

II - Communication TCP

Communication TCP (Transmission Control Protocol)

La communication TCP se fait en mode connecté. La connexion est réalisée en trois étapes qu'on appelle **hand-shake** (poignée de main) :

- 1 un hôte A envoie une demande de connexion à un hôte B,
- 2 l'hôte B répond en acceptant la connexion,
- 3 l'hôte A répond en accusant réception.

La communication entre A et B peut alors commencer...

Dans le modèle client/serveur

- le client initie la connexion par le hand-shake,
- le client doit donc connaître l'adresse du serveur et le numéro de port sur lequel le serveur **écoute** (c'est-à-dire attend les demandes de connexions des clients).

Socket

Une **socket** est un point de communication d'un hôte A vers un hôte B qui permet l'envoi de données de A vers B et la réception par A de données provenant de B.

Une socket est une abstraction de la couche application définie par

- une adresse *IP locale* : adresse source
- un numéro de *port local*
- une adresse *IP distante* : adresse de destination
- un numéro de *port distant*
- un protocole : *TCP* ou *UDP*

→ permet au système de savoir quelle application est destinataire des données reçues.

Propriétés d'une socket TCP

- transmission *fiable* : pas de données perdues
- *ordre* et *intégrité* des données

Exemples de services utilisant TCP :

- http/https,
- smtp/pop3/imap,
- telnet...

Un client TCP

Les étapes pour créer un client :

- 1 créer 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

Un client TCP IPv4

Déclarer une socket TCP

Pour créer une socket en C

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

- **domain** : famille de protocoles utilisée
 - **PF_INET** : socket IPV4
 - **PF_INET6** : socket IPV6
 - (**PF_LOCAL...** pour une socket « locale »...)
- **type** : type de socket → **SOCK_STREAM** pour une socket TCP
- **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.
- l'entier renvoyé est le *descripteur* de la socket si celle-ci est créée.

Un client TCP IPv4

Déclarer une socket TCP

On commence donc par déclarer une socket IPv4

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

En cas d'erreur lors de la création de la socket, la fonction retourne **-1** et **errno** est positionné → **vérifier la création de la socket !**

Si la socket est créée, on a maintenant une socket de communication.

Mais comment fait-on pour joindre le serveur ?

Il faut commencer par définir l'adresse et le port de destination (serveur).

Un client TCP IPv4

Définir l'adresse et le port

En C, une adresse IP est représentée par le type

```
struct sockaddr {  
    sa_family_t sa_family;  
    char        sa_data[14];  
}
```

C'est une structure générale où `sa_family` représente la famille d'adresses et peut prendre les valeurs suivantes :

- `AF_INET` pour des adresses IPv4
- `AF_INET6` pour des adresses IPv6

On va en fait utiliser des structures plus spécifiques.

Un client TCP IPv4

Définir l'adresse et le port

Pour des connexions IPv4, on utilise la structure `struct sockaddr_in`

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

avec

```
struct in_addr {  
    unsigned long s_addr;       //adresse IPv4  
}
```

Un client TCP IPv4

Définir l'adresse et le port

On commence par déclarer et initialiser la structure

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

Ne pas oubliez de remplir de 0 la structure !!!

Puis, on remplit les champs `sin_family` et `sin_port`

```
adrso.sin_family = AF_INET;  
adrso.sin_port = htons(2121);
```

Ne pas oubliez la conversion en big-endian du numéro de port !!!

Un client TCP IPv4

Définir l'adresse et le port

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

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

qui affecte à `dst` l'adresse au format réseau (entier codé en big-endian) correspondant à l'adresse représenté par la chaînes de caractères `src`.

- `af` : famille de protocole → `AF_INET` pour IPv4, `AF_INET6` pour IPv6
- `src` : chaîne de caractères représentant l'adresse IP de destination
- `dst` : adresse mémoire où sera stockée sous forme réseau l'adresse de destination → pointeur sur `struct in_addr` pour IPv4
- retourne `1` si l'adresse a été traduite

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

Un client TCP IPv4

Définir l'adresse et le port

La fonction inverse

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

- permet de traduire sous forme de chaîne de caractères l'adresse src
- `size` est la taille maximale la chaîne de caractères stockée dans `dst`
→ `INET_ADDRSTRLEN` pour IPv4 et `INET6_ADDRSTRLEN` pour IPv6

Si on utilise des adresses IPv4, il y a également les fonctions :

```
int inet_aton(const char *cp, struct in_addr *inp); //non POSIX
in_addr_t inet_addr(const char *cp);                //obsolete
char *inet_ntoa(struct in_addr in);
```

Toutes ces fonctions traduisent des représentations d'adresses (chaînes de caractères ↔ réseau). Elles ne font pas appel à l'annuaire !!!

