Convertit un entier long au format local.*

Moyens mnémotechniques :
`h` pour *host*, `n` pour *network*, `to` pour vers, *s* pour *short* et `l` pour *long*

Structures pour représenter les adresses IP

- **Adresse IPv4** :

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

  - 4 entiers stockés sur un octet
  - Attention : stockés dans l'ordre inverse

- **Adresse IPv6** :

```
struct in6_addr {
    unsigned char   s6_addr[16];
};
```

  - Chaque case = 1 octet
  - Attention : stocké dans l'ordre de lecture "normal"

Convertir une chaîne de caractères en adresse réseau

- **En-tête de la fonction (S3)** :
  - `int inet_pton(int famille, const char *source, void *destination)`
  - *Inclusions* : `arpa/inet.h`
- **Paramètre(s)** :
  - `famille` : soit `AF_INET`, soit `AF_INET6`
  - `source` : adresse au format texte
  - `destination` : une structure
    ↪ `struct in_addr` pour IPv4 ou `struct in6_addr` pour IPv6
- **Valeurs retournées** :
  - 1 en cas de réussite, 0 si l'adresse ne correspond pas à la famille, -1 si la famille n'est pas correcte

Exemple d'utilisation pour IPv4

```
struct in_addr adresseIPv4;

if(inet_pton(AF_INET, "127.0.0.1", &adresseIPv4) != 1) {
  fprintf(stderr, "Erreur lors de la conversion\n");
  exit(EXIT_FAILURE);
}

printf("Adresse : %d\n", adresseIPv4.s_addr);
```

- **Résultat** :
  - Affichage : `16777343`
  - Justification : $127 \times 256^0 + 0 \times 256^1 + 0 \times 256^2 + 1 \times 256^3$

Exemple d'utilisation pour IPv6

```
struct in6_addr adresseIPv6;

if(inet_pton(AF_INET6, "::1", &adresseIPv6) != 1) {
  fprintf(stderr, "Erreur lors de la conversion\n");
  exit(EXIT_FAILURE);
}

printf("Adresse : ");
for(i = 0; i < 15; i++)
  printf("%d:", (int)adresseIPv6.s6_addr[i]);
printf("%d\n", (int)adresseIPv6.s6_addr[i]);
```

- **Résultat** :
  - Affichage : `0:0:0:0:0:0:0:1`
  - Justification : il s'agit de l'adresse locale en IPv6

Convertir une adresse réseau en chaîne de caractères (1/2)

- **En-tête de la fonction (S2)** :
  - `const char *inet_ntop(int famille, const void *source, char *destination, socklen_t taille)`
  - *Inclusions* : `arpa/inet.h`
- **Paramètre(s)** :
  - `famille` : soit `AF_INET`, soit `AF_INET6`
  - `source` : adresse au format réseau
    ↪ `struct in_addr` pour IPv4
    ↪ `struct in6_addr` pour IPv6
  - `destination` : chaîne de taille...
    - ...`INET_ADDRSTRLEN` octets pour IPv4
    - ...`INET6_ADDRSTRLEN` octets pour IPv6
  - `taille` : la taille de `dst` (`INET_ADDRSTRLEN` ou `INET6_ADDRSTRLEN`)

Convertir une adresse réseau en chaîne de caractères (2/2)

- **Valeur retournée et erreurs générées**
  - Pointeur vers `destination` en cas de réussite ou `NULL` en cas d'erreur
  - Les erreurs possibles :
    - `EAFNOSUPPORT` : famille non supportée
    - `ENOSPC` : chaîne de destination trop petite

Construire une adresse pour `bind`

- Adresse : constituée d'une adresse réseau + d'un numéro de port
- Différentes structures suivant le protocole :
  - `sockaddr_in` pour une adresse IPv4
  - `sockaddr_in6` pour une adresse IPv6
- Doivent être transtypées en `struct sockaddr*` pour `bind`
- Possible de laisser le système d'exploitation choisir une adresse IPv4 :
  - Utilisation de la constante `INADDR_ANY`
- Idem pour IPv6 :
  - Utilisation de la variable globale `in6addr_any`

Structures utilisées pour IPv4 et IPv6

- La structure `sockaddr_in` pour représenter une adresse IPv4 :

```
struct sockaddr_in{
  sa_family_t     sin_family;  /* Ici AF_INET */
  in_port_t       sin_port;    /* Numéro de port */
  struct in_addr  sin_addr;    /* Adresse IPv4 */
}
```

- La structure `sockaddr_in6` pour représenter une adresse IPv6 :

```
struct sockaddr_in6 {
  sa_family_t     sin6_family;   /* Ici AF_INET6 */
  in_port_t       sin6_port;     /* Numéro de port */
  uint32_t        sin6_flowinfo; /* Information de flux
      IPv6 */
  struct in6_addr sin6_addr;     /* Adresse IPv6 */
  uint32_t        sin6_scope_id; /* Scope ID */
};
```

Exemple complet avec IPv4

```
struct sockaddr_in adresse;
int fd;

if((fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) == -1) {
  perror("Erreur lors de la création de la socket ");
  exit(EXIT_FAILURE);
}

memset(&adresse, 0, sizeof(struct sockaddr_in));
adresse.sin_family = AF_INET;
adresse.sin_addr.s_addr = INADDR_ANY;
adresse.sin_port = 0;

if(bind(fd, (struct sockaddr*)&adresse,
        sizeof(struct sockaddr_in)) == -1) {
   perror("Erreur lors du nommage de la socket ");
   exit(EXIT_FAILURE);
}
```

Exemple complet avec IPv6

```
struct sockaddr_in6 adresse;
int fd;

if((fd = socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP)) == -1)
    {
  perror("Erreur lors de la création de la socket ");
  exit(EXIT_FAILURE);
}

memset(&adresse, 0, sizeof(struct sockaddr_in6));
adresse.sin6_family = AF_INET6;
adresse.sin6_addr.s_addr = in6addr_any;
adresse.sin6_port = 0;

if(bind(fd, (struct sockaddr*)&adresse,
        sizeof(struct sockaddr_in)) == -1) {
   perror("Erreur lors du nommage de la socket ");
   exit(EXIT_FAILURE);
}
```

Fonctions `getaddrinfo` et `getnameinfo`

- Possibilité de faire une correspondance nom/adresse (DNS) :
  - Utilisation de la fonction `getaddrinfo`
  - Retourne des structures de type `addrinfo`
  - Utilisables directement pour nommer des *sockets*
- Correspondance inverse possible (*reverse DNS*) :
  - Fonction `getnameinfo`

```
struct addrinfo {
  int              ai_flags;     /* Options */
  int              ai_family;    /* AF_INET, AF_INET6 */
  int              ai_socktype;  /* SOCK_STREAM ou SOCK_DGRAM
      */
  int              ai_protocol;  /* Protocole utilisé */
  size_t           ai_addrlen;   /* Taille de l'adresse */
  struct sockaddr *ai_addr;      /* Adresse */
  char            *ai_canonname; /* Nom canonique */
  struct addrinfo *ai_next;      /* Pointeur suivant */
};
```

Utilisation de `getaddrinfo`

- Objectif : créer une adresse utilisable pour une *socket*
- Nécessite donc de spécifier les informations de base :
  - Utilisation d'une structure `addrinfo` en entrée
  - Famille de l'adresse + type de *socket* + protocole
- Étapes de fonctionnement :
  1. Création d'une `addrinfo` pour indiquer ce qu'on attend
  2. Appel de `getaddrinfo` qui effectue la requête
  3. Traitement des adresses créées par `getaddrinfo`

Exemple d'utilisation de `getaddrinfo` (IPv4) (1/2)

```
struct addrinfo *resultat;
struct addrinfo demande;

memset(&demande, 0, sizeof(struct addrinfo));
demande.ai_family = AF_INET;
demande.ai_socktype = SOCK_DGRAM;
demande.ai_flags = AI_PASSIVE;
demande.ai_protocol = 0;
demande.ai_canonname = NULL;
demande.ai_addr = NULL;
demande.ai_next = NULL;

if(getaddrinfo(argv[1], NULL, &demande, &resultat) == -1) {
  perror("Erreur getaddrinfo ");
  exit(EXIT_FAILURE);
}
```

