

LA PROGRAMMATION RÉSEAU SOUS LINUX

NAÏM QACHRI, STÉPHANE FERNANDES
MEDEIROS, EYTHAN LEVY, CÉDRIC MEUTER

Université Libre de Bruxelles
2010/2011

Introduction aux *sockets*

- Les *sockets* sont des dispositifs UNIX permettant à des processus situés sur des machines différentes de communiquer entre eux.
- Les *sockets* fournissent une interface de programmation entre les processus et le réseau, et permettent d'envoyer des informations sur le réseau comme s'il s'agissait d'un fichier (fd, read, write, ...).
- Leur fonctionnement est comparable aux *pipes* (ordre FIFO, lecture destructive, ...).
- Les *sockets* sont universellement utilisés (FTP, Netscape, ...).

Structures de données utilisées (I)

- C'est la structure de base employée pour stocker les informations sur les adresses de *sockets* :

```
struct sockaddr {  
    unsigned short sa_family; // address family, AF_XXX  
    char sa_data[14]; // 14 bytes of protocol address  
};
```

- Ce struct étant difficile à utiliser, on utilise, en général, un autre struct, qui est convertible en struct sockaddr :

```
struct sockaddr_in {  
    short int sin_family; // Address family  
    unsigned short int sin_port; // Port number  
    struct in_addr sin_addr; // Internet address  
    unsigned char sin_zero[8]; // Same size as struct  
sockaddr  
};
```

Structures de données utilisées (II)

- Le struct `in_addr` contient l'adresse IP liée au *socket* :

```
struct in_addr {  
    unsigned long s_addr; // that's a 32-bit long, or 4 bytes  
};
```

Problèmes d'ordre de bytes

- Les bytes de nombres entiers sont représentés sur le réseau en ordre *gros-boutiste* (network byte order). Tout entier envoyé ou reçu sur un *socket* doit donc être converti.
- Des fonctions vous permettent d'effectuer ces conversions, pour des entiers courts et longs :
 - `htons()` – "Host to Network Short"
 - `htonl()` – "Host to Network Long"
 - `ntohs()` – "Network to Host Short"
 - `ntohl()` – "Network to Host Long"
- Note : dans le struct `sockaddr_in`, les champs `sin_addr` et `sin_port` doivent aussi être convertis.

Les adresses IP

- Les machines sur internet sont identifiées par des adresses IP codées sur 4 bytes, et sont souvent représentées sous forme pointée : “10.12.110.57”.
- Des fonctions intéressantes existent pour vous faciliter l’usage des adresses IP.
- La fonction `inet_aton` permet d’assigner une valeur au champ `sin_addr` d’un struct `sockaddr_in` à partir d’un *string* en notation pointée :

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

```
int inet_aton(const char *cp, struct in_addr *inp);
```

Les adresses IP (II)

- Exemple de création et d'initialisation d'un struct `sockaddr_in` :

```
struct sockaddr_in my_addr;
my_addr.sin_family = AF_INET; // host byte order
my_addr.sin_port = htons(MYPORT); // short, network byte order
net_aton("10.12.110.57", &(my_addr.sin_addr));
memset(&(my_addr.sin_zero), '\0', 8); // zero the rest
```

- Voici une variante : la fonction `inet_addr(const char* cp)`, utilisée de la façon suivante :

```
struct sockaddr_in ina;
ina.sin_addr.s_addr = inet_addr("10.12.110.57");
```



- La fonction `inet_ntoa(struct in_addr in)` permet de faire la conversion inverse.

Les system call's de base: *socket()*

- La fonction `socket ()` permet d'initialiser un *socket* et d'obtenir son *socket descriptor* :

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

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

- Le premier paramètre doit être mis à `PF_INET` pour permettre les communications sur le réseau. 
- Le deuxième paramètre indique le type de *socket*. Dans le cadre des TP's, nous utiliserons le type `SOCK_STREAM`. 
- Le troisième paramètre peut être mis à 0.

Les system call's de base: *bind()*


- La fonction `bind()` permet d'associer un *socket* avec un numéro de port sur la machine locale. Le numéro de port est utilisé par le kernel pour associer les paquets entrants dans la machine vers un *socket descriptor* particulier :

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

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

- Le premier paramètre est le *socket descriptor* renvoyé par `socket()`, le second paramètre contient le port et l'adresse IP de la machine locale et le troisième paramètre, on peut le mettre à `sizeof(struct sockaddr)`.

