# TP sur les protocoles Internet

Pour obtenir de l'aide sous Unix, il suffit d'utiliser le manuel, qui est disponible avec la commande `man`. La syntaxe d'utilisation est `man sujet`. Il existe parfois plusieurs sections concernant le même sujet : il faut alors utiliser la syntaxe `man section sujet`. Les sections les plus utiles sont : `1` pour les commandes, `2` pour les appels système, et `3` pour les fonctions des bibliothèques.

**Exercice 1.** *Dans quelles sections du manuel se trouvent la documentation de* `printf`, `pwd` *et* `read` *?*

**Exercice 2.** *Le sujet* `write` *se trouve dans plusieurs sections. Lesquelles ?*

**Exercice 3.** *Pourquoi certaines fonctions du C (comme* `printf`*) sont dans la section 3, alors que d'autres (comme* `write`*) sont dans la section 2 ?*

## 1 Programmation *socket*

Tout programme pouvant communiquer en réseau utilise, à la base, le mécanisme client/serveur (même si ce mécanisme semble parfois caché aux utilisateurs). L'outil système pour développer un client/serveur est la *socket* (pour connecteur ou prise), une *socket* étant l'extrémité d'un canal de communication. La *socket* est un fichier particulier (non régulier), au-dessus de la couche transport.

Pour que deux programmes puissent communiquer, il faut que les deux programmes utilisent le même protocole de transport (UDP ou TCP), le même port de communication, et que l'un lance un serveur (c'est-à-dire une *socket* en écoute) et que l'autre lance un client (c'est-à-dire une *socket* en connexion).

### 1.1 Programmation d'un client/serveur UDP

En pratique, très peu de programmes utilisent UDP. Nous allons toutefois commencer par étudier UDP, étant donné que la création de *sockets* UDP étant légèrement plus simple que la création de *sockets* TCP.

#### 1.1.1 Serveur UDP

Pour créer un serveur UDP sur le port 12345 (par exemple), il faut appeler la fonction `socket` avec pour paramètres `AF_INET`, `SOCK_DGRAM` (qui précise qu'il faut utiliser UDP), et `0`. Il faut ensuite faire un appel à `bind` de la manière suivante (`s` étant la *socket*) :

```
struct sockaddr_in sin;
int i, ret;
sin.sin_family = AF_INET;
sin.sin_port = htons(12345);
```

```
sin.sin_addr.s_addr = htonl(INADDR_ANY);
for (i=0; i<8; i++) {
sin.sin_zero[i] = 0;
}
ret = bind(s, (struct sockaddr *)&sin, sizeof(struct sockaddr_in));
if (ret<0) {
printf("error");
}
```

Lorsque l'appel à `bind` est effectué, on dit que le serveur est ouvert, puisqu'il est prêt à recevoir des messages.

Pour recevoir des données, il faut appeler la fonction `recvfrom`. Cette fonction est bloquante : elle attend que des données soient reçues. Pour envoyer des données, il faut appeler la fonction `sendto`. Finalement, Pour fermet la socket, il suffit de faire appel à `close`. Lorsque l'appel à `close` est effectué, le serveur est fermé.

**Exercice 4.** *Créez un serveur UDP qui écoute sur un port passé en paramètre, et qui affiche tous les messages reçus. Quand le message* `quit` *est reçu, le serveur UDP termine.*

**Exercice 5.** *Pourquoi utiliser* `htons` *pour le numéro de port ?*

**Exercice 6.** *Vérifiez avec* `netstat` *que votre serveur est bien lancé sur le bon port.*

### 1.1.2 Client UDP

Pour créer un client UDP, de la même manière, il faut appeler la fonction `socket` pour créer une *socket*. Toutefois, la différence réside dans l'appel à bind. Dans le champ `sin.sin_addr.s_addr`, il faut préciser l'adresse IP du serveur en utilisant la fonction `inet_aton`.

Les autres fonctions s'utilisent de la même manière : `recvfrom`, `sendto`, `close`.

**Exercice 7.** *La structure* `sockaddr_in` *doit contenir l'adresse au format réseau. Faut-il utiliser* `htonl` *sur l'adresse retournée par* `inet_aton` *?*

**Exercice 8.** *Créez un client UDP qui se connecte sur un serveur dont l'adresse est passée en paramètre, sur le port passé en paramètre et qui envoie au serveur un message tapé au clavier par l'utilisateur. Testez votre client avec votre serveur. Vous pouvez utiliser l'adresse* `127.0.0.1`*, qui est l'adresse localhost, c'est-à-dire l'adresse de la machine elle-même.*

**Exercice 9.** *Modifiez votre client pour qu'il envoie au serveur tous les messages tapés au clavier par l'utilisateur, plutôt qu'un seul. Le client s'arrête quand l'utilisateur ferme* `STDIN` *(ce qui se fait avec* `Ctrl-D`*). Est-ce que vous voyez une différence côté serveur ?*

### 1.1.3 Communications UDP

**Exercice 10.** *Testez votre programme en lançant votre serveur UDP, et en demandant à plusieurs personnes de vous envoyer des messages depuis leurs clients UDP (sans arrêter votre serveur).*

**Exercice 11.** *Considérons le protocole suivant : un client se connecte, envoie un message, le serveur répond avec un autre message, et les deux programmes quittent. Est-ce que ce protocole est adapté à UDP ? Pourquoi ?*

## 1.2 Programmation d'un client/serveur TCP

L'essentiel de la programmation réseau est basé autour de clients/serveurs TCP (avec, dans certains cas, de nombreux clients et/ou serveurs sur une même machine).

