UNIVERSITÉ PARIS SUD
IUT D'ORSAY

**DUT Informatique**
**M3101 Système S3**
**2015 / 2016**

**Travaux Dirigés n° 3 : Sockets stream**

*Objectifs : comprendre les principes et les mécanismes de communication par sockets stream, être capable de réaliser des systèmes client-serveur sur ce mode de communication.*

# 1 Communication par sockets

Les *sockets* sont des interfaces de communication bidirectionnelles entre différents processus, qu'ils appartiennent à un même système ou non. Dans le cas des sockets *datagramme*, cette interface s'apparente à une boîte aux lettres : on envoie des messages complets à n'importe qui pourvu qu'on en connaisse l'adresse. Dans le cas des sockets *stream*, cette interface s'apparente à un téléphone : les informations sont transmises entre deux parties en continu après établissement d'une connexion. Ce type de communication s'utilise entre un serveur (l'appelé) et un client (l'appelant), les rôles ne sont donc pas symétriques.

Les sockets sont représentés par des descripteurs de fichiers et les primitives génériques `read()`, `write()` et `close()` sont utilisables. Dans le cas des sockets stream, le serveur utilise une socket dite d'*écoute* dédiée à la mise en place de connexions, et une socket par connexion, dite socket de *service*, dédiée à l'échange d'information proprement dit. Le schéma général de communication par sockets stream est le suivant :

| Serveur (appelé) | Client (appelant) |
|---|---|
| Créer les sockets | |
| `secoute = socket(...)` | `sclient = socket(...)` |
| Attacher la socket à une adresse<br>`bind(secoute,...)` | |
| Mettre la socket en mode écoute<br>`listen(secoute,...)` | |
| Accepter une demande de connexion<br>`sservice = accept(secoute,...)` | Demander une connexion<br>`connect(sclient,...)` |
| Communiquer | |
| `read(sservice,...)`<br>`write(sservice,...)` | `write(sclient,...)`<br>`read(sclient,...)` |
| Fermer la connexion dans un ou deux sens | |
| `shutdown(sservice,...)` | `shutdown(sclient,...)` |
| Fermer la connexion et la socket associée | |
| `close(sservice)` | `close(sclient)` |
| Fermer la socket d'écoute<br>`close(secoute)` | |

Les primitives de communication avec les sockets ne sont pas nombreuses, mais elles sont hélas compliquées par la nécessité de connaître d'autres fonctions utilitaires et la structure de données de l'adresse d'une socket.

# 2 Création d'une socket

```
#include <sys/socket.h>
int socket(
  int domaine,   /* AF_UNIX, AF_INET, AF_INET6 */
  int type,      /* SOCK_DGRAM, SOCK_STREAM */
  int protocole  /* 0: protocole par defaut */);
```

La primitive `socket()` demande au système la création d'une socket et renvoie son descripteur en cas de succès ou $-1$ en cas d'échec. Le domaine définit l'ensemble de sockets avec lesquelles une communication pourra être établie :

— `AF_LOCAL` ou `AF_UNIX` pour la communication entre les processus sur le même système.

— `AF_INET` ou `AF_INET6` pour la communication via les réseaux Internet.

Le type définit la nature de la socket : datagramme avec `SOCK_DGRAM` et stream avec `SOCK_STREAM`. Dans ce TD nous allons toujours utiliser les sockets stream (`SOCK_STREAM`) qui communiquent dans les réseaux IPv4 (`AF_INET`).

# 3 Attachement d'une socket à une adresse

```
#include <sys/socket.h>
int bind(
  int descripteur,            /* descripteur de la socket */
  struct sockaddr * adresse,  /* pointeur vers l'adresse d'attachement */
  socklen_t longueur_adresse  /* longueur de l'adresse en octets */);
```

La primitive `bind()` effectue l'attachement à l'adresse pointée par le paramètre `adresse` de la socket `descripteur` et la rend ainsi accessible depuis des processus ne connaissant pas le descripteur (mais l'adresse). Elle renvoie 0 en cas de succès ou $-1$ en cas d'échec. Dans le cas du domaine `AF_INET`, cette adresse a la forme suivante :

```
#include <netinet/in.h>
struct in_addr {            /* adresse internet d'une machine */
  uint32_t s_addr;          /* - adresse IPv4 sur 32 bits */ };

struct sockaddr_in {        /* adresse internet d'une socket */
  short sin_family;         /* - AF_INET */
  u_short sin_port;         /* - numéro de port associé */
  struct in_addr sin_addr;  /* - voir ci-dessus */
  char sin_zero[8];         /* - 8 octets nuls */ };
```