Exemple d'utilisation de `getaddrinfo` (IPv4) (2/2)

```
char adresseIPv4[INET_ADDRSTRLEN];
int i = 0;

while(resultat != NULL) {
  printf("Entrée␣%d␣:\n", i);
  if(inet_ntop(AF_INET, resultat->ai_addr,
               adresseIPv4, INET_ADDRSTRLEN) == NULL)
    printf("Conversion␣impossible...\n");
  else
    printf("Adresse␣:␣%s,␣%s\n", resultat->ai_canonname,
                                 adresseIPv4);

  resultat = resultat->ai_next;
  ++i;
}
```

Exemple d'utilisation de `getnameinfo` (IPv4)

```
memset(&adresse, 0, sizeof(struct sockaddr_in));
adresse.sin_family = AF_INET;
inet_pton(AF_INET, argv[1], &adresse.sin_addr.s_addr);
adresse.sin_port = htons(atoi(argv[2]));

if(getnameinfo((struct sockaddr*)&adresse, sizeof(adresse),
               nomHote, TAILLE_MAX, nomService, TAILLE_MAX,
               NI_NAMEREQD) != 0)
  perror("Erreur lors de la conversion de nom ");
else
  printf("Nom : %s %s\n", nomHote, nomService);
```

Sortie écran

```
./getnameinfo 194.57.105.10 80
Nom : www.univ-reims.fr http
```

Notes sur `gethostbyname` et `gethostbyaddr`

- Fonctions permettant le même traitement
- **Attention** :
  - POSIX.1-2001 les marque comme obsolètes
  - POSIX.1-2008 les supprime totalement ! ! !
- **Rappels** :
  - De nombreux exemples sur Internet (et dans des livres) utilisent ces fonctions
  - Vérifiez la partie *Conformités* dans les pages du manuel

La fonction `getaddrinfo`

- **En-tête de la fonction :**
  - `int getaddrinfo(const char *node, const char *service, const struct addrinfo *hints, struct addrinfo **res)`
  - *Inclusions* : `sys/socket.h`, `sys/types.h`, `netdb.h`
- **Paramètre(s) :**
  - `node` : nom de l'hôte
  - `service` : nom du service
  - `hints` : paramètres pour les adresses retournées
  - `res` : les adresses retournées
- **Valeur retournée et erreurs générées :**
  - Retourne 0 en cas de succès, -1 sinon
  - Quelques erreurs :
    - `EAI_AGAIN` : résolution impossible à ce moment
    - `EAI_ADDRFAMILY` : l'hôte n'a pas d'adresse dans la famille demandée
    - `EAI_FAMILY` : famille demandée non supportée
    - `EAI_NONAME` : nom impossible à résoudre (ou nom+service à `NULL`)

La fonction `getnameinfo` (1/2)

- **En-tête de la fonction** :
  - `int getnameinfo(const struct sockaddr *sa, socklen_t salen, char *host, size_t hostlen, char *serv, size_t servlen, int flags)`
  - *Inclusions* : `sys/socket.h`, `sys/types.h`, `netdb.h`
- **Paramètre(s)** :
  - `sa` et `salen` : adresse à résoudre + taille
  - `host` et `hostlen` : tampon alloué pour le nom de l'hôte (+ taille)
  - `serv` et `servlen` : tampon alloué pour le service (+ taille)
  - `flags` :
    - `NI_NAMEREQD` : provoque une erreur si la résolution est impossible
    - `NI_NUMERICHOST` : retourne la forme numérique pour l'hôte
      ↪ Par défaut si la résolution est impossible
    - `NI_NUMERICSERV` : idem pour le service

La fonction `getnameinfo` (2/2)

- **En-tête de la fonction** :
  - `int getnameinfo(const struct sockaddr *sa, socklen_t salen, char *host, size_t hostlen, char *serv, size_t servlen, int flags)`
  - *Inclusions* : `sys/socket.h`, `sys/types.h`, `netdb.h`
- **Valeur retournée et erreurs générées** :
  - Retourne 0 en cas de succès,
  - Quelques erreurs :
    - `EAI_AGAIN` : résolution impossible à ce moment
    - `EAI_FLAGS` : options incorrectes
    - `EAI_NONAME` : nom impossible à résoudre avec les paramètres fournis
    - `EAI_OVERFLOW` : taille de tampon trop petite

Récupérer le nom associé à une *socket*

- **En-tête de la fonction (S2)** :
  - `int getsockname(int sockfd, struct sockaddr *adresse, socklen_t *taille)`
  - *Inclusion* : `sys/socket.h`
- **Paramètre(s)** :
  - `sockfd` : le descripteur de la *socket*
  - `adresse` : l'adresse (le nom) associée à la *socket*
  - `taille` : la taille de l'adresse
- **Valeur retournée et erreurs générées** :
  - 0 en cas de réussite ou -1 en cas d'erreur
  - Quelques erreurs :
    - `EBADF` : le descripteur est invalide
    - `ENOTSOCK` : le descripteur ne correspond pas à une *socket*

Récupérer le nom du correspondant connecté sur une *socket*

- **En-tête de la fonction (S2)** :
  - `int getpeername(int sockfd, struct sockaddr *adresse, socklen_t *taille)`
  - *Inclusion* : `sys/socket.h`
- **Paramètre(s)** :
  - `sockfd` : le descripteur de la *socket*
  - `adresse` : l'adresse (le nom) du correspondant connecté à la *socket*
  - `taille` : la taille de l'adresse
- **Valeur retournée et erreurs générées** :
  - 0 en cas de réussite ou -1 en cas d'erreur
  - Erreurs (en plus de `getsockname`) :
    - `ENOTCONN` : la *socket* n'est pas connectée

Exemple d'utilisation de getsockname (1/2)

```
int socketfd;
if((socketfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) ==
     -1) {
  perror("Erreur␣lors␣de␣la␣création␣de␣la␣socket␣");
  exit(EXIT_FAILURE);
}

struct sockaddr_in adresse;
memset(&adresse, 0, sizeof(struct sockaddr_in));
adresse.sin_family = AF_INET;
adresse.sin_port = 0;
adresse.sin_addr.s_addr = htonl(INADDR_ANY);
if(bind(socketfd, (struct sockaddr*)&adresse,
        sizeof(struct sockaddr_in)) == -1) {
  perror("Erreur␣lors␣du␣nommage␣de␣la␣socket␣");
  exit(EXIT_FAILURE);
}
...
```

Exemple d'utilisation de `getsockname` (2/2)

```
...
struct sockaddr_in adresse2;
socklen_t taille = sizeof(struct sockaddr_in);
if(getsockname(socketfd, (struct sockaddr*)&adresse2,
               &taille) == -1) {
  perror("Erreur lors de la récupération du nom ");
  exit(EXIT_FAILURE);
}
port = ntohs(adresse2.sin_port);
```

Mode non connecté

1. Création de la *socket* sur le serveur
2. Attribution d'un nom à la *socket*
3. Le client crée lui-aussi une *socket*
4. Des informations peuvent être échangées

Mode non connecté : plusieurs clients/serveurs

- *Socket* sur le client : indépendante du serveur !
- *Socket* sur le serveur : indépendante des clients !

