

# Réseaux Sockets

E. Jeandel

Emmanuel.Jeandel at lif.univ-mrs.fr

- 1 Généralités
- 2 Sockets BSD
- 3 Usage élaboré

- Une des extrémités d'une communication sur internet
- Une socket est donnée par :
  - L'adresse de transport locale (IP + port)
  - L'adresse de transport du destinataire (eventuellement)
  - Le protocole de transport (TCP, UDP, ...)

*Une socket connectée est déterminée par quatre informations :  
Adresse réseau/port source et Adresse réseau/port destination*

Plusieurs types. Les plus fréquents :

- Socket stream (connectée, TCP sur Internet)
- Socket datagramme (non connectée, non fiable, UDP sur internet)
- Socket *Raw* : niveau couche réseau

Sockets Raw accessibles en général uniquement par *root*.

- Conséquence :

```
ejeandel@monster:home/> ls -l /bin/ping
-rwsr-xr-x 1 root root 33576 août 28 2009 /bin/ping
```

Plusieurs API :

- BSD sockets (Standard sur MAC OS X, Linux)
- Windows Sockets
- TLI (Transport Layer Interface)/XTI (Standard sur MAC OS, Solaris... )

Normalisé dans POSIX.

- 1 Généralités
- 2 Sockets BSD
- 3 Usage élaboré

## 1 Généralités

## 2 Sockets BSD

- Création
- Destruction
- Adresses
- I/O
- Client/Serveur
- UDP/TCP

## 3 Usage élaboré

- Divers
- Appels non bloquants

Pour créer une socket, il faut savoir :

- La famille de protocoles
- Le type (voir plus haut)
- Le protocole

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

- Le résultat est un descripteur de fichiers
- On peut utiliser **write**, **read**, **dup2** sur le résultat

# Famille de protocoles

```
#define AF_UNSPEC          0
#define AF_UNIX           1    /* Unix domain sockets */
#define AF_LOCAL          1    /* POSIX name for AF_UNIX */
#define AF_INET           2    /* Internet IP Protocol */
#define AF_AX25           3    /* Amateur Radio AX.25 */
#define AF_IPX            4    /* Novell IPX */
#define AF_APPLETALK      5    /* AppleTalk DDP */
#define AF_NETROM         6    /* Amateur Radio NET/ROM */
#define AF_BRIDGE         7    /* Multiprotocol bridge */
#define AF_ATMPVC         8    /* ATM PVCs */
#define AF_X25            9    /* Reserved for X.25 project */
#define AF_INET6          10   /* IP version 6 */
#define AF_ROSE           11   /* Amateur Radio X.25 PLP */
#define AF_DECnet         12   /* Reserved for DECnet project */
#define AF_NETBEUI        13   /* Reserved for 802.2LLC project */
#define AF_SECURITY       14   /* Security callback pseudo AF */
#define AF_KEY            15   /* PF_KEY key management API */
...

```

(Source : Noyau Linux)

# Types et Protocoles

```
#define SOCK_STREAM      1      /* stream socket */
#define SOCK_DGRAM      2      /* datagram socket */
#define SOCK_RAW        3      /* raw-protocol interface */

#define IPPROTO_ICMP    1      /* control message protocol */
#define IPPROTO_IPv4    4      /* IPv4 encapsulation */
#define IPPROTO_TCP     6      /* tcp */
...
#define IPPROTO_UDP     17     /* user datagram protocol */
#define IPPROTO_RAW     255    /* raw IP packet */
```

(Source : Noyau XNU)

- Souvent, famille + type suffit à identifier le protocole
- AF\_INET + SOCK\_STREAM → IPPROTO\_TCP
- AF\_INET + SOCK\_DGRAM → IPPROTO\_UDP
- Mettre : protocol = 0

# Fonctions de conversion

- Le codage des entiers sur le réseau doit être défini de façon cohérente
- *network byte order*
- Grand boutiste

## Fonctions pour convertir

```
uint32_t  htonl (uint32_t  hostlong);  
uint16_t  htons (uint16_t  hostshort);  
uint32_t  ntohl (uint32_t  netlong);  
uint16_t  ntohs (uint16_t  netshort);
```

## 1 Généralités

## 2 Sockets BSD

- Création
- Destruction
- Adresses
- I/O
- Client/Serveur
- UDP/TCP

## 3 Usage élaboré

- Divers
- Appels non bloquants

```
int shutdown(int sockfd, int how);
```

- `how = SHUT_{RD,WR,RDWR}`, désactive réception, envoie, ou les deux.
- `SHUT_WR` envoie un paquet `FIN`
- Ensuite on peut faire `close`

On peut aussi faire `close` directement.

