Programmation Réseaux

En C sous Linux

V. Limouzy

Utilisation des sockets

- Socket est utilisé pour communiquer en réseaux
- Différents types unifiés dans une même structure:
  - TCP
  - UDP
  - RAW

Fonction `socket()`

- Permet de créer une socket
  - i.e. un identifiant unique pour la communication réseau (comme un descripteur de fichier)
- `int socket(int domain, int type, int protocol);`
- Domain: type de réseaux
  on utilisera `PF_INET (pour IPv4)` ou `PF_INET6 (pour IPv6)` (PF pour Procotol Family)
- Type:
  - `SOCK_STREAM` pour TCP
  - `SOCK_DGRAM` pour UDP
  - `SOCK_RAW` pour faire de l'ICMP par exemple (nécéssite des droits particuliers)
- Protocol: 0 dans le doute Correspond au protocole de couche 4
  - Liste disponible dans *`/etc/protocols`*
  - `Ip`: 0; `icmp`: 1; `tcp`: 6; `udp`: 17 ….
- Renvoie -1 si erreur et erreur disponible avec `perror() ;` descripteur sinon

Affectation d'une Adresse: `bind()` 1/2

- Pour être contacté on doit affecter une adresse à la socket
  - Choix sur le n° de port (>1024)
  - Choix sur l'adresse IP si la machine a plusieurs adresses IPs.
    - Sinon affectation Automatique
    - On ne peut pas utiliser l'adresse d'une autre machine !
- `int bind(int socket, const struct sockaddr *addr, socklen_t addrlen);`

`bind()` 2/2

- socket est la socket qui va être traitée
- addr est une structure d’adresse générique (qui fonctionnera avec IPv4 et IPv6)
- `addrlen` correspond à la taille de la structure précédente
- Renvoie :

Structure sockaddr

- Structure générique pour stocker les adresses
- Utilisée comme paramètre dans les fonctions qui manipulent des adresses
- Des cast sont nécessaires avec les vrais structures d'adresses...

```
struct sockaddr{
sa_family_t sa_family;
// famille de sockaddr
// champ commun à toutes
// les struct d'adresses
// AF_INET;AF_INET6 ;

// AF_UN...


char sa_data[14];
// Données de l'adresse
// réelle
}
```

sockaddr_in et sockaddr_in6

```
struct sockaddr_in {
sa_family_t sin_family;
uint16_t  sin_port;
struct in_addr sin_addr;

};
```

- Pour IPv4
- famille d'adresses : AF_INET
- Port dans l'ordre d'octets réseau
- Adresse Internet (entier 32bits)

sockaddr_in et sockaddr_in6

```
struct sockaddr_in6 {
uint16_t   sin6_family;
uint16_t   sin6_port;
uint32_t   sin6_flowinfo;

struct in6_addr sin6_addr;
uint32_t  in6_scope_id;

};
```

- Pour IPv6
- AF_INET6
- Numéro de port
- Information de flux IPv6
- Adresse IPv6 (struct)
- Scope ID (nouveauté 2.4)

sockaddr_in et sockaddr_in6

```
struct in6_addr {

unsigned char
s6_addr[16];
};
```

- Adresse IPv6

Pour coder les 128 bits d’adresses

Stockage des entiers (1/2)

- Exemple un entier (en héxa): OxA0B1C2D3
- Big Endian / Gros Boutisme

  Ordre des octets:

  00|01 |02|03

  A0|B1|C2|D3

- Little Endian / Petit Boutisme

  Ordre des octets:

  00|01 |02|03

  D3|C2|B1|A0

- Dépend du processeur
- Big Endian:
  - IBM
  - Motorola
  -
- Little endian
  - Archi x86 (Intel, Amd,…)
  - Arm
  -

Stockage des entiers (2/2)

- Nécéssité de s’accorder sur un standard
- Ordre des bits du réseaux: BigEndian
- Nécéssite de convertir les entiers
  - À l’envoie
  - À la réception
  - Pour les adresses IPv4
- Fonctions Hôtes vers réseaux: `htnol  et htons`
- Fonctions Réseaux vers hôtes: `ntol et ntos`

Initialisation de structure d'adresse

Exemple pour l'affectation côté serveur

```
struct sockaddr_in addr ;

addr.sin_family=AF_INET ;

addr.sin_port= htons(15550);

addr.sin_addr= htonl(INADDR_ANY);
```

- Création d'une `struct` d'adresse
- Indique que c'est une adresse IPv4
- Indique le port (format réseau)
- Indique que l'adresse IP est toutes les Ips dispo de la machine