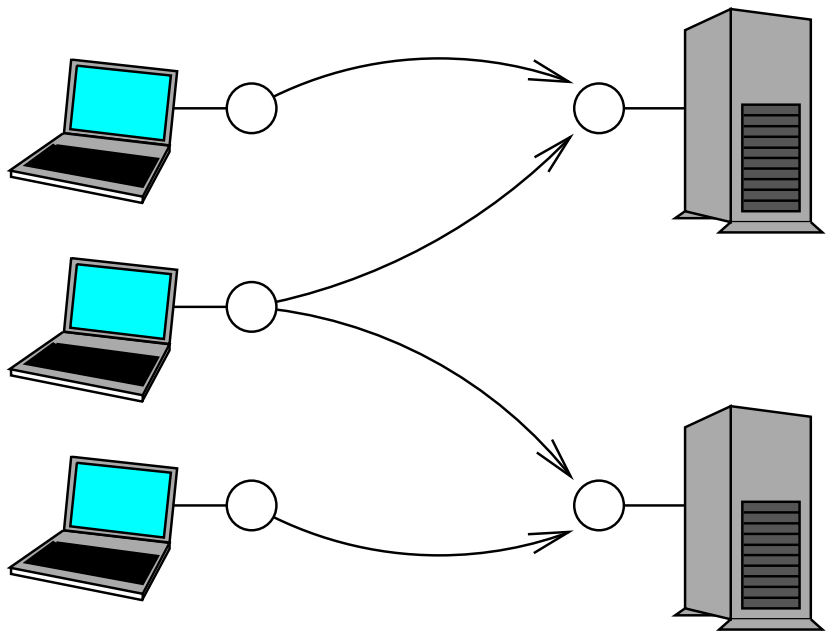
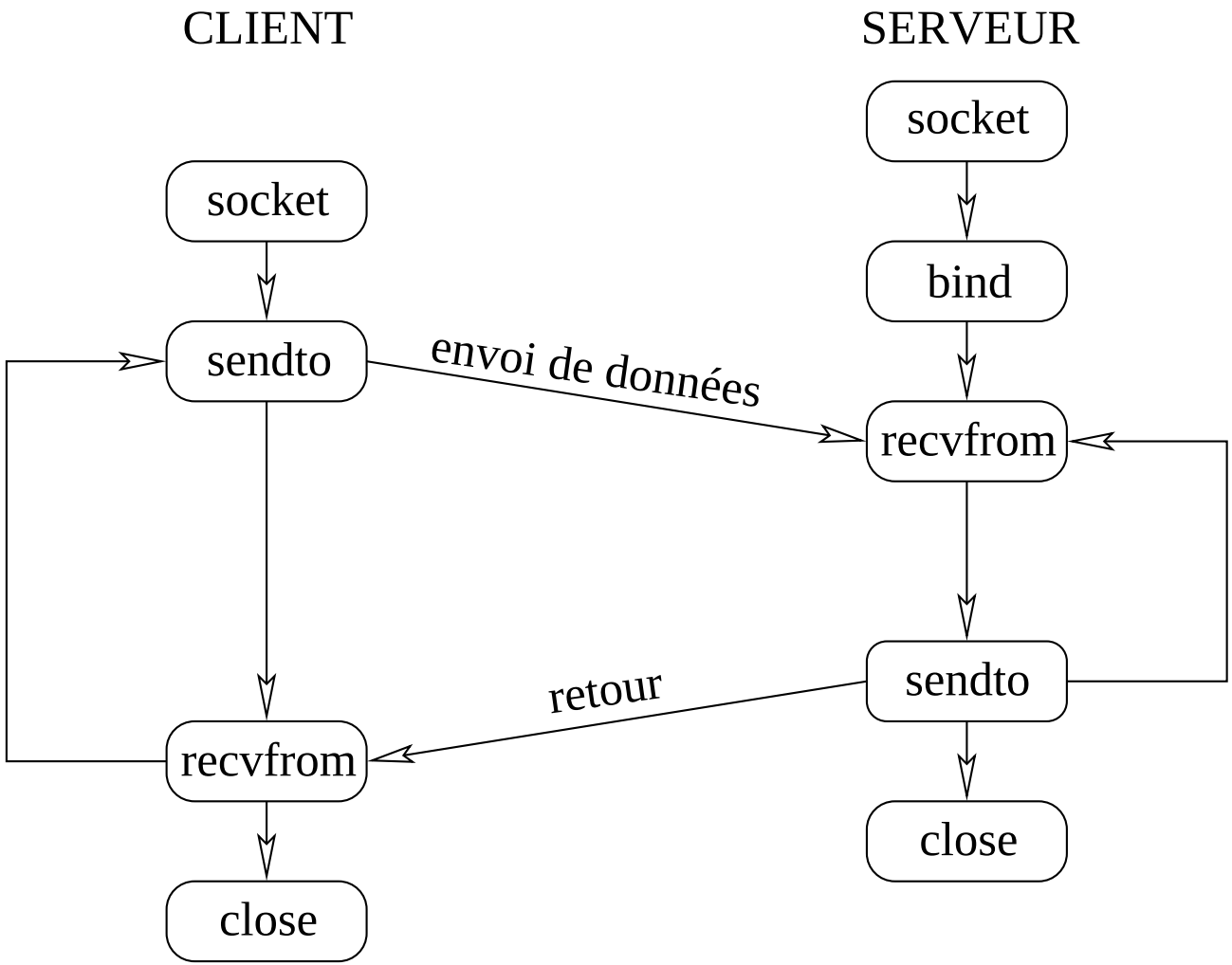


Diagramme des appels systèmes en mode non connecté



La fonction `sendto`

- **En-tête de la fonction (S2) :**
  - `ssize_t sendto(int sockfd, const void *message, size_t taille, int flags, const struct sockaddr *adresse_dest, socklen_t taille_adresse)`
  - *Inclusions* : `sys/types.h` et `sys/socket.h`
- **Paramètre(s) :**
  - `sockfd` : descripteur de la *socket*
  - `message` et `taille` : message de taille `taille`
  - `flags` : options (0 pour nous)
  - `adresse_dest` : adresse de destination
  - `taille_adresse` : longueur adresse de destination
- **Valeur retournée et erreurs générées**
  - Nombre de caractères envoyés ou -1 en cas d'erreur
  - Quelques erreurs :
    - `EBADF` : descripteur incorrect
    - `EINTR` : signal reçu avant l'envoi des données

La fonction `recvfrom`

- **En-tête de la fonction (S2)** :
  - `ssize_t recvfrom(int sockfd, void *message, size_t taille, int flags, struct sockaddr *adresse_source, socklen_t *taille_adresse)`
  - *Inclusions* : `sys/types.h` et `sys/socket.h`
- **Paramètre(s)** :
  - `sockfd` : descripteur de la *socket*
  - `message` et `taille` : message de taille `taille`
  - `flags` : options (0 pour le moment)
  - `adresse_source` : adresse source (remplie si non nulle)
  - `taille_adresse` : longueur adresse source
- **Valeur retournée et erreurs générées** :
  - Nombre de caractères reçus ou -1 en cas d'erreur
  - Quelques erreurs :
    - `EBADF` : descripteur incorrect
    - `EINTR` : signal reçu avant la réception des données

Exemple d'utilisation de `sendto` et `recvfrom` (1/2)

```
  /* Code serveur */
  int sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
...
  struct sockaddr_in adresseServeur;
  memset(&adresseServeur, 0, sizeof(struct sockaddr_in));
  adresseServeur.sin_family = AF_INET;
  adresseServeur.sin_port = htons(1234);
  adresseServeur.sin_addr.s_addr = htonl(INADDR_ANY);
  bind(sockfd, (struct sockaddr*)&adresseServeur,
       sizeof(struct sockaddr_in));
...
  requete_t requete;
  struct sockaddr_in adresseClient;
  socklen_t longueurAdresse = sizeof(struct sockaddr_in);
  recvfrom(sockfd, &requete, sizeof(requete_t), 0,
           (struct sockaddr*)&adresseClient, &longueurAdresse)
...
  reponse_t reponse; /* A initialiser */
...
  sendto(sockfd, &reponse, sizeof(reponse_t), 0,
         (struct sockaddr*)&adresseClient, longueurAdresse);
```

Exemple d'utilisation de `sendto` et `recvfrom` (2/2)

```
  /* Code client */
  int sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

  struct sockaddr_in adresseServeur;
  memset(&adresseServeur, 0, sizeof(struct sockaddr_in));
  adresseServeur.sin_family = AF_INET;
  adresseServeur.sin_port = htons(1234);
  inet_pton(AF_INET, "127.0.0.1", &adresseServeur.sin_addr);
...
  requete_t requete; /* A initialiser */
...
  sendto(sockfd, &requete, sizeof(requete_t), 0,
         (struct sockaddr*)&adresseServeur, sizeof(struct
             sockaddr_in));
...
  reponse_t reponse;
  recvfrom(sockfd, &reponse, sizeof(reponse_t), 0, NULL, 0);
```

Exemple d'une application client-serveur

- Serveur en attente de requêtes de la part de clients
- Récupération de l'heure ou de la date