Les system call's de base: *bind()*

- Le numéro de port doit être un numéro non déjà utilisé par une autre application, et être entre 1024 et 65535. Un port mis à 0 signifie que l'OS peut choisir un port libre au hasard.
- La constante `INADDR_ANY` pour l'adresse IP symbolise l'adresse de la machine locale. 
- Deux appels successifs à `bind()` peuvent entraîner des délais d'attente (1 minute) et un message d'erreur.
- On peut s'en débarrasser grâce à la fonction `setsockopt()` :

```
int yes=1;
```

```
setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int));
```




Les system call's de base: *connect()*

- Le system call `connect()` permet de se connecter à un ordinateur distant :

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

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

- Les paramètres sont les mêmes que pour le `bind()`, sauf que le struct `sockaddr` passé en paramètre contient l'adresse IP et le port de la machine ***distante***.
- Un appel à `connect()` doit toujours être précédé d'un appel à `socket()`, mais pas forcément d'un appel à `bind()`. En cas d'absence de `bind()`, un numéro de port sera assigné automatiquement à notre socket.

Les system call's de base: *listen()*

- Ce *system call* est utile pour des processus attendant des connexions de l'extérieur. Il permet d'écouter sur le *socket* si des ordinateurs distants essayent de se connecter sur notre numéro de port. L'acceptation de connexions se fait au moyen de deux *system calls* consécutifs : `listen()` et `accept()`.

```
int listen(int sockfd, int backlog);
```

- Le premier paramètre est le *socket descriptor* renvoyé par `socket()`, le second est le **nombre maximum de demandes de connexions** dans la file d'attente. Les demandes de connexion sont mises en file d'attente jusqu'au moment où la machine fait un `accept()`. Une valeur typique pour `backlog` est 20.

Les system call's de base: *listen()*

- Il convient de faire un `bind()` avant un `listen()`, sinon le *kernel* nous associera un port aléatoire, ce qui posera problème pour les clients éventuels (deviner un port est assez...difficile).

Les system call's de base: *accept()*

- Des ordinateurs distants vont, à présent, essayer de se connecter sur notre machine à un port sur lequel nous écoutons (grâce à `listen()`). Leurs demandes de connexions seront mises en file d'attente jusqu'à ce que nous les acceptions (une à la fois), avec l'appel système suivant :

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

```
int accept(int s, struct sockaddr *addr, int *addrlen);
```


Les system call's de base: *accept()*

- Le premier paramètre est le *socket descriptor* sur lequel on a fait le `listen()`. Le second est un *pointeur vers un struct sockaddr_in* local au programme, qui contiendra des informations sur la machine qui tente de se connecter. Le troisième paramètre est l'adresse d'une variable locale dont la valeur doit être initialisée à `sizeof(struct sockaddr_in)`.
- `accept()` renvoie un nouveau *socket descriptor*, qui devra être utilisé pour toutes les communications avec la machine qui se connecte.

Les system call's de base: *send()*

- La fonction `send()` permet d'envoyer des informations à un ordinateur distant au travers du *socket descriptor* associé à la connexion établie avec cet ordinateur :

```
int send(int sockfd, const void *msg, int len, int flags);
```

- Le deuxième paramètre contient l'information à envoyer, le troisième contient sa longueur, le quatrième peut être mis à 0 (voir la *man page* pour plus de détails sur les flags disponibles).
- `send()` renvoie le nombre de bytes envoyés. En fait `send()` envoie le maximum de données possible, mais cela peut encore être inférieur à `len`. Il faudra alors envoyer à nouveau le reste.

Les system call's de base: *send()*

- L'utilisation de `send` avec `flags` mis à 0 est équivalent à l'utilisation de `write()` avec notre *socket descriptor* utilisé comme `fd`.

