# TRAVAUX PRATIQUES
# Communication Socket entre processus

## Exercice 1 : Première communication socket

On désire réaliser un serveur itératif recevant des connexions de clients en mode connecté à l'aide de Socket TCP. A chaque nouvelle connexion d'un client, le serveur doit lui envoyer le message suivant d'une taille de 29 caractères : « Bienvenue, connexion réussie. »
De son côté le client reçoit le message envoyé par le serveur et l'affiche sur une console.
Reprenez les squelettes vu en cours et donnez les programmes associés au serveur et au client. On considérera que le serveur peut être contacté sur le numéro de port 55555, et pour simplifier la mise en œuvre les programmes client(s) et serveur seront exécutés sur la même machine.
**Rappel :** les programmes pourront être compilés avec la commande suivante : `gcc –o prog prog.c`

Dans l'exercice précédent, vous avez réalisé un premier programme permettant de faire communiquer deux processus à l'aide de l'API Socket. Afin d'échanger correctement des informations avec cet outil, il est nécessaire de mettre en place un protocole de communication au-dessus de l'API Socket. Ainsi, dans la suite de ce TP nous allons nous intéresser au formatage et au traitement des informations en vu de leur échange à travers une Socket.

## Pré-requis

Pour échanger les données en toute généralité, c'est-à-dire quelque soit le contenu et la taille des données, nous devons mettre en place un protocole d'échange d'informations entre les processus qui désirent communiquer.
Le protocole le plus simple pour faire un échange d'informations à travers une Socket est de signaler avec un premier message la taille des données à recevoir au destinataire avant l'envoie des données dans un second message. Cependant, l'échange de valeurs numériques entre deux machines en réseau n'est pas aussi simple.

### *Formatage des entiers*

Vous avez vu en cours que d'une machine à l'autre les entiers peuvent être stockés et lus suivant la convention *Big-Endian* ou *Little-Endian*. Pour simplifier le développement d'applications communiquant à travers le réseau, une convention a été introduite, appelée *Network Byte Order*.
De plus, d'une machine à une autre les entiers peuvent être encodés avec un nombre de bits différent. Pour pallier à ce problème, deux types ont été introduit en langage C afin de spécifier le nombre de bits utilisés pour stocker des entiers non signés (dans la librairie `inttypes.h`) :

- `uint16_t` : utilisation de 16 bits pour stocker un entier court non signé,
- `uint32_t` : utilisation de 32 bits pour stocker un entier long non signé.

On rappel également que plusieurs primitives vous permettent de passer d'un format à l'autre :

- `htons` (Host TO Network Short) permet de convertir un entier court (16 bits) du format local vers le format réseau dont le prototype est donné ci-dessous :

```
uint16_t htons(uint16_t hostshort);
```

- `htonl` (Host TO Network Long) permet de convertir un entier long (32 bits) du format local vers le format réseau dont le prototype est donné ci-dessous :

```
uint32_t htonl(uint32_t hostlong);
```

- `ntohs` (Host TO Host Short) permet de convertir un entier long (16 bits) du format réseau vers le format local dont le prototype est donné ci-dessous :

```
uint16_t ntohs(uint16_t netshort);
```

- `ntohl` (Host TO Host Long) permet de convertir un entier long (32 bits) du format réseau vers le format local dont le prototype est donné ci-dessous :

```
uint32_t ntohl(uint32_t netlong);
```

Les données sont échangées comme un flot d'octets qui peuvent être encodés à l'aide de chaînes de caractères. Il est nécessaire de pouvoir transformer les données en chaîne de caractères et vice-versa. On parle aussi de sérialisation des données, c'est-à-dire le fait d'aplatir (dans le cas d'une structure) ou mettre bout à bout différentes données dans une chaîne afin de l'envoyer.

*Transformation en flot de données*

**Les entiers**

Nous allons tout d'abord nous intéresser à la transformation d'un entier court ou long en chaîne de caractères en vu de son envoi à travers une Socket.
Un entier court (resp. long) est stocké sur 16 bits (resp. 32 bits) et nécessite au moins une chaîne de taille 2 octets (resp. 4 octets) pour son stockage.
Pour l'envoi d'un entier, nous allons tout d'abord convertir cet entier au format réseau puis copier les octets dans une chaîne de caractère ayant la taille adaptée pour l'envoyer à travers la Socket. Cela peut être réalisé avec les appels suivants :

```
int nb1;
uint16_t nb2;
char tab[2];
…
nb2=htons((uint16_t) nb1);
memcpy(tab, &nb2, 2);

send(sockfd, tab, 2*sizeof(char), 0);
```