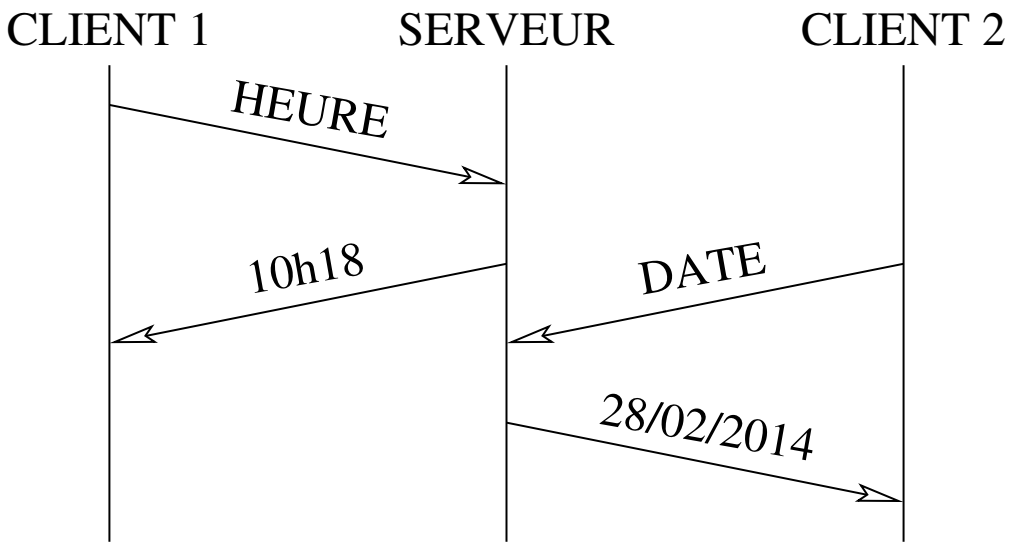


Diagramme d'échange entre un client et un serveur

Les structures utilisées : `structures.h`

```
#ifndef _STRUCTURES_
#define _STRUCTURES_

#define CODE_HEURE 1
#define CODE_DATE  2

typedef struct {
  int id;
  int code;
} requete_t;

typedef struct {
  int id;
  char resultat[256];
} reponse_t;

#endif
```

Le client : `client.c` (1/2)

```
int main(int argc, char *argv[]) {
  int sockfd;
  struct sockaddr_in adresseServeur;
  requete_t requete;
  reponse_t reponse;

  sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

  memset(&adresseServeur, 0, sizeof(struct sockaddr_in));
  adresseServeur.sin_family = AF_INET;
  adresseServeur.sin_port = htons(1234);
  inet_pton(AF_INET, "127.0.0.1", &adresseServeur.sin_addr);
```

Ici, nous considérons que le serveur et le client s'exécutent sur la même machine, d'où le 127.0.0.1 (adresse *localhost*)

Le client : `client.c` (2/2) (suite)

```
  requete.id = getpid();
  if(strcmp(argv[1], "HEURE") == 0)
    requete.code = CODE_HEURE;
  else
    requete.code = CODE_DATE;
  sendto(sockfd, &requete, sizeof(requete_t), 0,
         (struct sockaddr*)&adresseServeur,
         sizeof(struct sockaddr_in));

  recvfrom(sockfd, &reponse, sizeof(reponse_t), 0, NULL, 0);

  printf("Client : reponse reçue = (%d) : %s\n",
         reponse.id, reponse.resultat);

  close(sockfd);
  return EXIT_SUCCESS;
}
```

Le serveur : serveur.c (1/2)

```
int main(int argc, char *argv[]) {
  int sockfd;
  struct sockaddr_in adresseServeur, adresseClient;
  socklen_t longueurAdresse = sizeof(struct sockaddr_in);
  requete_t requete;
  reponse_t reponse;
  struct tm *date;
  time_t heure;

  sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

  memset(&adresseServeur, 0, sizeof(struct sockaddr_in));
  adresseServeur.sin_family = AF_INET;
  adresseServeur.sin_port = htons(1234);
  adresseServeur.sin_addr.s_addr = htonl(INADDR_ANY);
  bind(sockfd, (struct sockaddr*)&adresseServeur,
       sizeof(struct sockaddr_in));
```

Le serveur : serveur.c (2/2) (suite)

```
  recvfrom(sockfd, &requete, sizeof(requete_t), 0,
           (struct sockaddr*)&adresseClient, &longueurAdresse)
              ;

  heure = time(NULL);
  date = gmtime(&heure);
  reponse.id = requete.id;
  if(requete.code == CODE_HEURE)
    sprintf(reponse.resultat, "%.2dh%.2d",
            date->tm_hour, date->tm_min);
  else
    sprintf(reponse.resultat, "%.2d/%.2d/%4d", date->tm_mday,
            date->tm_mon + 1, date->tm_year + 1900);

  sendto(sockfd, &reponse, sizeof(reponse_t), 0,
    (struct sockaddr*)&adresseClient, longueurAdresse);

  close(sockfd);
  return EXIT_SUCCESS;
}
```

Gestion de requêtes multiples

- Dès qu'une requête est reçue : fin du serveur
- Solution : boucle infinie

```
while(1) {
  recvfrom(...);
  ...
  sendto(...);
}
```

Arrêt "propre" du serveur (1/2)

- Arrêt du serveur avec un signal : par exemple "`SIGINT`"
- Attention à `recvfrom` : interrompu !

```
while(stop == 0) {
  if(recvfrom(...) == -1) {
    if(errno != EINTR) {
      perror("Erreur lors de la réception d'un message ");
      exit(EXIT_FAILURE);
    }
  }
  else {
    ...
    sendto(...);
  }
}
```

Arrêt "propre" du serveur (2/2)

```
int stop = 0;

void handler(int signum) {
  stop = 1;
}
...
  struct sigaction action;
  sigemptyset(&action.sa_mask);
  action.sa_flags = 0;
  action.sa_handler = handler;
  if(sigaction(SIGINT, &action, NULL) == -1) {
    perror("Erreur lors du placement du gestionnaire ");
    exit(EXIT_FAILURE);
  }
...
```

Blocage du client (1/3)

- Avec UDP : pas de gestion de la perte de message
  ↪ Impossible de savoir si la requête ou la réponse ont été perdues
- Si la requête est perdue : client bloqué !
  ↪ `recvfrom` bloquant par défaut
- Idem si le serveur n'est pas connecté !
- Solution proposée ici :
  - Utilisation d'une alarme de 1s
  - Si pas de réponse, nouvelle tentative
    ↪ Attention ! Il faut renvoyer la requête
  - Au bout de trois essais, on arrête

Blocage du client (2/3)

```
int alarme = 0;

void handler(int signum) {
  alarme++;
}
...
  struct sigaction action;
  sigemptyset(&action.sa_mask);
  action.sa_flags = 0;
  action.sa_handler = handler;
  if(sigaction(SIGALRM, &action, NULL) == -1) {
    perror("Erreur lors du placement du gestionnaire ");
    exit(EXIT_FAILURE);
  }
...
```

Blocage du client (3/3)

```
while(stop == 0) {
  sendto(...);
  alarm(1);
  if(recvfrom(...) == -1) {
    if(errno == EINTR) {
      if(alarme == 3) {
        printf("Pas de reponse de la part du serveur...\n");
        exit(EXIT_FAILURE);
      }
    }
    else {
      perror("Erreur lors de la reception de la reponse ");
      exit(EXIT_FAILURE);
    }
  }
  else {
    alarm(0); stop = 1;
  }
}
```

Problématique des échanges multiples

- Rappel :
  - Le serveur se met en attente de la réception de messages
  - Tous les messages des clients sont reçus sur la même *socket*
- Que se passe-t-il si les deux intervenants doivent échanger plus qu'un couple requête/réponse ?
  - Plusieurs messages envoyés par les clients
  - Mélange des dialogues entre les clients

Première solution : session

- Le serveur garde une trace des clients :
  $\hookrightarrow$ Couples adresse source/port source uniques
- À la réception du premier message :
  - Création d'une session sur le serveur
  - Identification de la session + état
- Le multiplexage est géré par l'application