Un client TCP

Établir une connexion avec le serveur

Pour faire une demande de connexion, le client utilise la fonction

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

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

- Il faut faire un transtypage explicite du pointeur d'adresse vers une adresse de type struct sockaddr *.
- r vaut 0 en cas de succès et -1 sinon avec errno positionné

Ne pas oubliez de tester si la connexion a été établie !!!

Lorsque la connexion est établie, la conversation entre le client et le serveur peut commencer.

Un client TCP

Communication client/serveur

Pour envoyer et recevoir un message, on utilise les fonctions

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

```
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 (send)
 - le message reçu (recv)
- `len` : nombre d'octets
 - à envoyer (send)
 - max d'octets à recevoir (recv)
- `flags` : options → mettre ce champs à **0** si pas d'option

Un client TCP

Communication client → serveur

On prépare le message à envoyer, puis on l'envoie

```
char buf[SIZE_MESS];  
memset(buf, 0, SIZE_MESS);  
sprintf(buf, "Hello %s", NOM);  
  
int ecrit = send(sock, buf, strlen(buf), 0);
```

ecrit est égal au nombre d'octets envoyés si l'envoi s'est fait, -1 sinon et errno est positionné

Ne pas oublier de vérifier le nombre d'octets envoyés!!!

send avec le paramètre flags à 0 est équivalent à la fonction write :

```
int ecrit = write(sock, buf, strlen(buf));
```

Un client TCP

Communication serveur → client

On prépare le tampon pour recevoir le message, puis on le remplit

```
char buf[SIZE_MESS+1];  
memset(buf, 0, SIZE_MESS+1);  
  
int recu = recv(sock, buf, SIZE_MESS * sizeof(char), 0);
```

recu est égal :

- -1 en cas d'erreur et errno est positionné
- 0 si le serveur a terminé proprement la connexion
- au nombre d'octets reçus sinon

Ne pas oubliez de vérifier le nombre d'octets reçus!!!

recv avec le paramètre flags à 0 est équivalent à la fonction read :

```
int recu = read(sock, buf, strlen(buf));
```


Un client TCP

Fermeture de la socket

Lorsque le client termine la conversation ou avant de quitter le processus, on ferme la socket en faisant appel à la fonction `close` que vous connaissez (`man 2 close`)

```
close(sock);
```

Cela libère le descripteur `sock` et si plus aucun autre descripteur ne pointe sur la connexion, celle-ci est terminée proprement.

Comment tester son client ?

on peut simuler un serveur simple avec

```
netcat -l <num_port>
```

Cela signifie que **netcat** écoute sur le port **<num_port>**.
L'adresse ip est celle de la machine sur laquelle tourne **netcat**.

Si on ajoute de plus l'option :

- **-k** : on obtient une connexion persistente
- **-6** : le serveur attend des connexions en IPv6

Il est important de tester vos applications en réseau et pas seulement en local !!!

Qu'est ce qui change pour un client TCP IPv6 ?

Les étapes pour créer un client :

- 1 **créer 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

On commence par déclarer une socket IPv6

```
int sock = socket(PF_INET6, SOCK_STREAM, 0);
```

Un client TCP IPv6

Définir l'adresse et le port

On prépare ensuite l'adresse et le port de destination (serveur).

On utilise pour cela la structure `struct sockaddr_in6`

```
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;    //numéro de l'interface
};

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

Un client TCP IPv6

Définir l'adresse et le port

On commence par déclarer et initialiser la structure avec l'adresse et le port de destination

```
struct sockaddr_in6 adrso;  
memset(&adrso, 0, sizeof(adrso));  
  
address_sock.sin6_family = AF_INET6;  
address_sock.sin6_port = htons(2121);  
inet_pton(AF_INET6, "fdc7:9dd5::3f:fe49:79bf", &adrso.sin6_addr);
```

Un client TCP IPv6

Et ensuite...

Le type de connexion n'a pas besoin d'être à nouveau spécifié lors des étapes de connexion (`connect`), de conversation (`send`, `write`, `recv`, `read`) et de fermeture de la socket (`close`).

⇒ même code en IPv4 ou IPv6

Un client TCP IPv6

adresse locale au lien

Si l'adresse est locale au lien (fe80::/10), il est nécessaire de donner le numéro de l'interface utilisée par le serveur.

