Fiche réseaux

Sommaire :

Les adresses

Différentes types d'adresses

> - Adresse IP : adresse donnée temporairement à une entité sur un réseau donné. Une entité peut avoir plusieurs adresses IP éventuellement sur des réseaux différents.
> - Adresse MAC : adresse physique d'une entité stockée en général dans la carte réseau. Utilisée pour l'adressage sur un même réseau local. L'adresse est unique et pérenne.

Nous allons évoqués les adresses IP. Il y a deux différentes familles d'adresses IP :

- IPv4 : 32 bits, 4 octets, 4 chiffres de 0 à 255 séparés par des points. Exemple : 10.0.175.225
- IPv6 : 128 bits, 16 octets, 4 chiffres de 0 à 255 séparés par des points. Exemple : 2001:0db8:85a3:0000:0000:8a2e:0370:7334 (on peut omettre les zéros inutiles et remplacer les 0 consécutifs par des `:`)
  - Par exemple 1b01:fa06:842:0:0:0:189a:31af → 1b01:fa06:842::189a:31af

Résolution de nom

Pour l'humain, difficile de retenir une adresse IPv4 ou IPv6 → utilisation de noms de domaine pour désigner une adresse IP Exemple :

> www.informatique.univ-paris-diderot.fr

- **www** est la machine
- dans le sous-domaine **informatique**
- dans le sous-domaine **univ-paris-diderot**
- dans le domaine **fr**

localhost désigne en général l'interface locale à l'hôte.

> **Serveur DNS** (*Domain Name System*) : serveur permettant d'associer les adresses de noms avec leurs adresses IP.

- annuaire distribué (*il y a donc plusieurs serveurs DNS*)
- Le DNS contient aussi d'autres informations lié à un nom de domaine :
  - l'adresse IPV4 (A record)
  - l'adresse IPv6 (AAAA record)
  - les serveurs de courrier électronique pour le domaine (MX record)
  - les serveurs DNS du domaine (NS record)

commandes : **host et dig**

> **Définition d'un port :** Une machine peut avoir plusieurs applications réseaux qui tournent en parallèle. De plus, ces applications peuvent utiliser la même adresse IP.

→ **port** : entier non-signé de 16 bits qui correspond à une adresse locale à la machine.

Simuler un serveur ou client simple :

Les commandes telnet et netcat (ou nc) permettent entre autre de simuler un client simple.

```
telnet <adresse> <port>
netcat <adresse> <port>
```

Avec l'option -l, la commande netcat permet de simuler un serveur simple. Dans ce cas, on ne spécifie pas l'adresse.

Relation Client/Serveur

Définition d'une socket : Une socket est un point de communication d'un hôte A vers un hôte B qui permet l'envoie de données de A vers B et la reception par A de données provenant de B.

Étape générale pour créer un client TCP

Les étapes pour créer un client :

1. Création de la socket
2. Préparer l'adresse (IP + port) du destinataire (le serveur)
3. Faire une demande de connexion au serveur
4. Une fois la connexion établie, la conversation peut commencer...
5. Fermer la socket à la fin de la conversation

Client TCP IPv4/IPv6

Définition d'une socket : Une socket est un point de communication d'un hôte A vers un hôte B qui permet l'envoie de données de A vers B et la reception par A de données provenant de B.

Pour créer une socket en C :

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

- **domain** : famille de protocoles utilisée pour la communication
  - `PF_INET` : IPv4
  - `PF_INET6` : IPv6
- **type** : type de socket
  - `SOCK_STREAM` : TCP (TOUJOURS UTILISER CELUI-LÀ)
  - `SOCK_DGRAM` : UDP
- **protocol** : précise le protocole à utiliser s'il y en a plusieurs dans la famille. En général, il n'y en a qu'un et on peut alors mettre la valeur `0` (zéro). C'est le cas pour une socket TCP.
- **retour** : un descripteur de fichier (entier) qui correspond à la socket créée.

Client TCP IPv4

Étape 1 : Création de la socket

```
int sockfd = socket(PF_INET, SOCK_STREAM, 0);
```

On a maintenantune socket de communication.