Deuxième solution : utilisation d'un second numéro de port

- À la réception du premier message :
  - Création d'une nouvelle *socket* (autre numéro de port)
  - Réponse au client via la nouvelle *socket*
  - Récupération de la nouvelle adresse par le client
  - Le client peut répondre sur la nouvelle adresse
- Avantage : possible de gérer chaque "connexion" par des processus différents
- Inconvénient : identification nécessaire des clients

Troisième solution : ne pas utiliser UDP

- UDP est utilisé pour sa légèreté :
  ↪ DNS, TFTP, voix sur IP, télévision...
- Pas pratique pour le transfert de données plus volumineuses :
  - Pas de gestion de la congestion
  - Pas de gestion de la perte des données
  - Pas pratique pour les échanges multiples
- Dans ce cas (et uniquement), utilisation de TCP

Mode connecté (TCP)

1. Création de la *socket* sur le serveur
2. Attribution d'un nom à la *socket*
3. Placer la *socket* en écoute
4. Attente de connexions
5. Le client crée lui-aussi une *socket*
6. Demande de connexion au serveur
7. Une connexion est établie

Serveur : mise en écoute

- **En-tête de la fonction (S2)** :
  - `int listen(int sockfd, int taille)`
  - *Inclusions* : `sys/types.h` et `sys/socket.h`
- **Explications** :
  - Place la *socket* comme étant passive (en attente de connexions)
  - Fixe la taille de la file d'attente des connexions :
    ↪ Tant qu'une connexion n'est pas acceptée, elle est mise en attente
    ↪ Si le nombre de connexion en attente est trop grand, la connexion est refusée (suivant le protocole)
- **Paramètres** :
  - `sockfd` : le descripteur de fichier correspondant à la *socket*
  - `taille` : taille de la file d'attente
- **Valeur retournée et erreurs générées** :
  - 0 en cas de réussite ou -1 en cas d'erreur
  - Quelques erreurs :
    - `EBADF` : descripteur invalide
    - `ENOTSOCK` : le descripteur ne correspond pas à une *socket*
    - `EOPNOTSUPP` : `listen` non supporté par ce type de *socket*

Client : établissement d'une connexion (1/2)

- **En-tête de la fonction (S2) :**
  - `int connect(int sockfd, struct sockaddr * adresseServeur, socklen_t taille)`
  - *Inclusions* : `sys/types.h` et `sys/socket.h`
- **Explications :**
  - Connecte la *socket* à l'adresse indiquée
  - Peut aussi être utilisé avec un protocole non connecté :
    ↪ Il s'agira de l'adresse de destination par défaut
    ↪ Possible d'appeler plusieurs fois cette fonction
- **Paramètres :**
  - `sockfd` : le descripteur de fichier correspondant à la *socket*
  - `adresseServeur` : l'adresse du serveur
  - `taille` : la taille de l'adresse du serveur

Client : établissement d'une connexion (2/2)

- **En-tête de la fonction (S2) :**
  - `int connect(int sockfd, struct sockaddr * adresseServeur, socklen_t taille)`
  - *Inclusions* : `sys/types.h` et `sys/socket.h`
- **Valeur retournée et erreurs générées :**
  - 0 en cas de réussite ou -1 en cas d'erreur
  - Quelques erreurs :
    - `EBADF` : mauvais descripteur
    - `EAGAIN` : pas de port local disponible
    - `EALREADY` : *socket* non bloquante et tentative précédente non terminée
    - `ECONNREFUSED` : connexion refusée par le serveur
    - `EINTR` : appel interrompu par un signal

Serveur : accepter une connexion (1/2)

- **En-tête de la fonction (S2) :**
  - `int accept(int sockfd, struct sockaddr *adresseClient, socklent_t *taille)`
  - *Inclusions* : `sys/types.h` et `sys/socket.h`
- **Explications :**
  - Met en attente de connexions le processus courant
  - Dès qu'une demande est reçue :
    ↪ Connexion établie avec le client
    ↪ Une nouvelle *socket* est créée, associée à la connexion
  - Appel bloquant sauf si la *socket* est créée comme non bloquante

Serveur : accepter une connexion (2/2)

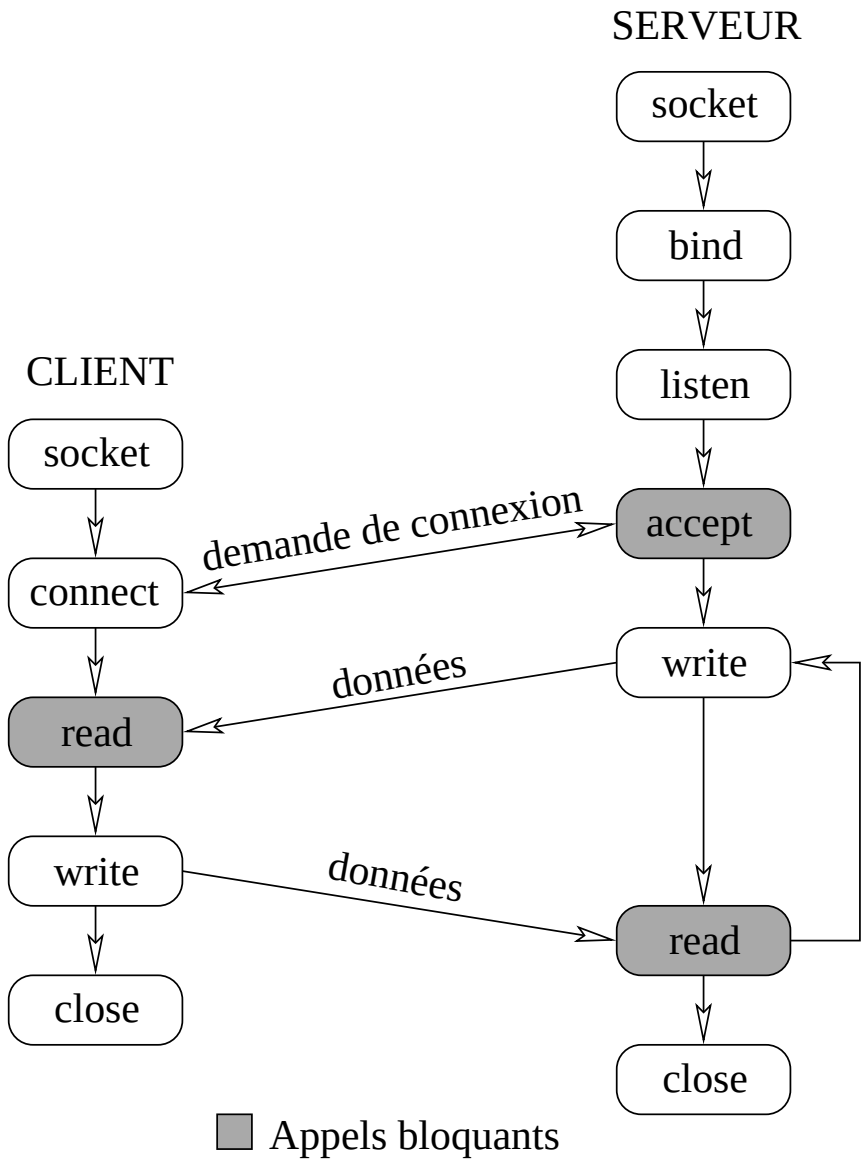
- **En-tête de la fonction (S2)** :
  - `int accept(int sockfd, struct sockaddr *adresseClient, socklent_t *taille)`
  - *Inclusions* : `sys/types.h` et `sys/socket.h`
- **Paramètres** :
  - `sockfd` : le descripteur de fichier correspondant à la *socket*
  - `adresseClient` : l'adresse du client
  - `taille` : la taille réelle de l'adresse du client
- **Valeur retournée et erreurs générées** :
  - 0 en cas de réussite ou -1 en cas d'erreur
  - Quelques erreurs :
    - `EAGAIN` : pas de connexion et *socket* non bloquante
    - `EBADF` : descripteur invalide
    - `EINTR` : appel interrompu par un signal
    - `EMFILE` : nombre maximal de descripteurs ouverts atteint

Envoi et réception de données

- **En-têtes des fonctions :**
  - `size_t read(int sockfd, void *tampon, size_t taille)`