Certaines valeurs « joker » sont possibles : `INADDR_ANY` pour `sinaddr.s_addr` attache la socket à toutes les adresses possibles de la machine. La valeur 0 pour `sin_port` laisse le système décider du numéro de port. Par exemple, pour un client d'une application client-serveur le choix de numéro de port n'a aucune importance.

Une nouvelle difficulté à relever est la traduction des informations du système hôte (*host*) vers des données dans le format du réseau (*network*) pour remplir les structures de données ci-dessus. En effet, des différents systèmes peuvent utiliser des représentations très différentes de données numériques (nombre de bits différent, octet de poids fort à un endroit différent, complément à un ou à deux...). Quand ces données sont transmises par le réseau, l'utilisation d'un format commun s'impose. Pour cela existent quatre primitives de traduction :

| | |
|---|---|
| `ntohs()` — network-to-host 16 bit | `htons()` — host-to-network 16 bit |
| `ntohl()` — network-to-host 32 bit | `htonl()` — host-to-network 32 bit |

Ainsi connaissant le numéro de port, `port`, et ayant auparavant préparé une structure `adresse` de type **`struct`** `sockaddr_in`, nous pourrions la remplir de la manière suivante :

```
#include <arpa/inet.h> /* pour htonl(), htons() ou inet_addr() */
...
struct sockaddr_in adresse = {0}; /* initialisation à zéro */
adresse.sin_family = AF_INET;
adresse.sin_addr.s_addr = htonl(INADDR_ANY);
adresse.sin_port = htons(port);
```

Si on ne choisit pas `INADDR_ANY`, `adresse.sin_addr.s_addr` doit contenir l'adresse IP de la machine sous forme entière. Par exemple, pour l'adresse 130.57.12.30 on utilisera `htonl(`$130 \times 2^{24} + 57 \times 2^{16} + 12 \times 2^8 + 30$`)`. Plus simplement, on peut aussi utiliser la chaîne de caractères de l'adresse IP de la manière suivante : `inet_addr("130.57.12.30")`. Si on ne connaît que le nom de la machine, on peut utiliser les instructions suivantes :

```
#include <netdb.h> /* pour hostent et gethostbyname() */
...
struct hostent * host;
host = gethostbyname("nom_machine.iut-orsay.fr");
memcpy((void *)&adresse.sin_addr, (void *)host->h_addr, host->h_length);
adresse.sin_family = host->h_addrtype;
```

# 4 Écoute sur une socket

```
#include <sys/socket.h>
int listen(
  int descripteur,      /* descripteur de la socket d'ecoute */
  struct nb_pendantes   /* nombre maximum de connexions pendantes */);
```

La primitive `listen()` permet à un processus serveur de déclarer au système d'exploitation qu'il est prêt à accepter les demandes de connexion arrivant à sa socket d'écoute (désignée par `descripteur`). Il peut y avoir des délais entre l'arrivée de demandes de connexion et l'acceptation de ces connexions (primitive `accept()`, voir section 6). Les connexions en attente d'acceptation sont dites *pendantes*. Le paramètre `nb_pendantes` permet de préciser le nombre maximum de telles connexions. Cette primitive renvoie 0 en cas de succès ou $-1$ en cas d'échec.

# 5 Demande de connexion

```
#include <sys/socket.h>
int connect(
  int descripteur,           /* descripteur de socket de l'appelant */
  struct sockaddr * adresse, /* pointeur vers l'adresse de l'appelé */
  int longueur_adresse       /* longueur de l'adresse de l'appelé */);
```

La primitive `connect()` permet à un processus client de demander une connexion entre sa socket désignée par `descripteur` et la socket d'un serveur dont l'adresse est pointée par `adresse` et a pour longueur `longueur_adresse`. L'appel à cette primitive est bloquant : le processus est bloqué jusqu'à ce que la communication s'établisse (elle renvoie alors 0) ou échoue (elle renvoie alors $-1$).