Étape 2 : Préparer l'adresse (IP + port) du destinataire (le serveur). Utilisation de la structure `sockaddr_in` pour les adresses IPv4.

```
struct sockaddr_in {
short sin_family; //famille d'addresses: AF_INET
unsigned short sin_port; //numero de port
struct in_addr sin_addr;
char sin_zero[8]; //remplir de zero
};
```

D'abord, on crée une structure `sockaddr_in` et **on la remplit avec les 0**.

```
struct sockaddr_in adrso;
memset(&adrso, 0, sizeof(adrso));
```

On remplit ensuite les champs de la structure :

```
adrso.sin_family = AF_INET;
adrso.sin_port = htons(1234);
```

> *NE PAS OUBLIER DE CONVERTIR LE PORT EN BIG ENDIAN AVEC `htons` !!*

Puis, on remplit le champs sin_addr en utilisant la fonction :

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

- `af` : famille d'adresse (AF_INET ou AF_INET6)
- `src` : adresse IP sous forme de chaîne de caractères
- `dst` : adresse mémoire où sera stockée sous forme réseau l'adresse de destination →pointeur sur struct in_addr pour IPv4

Exemple dans le cas d'une adresse IPv4:

```
inet_pton(AF_INET, ''192.168.70.73'', &adrso.sin_addr);
```

Étape 3 : Faire une demande de connexion au serveur

```
int r = connect(sock, (struct sockaddr *) &adrso, sizeof(adrso));
```

Tester la connexion établie avec `r` (vaut -1 si il y a une erreur)

Etape 4 : Une fois la connexion établie, la conversation peut commencer.

- Pour envoyer des données, on utilise la fonction :

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

- Pour recevoir des données, on utilise la fonction :

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

- `sockfd` : numéro de la socket
- `buf` : adresse mémoire où se trouve
  - le message à envoyer pour `send`
  - le message reçu pour `recv`
- `len` : taille du message à envoyer ou à recevoir
- `flags` : optionnel, à mettre à 0

Exemple :

```
//*** envoie d'un message ***
  char bufsend[SIZE_MESS];
  memset(bufsend, 0, SIZE_MESS);

  sprintf(bufsend, "Hello %s\n", NOM);
  int ecrit = send(fdsock, bufsend, strlen(bufsend), 0);
  if(ecrit <= 0){
    perror("erreur ecriture");
    exit(3);
  }

  //*** reception d'un message ***
   char bufrecv[SIZE_MESS+1];
   memset(bufrecv, 0, SIZE_MESS+1);

   int recu = recv(fdsock, bufrecv, SIZE_MESS * sizeof(char), 0);
   //Vérification du nombre d'octets reçus
   if (recu < 0){
     perror("erreur lecture");
     exit(4);
   }
   if (recu == 0){
     printf("serveur off\n");
     exit(0);
   }
```

Etape 5 : Fermer la socket à la fin de la conversation

```
close(sockfd);
```

Client TCP IPv6

A la différence du client TCP IPv4, on utilise la structure `sockaddr_in6` pour préparer l'adresse (IP + port) du destinataire (le serveur).

```
struct sockaddr_in6 {
    u_int16_t sin6_family; //AF_INET6
    u_int16_t sin6_port; //numéro de port
    u_int32_t sin6_flowinfo;
    struct in6_addr sin6_addr; //adresse IPv6
    u_int32_t sin6_scope_id;
};
```

> **On initialise de manière différente la structure `sockaddr_in6` :**

```c
struct sockaddr_in6 adrso;
memset(&adrso, 0, sizeof(adrso));
address_sock.sin6_family = AF_INET6; //Différence
address_sock.sin6_port = htons(2121);
inet_pton(AF_INET6, "fe80:::43ff:fe49:79bf", &adrso.sin6_addr);
//Différence
```

Serveur TCP IPv4/IPv6

Étape générale pour créer un serveur TCP

Les étapes pour créer un serveur :

1. Création de la `socket serveur`
2. préparation de l'adresse (IP + port) d'écoute du serveur
3. lier la `socket serveur` à l'adresse et au port d'écoute
4. préparer la `socket serveur` à recevoir des connexions
5. accepter une connexion et créer la `socket client`. La conversation commence
6. fermer la `socket client` à la fin de la conversation