La fonction `memcpy()` permet de copier un ensemble d'octets d'une zone mémoire à une autre (ici copier les données situées à l'adresse &nb2 vers l'adresse où est stockée le tableau tab).
Les étapes sont inversées lors de la réception de l'entier à l'autre bout de la Socket, comme indiqué ci-dessous :

```
int nb1;
uint16_t nb2;
char buf[2];
…
recv(sockfd, buf, 2*sizeof(char), 0);

memcpy(&nb2, buf, 2);
nb1=(int) ntohs(nb2);
```

## Exercice 2 : Seconde communication socket

On désire réaliser un serveur itératif gérant des connexions de clients en mode connecté à l'aide de Socket TCP. A chaque nouvelle connexion d'un client, le serveur doit lui renvoyer son numéro de connexion (défini par un compteur géré et maintenu par le serveur). De son côté le client reçoit son numéro de connexion et l'affiche sur une console.
Donnez les programmes associés au serveur et au client. On considérera que le serveur peut être contacté sur le numéro de port 55555, et pour simplifier la mise en œuvre les programmes client(s) et serveur seront exécutés sur la même machine.

## Exercice 3 : Communication socket avec serveur parallèle

On reprend le contexte de l'exercice 2, mais cette fois-ci on désire réaliser un serveur parallèle pour traiter les réponses aux clients.

1. Reprenez le programme réalisé à l'exercice 2 et modifiez le code du serveur de sorte à réaliser un serveur parallèle.
2. Après avoir mis en place le serveur parallèle, exécutez la commande dans un terminal : `ps aux`
   Quel problème constatez-vous ? Quelle solution proposez-vous pour régler le problème ?

## Sérialisation des données

Pour sérialiser un ensemble de données, c'est-à-dire les mettre à la suite, dans une chaîne de caractères il est possible d'utiliser la fonction `memcpy()` en indiquant la bonne adresse de destination en mémoire décalée à chaque étape du nombre de caractères copiés précédemment. Cela peut être fastidieux et surtout source d'erreurs si l'on n'est pas méticuleux.

Nous allons utiliser une autre primitive qui permet d'arriver au même résultat sans avoir à manipuler les adresses en mémoire. La primitive `sprintf()` fonctionne de façon similaire à la primitive `printf()` (permettant de réaliser des affichages à l'écran dans un terminal) mais au lieu d'afficher les données à l'écran cela est écrit en mémoire à l'adresse indiquée en paramètre. Il est également possible d'indiquer la taille de la zone en mémoire afin d'éviter les dépassements en mémoire à l'aide de la primitive `snprintf()`.
Voici ci-dessous un exemple d'utilisation de cette seconde primitive :

```
uint16_t nb1, nb2;
char chaine[5];
char msg[TAILLE_MAX];
…
snprintf(msg, TAILLE_MAX, "%u,%s,%u", nb1, chaine, nb2);
```

Dans l'exemple ci-dessus, deux entiers non signés et une chaine de caractères sont mis bout à bout et stockés en sortie pour produire la chaine de caractères `msg`. La fonction `snprintf()` permet de convertir différents types de données pour cela un certain nombre de symboles précédés de `%` permettent d'indiquer le type de données à convertir.
Voici ci-dessous quelques-uns de ces symboles :

- `%d` : pour un entier signé
- `%u` : pour un entier non signé
- `%c` : pour un caractère
- `%s` : pour une chaine de caractères