# 6 Acceptation d'une connexion

```
#include <sys/socket.h>
int accept(
  int descripteur,           /* descripteur de la socket d'écoute */
  struct sockaddr * adresse, /* pointeur vers l'adresse de l'appelant */
  int * longueur_adresse     /* pointeur vers la longueur de l'adresse */);
```

La primitive `accept()` permet à un processus serveur d'accepter une demande de connexion arrivée sur sa socket d'écoute (désignée par `descripteur`), de la part d'un client ayant invoqué la primitive `connect()`. Au moment de l'appel à cette primitive, le champ `longueur_adresse` pointe sur une donnée indiquant la taille de l'espace alloué à l'adresse `adresse`, il faut donc l'initialiser (typiquement à `sizeof(struct sockaddr)`). Au retour de cette fonction, les champs `adresse` et `longueur_adresse` permettront de connaître les coordonnées du client dont la demande a été acceptée. Si le serveur ne veut pas les savoir, les deux paramètres peuvent être `NULL`. Cette primitive renvoie le descripteur vers la socket de service dédiée à cette connexion, ou $-1$ en cas d'échec.

# 7 Envoi et réception de données

Les primitives `read()` et `write()` peuvent être utilisées pour l'échange de données via les sockets. Le comportement est similaire au cas des tubes. La primitive `read()` est bloquante : en absence de données reçues le processus appelant `read()` se met en attente de nouvelles données. Si la socket locale est fermée en lecture ou la socket distante est fermée en écriture (voir la section suivante), alors `read()` renvoie 0 en signalant ainsi que la réception de données n'est plus possible.

La primitive `write()` est également bloquante : si le tampon d'envoi (attribué à la connexion par le système) est plein, le processus appelant `write()` se met en attente jusqu'à ce que le processus distant acquitte la réception de données transmises. Si la socket locale est fermée en écriture ou la socket distante est fermée en lecture, alors le processus appelant reçoit le signal `SIGPIPE` et `write()` renvoie $-1$.

En dehors de `read()` et `write()` il y a plusieurs primitives de réception et d'envoi de données spécifiques aux sockets : `recv()`, `recvfrom()`, `recvmsg()` et `send()`, `sendto()`, `sendmsg()`. Ces primitives acceptent des paramètres supplémentaires contrôlant la transmission. Par exemple, pour les sockets datagramme où il n'y a pas d'établissement de connexion, les appels `sendto()` et `recvfrom()` permettent de passer l'adresse de la socket distante. Voir les pages de `man` correspondantes pour plus d'information.

# 8 Terminaison de connexion

```
#include <sys/socket.h>
int shutdown(
  int descripteur, /* descripteur de la socket */
  int sens         /* 0 : lecture, 1 : écriture, 2 : les deux */);
```

La primitive `shutdown()` ferme la connexion établie par la socket `descripteur` dans le sens défini par le paramètre `sens`. Si `sens` vaut `SHUT_RD`, la connexion est fermée en lecture, si `sens` vaut `SHUT_WR`, elle sera fermée en écriture, et elle sera enfin fermée dans les deux sens pour `SHUT_RDWR`.

# 9 Exercices

**Exercice 1 : Client/serveur simple.** Construisez un serveur qui à la réception d'un caractère ASCII renvoie le même caractère s'il ne s'agît pas d'une lettre de l'alphabet et de la majuscule correspondante dans le cas contraire. Les communications se feront par les sockets `SOCK_STREAM` et dans le domaine `AF_INET`. Le serveur ne pourra traiter qu'une seule communication à la fois et ne pourra attendre que 5 communications au maximum. Écrivez de même le programme d'un client.

**Exercice 2 : Client/serveur évolué.** On se propose d'écrire les codes d'un client et d'un micro-serveur utilisant les sockets de type `SOCK_STREAM` dans le domaine `AF_INET` pour établir les communications avec un nombre quelconque de clients. Le serveur s'exécute sur une machine de nom réseau `tests.u-psud.fr` et utilise pour le service qu'il rend le port 5015. Les clients peuvent s'exécuter sur n'importe quelle machine du réseau.