Serveur TCP IPv4

Étape 1 : Création de la socket serveur. Le début est le même que pour le client TCP IPv4.

```c
//*** creation de la socket serveur ***
int sock = socket(PF_INET, SOCK_STREAM, 0);
if(sock < 0){ perror("creation socket"); exit(1);}
//*** creation de l'adresse du destinataire (serveur) ***
struct sockaddr_in adrsock;
memset(&adrsock, 0, sizeof(adrsock));
adrsock.sin_family = AF_INET;
adrsock.sin_port = htons(2121);
adrsock.sin_addr.s_addr = htonl(INADDR_ANY);
```

On veut génaralement accepter les connexions sur toutes les interfaces réseaux de la machine. Pour cela, on utilise la constante `INADDR_ANY` pour le champs `sin_addr` de la structure `sockaddr_in`.

Étape 2 : On lie la socket serveur à l'adresse et au port d'écoute. Utilisation de la fonction `bind` :

```c
int r = bind(sock, (struct sockaddr *) &adrso, sizeof(adrso));
```

Ne pas oublier de tester (-1 si erreur)

Étape 3 : On prépare la socket serveur à recevoir des connexions. Utilisation de la fonction `listen` :

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

- `sockfd` : numéro de la socket
- `backlog` : nombre de connexions en attente. `0` signifie que le serveur est sans limite de connexions en attente.

Étape 4 : On accepte une connexion et on crée la socket client. Utilisation de la fonction `accept` :

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

Exemple :

```
struct sockaddr_in adrclient;
memset(&adrclient, 0, sizeof(adrclient));
socklen_t size=sizeof(adrclient);
int sockclient = accept(sock, (struct sockaddr *) &adrclient, &size);
if(sockclient == -1){
    perror("probleme socket client");
    exit(1);
}
```

Étape 5 *(facultative)* **:** Afficher la connexion acceptée

```
char adr_buf[INET_ADDRSTRLEN];
memset(adr_buf, 0, sizeof(adr_buf));

inet_ntop(AF_INET, &(adrclient.sin_addr), adr_buf, sizeof(adr_buf));
printf("adresse client : IP: %s port: %d\n", adr_buf,
ntohs(adrclient.sin_port));
```

On récupère l'adresse IPv4 du client dans `adrclient` → on récupère l'adresse IPv4 du client sous forme de chaîne de caractères avec la fonction `inet_ntop` *(permet de traduire sous forme de chaîne de caractères l'adresse src)*

Récupérer le port du client avec la fonction `ntohs` *(permet de traduire dans l'encodage de notre machine)*

Étape 6 : La conversation commence. On peut envoyer et recevoir des données.

Exemple :

```
if (sockclient >= 0) {
    //*** reception d'un message ***
    char buf[SIZE_MESS+1];
    memset(buf, 0, SIZE_MESS+1);
```

```
    int recu = recv(sockclient, buf, (SIZE_MESS) * sizeof(char), 0);
    if (recu <= 0){
      perror("erreur lecture");
      return(1);
    }
    buf[recu] = '\0';
    printf("%s\n", buf);

    //*** envoie d'un message ***
    memset(buf, 0, SIZE_MESS+1);
    sprintf(buf, "Salut %s", NOM);
    int ecrit = send(sockclient, buf, strlen(buf), 0);
    if(ecrit <= 0){
      perror("erreur ecriture");
      return(2);
    }
}
```

Serveur TCP IPv6

La manière de faire est la même que pour le serveur TCP IPv4. La seule différence est que l'on utilise la structure `sockaddr_in6` pour préparer l'adresse (IP + port) du serveur.

```
int sock = socket(PF_INET6, SOCK_STREAM, 0);
if(sock < 0){
perror("creation socket");
exit(1);
}
struct sockaddr_in6 address_sock;
memset(&address_sock, 0, sizeof(address_sock));
address_sock.sin6_family = AF_INET6; // Différence
address_sock.sin6_port = htons(2121);
address_sock.sin6_addr = in6addr_any; // Différence
```

> **ATTENTION :** `in6addr_any` est une constante qui permet d'accepter les connexions sur toutes les interfaces réseaux de la machine. *Différent de `INADDR_ANY` pour IPv4*

Si on veut affciher l'adresse IPv6 du client, on utilise la fonction `inet_ntop` comme ceci:

```
char addr_buf[INET6_ADDRSTRLEN];
inet_ntop(AF_INET6, &(adrclient.sin6_addr), adr_buf, sizeof(adr_buf));
//AF_INET6
printf("client connecte : %s %d\n", addr_buf, adrclient.sin6_port);
//sin6_port
```