Pour définir un protocole d'échange de données, il faut être capable à la réception du message sérialisé de récupérer chacune des données. Il y a deux moyens soit nous connaissons à l'avance la taille de chaque données mais cela n'est pas toujours évident en fonction des types de données convertis, ou alors nous utilisons des symboles particuliers permettant de séparer chaque donnée.
Nous allons utiliser ici la seconde solution. Le symbole de séparation utilisé dans cet exemple est le symbole « , ». Après réception d'un message en sortie de la Socket, nous utilisons la primitive `strtok_r()` qui permet de découper notre message en fonction d'un symbole séparateur indiqué. Nous utiliserons la primitive `sscanf()` qui nous permet de convertir en sens inverse les données converties avec la primitive `snprintf()` en utilisant les mêmes symboles pour indiquer les types désirés.
Voici ci-dessous un exemple d'utilisation permettant de récupérer en sortie de la Socket le message envoyé dans l'exemple précédent :

```
uint16_t nb, nb1, nb2;
char chaine[5];
char *token, *saveptr;
…
recv(sock_fd, msg, TAILLE_MAX, 0);
```

```
// recuperation premier champ message jusqu'au seperateur ","
token=strtok_r(msg, ",", &saveptr);
// stockage du champ dans variable de type entier non signé
sscanf(token, "%u", &nb);
// conversion entier court avec codage local (little ou big endian)
nb1=ntohs(nb);
// recuperation deuxieme champ jusqu'au seperateur ","
token=strtok_r(NULL, ",", &saveptr);
// stockage du champ dans une variable chaine de caracteres
sscanf(token, "%s", chaine);
// recuperation troisieme champ message jusqu'au seperateur ","
token=strtok_r(NULL, ",", &saveptr);
// stockage du champ dans variable de type entier non signé
sscanf(token, "%u", &nb);
// conversion entier court avec codage local (little ou big endian)
nb2=ntohs(nb);
```

## Exercice 4

On reprend l'exercice 2 que l'on adapte de sorte à réaliser les échanges suivants.
On désire réaliser un serveur itératif gérant des connexions de clients en mode connecté à l'aide de Socket TCP. A chaque nouvelle connexion d'un client, le serveur doit lui renvoyer un message contenant son numéro de connexion (défini par un compteur géré et maintenu par le serveur) : « Bienvenue client : num_client » où num_client est à remplacer par la valeur au moment de la connexion du compteur maintenu par le serveur.
De son côté, le client reçoit son numéro de connexion et l'affiche sur sa console.
On considérera que les processus serveur et clients sont exécutés sur la même machine physique et que le serveur peut être contacté sur le numéro de port 55555.
Reprenez le programme réalisé à l'exercice 1 et mettez en place un protocole d'échange d'information entre le client et le serveur en suivant les indications données précédemment.

## Exercice 5

On reprend le programme serveur réalisé dans l'exercice 4 et on désire intégrer la primitive select() afin de permettre la prise en compte d'un ensemble de socket par le serveur.
Modifier le code du programme serveur pour réaliser cela.

## Exercice 6

On désire développer une petite application de tchat entre un groupe d'utilisateurs en mode texte (en mode console). L'approche la plus simple est de choisir une architecture client-serveur, et les programmes clients et serveur seront exécutés sur une même machine. A savoir, de réaliser une partie client qui permet d'envoyer et recevoir des messages, et une partie serveur qui est chargée de collecter tous les messages et de les redistribuer à tous les clients connectés au serveur. Nous ne considérerons qu'un seul fil de discussion sur le serveur, ainsi tous les clients connectés participeront à la même discussion.
Les communications devront être effectuées en utilisant l'API Socket en langage C. Nous utiliserons le protocole de transport TCP afin de s'assurer de la bonne transmission et de l'ordre des messages affichés sur les interfaces clients. Le serveur sera en écoute sur le port 55 555 sur l'adresse locale 127.0.0.1.

Le serveur aura la charge de maintenir la liste des utilisateurs connectés, on supposera qu'au plus 10 utilisateurs sont connectés simultanément au serveur de tchat. Il devra aussi centraliser les messages envoyés par les clients et les renvoyer à tous les utilisateurs ensuite. Les clients n'afficheront les messages qu'une fois que le serveur leur aura envoyé celui-ci (même si c'est le client lui-même qui vient d'envoyer le message).

Pour réaliser cette application, deux types de requêtes seront utilisés encodés via la structure **`msg`** :

- **Type 1 :** Requête d'ajout d'un utilisateur au fil de discussion (envoie du pseudo de l'utilisateur de taille au plus 9 caractères) envoyée par le client au serveur,
- **Type 2 :** Requête d'ajout d'un nouveau message (composé du pseudo et d'une chaîne d'au plus 255 caractères) envoyée par le client au serveur puis du serveur vers les clients.

Complétez les squelettes des programmes client et serveur qui vous sont fournis afin de réaliser un tchat fonctionnel avec l'API Socket en C. Le serveur fonctionne en ***mode itératif*** pour :

3. gérer la connexion de nouveaux clients et mettre à jour la liste des clients affichée sur la console (terminal) du serveur,
4. recevoir un nouveau message de la part d'un client entré au clavier,
5. distribuer un nouveau message reçu à tous les clients (le client émetteur également).

**Remarque :** Vous pourrez utiliser la fonction `select()` pour vous permettre de gérer un ensemble de sockets ouverts côté serveur.