### 1.2.1 Serveur TCP

La création d'un serveur TCP se fait en appelant les fonctions suivantes, dans l'ordre : `socket` (avec comme deuxième paramètre `SOCK_STREAM`), `bind`, `listen`. La fonction `listen` prend en paramètre le nombre de demandes de connexions simultanées que le serveur peut gérer (qui est différent du nombre de connexions simultanées que le serveur peut gérer). En général, choisir une valeur de 5 est suffisant.

Pour accepter la communication d'un nouveau client, il faut appeler la fonction `accept`. Cette fonction est bloquante et retourne le descripteur d'une nouvelle *socket* à chaque fois qu'un nouveau client se connecte sur le serveur. Quand le serveur est en attente sur le `accept`, on dit qu'il est ouvert. Quand la *socket* faisant les `accept` est fermée, on dit que le serveur est fermé. Un serveur fermé peut avoir des clients connectés ; en revanche, il ne peut pas accepter de nouveaux clients.

L'envoi et la réception de messages se font *via* les fonctions `recv` et `send` (mais pourrait aussi bien se faire *via* les fonctions `read` et `write`).

**Exercice 12.** *Considérons un serveur TCP ouvert et sur lequel trois clients sont connectés. Combien de sockets doit-il gérer ?*

**Exercice 13.** *Écrivez un programme qui ouvre un serveur TCP sur un port passé en paramètre, attend un client, affiche tous les messages que le client envoie, et recommence à attendre un nouveau client quand le précédent a quitté. Quand un client tape* `quit`*, le serveur quitte.*

**Exercice 14.** *Testez votre programme avec* `telnet`*.*

**Exercice 15.** *Pourquoi le serveur peut-il détecter quand un client quitte en TCP, alors qu'il ne pouvait pas le détecter en UDP ?*

**Exercice 16.** *Pourquoi votre programme ne peut pas gérer plusieurs clients en parallèle ? Que se passe-t'il si un grand nombre de clients (une demi-douzaine) essayent de se connecter en même temps ? Comment augmenter ce nombre ?*

### 1.2.2 Client TCP

Pour programmer un client TCP, il suffit de faire appel aux fonctions `socket`, puis `connect` (qui a une syntaxe très proche de `bind`). Les fonctions `recv`, `send` et `close` peuvent ensuite être utilisées.

**Exercice 17.** *Programmez un client TCP, qui se connecte sur un serveur dont l'adresse IP et le port sont passés en paramètre. Le client envoie au serveur tous les messages tapés par l'utilisateur. Testez votre client avec votre serveur.*

## 1.3 Support de clients TCP multiples

La plupart des serveurs doivent être en mesure de gérer des clients multiples, simultanément. Cela se fait en manipulant plusieurs *sockets* en parallèle. Plus généralement, le besoin de manipuler plusieurs descripteurs de fichiers (réguliers ou non) intervient dans de nombreuses situations :

- un client TCP (comme `telnet`) qui souhaite afficher les messages reçus depuis la *socket*, et envoyer les messages tapés sur `STDIN`,
- un client muni d'une interface utilisateur graphique, qui souhaite gérer les messages reçus depuis la *socket*, et traiter les événements de l'interface graphique,
- un programme qui souhaite faire des vérifications sur le réseau (comme des recherches de mise à jour) en tâche de fond.

Il existe trois techniques pour gérer de nombreux descripteurs en parallèle : l'attente active, la programmation multi-processus, et l'attente multiple bloquante.

### 1.3.1 Attente active

L'attente active consiste à rendre les descripteurs de fichiers non bloquants en lecture, et à les interroger successivement, avec une grande fréquence. Il s'agit d'une méthode très simple à programmer, mais inefficace.

Pour rendre un descripteur de fichier non bloquant, il faut utiliser la fonction `fcntl` de la manière suivante : on utilise la commande `F_GETFL` de `fctnl` pour obtenir les *flags* actuels de la *socket*, on ajoute le bit `O_NONBLOCK` dans les *flags* (par opération binaire), puis on utilise la commande `F_SETFL` avec ces nouveaux *flags*.

Une fois que la *socket* est en mode non bloquant, les appels à `recv` retournent `-1` et `errno` vaut `EWOULDBLOCK` si aucune donnée n'est disponible. Entre deux appels à `recv`, on utilise généralement une attente courte (obtenue avec `sleep` ou `usleep`).