La structure du serveur est basée sur le multitâche. Un premier processus est lancé pour créer la socket d'écoute et faire fonction de superviseur des demandes de connexions. Lorsqu'une demande se présente, il crée un processus « communication » pour accepter l'appel et gérer la communication jusqu'à sa libération. Le premier processus est prévenu de cette terminaison par le signal `SIGCHLD`. On limitera le nombre de clients simultanés à 5.

Le squelette du serveur est le suivant :

```
Mettre en place un traitant pour le signal SIGCHLD (primitive signal())
Créer une socket d'écoute (primitive socket())
Attacher la socket à une adresse (primitive bind())
Attendre des demandes de communications (primitive listen())
tant_que vrai faire
| si demande de connexion (primitive accept())
| | Créer un processus communication (c)
| | (c) tant_que pas_fini faire
| | (c) | Lecture des requêtes (primitive read())
| | (c) | Envoi des réponses (primitive write())
| | (c) Fermeture des flux (primitive shutdown())
| | (c) Destruction de la socket de service (primitive close())
```

Le squelette du client est le suivant :

```
Créer une socket (primitive socket())
tant_que non_connecté faire
| Tenter de se connecter au serveur (primitive connect())
tant_que pas_fini faire
| Envoi des requêtes (primitive write())
| Lecture des réponses (primitive read())
| Traitement des réponses
Fermeture des flux (primitive shutdown())
Destruction de la socket (primitive close())
```

| Pourquoi est-il nécessaire de traiter le signal `SIGCHLD` dans une telle application ? |
|---|
| |

Implémentez le serveur et le client.

# 10 Entraînement : exercice corrigé

**Calculateur.** L'objectif de cet exercice est de réaliser un serveur de calcul inspiré des calculatrices HP à notation polonaise inversée (on se limitera au support des 4 opérations de base, +, -, ×, /). On rappelle que ce type de calculatrice utilise une pile pour effectuer les calculs. Le client se connecte au serveur puis envoie des commandes. Une commande est :

— soit un nombre ;
— soit un opérateur.

Le protocole (pour chaque envoi de commande) est le suivant :

1. le client envoie la commande terminée par un caractère de retour à la ligne ;
2. le serveur traite la commande :
   — nombre : le nombre est empilé,
   — opérateur : les opérandes de l'opérateur sont dépilés, puis le résultat de l'application de l'opérateur à ces opérandes est empilé.
3. le contenu du niveau 1 de la pile est retourné au client.

À la fermeture du client, le serveur repasse en mode acceptation de client. On suppose pour l'implémentation que les opérations de base sur les piles sont disponibles dans une bibliothèque.

**Correction.** L'implémentation est principalement la simple traduction du schéma des notes de cours à l'aide de primitives C. Le client envoie son chiffre ou sa commande et reçoit un message de réponse de la part du serveur tant qu'il ne souhaite pas s'arrêter (envoi de la commande `f`). Le serveur fait une boucle infinie autour de l'acceptation d'une connexion et du traitement des messages qu'il reçoit. Ainsi lorsqu'un client se déconnecte, c'est le suivant qui sera servi par le serveur.