Si on ne connaît que le nom de l'interface (et pas son numéro), on peut, pour déterminer son numéro, utiliser la fonction :

```
unsigned int if_nametoindex(const char *ifname)
```

où `ifname` représente le nom de l'interface.

```
address_sock.sin6_scope_id = if_nametoindex("eth0");
```

Un client TCP

Préparer l'adresse du destinataire

ipv4 globale

<code>ADDR = "194.1.2.3"</code>	<code>ADDR = "192.168.13.56"</code>
<code>dev = "eth0"</code>	<code>dev = "eth1"</code>
<pre>struct sockaddr_in adso; memset(&adso, 0, sizeof(adso)); adso.sin_family = AF_INET; adso.sin_port = htons(PORT); inet_pton(AF_INET, ADDR, &adso.sin_addr);</pre>	<pre>struct sockaddr_in adso; memset(&adso, 0, sizeof(adso)); adso.sin_family = AF_INET; adso.sin_port = htons(PORT); inet_pton(AF_INET, ADDR, &adso.sin_addr);</pre>

ipv4 locale

ipv6 globale

<code>ADDR = "2001:d0:304:400::67c"</code>	<code>ADDR = "fdc7:9dd5:2c66::43ff:fe49:79bf"</code>
<code>dev = "eth0"</code>	<code>dev = "eth0"</code>
<pre>struct sockaddr_in6 adso; memset(&adso, 0, sizeof(adso)); adso.sin6_family = AF_INET6; adso.sin6_port = htons(PORT); inet_pton(AF_INET6, ADDR, &adso.sin6_addr);</pre>	<pre>struct sockaddr_in6 adso; memset(&adso, 0, sizeof(adso)); adso.sin6_family = AF_INET6; adso.sin6_port = htons(PORT); inet_pton(AF_INET6, ADDR, &adso.sin6_addr);</pre>

ipv6 locale

Un client TCP

Préparer l'adresse du destinataire

ipv4 globale

<code>ADDR = "194.1.2.3"</code>	<code>ADDR = "192.168.13.56"</code>
<code>dev = "eth0"</code>	<code>dev = "eth1"</code>
<pre>struct sockaddr_in adso; memset(&adso, 0, sizeof(adso)); adso.sin_family = AF_INET; adso.sin_port = htons(PORT); inet_pton(AF_INET, ADDR, &adso.sin_addr);</pre>	<pre>struct sockaddr_in adso; memset(&adso, 0, sizeof(adso)); adso.sin_family = AF_INET; adso.sin_port = htons(PORT); inet_pton(AF_INET, ADDR, &adso.sin_addr);</pre>

ipv4 locale

ipv6 globale

<code>ADDR = "2001:d0:304:400::67c"</code>	<code>ADDR = "fe80::4849:43ff:fe49:79bf"</code>
<code>dev = "eth0"</code>	<code>dev = "eth0"</code>
<pre>struct sockaddr_in6 adso; memset(&adso, 0, sizeof(adso)); adso.sin6_family = AF_INET6; adso.sin6_port = htons(PORT); inet_pton(AF_INET6, ADDR, &adso.sin6_addr);</pre>	<pre>struct sockaddr_in6 adso; memset(&adso, 0, sizeof(adso)); adso.sin6_family = AF_INET6; adso.sin6_port = htons(PORT); inet_pton(AF_INET6, ADDR, &adso.sin6_addr); adso.sin6_scope_id = if_nametoindex(dev);</pre>

Un client TCP

Retour sur `struct sockaddr`

- `struct sockaddr_in` a la même taille que `struct sockaddr`
- `struct sockaddr_in6` a une taille plus grande que `struct sockaddr`

Les fonctions comme `connect`, ou encore `bind` ou `accept` que nous verrons la semaine prochaine, doivent fonctionner pour des adresses IPv4 ou IPv6. Le 2ème paramètre est donc de type `struct sockaddr *` et le 3ème paramètre contient sa **taille réelle**.

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

`connect` connaît l'adresse mémoire où est stockée l'adresse de socket et connaît également la taille réelle de cette dernière \Rightarrow il a toutes les informations nécessaires pour récupérer l'adresse de connexion.

Un client TCP

Remarques importantes sur `send` et `recv`

- Sur le réseau, on envoie et on lit des flux d'octets. Ces octets peuvent former des chaînes de caractères, des entiers, ...
- On ne peut pas considérer que tous les octets à envoyer seront bien envoyés lors d'un appel à `send` \Rightarrow boucle sur `send` si on veut être sûr de bien tout envoyer :

```
int ecrit = 0, lg = strlen(buf);
while(ecrit >= 0 && ecrit < lg)
    ecrit += send(sock, buf+ecrit, lg-ecrit, 0);
```

- `send` peut bloquer si le tampon d'écriture du système est plein.
- Un appel à `send` peut nécessiter plusieurs appels à `recv` \Rightarrow boucle sur `recv` si on veut être sûr de bien tout lire