Les system call's de base: *recv()*

- La fonction `recv()` permet de recevoir des informations envoyées par un ordinateur distant au travers du *socket descriptor* associé à la connexion établie avec cet ordinateur :

```
int recv(int sockfd, void *buf, int len, unsigned int flags);
```

- Le deuxième paramètre est un vecteur qui contiendra l'information lue, le troisième est la longueur du vecteur. Le quatrième peut être mis à 0 ou à l'un des l'une des valeurs permettant de contrôler de manière plus fine le comportement du `recv()` (voir `man recv`).
- Le comportement par défaut de `recv()` est d'attendre jusqu'à ce qu'une information soit disponible (lecture bloquante).

Les system call's de base: *recv()*

- `recv()` renvoie le nombre de bytes reçus ou 0 si la machine distante a déjà fermé son socket (à l'aide de `close()`).
- Le système call `read()` peut être utilisé de manière équivalente (sauf pour les flags).

getpeername() et *gethostname()*

- La fonction `getpeername()` permet de savoir qui est à l'autre bout d'une connexion par socket :

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

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

- Le second paramètre contiendra les informations concernant la machine distante.
- Une fois l'adresse obtenue, les fonctions `inet_ntoa()` et `gethostbyaddr()` permettent de l'afficher ou d'obtenir des informations supplémentaires.
- La fonction `gethostname()` permet de connaître le nom de la machine locale :

```
#include <unistd.h>
```

```
int gethostname(char *hostname, size_t size);
```


gethostbyname()

- Cette fonction prend en paramètre un nom de machine et renvoie son adresse IP :

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

```
struct hostent *gethostbyname(const char *name);
```

- Le struct renvoyé est le suivant :

```
struct hostent {
    char *h_name;
    char **h_aliases;
    int h_addrtype;
    int h_length;
    char **h_addr_list; //vecteur d'adresses
};
```

Le modèle client-serveur

- De nombreuses applications fonctionnent selon le modèle *client-serveur*, où un processus serveur répond aux requêtes de nombreux processus clients. Ceci est particulièrement vrai pour les applications de type réseau (telnet, ftp, ...).
- Fonctionnement typique d'une application clients-serveur sur un réseau :
 - Le serveur exécute une boucle infinie qui contient un `accept ()`.
 - Après cet `accept ()`, le serveur « *forke* », et le processus fils s'occupe de la demande effectuée par le processus client de la machine distante, tandis que le processus père revient à sa boucle infinie et effectue l'`accept ()` suivant.

Exercice

- Ecrire un programme serveur et un programme client, où le serveur attend une connexion, puis « forke ». Son fils envoie un message vers le client. Le client affichera le message reçu.

La fonction `select()`

- Un serveur doit parfois en même temps recevoir des données sur une *socket* (ou un fichier) et pouvoir recevoir des nouvelles connexions. Cela peut poser un problème, car les `recv()` et les `accept()` sont bloquants. On peut les rendre non-bloquants, mais on monopoliserait le processeur.
- Solution : la fonction `select()` permet d'écouter sur plusieurs *socket descriptors* en même temps et d'être réveillé par le premier qui devient actif :

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

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

La fonction `select()`

- Le type `fd_set` représente un ensemble de *file descriptors*. Ces ensembles peuvent être mis à jour avec les macros suivantes :
 - `FD_SET(int fd, fd_set *set)` – ajouter `fd` à un ensemble
 - `FD_CLR(int fd, fd_set *set)` – enlever `fd` d'un ensemble.
 - `FD_ZERO(fd_set *set)` – vider un ensemble.
 - `FD_ISSET(int fd, fd_set *set)` – tester si `fd` appartient à l'ensemble.
- `select()` écoute trois ensembles de *file descriptors* : `readfds`, `writelfds`, et `exceptfds` et attend que (au moins) l'un d'entre eux devienne actif.

La fonction `select()`

- Après l'appel de la fonction, `readfds` contiendra les *fd* prêts à être lus (ou *acceptés*), et `writefds` contiendra ceux prêts à être écrits.
- Le paramètre `n` doit être initialisé au *max_des_fd+1*.
- Le paramètre `timeval`, utilisé pour spécifier un *timeout*, est du type suivant :

```
struct timeval {  
    int tv_sec;    // seconds  
    int tv_usec;  // microseconds  
};
```

Exercice

- Ecrire un petit programme qui attend 3 secondes pour que quelque chose apparaisse sur l'input standard et dans ce cas l'affiche. Dans le cas contraire votre programme affichera que rien n'était présent sur l'input standard.