Connexion par adresse de nom

Si nous souhaitons nous connecter à un serveur dont nous ne connaissons que son nom, nous devons utiliser la fonction `getaddrinfo` pour récupérer l'adresse IP du serveur. Pour cela nous devons remplir une structure `addrinfo` :

```
struct addrinfo {
int ai_flags; /* options */
int ai_family; /* famille de protocols pour la socket */
int ai_socktype; /* type de socket */
int ai_protocol; /* protocole pour la socket */
socklen_t ai_addrlen; /* taille de l'adresse */
struct sockaddr *ai_addr; /* addresse: IP, port... */
char *ai_canonname; /* nom canonique de l'adresse */
struct addrinfo *ai_next; /* pointeur sur le suivant dans la liste */
};
```

Fonction `getaddrinfo` :

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

Avec :

- `node` : nom du serveur
- `service` : port du serveur
- `hints` : structure `addrinfo` initialisée
- `res` : pointeur sur une structure `addrinfo` qui contiendra l'adresse du serveur. **C'est ce que cette `getaddrinfo` va remplir.**

> **Attentiion** : Ne pas oublier de créer la socket (`int *sock`) et sa taille (`int *addrlen`) avant d'appeler la fonction `getaddrinfo`.

```
// Créer la socket int *sock et
//Initialisation de la structure hints
struct sockaddr_in *addr; // struct sockaddr_in6 *addr;
struct addrinfo hints,*r,*p;
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET; //hints.ai_family = AF_INET6;
hints.ai_socktype = SOCK_STREAM;

//Verification de la validite de l'adresse
if ((getaddrinfo(hostname, port, &hints, &r)) != 0 || NULL == r) return
-1;
*addrlen = sizeof(struct sockaddr_in);//*addrlen = sizeof(struct
sockaddr_in6);
p=r;
while( p != NULL ){
  if((sock = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) > 0){
```

```
    if(connect(sock, p->ai_addr, addrlen) == 0)
      break;
  close(sock);
  }
  p = p->ai_next;
}
if (NULL == p) return -2;
//on stocke l'adresse de connexion
*addr = p->ai_addr; //*addr = (struct sockaddr_in6 *) p->ai_addr;
//on libère la mémoire allouée par getaddrinfo
freeaddrinfo(r);
```

Concurrence

Afin de créer un serveur qui peut gérer plusieurs clients en même temps, nous avons différentes solutions :

- **Solution 1 :** Créer un processus par client
- **Solution 2 :** Créer un thread par client
- **Solution 3 :** Utiliser une socket non bloquante

Créer un processus par client

> **Rappel :** Un `processus` est un *programme* en cours d'exécution.

Après avoir accepté une connexion, on crée un processus fils qui va gérer la conversation avec le client. Le processus père va continuer à accepter les connexions.

> **Rappel concernant la fonction `waitpid`:**

```
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

- `pid = -1` : attend n'importe quel fils
- `wstatus = NULL` : ne renvoie pas d'information sur le fils
- `options = WNOHANG` : ne pas bloquer si pas de fils terminé

```
while(1){
  int sockclient = accept(sock, (struct sockaddr *) &adrclient, &size);
  if(sockclient == -1){}; //gérer l'erreur...
  switch(fork()){
  case -1 : break; //gérer l'erreur...
  case 0 : //fils
    close(sock);
    int ret = communication(sockclient); // Récéption / Envoi
    exit(ret);
  default : //père
  close(sockclient);
  affiche_connexion(adrclient);
  while(waitpid(-1, NULL, WNOHANG) > 0); //récupération des zombies
```

```
  }
}
```

Créer un thread par client

> **`Thread` : Un thread est un fil d'exécution dans un programme, le programme étant lui même exécuté par un processus.

On utilise la bibliothèque `pthread` pour créer des threads. Pour cela, on doit compiler un programme avec l'option `-pthread`.