- Différence sur le traitement des messages dans les tampons

## 1 Généralités

## 2 Sockets BSD

- Création
- Destruction
- Adresses
- I/O
- Client/Serveur
- UDP/TCP

## 3 Usage élaboré

- Divers
- Appels non bloquants

- Une adresse est stockée dans une structure `sockaddr`
- Dépend de l'implémentation
- Contient un champ `sa_family`
- Pour chaque `sa_family` différent, une sous-structure :
  - **struct** `sockaddr_in` (`AF_INET`)
  - **struct** `sockaddr_in6` (`AF_INET6`)
  - **struct** `sockaddr_un` (`AF_UNIX`)
- Structure générique **struct** `sockaddr_storage` de taille plus grande que toutes les autres.
- La façon d'aller chercher l'adresse et le port diffèrent suivant la famille. . .
- On peut tout de même les rentrer à la main. . .

# Entrer les adresses

- L'utilisateur préfère avoir les adresses de façon "claire" (chaîne de caractères) plutôt que binaire
- Fonctions de conversion
- IPV4 : `inet_ntoa`, `inet_aton` (déprécié)

```
char *inet_ntop(int af, void *src, char *dst, socklen_t size);  
int inet_pton(int af, const char *src, void *dst);
```

- `af` : Address family (`AF_INET`, `AF_INET6`)
- Pénible à utiliser en pratique

# Solution propre

- `getaddrinfo`
- S'occupe également de la résolution des adresses/ports
- Remplace aussi `getservbyname`, `gethostbyname`

```
int getaddrinfo(char *node, char *service,  
                struct addrinfo *hints, struct addrinfo **res);
```

- `hints` : Paramètres voulus pour la socket  
`ai_{family, socktype, protocol}` ainsi que `ai_flags`
- Résultat : liste des possibilités :  
`ai_{family, socktype, protocol, sockaddr}`
- `ai_next` pour passer à la suivante.
- Libéré avec `freeaddrinfo`.

- Retrouver adresse et port à partir de `sockaddr`
- `getnameinfo`
- Remplace `gethostbyaddr`, `getservbyport`

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

## 1 Généralités

## 2 Sockets BSD

- Création
- Destruction
- Adresses
- I/O
- Client/Serveur
- UDP/TCP

## 3 Usage élaboré

- Divers
- Appels non bloquants

```
ssize_t send(int sockfd, void *buf, size_t len, int flags);  
ssize_t sendto(int sockfd, void *buf, size_t len, int flags,  
               struct sockaddr *dest_addr, socklen_t addrlen);
```

- `sendto` utilisé pour UDP.
- Si le destinataire est connu (eg TCP), on peut aussi utiliser `write`

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);  
  
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,  
                  struct sockaddr *src_addr, socklen_t *addrlen);
```

- On peut également utiliser **recv** (et même **read**) sur une socket UDP.

## 1 Généralités

## 2 Sockets BSD

- Création
- Destruction
- Adresses
- I/O
- Client/Serveur
- UDP/TCP

## 3 Usage élaboré

- Divers
- Appels non bloquants

Client :

```
int connect(int sockfd, struct sockaddr *addr, socklen_t addrlen);
```

Serveur :

```
int bind(int sockfd, struct sockaddr *addr, socklen_t addrlen);  
int listen(int sockfd, int backlog);  
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

- **bind** : Donne une adresse locale à la socket
- **listen** : La socket est passive et attend les connexions.
- **accept** : Attend une connexion, puis crée une nouvelle socket permettant de traiter la connexion

Note : Pour le serveur, il faut ajouter le flag `AI_PASSIVE` lors de **getaddrinfo**

- On peut également faire **bind** chez le client (sémantique ?)

```
int bind(int sockfd, struct sockaddr *addr, socklen_t addrlen);
```

- **bind** : Donne une adresse locale à la socket (permet ensuite d'utiliser **recvfrom**)
- On peut utiliser **connect** (fd, addr, addrlen)
  - Seuls les paquets émis depuis `addr` seront reçus.
  - Les paquets sont envoyés par défaut (avec **send**) à `addr`
- Pas vraiment de client ni de serveur..