Manipulation d’adresse

- Les adresses sont soit au format IP (125.10.10.25 IPv4 ou `fe80::c1f3:be71:f1a8:d8c6` IPv6
- Soit sous forme d’url (`www.uca.fr ; www.lamontagne.fr`…)
- Pour établir une connexion, on doit renseigner une adresse IP
- Deux fonctions permettent de faire la conversion
  - `gethostbyname()` fonctionne uniquement pour IPv4
  - `getaddrinfo()` fonctionne pour IPv4 et IPv6

Structure addrinfo

```
struct addrinfo {
    int ai_flags;
    int ai_family;
    int ai_socktype;
    int ai_protocol;
    size_t  ai_addrlen;
    struct sockaddr *ai_addr;
    char  *ai_canonname;
    struct addrinfo *ai_next;
};
```

- options
- Indique la famille d’adresse AF_INET; AF_INET6 ou AF_UNSPEC
- Type de socket, (SOCK_STREAM ou SOCK_DGRAM) dans le doute 0
- Protocole, mettre à 0 pour tout accepter
- Taille de la structure d’adresse
- Structure d’adresse
- Nom canonique
- Prochaine structure d’adresse (comme dans une liste).

Fonction `getaddrinfo()`

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

- Node adresse forme pointée (125.6.6.7) ou texte (www.uca.fr)
- Service: protocole visée ou numéro de port
- Hints: contient les infos types demandés (cf. Exemple)
- Res: tableau de pointeur résultat, chaque case contient une antrée
- `freeaddrinfo(res)` libère le tableau utilisé

Utilisation getaddrinfo (1/3)

```
struct addrinfo hints;
struct addrinfo *result, *rp;

int sfd, s;
struct sockaddr_storage peer_addr;
socklen_t peer_addr_len;
ssize_t nread;
char buf[BUF_SIZE];
```

```
//  Mise à 0 de la structure
    memset(&hints, 0, sizeof(struct
    addrinfo));
    hints.ai_family = AF_UNSPEC;
   /* Allow IPv4 or IPv6 */
    hints.ai_socktype = SOCK_DGRAM;
     /* Datagram socket */
    hints.ai_flags = AI_PASSIVE;
    hints.ai_protocol = 0;
    /* Any protocol */
    hints.ai_canonname = NULL;
    hints.ai_addr = NULL;
    hints.ai_next = NULL;
```

Utilisation getaddrinfo (2/3)

```
s = getaddrinfo(addr,port , &hints, &result);
if(s!=0){
  fprintf(stderr, "getaddrinfo: %s\n",
    gai_strerror(s));
  exit(EXIT_FAILURE);
  }
return  result ;
```

Utilisation getaddrinfo (3/3)

```
Struct addrinfo * Addr ;//structure renseignée par getaddrinfo
struct addrinfo * rp ;
char message[BUF_SIZE] ;
memset(message,0,BUF_SIZE) ;
for(rp=Addr;rp!=NULL;rp=rp->ai_next){
      if(rp->ai_family==AF_INET){
        char * adress_ip = (char * )
malloc(sizeof(char)*INET_ADDRSTRLEN) ;
    struct sockaddr_in * sai =(struct sockaddr_in *) rp-
>ai_addr ;

    struct in_addr  Adresse= sai->sin_addr ;
    inet_ntop(rp->ai_family,& Adresse,adress_ip,
INET_ADDRSTRLEN);
    printf("IPv4 : %s | port: %d\n", adress_ip,ntohs
(sai->sin_port)) ;
free(adress_ip) ;
}
```

Conversion Adresse IP ↔ ASCII (IPv4)

- Fonction qui permettent de convertir

  une adresse IP en notation pointée

  `inet_ntoa()` pour Network to Ascii
- Fonction qui permet de transformer une adresse Ascii (i.e. `‘‘148.56.77.12’’`) en version réseau `inet_aton()` pour Ascii to Network
- Ne fonctionne qu’avec IPv4

Conversion Adresse IP ↔ ASCII (IPv4 et IPv6) (1/2)

- Convertir une adresse Binaire en chaîne de caractères:

  inet_ntop()

- `const char *inet_ntop(int af, const void *src,char *dst, socklen_t cnt);`
- Af pour Adresse Family: AF_INET (pour IPv4), AF_INET6 (pour IPv6)
- Src: Structure d'adresse (src pointe soit vers une struct `in_addr` ou `in_addr6`)
- Cnt: longueur de la structure
- Renvoie une chaîne de caractère qui représente l'adresse