    `size_t write(int sockfd, const void *tampon, size_t taille)`
- **Explications :**
  - Mêmes comportements que pour la manipulation des tubes :
    ↪ Lecture bloquante tant qu'aucune donnée n'est reçue
    ↪ Lecture destructrice
  - D'autres spécificités, propres aux *sockets*

Diagramme des appels systèmes en mode connecté



Exemple d'une application client-serveur

- Le serveur se met en attente de connexions
- Le client se connecte au serveur :
  - Envoi d'un entier (valeur quelconque)
  - Réception d'un entier (le double de la valeur envoyée)
  - Fermeture de la connexion
- Le serveur, dès qu'une connexion est établie :
  - Réception d'un entier
  - Envoi du double de la valeur reçue
  - Fermeture de la connexion

Exemple : le client

```
int socketfd;
struct sockaddr_in adresseServeur;

memset(&adresseServeur, 0, sizeof(adresseServeur));
adresseServeur.sin_family = AF_INET;
adresseServeur.sin_port = htons(12340);
inet_pton(AF_INET, "127.0.0.1", &adresseServeur.sin_addr);

socketfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

connect(socketfd, (struct sockaddr*)&adresseServeur,
                  sizeof(adresseServeur));

int n = 5;
write(socketfd, &n, sizeof(int));
read(socketfd, &n, sizeof(int));

close(socketfd);
```

Exemple : le serveur (1/2)

```
int ecoutefd, connexionfd;
struct sockaddr_in adresseServeur;

ecoutefd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

memset(&servaddr, 0, sizeof(adresseServeur));
adresseServeur.sin_family = AF_INET;
adresseServeur.sin_port = htons(12340);
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
bind(ecoutefd, (struct sockaddr*)&adresseServeur, sizeof(
    adresseServeur));

listen(ecoutefd, 1024);
...
```

Exemple : le serveur (2/2) (suite)

```
int n;
while(1) {
  connexionfd = accept(ecoutefd, (struct sockaddr*)NULL,
                                 NULL));

  read(connexionfd, &n, sizeof(int));

  n = n * 2;
  write(connexionfd, &n, sizeof(int));

  close(connexionfd);
}
```

Fermeture d'une connexion

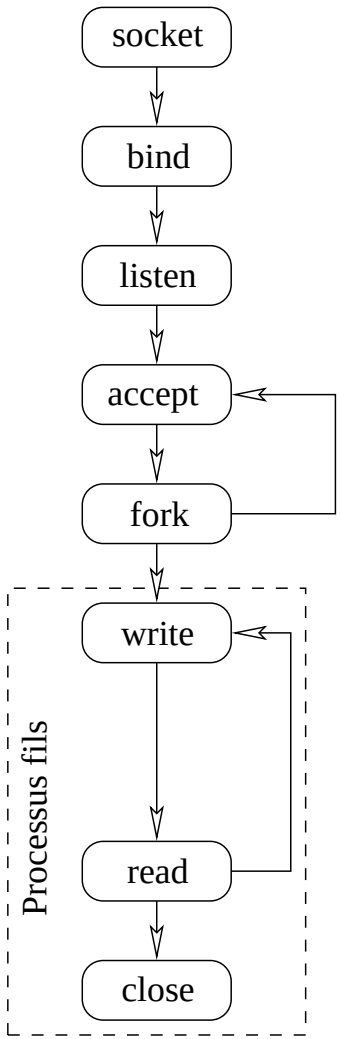
- Pour fermer une connexion, utilisation de `close` :
  ↪ Fermeture dans les deux sens
  ↪ Génération d'une erreur si elle n'est pas négociée
- Possible de fermer en lecture ou fermeture uniquement :
  ↪ Utilisation de `shutdown`

Fonction `shutdown`

- **En-tête de la fonction (S2) :**
  - `int shutdown(int sockfd, int comment)`
  - *Inclusions* : `sys/socket.h`
- **Paramètres :**
  - `sockfd` : le descripteur de fichier correspondant à la *socket*
  - `comment` : indication sur le mode de fermeture
    ↪ `SHUT_RD`, `SHUT_WR`, `SHUT_RDWR`
- **Valeur retournée et erreurs générées :**
  - 0 en cas de réussite ou -1 en cas d'erreur
  - Quelques erreurs :
    - `EBADF` : descripteur invalide
    - `ENOTCONN` : *socket* non connectée
    - `ENOTSOCK` : le descripteur ne correspond pas à une *socket*

Gestion de connexions simultanées de clients

- En TCP, il est possible de gérer plusieurs clients sur le serveur :
  ↪ Utilisation d'un seul numéro de port
- Le multiplexage est géré par le système (et par TCP)
- Par défaut, le traitement est séquentiel :
  ↪ Utilisation de processus/threads
- Procédure :
  1. Attente d'une demande de connexion
  2. Dès qu'une connexion est établie, création d'un processus fils



Exemple : le serveur multi-clients

```
while(1) {
  connexionfd = accept(ecoutefd, (struct sockaddr*)NULL,
                                 NULL));

  if(fork() == 0) {
    close(ecoutefd);

    int n;
    read(connexionfd, &n, sizeof(int));
    n = n * 2;
    write(connexionfd, &n, sizeof(int));

    close(connexionfd);
    exit(EXIT_SUCCESS);
  }

  close(connexionfd);
}
```

Problèmes

- Fils créés, mais pas de `wait` :
  ↪ Fils en état zombie
- Solution :
  - Ignorer les signaux `SIG_CHLD` : libération des ressources auto.
    ↪ Pas forcément portable
  - Création d'un gestionnaire sur `SIG_CHLD` avec un `waitpid` non bloquant
- Comment couper le serveur ?
  ↪ Arrêt brutal avec un signal
- Solution :
  - Utilisation d'un gestionnaire de signal :
    ↪ Variable globale `int stop = 0`
    ↪ À la réception d'un signal, `stop = 1`
    ↪ Boucle principale : `while(!stop)`

Remarque sur la lecture

- La fonction `read` est bloquante :
  ↪ Déblocage lorsque des données sont reçues
- On spécifie la taille du tampon de données

```
char donnees[255];
ssize_t lus;
if((lus = read(sockfd, donnees, 255)) == -1) {
  perror("Erreur lors de la lecture ");
  exit(EXIT_FAILURE);
}
```

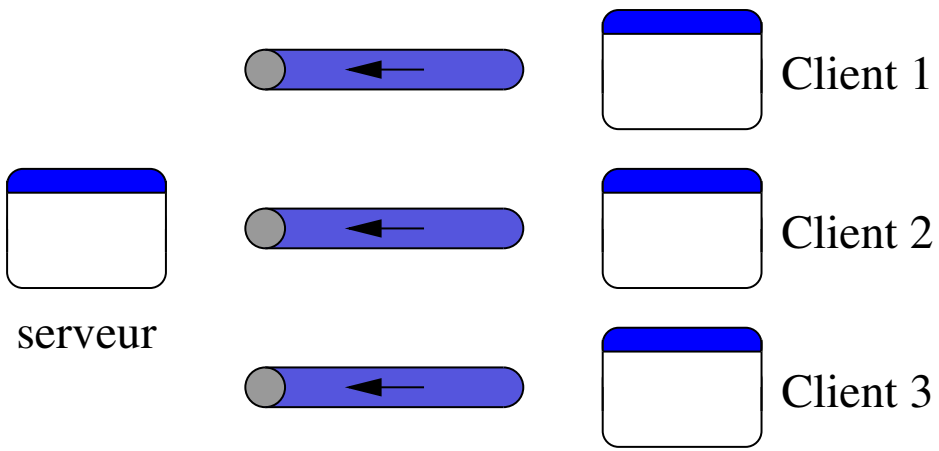
Problème

- Rien n'oblige que la lecture soit bloquante tant que la quantité de données attendue n'est pas reçue
- Lire moins de données que nécessaire n'est pas une erreur !

```
char donnees[255];
ssize_t lus, totallus = 0;
while(totallus != 255) {
  if((lus = read(sockfd, &(donnees[totallus]), 255 -
      totallus) == -1) {
    perror("Erreur lors de la lecture ");
    exit(EXIT_FAILURE);
  }
  totalus += lus;
}
```

Présentation de l'application

- Un serveur communique avec plusieurs clients :
  ↪ Données reçues via plusieurs descripteurs de fichier (tubes, sockets, *etc.*)
- Les fils envoient des données au père qui les traite au fur-et-à-mesure
- Problème : la lecture est bloquante



Solutions

1. Utilisation de signaux envoyés au serveur :
   ↪ Nécessité d'utiliser un gestionnaire + variable globale
   ↪ OK pour les tubes, mais que faire avec les *sockets* ?
2. Utilisation d'un seul descripteur :
   ↪ Attention au problème d'atomicité de l'écriture (tubes)
   ↪ Une *socket* créée par client en mode connecté
3. Utilisation de fils sur le serveur :
   ↪ OK si le traitement est cloisonné
   ↪ Mêmes problèmes sinon
4. Rendre la lecture non bloquante
   ↪ A suivre...

