

TRAVAUX PRATIQUES 3

Outil de communication entre processus distants sous Linux

L'objectif de ce TP est de voir les principes importants associés à l'utilisation de l'API Socket sous Linux pour permettre la communication entre processus exécutés sur des machines distantes et communiquant à travers un réseau.

A l'aide de l'API Socket, vous utiliserez le protocole de transport TCP et la couche IP.

Dans ce TP vous aurez à écrire des programmes en langage C pour utiliser l'API Socket sous Linux. Pour cela, vous pouvez utiliser un éditeur de texte comme `kwrite` en exécutant la commande ci-dessous dans le répertoire contenant `prog.c` : `$>kwrite prog.c &`

Rappel : Pour compiler vos programmes en langage C afin de les exécuter vous pouvez utiliser le compilateur `gcc` comme suit : `$> gcc -o ex01 ex01.c`

EXERCICE 1 : Première communication socket

On désire réaliser un serveur itératif gérant des connexions de clients en mode connecté (via TCP) à l'aide de l'API Socket. 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.

Question 1 – En utilisant les squelettes de programmes fournis, donnez les programmes complet associés au serveur et au client.

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.

Question 2 – On désire modifier le mode de fonctionnement du serveur de sorte à le rendre parallèle. Reprenez le programme serveur de la question précédente et modifiez le afin d'obtenir un serveur parallèle.

Question 3 – Quel problème pose éventuellement les processus fils du serveur ?

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 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'envoi 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 rappelle 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 court (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 (resp. 4) 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 stocké 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);
```

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 chaîne de caractères sont mis bout à bout et stockés en sortie pour produire la chaîne 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 chaîne 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 separeteur ","
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 separeteur ","
token=strtok_r(NULL, ",", &saveptr);
// stockage du champ dans une variable chaîne de caracteres
sscanf(token, "%s", chaine);
// recuperation troisieme champ message jusqu'au separeteur ","
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 2 : Communication avancée en mode connecté

Des clients C_i envoient des messages à deux serveurs S1 sur le port PORTS (par exemple de numéro 55558). Les messages sont typés et composés :

- soit d'un opérande x
- soit de deux opérandes x et y

Le serveur S1 traite ces requêtes composées de 1 ou de 2 opérandes. Pour chaque requête, le serveur S1 crée un processus fils P_i . Si le message comporte un seul opérande x , le processus P_i calcule x^3 , puis renvoie le résultat au client avant de se terminer. Si le message comporte deux opérandes x et y , le processus P_i calcule $x^2 + y^2$, puis renvoie le résultat au client avant de se terminer. S'il y a plusieurs messages en attente, le serveur S1 crée autant de processus P_i que de messages afin de traiter les messages en parallèle.

Question 1 – On choisit de faire communiquer les clients et le serveur S1 selon un mode de communication TCP.

Donnez les programmes du client et du serveur S1 en faisant apparaître clairement l'utilisation des primitives de communication. De plus, en utilisant les indications données précédemment, vous sérialiserez la structure des messages envoyés aux serveurs.