Conversion Adresse IP ↔ ASCII (IPv4 et IPv6) (2/2)

- Convertir une chaîne de caractères en Binaire :inet_ntop()
- `const char *inet_ntop(int af, const void *src,char *dst, socklen_t cnt);`
- Af pour Adresse Family: AF_INET (pour IPv4), AF_INET6 (pour IPv6)
- Src: Structure d'adresse (src pointe soit vers une struct `in_addr` ou `in_addr6`)
- Cnt: longueur de la structure
- Renvoie une chaîne de caractère qui représente l'adresse

connect()

- Permet de d'initier la connexion vers une autre machine
- S'effectue coté Client
- `int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);`

  1) Socket par laquelle faire la connexion
  2) Adresse de l'hôte distant (IPv4 ou IPv6)
  3) Taille de la structure d'adresse

- Renvoie 0 si Ok, -1 sinon; erreur dispo avec `perror()`

accept()

- Permet d'accepter une nouvelle connexion (coté Serveur)
- `int accept(int sockfd, struct sockaddr *adresse, socklen_t *longueur);`

1) Socket sur laquelle accepter la connexion
2) Adresse du client qui demande la connexion
3) Taille de la structure d'adresse (pour déterminer si IPv4 ou IPv6)
4) Renvoie -1 si erreur ; une valeur positive (nouvelle socket)

`accept()`

- Crée un nouveau socket pour la communication dédiée
- La fonction est bloquante
- En général utilisé avec un `fork()` ou des `threads`
- La fonction `getpeername()` permet de connaître l'identité de de l'autre côté de la socket
- La fonction `getsockname()` permet de connaître l'identité de notre socket. Utile quand on a fait un `connect()` sans `bind()`

Schéma d'utilisation de `accept()`

```
int so= accept(fd,...) ;
switch(fork()){
   case 0: // fils
      close(fd);
      //communiquer sur so
   case 1://erreur
      exit(-1) ;
   default: //père
      close(so) ;
   }
```

- Dans le processus père on gère les connexions entrantes
- Dans le processus fils on s'occupe de la communication sur la socket dédiée

Flonction `close()`

- Une fois les communications terminées, on doit fermer les socket (comme des fichiers
  - Libère les ressources au niveaux du système (buffer;descripteur de fichiers;…)
  - Libère l'adresse et surtout le port utilisé
  -
- En TCP, l'adresse ne redevient pas disponible tout de suite !
  - Dû au protocole qui doit être fiable et donc possiblement demander de ré-expédier les dernières données envoyées.
  - Peut poser des problèmes en phase de test. Bind() impossible

Fonctions `read()/write()`

- La communications peut se faire avec les fonctions classiques:
  - Read: `ssize_t read(int fd, void *buf, size_t count);`
    - Lit depuis le descripteur fd.
    - Écrit le résultat dans buf.
    - De taille au plus count.
    - Renvoie le nombre d’octets réellement lus.
    - Fonction bloquante
  - Write: `ssize_t write(int fd, const void *buf, size_t count);`
    - Même paramètre que read()
    - Renvoie le nombre d’octets réellement écrits
    - Peut être bloquante si pas de lecture de l’autre côté.

Fonctions `send()`

- Des fonctions plus orientées réseaux sont dispo
- `ssize_t send(int s, const void *buf, size_t len, int flags);`
- Fonctionne en Mode Connecté (i.e. `SOCK_STREAM`)
- Fonctionne comme write mais avec des possibilités d'options
- Parmis les flags dispo on notera
  - `MSG_DONTWAIT` // Rend l'écriture non bloquante
  - `MSG_OOB` // Permet d'envoyer des données Hors bande

Fonction `recv()`

- Analogue à la fonction read()
- `ssize_t recv(int s, void *buf, ssize_t len, int flags);`
- Même type d'options que pour send()
  - `MSG_PEEK` permet de lire dans la file sans enlever les données
  - `MSG_DONTWAIT` lecture non bloquante mais possiblement incomplète
  - `MSG_WAITALL` Attend que toutes les données demandées soient transmises (bloque en attendant)

Entrées/Sorties non bloquantes

- On peut avec `send()` et `recv()` avoir des entrées sorties non bloquantes
- On doit gérer le fait que
  - Aucune donnée n'arrive
  - La donnée arrive de manière parcellaire (donner à reconstituer)
- On peut mettre les entrées/sorties classiques (read/write)

  en mode non bloquant avec `fcntl()` (mise en place et gestion compliquée)

`select()`

- La fonction select permet de « *surveiller* » les descripteurs de fichiers/sockets
- Surveille plusieurs éléments
- Indique quels éléments sont prêts en lectures
- Permet de garder des lectures bloquantes et de ne pas passer en mode non bloquant...