Lecture non bloquante (rappels)

- Utilisation de la fonction `fcntl` :
  ↪ Modification des options du descripteur de fichier
- Le `read` devient non bloquant :
  ↪ Lecture de 0 octet si aucune donnée à lire
- Algorithme principal :
  1. Lecture du tube 1, puis du tube 2, puis du tube 3
  2. On recommence l'étape 1
- Problème : consommation CPU inutile
- Utilisation de pauses (`sleep`) : OK, mais délai dans le traitement !

Traitements inutiles = consommation de CPU inutile = le diable

Surveillance de descripteurs de fichiers

- Pour éviter l'attente active, utilisation d'une surveillance de descripteurs
- Processus bloqué tant qu'aucun descripteur n'est prêt :
  ↪ En lecture (données reçues)
  ↪ En écriture (tampon vidé)
  ↪ En exception (erreurs, données urgentes, *etc.*)
- Deux solutions :
  - `select` : cas "classique"
  - `pselect` : préférence si utilisation des signaux

Ensembles de descripteurs de fichier

- Utilisation d'ensembles de descripteurs :
  ↪ `fd_set`
- Manipulation des ensembles :
  - `void FD_ZERO(fd_set *ensemble)` : vidage de l'ensemble
  - `void FD_SET(int fd, fd_set *ensemble)` : ajout d'un descripteur
  - `void FD_CLR(int fd, fd_set *ensemble)` : suppression d'un descripteur
  - `int FD_ISSET(int fd, fd_set *ensemble)` : vérifie si le descripteur est présent

Utilisation de `select`/`pselect`

1. Création d'un ensemble de descripteurs à surveiller
2. Appel à `select` / `pselect` :
   - Processus bloqué tant qu'aucun descripteur n'est prêt
   - Débloqué au bout d'un temps donné (fin du compte-à-rebours)
3. Après l'appel, les ensembles sont mis à jour :
   ↪ `select` / `pselect` retourne le nombre de descripteurs concernés
4. Lecture, écriture, traitement puis retour à l'étape 1

La fonction `select` (1/2)

- **En-tête de la fonction (S2)** :
  - `int select(int nfds, fd_set fd_read, fd_set fd_write, fd_set fd_except, struct timeval *timeout)`
  - *Inclusions* : `sys/select.h`, `sys/types.h`, `unistd.h`
- **Explications** :
  - Blocage du processus tant qu'aucun descripteur n'est prêt en lecture, en écriture ou en exception
- **Les attributs** :
  - `nfds` : le numéro du plus grand descripteur + 1 (!!!)
  - `fd_read` : descripteurs à surveiller en lecture
  - `fd_write` : descripteurs à surveiller en écriture
  - `fd_except` : descripteurs à surveiller pour des exceptions
  - `timeout` : délai maximum

```
struct timeval {
  long tv_sec;  /* secondes      */
  long tv_usec; /* microsecondes */
};
```

La fonction `select` (2/2)

- **Retour et erreurs générées** :
  - Nombre total de descripteurs dans les trois ensembles
  - 0 si aucun descripteur n'est présent (si le délai est écoulé)
  - -1 en cas d'erreur :
    - `EBADF` : descripteur invalide dans l'un des ensembles
    - `EINTR` : signal intercepté
    - `EINVAL` : `nfds` négatif ou `timeout` invalide
    - `ENOMEM` : pas assez de mémoire pour le noyau

La fonction `pselect` (1/3)

- **En-tête de la fonction (S2)** :
  - `int pselect(int nfds, fd_set fd_read, fd_set fd_write, fd_set fd_except, const struct timespec *timeout, const sigset_t *sigmask)`
  - *Inclusions* : `sys/select.h`, `sys/types.h`, `unistd.h`
  - *Macros* : `POSIX_C_SOURCE >= 200112L` (pour `pselect`)
- **Explications** :
  - Blocage du processus tant qu'aucun descripteur n'est prêt.
  - Blocage de signaux lors de l'appel.

La fonction `pselect` (2/3)

- **Les attributs** :
  - `nfds` : le numéro du plus grand descripteur + 1 (!!!)
  - `fd_read` : descripteurs à surveiller en lecture
  - `fd_write` : descripteurs à surveiller en écriture
  - `fd_except` : descripteurs à surveiller pour des exceptions
  - `timeout` : délai maximum
  - `sigmask` : le masque de signaux à bloquer
- **La structure `timeval` (`sys/time.h`)**

```
struct timespec {
  long tv_sec;  /* secondes     */
  long tv_nsec; /* nanosecondes */
};
```

La fonction `pselect` (3/3)

- **Retour et erreurs générées** :
  - Nombre total de descripteurs dans les trois ensembles
  - 0 si aucun descripteur n'est présent (si le délai est écoulé)
  - -1 en cas d'erreur :
    - `EBADF` : descripteur invalide dans l'un des ensembles
    - `EINTR` : signal intercepté
    - `EINVAL` : `nfds` négatif ou `timeout` invalide
    - `ENOMEM` : pas assez de mémoire pour le noyau

Exemple de code (1/2)

```
/* Création des fils + tubes (sans gestion d'erreur) */
int nbFils, i, j;
int *tubes;
pid_t *pids;

nbFils = atoi(argv[1]);
tubes = (int*)malloc(sizeof(int) * 2 * nbFils);
pids = (int*)malloc(sizeof(pid_t) * nbFils)

for(i = 0; i < nbFils; i++) {
  pipe(&tubes[i * 2]);
  if((pids[i] = fork()) == 0) {
    for(j = 0; j <= i; j++)
          close(tubes[j * 2]);
    fils(i, tubes[i * 2 + 1]);
  }
  else
    close(tubes[i * 2 + 1]);
}
```

Exemple de code (2/2)

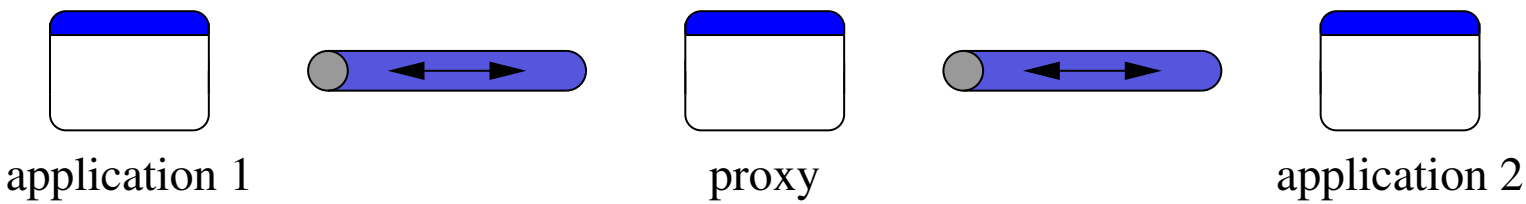
```
/* Surveillance des descripteurs (sans gestion d'erreur) */
fd_set ensemble;
int maxFd, valeur;
while(stop == 0) {
    FD_ZERO(&ensemble);
    maxFd = 0;
    for(i = 0; i < nbFils; i++) {
      FD_SET(tubes[i * 2], &ensemble);
      if(tubes[i * 2] > maxFd)
        maxFd = tubes[i * 2];
    }
    nb = select(maxFd + 1, &ensemble, NULL, NULL, NULL);
    printf("Données dans %d tube(s).\n", nb);
    for(i = 0; i < nbFils; i++) {
      if(FD_ISSET(tubes[i * 2], &ensemble)) {
        read(tubes[i * 2], &valeur, sizeof(int));
        printf("Lecture de %d depuis le tube %d\n", valeur, i)
            ;
      }
    }
}
```