On supposera que le client prend en argument sur la ligne de commande l'adresse IP et le port du serveur. Le serveur prend simplement en argument le port sur lequel il doit être attaché. Des codes possibles sont les suivants, on présente tout d'abord le code d'un client puis celui du serveur. Les tests d'échec des primitives ne sont pas faits, ils sont à rajouter dans tout travail sérieux.

```c
/* CLIENT. Donner l'adresse IP et le port du serveur en arguments, */
/* par exemple "client 127.0.0.1 5000", lancer en dernier. */

#include <stdio.h>            /* fichiers d'en-tête classiques */
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

#include <sys/socket.h>       /* fichiers d'en-tête "réseau" */
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define BUFFER_SIZE  1024     /* Taille maximum des messages */

int main(int argc, char *argv[]) {
  int  sclient;                     /* Descripteur de la socket client  */
  char commande[BUFFER_SIZE] = {0},  /* Commande a envoyer */
       message[BUFFER_SIZE] = {0};   /* Message reçu */
  struct sockaddr_in saddr = {0};    /* Adresse du serveur */

  /* 1. On crée la socket du client. */
  sclient = socket(AF_INET,SOCK_STREAM,0);

  /* 4.1. On prépare l'adresse du serveur. */
  saddr.sin_family = AF_INET;
  saddr.sin_port = htons(atoi(argv[2]));
  saddr.sin_addr.s_addr = inet_addr(argv[1]);

  /* 4.2. On demande une connexion au serveur, tant qu'on y arrive pas. */
  while (connect(sclient,(struct sockaddr *)&saddr,sizeof(saddr)) == -1) ;

  /* 6. On communique. */
  while(commande[0] != 'f') {
    printf("Entrez un nombre, un opérateur (+ ou -) ou f pour sortir.\n");
    scanf(" %1023s",commande);                  /* Lecture commande clavier */
    write(sclient,commande,strlen(commande)+1);  /* Envoi de la commande */
    read(sclient,message,BUFFER_SIZE-1);         /* Réception du message */
    printf("Message reçu : %s\n",message);       /* Affichage */
  }
  /* 7 et 8. On termine et libère les ressources. */
  shutdown(sclient,SHUT_RDWR);
  close(sclient);
  return 0;
}
```

```c
/* SERVEUR. Donner le port d'attachement du processus en argument,  */
/* par exemple "serveur 5000", lancer ce programme en premier. */
#include <stdio.h>                  /* fichiers d'en-tête classiques */
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>
#include "pile.h"                   /* Bibliothèque de pile */
#include <sys/socket.h>             /* fichiers d'en-tête "réseau" */
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define BUFFER_SIZE  1024           /* Taille maximum des messages */

int main(int argc, char *argv[]) {
  int  secoute,                     /* Descripteur socket d'écoute */
       sservice;                    /* Descripteur socket de service */
  socklen_t caddrlen;               /* Longueur de l'adresse du client  */
  float resultat;                   /* Résultat pour les calculs */
  char message[BUFFER_SIZE] = {0};  /* Message reçu ou a envoyer */
  struct sockaddr_in saddr = {0},   /* Mon adresse serveur */
                     caddr = {0};   /* Adresse du client */
  pile p;                           /* La pile */

  /* 1. On crée la socket d'écoute. */
  secoute = socket(AF_INET,SOCK_STREAM,0);
  /* 2.1 On prépare l'adresse du serveur. */
  saddr.sin_family = AF_INET;
  saddr.sin_addr.s_addr = htonl(INADDR_ANY);
  saddr.sin_port = htons(atoi(argv[1]));
  /* 2.2. On attache la socket a l'adresse du serveur. */
  bind(secoute,(struct sockaddr *)&saddr,sizeof(saddr));
  /* 3. On attend les demandes de connexion. */
  listen(secoute,5);

  while (1) {
    /* 5. On accepte une connexion. */
    caddrlen = sizeof(caddr);       /* Initialisation de caddrlen */
    sservice = accept(secoute,(struct sockaddr *)&caddr,&caddrlen);
    /* 6. On communique. */
    init_pile(&p);                  /* On initialise la pile */
    while (message[0] != 'f') {
      memset(message,'\0',BUFFER_SIZE);  /* Mise a 0 du message */
      read(sservice,message,BUFFER_SIZE);
      switch(message[0]) {         /* Traitement du message (très basique !) */
        case '+' : resultat = depiler(&p)+depiler(&p); /* Cas opérateur + */
                   empiler(&p,resultat);
                   sprintf(message,"%f",resultat);
                   write(sservice,&message,strlen(message)+1);
                   break;
        case '-' : resultat = depiler(&p)-depiler(&p); /* Cas opérateur - */
                   empiler(&p,resultat);
                   sprintf(message,"%f",resultat);
                   write(sservice,&message,strlen(message)+1);
                   break;
        case 'f' : break;                           /* Cas 'f' : fin */
        default  : empiler(&p,atof(message));       /* Cas nombre */
                   write(sservice,"Empilement OK",14);
      }
    }
    /* 7 et 8. Terminaison et libération des ressources. */
    shutdown(sservice,SHUT_RDWR);
    close(sservice);
  }
  close(secoute);
  return 0;
}
```