**Exercice 18.** *Programmez un serveur TCP qui attend la connexion de trois clients TCP, puis met en place un système de communication de type chat entre ces trois clients, en utilisant l'attente active. Quand un client envoie un message, le message est transmis aux deux autres clients. Quand un client quitte, le serveur quitte.*

**Exercice 19.** *Quelle est la durée que vous avez choisie pour l'attente entre deux tentatives de réception ? Quelle est la durée optimale ?*

### 1.3.2 Programmation multi-processus

La programmation multi-processus consiste à lancer un processus (ou un *thread*) par descripteur à gérer. La programmation est d'une difficulté moyenne, la méthode est efficace pour un nombre raisonnable de processus, mais elle pose de gros problèmes de communications inter-processus qui sont souvent difficiles à résoudre.

La programmation multi-processus n'est pas traitée dans cette UE (cf l'UE système du S6).

### 1.3.3 Attente multiple bloquante

L'attente multiple bloquante consiste à attendre de manière bloquante sur plusieurs descripteurs simultanément. La mise en place de cette solution nécessite beaucoup de

manipulation de structures d'ensembles de descripteurs, mais il s'agit d'une solution efficace (car elle est bloquante, et ne consomme donc pas de ressources quand aucune donnée n'est disponible).

L'attente multiple bloquante se fait au moyen de l'appel système `select`. Cet appel système prend en paramètre un ensemble de descripteurs (initialisé avec les macros `FD_ZERO`, `FD_SET` et `FD_ISSET`), et retourne un ensemble de descripteurs ayant des données disponibles en lecture (c'est-à-dire, sur lesquels on peut faire un appel à `recv` sans bloquer).

**Exercice 20.** *Programmez un serveur TCP qui attend la connexion de trois clients TCP, puis met en place un système de communication de type chat entre ces trois clients, en utilisant l'attente multiple bloquante. Quand un client envoie un message, le message est transmis aux deux autres clients. Quand un client quitte, le serveur quitte.*

# 2 Programmation de protocoles Internet

Nous allons à présenter programmer des clients et serveurs pour les protocoles Internet.

## 2.1 Client HTTP

Le protocole HTTP *(Hypertext Transfer Protocol)* est un protocole basé sur TCP et utilisant le port 80 par défaut. Il s'agit d'un protocole textuel de type requête/réponse. Pour accéder à une URL *(Uniform Resource Locator)*, le client (appelé un navigateur dans le contexte HTTP) décompose l'URL en un nom de serveur et en une URI *(Uniform Resource Identifier)*. Le client se connecte alors sur le serveur en question, sur le port 80 par défaut, et demande la page correspondant à l'URI au moyen de la commande HTTP `GET`, en précisant la version du protocole HTTP qu'il utilise. Le client peut envoyer d'autres informations au serveur (comme l'adresse du serveur destination lorsque l'on passe par un proxy, les encodages acceptés, etc), en indiquant une information par ligne. Quand le client a terminé sa requête, il envoie une ligne vide.

Par exemple, pour se connecter sur `http://wwww.google.com/` en passant par un proxy, la requête HTTP minimale (envoyée au proxy, sur le port du proxy) est :

```
GET / HTTP/1.0
Host: www.google.com
```

Notez que cette requête doit se terminer par une ligne vide.

Le serveur répond en deux parties : un entête et un corps. L'entête commence par un code indiquant si la réponse est positive ou négative, et par des informations pour le client (comme la taille du corps, le type de fichier, etc). Chaque information est sur une ligne séparée. Pour indiquer la fin de l'entête, le serveur envoie une ligne vide. Ensuite, le serveur envoie dans le corps le fichier qui était demandé par le client.

**Exercice 21.** *À quoi correspondent les codes HTTP suivants : 200, 301, 302, 400, 403, 404, 418 et 500 ?*

**Exercice 22.** *En utilisant* `telnet`*, sauvegardez dans un fichier la page HTML correspondant à la page d'accueil de* `www.google.com`*. Ouvrez le fichier obtenu avec un navigateur web.*

**Exercice 23.** *Écrivez une fonction* `splitURL` *qui prend en paramètre une chaîne de caractère correspondant à une URL, et qui renvoie le nom de serveur (d'une part) et l'URI (d'autre part). Vous aurez à gérer des URL contenant* `http://` *ou non. Notez que l'URI vide n'existe pas, il faut utiliser* `/` *à la place.*

**Exercice 24.** *Écrivez une fonction* `downloadURI` *qui prend en paramètre un nom de serveur, une URI, et un nom de fichier, et qui enregistre la page correspondant à l'URI (sur le serveur indiqué) dans le fichier.*

**Exercice 25.** *Écrivez une fonction* `getImagesURLs` *qui prend en paramètre un nom de fichier, et qui retourne toutes les URL d'images contenues dans ce fichier. Une image HTML est désignée par* `img src=`*, suivi d'une URL (entre guillemets simples ou doubles). Il peut y avoir des informations supplémentaires (inutiles dans notre cas) entre la balise* `img` *et l'attribut* `src`*.*

**Exercice 26.** *Écrivez un programme qui prend en paramètre une URL, et qui télécharge l'URL correspondante, ainsi que toutes les images contenues dans le fichier HTML. Testez le sur* `www.google.fr`*.*