Premier exemple : tubes nommés

- 4 clients qui écrivent chacun dans un tube nommé :
  ↪ Ouverture du tube
  ↪ Attente de données de la part de l'utilisateur
- Le serveur affiche les données reçues :
  ↪ Utilisation de `ncurses` avec 4 fenêtres
  ↪ Pas de fils sur le serveur (problème de `ncurses`)
- Le serveur :
  ↪ Création des 4 tubes
  ↪ Création de l'interface `ncurses`
  ↪ Traitement des données : `select` + affichage

Deuxième exemple : *sockets*

- Communication entre deux applications via un proxy
- Connexions sur le proxy :
  ↪ Données envoyées par une des applications, transférées à l'autre
- Possibilité d'avoir un fils sur le proxy pour chaque lien :
  ↪ Mais un seul fils par lien



Les *sockets* locales

- Utilisées pour communication entre processus locaux
- Comme des tubes mais :
  ↪ Communication bidirectionnelles
  ↪ Utilisation des fonctions liées aux *sockets* Internet
  ↪ Possible de les utiliser en mode paquet (`SOCK_DGRAM`) ou en mode flux (`SOCK_STREAM`)
- Deux types de *sockets* locales :
  ↪ *Sockets* locales anonymes
  ↪ *Sockets* locales nommées

Les *sockets* locales anonymes

- Fonctionnement similaire aux tubes anonymes
- Création d'une paire de descripteurs :
  ↪ Transmission par héritage
- Fonction `socketpair` :
  ↪ Crée deux descripteurs (comme `pipe`)
  ↪ Mais utilisation en mode bidirectionnel
  ↪ Une *socket* en lecture/écriture pour le père
  ↪ Une *socket* en lecture/écriture pour le fils
- Domaine : `AF_LOCAL` (ou `AF_UNIX`)

Fonction `socketpair`

- **En-tête de la fonction (S2)** :
  - `int socketpair(int domaine, int type, int protocol, int fd[2])`
  - *Inclusions* : `sys/socket.h`, voire `sys/types.h`
- **Paramètres** :
  - `domaine` : pour une *socket* locale, `AF_INET`
  - `type` : `SOCK_STREAM`, `SOCK_DGRAM`
  - `protocol` : 0
  - `fd` : les deux descripteurs de fichier créés
- **Valeur retournée et erreurs générées** :
  - 0 en cas de réussite ou -1 en cas d'erreur
  - Quelques erreurs :
    - `EAFNOSUPPORT`, `EOPNOTSUPP` : famille ou protocole incompatible
    - `ENFILE` : nombre maximum de descripteurs de fichier atteint

Exemple d'une *socket* locale en mode connecté (1/2)

```
int fd[2], i, valeur;

socketpair(AF_LOCAL, SOCK_STREAM, 0, fd);
if(fork() == 0) {
  close(fd[0]);
  fils(fd[1]);
}
close(fd[1]);
for(i = 0; i < 3; i++) {
  sleep(aleatoire(1, 3));
  read(fd[0], &valeur, sizeof(int));
  printf("Père␣:␣réception␣de␣%d\n", valeur);
  sleep(aleatoire(1, 3));
  write(fd[0], &i, sizeof(int));
  printf("Père␣:␣envoi␣de␣%d\n", i);
}
wait(NULL);
close(fd[0]);
printf("Père␣terminé.\n");
```

Exemple d'une *socket* locale en mode connecté (2/2)

```
void fils(int sockfd) {
  int i, valeur;

  srand(time(NULL) + getpid());

  for(i = 0; i < 3; i++) {
    sleep(aleatoire(1, 3));
    write(sockfd, &i, sizeof(int));
    printf("Fils : envoi de %d\n", i);
    sleep(aleatoire(1, 3));
    read(sockfd, &valeur, sizeof(int));
    printf("Fils : réception de %d\n", valeur);
  }
  close(sockfd);

  printf("Fils terminé.\n");

  exit(EXIT_SUCCESS);
}
```

Les *sockets* locales nommées

- Fonctionnement comme les *sockets* Internet
  ↪ Famille `AF_LOCAL`
  ↪ En mode connecté ou non connecté
- Nommage avec `bind`
- Adresse spécifiée :
  ↪ Correspond à une adresse locale
  ↪ Création d'une entrée dans le système de fichiers
  ↪ Suppression avec `unlink` (comme pour les tubes nommés)

Les adresses locales

```
#define UNIX_PATH_MAX    108

struct sockaddr_un {
  sa_family_t sun_family;               /* AF_UNIX */
  char        sun_path[UNIX_PATH_MAX]; /* chemin accès */
};
```

- Comme pour les tubes nommés, le nom correspond à un chemin
- Attention à l'utilisation de `getsockname` :
  ↪ Taille = `sizeof(sa_family_t) + strlen(sun_path) + 1`

Exemple d'une *socket* locale en mode non connecté (1/2)

```
/* Serveur */
int sockfd;
struct sockaddr_un adresse;

sockfd = socket(AF_LOCAL, SOCK_DGRAM, 0);

memset(&adresse, 0, sizeof(struct sockaddr_un));
adresse.sun_family = AF_LOCAL;
snprintf(adresse.sun_path,
         sizeof(struct sockaddr_un) - sizeof(sa_family_t),
         "%s",
         argv[1]);
bind(sockfd, (struct sockaddr*)&adresse, sizeof(struct
    sockaddr_un));

recvfrom(sockfd, ..., ..., 0, NULL, NULL);
...
close(sockfd)
unlink(adresse.sun_path);
```

Exemple d'une *socket* locale en mode non connecté (2/2)

```
/* Client */
int sockfd;
struct sockaddr_un adresse;

sockfd = socket(AF_LOCAL, SOCK_DGRAM, 0);

memset(&adresse, 0, sizeof(struct sockaddr_un));
adresse.sun_family = AF_LOCAL;
snprintf(adresse.sun_path,
         sizeof(struct sockaddr_un) - sizeof(sa_family_t),
         "%s",
         argv[1]);

sendto(sockfd, ..., ..., 0, (struct sockaddr*)&adresse,
    sizeof(struct sockaddr_un));

close(sockfd);
```

Résumé sur les *sockets* et le nommage

- `socket` : création d'une socket aller
- Structures pour représenter des adresses IPv4 et IPv6 aller
- Fonctions pour remplir ces structures :
  - conversions diverses d'entiers aller
  - `inet_pton` : conversion d'une chaîne représentant une adresse en adresse au format réseau aller
  - `inet_ntop` : l'inverse aller
- `bind` : nommage d'une socket aller
- DNS :
  - `getaddrinfo` : récupérer l'adresse correspondant à un URL aller
  - `getnameinfo` : récupérer le nom correspondant à une adresse aller
- `getsockname` : récupérer l'adresse associée à une socket aller
- `getpeername` : récupérer l'adresse de l'hôte distant connecté à une socket aller

Résumé sur les *sockets* en mode non connecté

- `sendto` : envoi d'un message aller
- `recvfrom` : réception d'un message aller

Résumé sur les *sockets* en mode connecté

- `listen` : place la socket en mode écoute aller
- `connect` : connexion à une socket distante aller
- `accept` : acceptation d'une connexion aller
- Lecture/écriture dans une socket aller
- `shutdown` : fermeture d'une connexion aller

Résumé sur les sockets locales

Permettent de communiquer entre processus de la même machine
Similaires aux tubes mais avec le comportement de *sockets*

- `socketpair` : crée une *socket* anonyme ▸ aller
- Autres fonctions : celles utilisées pour les *sockets* Internet

Résumé sur la surveillance de descripteurs

Possible de bloquer un processus jusqu'à la réception d'un évènement sur un ensemble de descripteurs de fichier

- Manipulation des ensembles de descripteurs aller
- `select` : attente d'une activité sur un ensemble de descripteurs aller
- `pselect` : attente d'une activité sur un ensemble de descripteurs avec gestion des signaux aller