Fonctions pour `select()` (1/3)

- `int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);`
- `nfds`: Plus grand descripteur des 3 ensembles + 1
- `readfds`: Ensemble de descripteurs à surveiller en lecture
- `writefds`: Ensemble de descripteurs à surveiller en écriture
- `exceptfds` : Utilisé pour les sockets pour les données hors bande
- Structure de timeout : temps maximum à attendre avant de renvoyer une valeur

Fonctions pour `select()` (2/3)

- Retour :
  - Renvoie le nombre de descripteurs qui sont prêts (à liré/écrire ou autre)
  - 0 Si le timeout est arrivé à son terme
  - -1 sinon
- 4 Macros sont disponibles pour manipuler les FDSet
  - `FD_ZERO()` : Mets la structure à 0
  - `FD_CLR()` : Enlève un descripteur du set
  - `FD_SET()` : Ajoute le descripteur au set
  - `FD_ISSET()` : détermine si un descripteur a été activé

Fonctions pour `select()` (3/3)

- Select modifie les sets
  - Avant d'invoquer select on indique les descripteur à surveiller dans le set avec `FD_SET()`
  - Après le retour de la fonction on teste sur avec `FD_ISSET()` si un descripteur a déclenché une action
  - On doit réinitialiser la liste des descripteurs à surveiller si on souhaite ré-invoquer `select()`

Exemple d’utilisation de `select()`

```
fd_set rfds;
struct timeval tv;
int retval;
/* Surveiller stdin (fd 0) en attente d'entrées */
FD_ZERO(&rfds); //initialise la strucutre à 0
FD_SET(0, &rfds); //indique de surveiller sur 0
/* Pendant 5 secondes maxi */
    tv.tv_sec = 5;
    tv.tv_usec = 0;
    retval = select(1, &rfds, NULL, NULL, &tv);
    /* Considérer tv comme indéfini maintenant ! */
    if (retval == -1)
        perror("select()");
    else if (retval)
        printf("Des données sont disponibles maintenant\n");
        /* FD_ISSET(0, &rfds) est vrai */
    else
        printf("Aucune données durant les 5 secondes\n");
    exit(EXIT_SUCCESS);
```

Schéma Client/Serveur TCP

- Serveur
  1)Socket
  2)Bind
  3)Listen
  4)Accept
  5)send/recv
  6)Close
- Client
  1)Socket
  2)Connect
  3)send/recv
  4)Close

Communication UDP

- En UDP (`SOCK_DGRAM`) mode déconnecté
- Pas de `listen()/accept()` côté serveur
- Pas de `connect()` côté client.
- Par contre `bind()` est toujours nécessaire
- Deux variantes de `send` et `recv` :
  - `sentdo()`
  - `recvfrom()`

Fonction sendto()

- `ssize_t sendto(int sockfd, const void *buf, size_t len, int flags, const struct sockaddr *dest_addr, socklen_t addrlen);`
- Les 4 premiers paramètres commun à `send()`
- Les 2 derniers correspondent à l'adresse de l'hôte que l'on veut joindre
  - Il faut donc conserver les structures d'adresses

Fonction `recvfrom()`

- `ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t *addrlen);`
- Idem que pour sendto
- L'adresse `src_addr` doit exister (ce n'est pas la fonction qui la créé)
- Utile côté serveur pour savoir à quelle machine répondre
- Peut être laissé à `NULL` (ainsi que `addrlen`)
  - Fonctionnera quand même / Ne permettra pas de répondre

Inclusion de fichiers d’en tête

- `#include <stdio.h>`
- `#include <stdlib.h>`
- `#include <unistd.h>`
- `#include <string.h>`
- `#include <sys/types.h>`
- `#include <sys/socket.h>`
- `#include <netinet/in.h>`
- `#include <arpa/inet.h>`
- `#include <netdb.h>`

Ressources utiles:

- http://manpagesfr.free.fr
  - Contient la doc des fonctions réseaux et systèmes de manières détaillée.
- Livre de Cristophe Blaess : *Développement système sous Linux*

  Ordonnancement multitâche, gestion mémoire, communications, programmation réseau