## 1 Généralités

## 2 Sockets BSD

- Création
- Destruction
- Adresses
- I/O
- Client/Serveur
- **UDP/TCP**

## 3 Usage élaboré

- Divers
- Appels non bloquants

- Tout ce qu'on a déjà dit les semaines précédentes..

# UDP vs TCP

- UDP : A chaque appel de **send**, un paquet va être envoyé
- UDP : **recv** reçoit un paquet en entier
- Si A appelle **send** deux fois sur un buffers d'un bit, et B fait un seul **recv** :
  - TCP : B recevra les deux bits.
  - UDP : B recevra le premier bit, doit faire un second **recv** pour le deuxième
- UDP : Si le buffer de **recv** est trop petit, le reste du paquet est perdu.

- UDP n'a pas de tampon pour les paquets envoyés.
- Pour un envoi de plusieurs paquets rapidement sur un réseau local, les premiers vont être supprimés
  - Temps que le cache ARP se mette à jour

- 1 Généralités
- 2 Sockets BSD
- 3 Usage élaboré

## 1 Généralités

## 2 Sockets BSD

- Création
- Destruction
- Adresses
- I/O
- Client/Serveur
- UDP/TCP

## 3 Usage élaboré

- Divers
- Appels non bloquants

# Options des sockets

```
int getsockopt(int sockfd, int level, int optname,  
               void *optval, socklen_t *optlen);  
int setsockopt(int sockfd, int level, int optname,  
               const void *optval, socklen_t optlen);
```

- level = SOL\_SOCKET, IPPROTO\_IP, IPPROTO\_TCP, etc
- Options utiles :
  - SO\_REUSEADDR. Permet de faire **bind** sur une adresse déjà utilisée.
  - IP\_TTL. Change le TTL des prochains paquets envoyés.
  - TCP\_NODELAY. Désactive l'algorithme de Nagle.
- D'autres options spécifiques à l'OS (TCP\_MAXSEG, TCP\_QUICKACK..)

# OOB (Out of Band)

- Permet d'envoyer des données sur un deuxième “canal” en même temps que le premier
- TCP : canal limité à 1 **bit** à la fois.
- Option `MSG_OOB` de **send/recv**

## 1 Généralités

## 2 Sockets BSD

- Création
- Destruction
- Adresses
- I/O
- Client/Serveur
- UDP/TCP

## 3 Usage élaboré

- Divers
- Appels non bloquants

- Faire des appels non bloquants, par exemple pour traiter plusieurs connexion simultanément

- Rendre la socket non bloquante

POSIX :

```
flags = fcntl(socket, F_GETFL, 0);  
fcntl(socket, F_SETFL, flags | O_NONBLOCK);
```

Autres :

```
flags = 1;  
ioctl(socket, FIONBIO, &flags);
```

# Solution 2

select

```
int select(int nfd, fd_set *readfds, fd_set *writefds,  
           fd_set *exceptfds, struct timeval *timeout);
```

Un évènement de lecture est enclenché :

- S'il y a quelque chose à lire
- Sur une socket passive, si un nouveau client s'est connecté
- Si la socket est fermée

Un évènement d'écriture est enclenché :

- Si on peut écrire (le tampon est vide)
- Si la socket est fermée
- Si un **connect** (non bloquant) est fini

Un évènement exceptionnel se produit

- Lorsqu'on reçoit un paquet OOB.

- 1 Généralités
- 2 Sockets BSD
- 3 Usage élaboré

Essentiellement la même chose que les sockets BSD

- Différence principale : Le numéro de la socket n'est pas un descripteur de fichier
- On ne peut donc pas utiliser `read`, `write`, `close`.

Différences dans l'API :

- `close` VS `closesocket`
- `ioctl` VS `ioctlsocket`
- Appel initial/final de toute application à `WSAStartup`/`WSACleanup`

Grosses différences dans la gestion des erreurs.