```
gcc -Wall -pthread serveur.c -o serv
```

On utilise la fonction `pthread_create` pour créer un thread.

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *
(*start_routine) (void *), void *arg);
```

Après que le serveur soit prêt a écouter les connexions sur le port a l'aide de la fonction `listen(sock,0)`, voici comment on implémente les threads :

```
while (1)
  {
    struct sockaddr_in6 addrclient;
    socklen_t size = sizeof(addrclient);

    //*** on crée la varaiable sur le tas ***
    int *sock_client = malloc(sizeof(int));

    //*** le serveur accepte une connexion et initialise la socket de
communication avec le client ***
    *sock_client = accept(sock, (struct sockaddr *)&addrclient, &size);

    if (sock_client >= 0)
    {
      pthread_t thread;
      //*** le serveur cree un thread et passe un pointeur sur socket
client à la fonction *serve(void *arg) qui recoit et envoie ***
      if (pthread_create(&thread, NULL, serve, sock_client) == -1)
      {
        perror("pthread_create");
        continue;
      }
      //*** affichage de l'adresse du client ***
      char nom_dst[INET6_ADDRSTRLEN];
      printf("client connecte : %s %d\n", inet_ntop(AF_INET6,
&addrclient.sin6_addr, nom_dst, sizeof(nom_dst)),
```

```
htons(addrclient.sin6_port));
    }
  }
  //*** fermeture socket serveur ***
  close(sock);
```

> **Attentinon :** il faut passer en argument de la fonction `serve` une variable allouée sur le tas

Cas où nous n'avons pas de pointeur de fonction

```
//fonction game(int)
printf("Connexion acceptee, nouvelle partie lancee.\n");
pthread_t thread;
int *sock = malloc(sizeof(int));
*sock = client_sock;
int a = pthread_create(&thread, NULL, game_1p_point, sock);
// Avec game_1p_point :
void *game_1p_point(void *arg)
{
  int *joueurs = (int *)arg;
  game_1p(sock);
  return NULL;
}
```

Thread ayant une fonction a plusieurs arguments

```
// 1.Créez une structure qui contient tous les paramètres que vous voulez
passer à votre fonction.
typedef struct {
  int parametre1;
  char* parametre2;
  float parametre3;
} Parametres;

// 2.Initialisez une variable de cette structure en y affectant les
valeurs de vos paramètres.

Parametres mesParametres;
mesParametres.parametre1 = 10;
mesParametres.parametre2 = "Bonjour";
mesParametres.parametre3 = 3.14;

// 3. Utilisez la fonction "pthread_create" pour créer un nouveau thread
et passer l'adresse de la variable de structure en tant qu'argument.
pthread_t monThread;
pthread_create(&monThread, NULL, maFonction, (void*)&mesParametres);

// 4. Dans votre fonction, récupérez les paramètres de la structure.
```

```c
void* maFonction(void* arg) {
  Parametres* mesParametres = (Parametres*)arg;
  int parametre1 = mesParametres->parametre1;
  char* parametre2 = mesParametres->parametre2;
  float parametre3 = mesParametres->parametre3;

  // Faites quelque chose avec les paramètres ici

  return NULL;
}
```

Les options de socket

La fonction `setsockopt` permet de modifier les options de socket.

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

- `sockfd` : socket a modifier
- `level` : indique le niveau du protocole auquel s'applique l'option spécifiée par `optname` qui est le nom de l'option a modifier
- `optval` : pointeur vers la variable contenant la nouvelle valeur de l'option

> Cette valeur courante peut être récupérée avec la fonction `getsockopt`.

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

Nous sommes en pleine transition entre les adresses IPv4 et IPv6. On veut donc écrire des applications *double stack* qui peuvent fonctionner avec les deux types d'adresses.

> - Soit on utilise deux sockets séparées, une pour IPv4 et une pour IPv6
> - Soit on utilise une socket polymorphe

Une socket polymorphe qui traduit les adresses IPv4 en IPv6 est appelée **IPv4-mapped**

```
127.2.3.4 →::ffff:127.2.3.4 (ou ::ffff:ff02:304)
```

Pour que la socket soit polymorphe, il faut utiliser la fonction `setsockopt`. Il faut désactiver l'option `IPV6_V6ONLY` au niveau du protocole `IPPROTO_IPV6`.

```c
int no = 0;
int r = setsockopt(sock, IPPROTO_IPV6, IPV6_V6ONLY, &no, sizeof(no));
```

> Cela doit être fait avant le `bind`