iUT ORSAY
Département Informatique

DUT Informatique
Module Système S4 C
2009 / 2010

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 Notes de cours

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 *stream*, cette interface s'apparente à un téléphone : les informations sont transmises avec fiabilité 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.
La principale caractéristique, en comparaison des sockets datagramme, est l'utilisation par le serveur d'au moins deux sockets : une dite d'*écoute* dédiée à la mise en place de connexions, et une autre dite 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)
1	Préparation de la connexion `secoute = socket(...)`	`sclient = socket(...)`
2	Liaison `bind(secoute,...)`	
3	Écoute sur la socket `listen(secoute,...)`	
4		Demande de connexion `connect(sclient,...)`
5	Acceptation de la connexion `sservice = accept(secoute,...)`	

6	Communication
	`write(sservice,...)`
	`read(sservice,...)`
		`write(sclient,...)`
		`read(sclient,...)`

7	Terminaison `shutdown(sservice,...)`	`shutdown(sclient,...)`

8	Libération des ressources
	`close(sservice)`
	`close(secoute)`
		`close(sclient)`

Il y a donc simplement 9 primitives fondamentales pour communiquer avec les sockets stream. Trois d'entre elles, `socket`, `bind` et `close`, sont communes à la communication par sockets datagramme (qui est supposée connue). L'unique différence est que nous utiliserons le type `SOCK_STREAM` dans le champ `type` de la primitive `socket`.

1.1 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 connections arrivant à sa socket d'écoute (désignée par `descripteur`). Il peut y avoir des délais entre l'arrivée de demandes de connexions et l'acceptation de ces connexions (primitive `accept`, voir section 1.2). Les connections en attente d'acceptation sont dites *pendantes*. Le paramètre `nb_pendantes` permet de préciser le nombre maximum de telles connexions. Cette primitive retourne 0 en cas de succès ou -1 en cas d'échec.

1.2 Acceptation d'une connexion

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

La primitive `accept` permet à un processus serveur d'accepter une demande de connexion arrivée sur sa socket d'écoute (désignée par le paramètre `descripteur`), de la part d'un client ayant invoqué la primitive `connect` (voir section 1.3). 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. Cette primitive retourne le descripteur vers la socket de service dédiée à cette connection, ou -1 en cas d'échec.

1.3 Demande de connexion

```
#include <sys/socket.h>
int connect(
    int descripteur,            /* Descripteur de socket de l'appelant */
    struct sockaddr * adresse,  /* Pointeur vers adresse de la socket */
                                /* de l'appele */
    int longueur_adresse        /* Longueur de l'adresse de la socket */
                                /* de l'appele */
);
```

La primitive `connect` permet à un processus client de demander une connection entre sa socket désignée par le paramètre `descripteur` et la socket (d'un serveur) dont l'adresse est pointée par `adresse` et a pour longueur `longueur_adresse` et . 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).

1.4 Réception d'information

```
#include <unistd.h>
ssize_t read(
    int descripteur, /* Descripteur de la socket */
    void * message,  /* Adresse de reception */
    size_t longueur  /* Longueur maximum du message */
);
```

La primitive `read` est simplement la primitive standard de lecture. Dans le cas d'une socket, on lira sur la socket désignée par `descripteur` au plus `longueur` caractères contenus dans le tampon de réception de la socket et on les écrira à l'adresse `message`. Cette primitive retourne le nombre de caractères effectivement lus.

1.5 Envoi d'information

```
#include <unistd.h>
ssize_t write(
    int descripteur, /* Descripteur de la socket */
    void * message,  /* Adresse de reception */
    size_t longueur  /* Longueur du message */
);
```

La primitive `write` est simplement la primitive standard d'écriture. Dans le cas d'une socket, on écrira sur la socket désignée par `descripteur` les `longueur` caractères se trouvant à l'adresse `message`. Cette primitive retourne le nombre de caractères écrits.

1.6 Terminaison

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

La primitive `shutdown` ferme la connexion établie par la socket désignée par `descripteur` dans le sens défini par la paramètre `sens`. Si `sens` vaut 0, la socket est fermée en lecture, si `sens` vaut 1, elle sera fermée en écriture, et elle sera enfin fermée dans les deux sens pour 2.

2 Exercices

2.1 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 donaine `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.

2.2 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.iut-orsay.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 connections. Lorsqu'une demande se présente, il crée un processus «communication» pour accepter l'appel et gérer la communication jusquà 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 connection (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 seveur (primitive connect)
tant_que pas_fini faire
| Envoi des requêtes (primitive read)
| Lecture des réponses (primitive write)
| Traitement des réponses
Fermeture des flux (primitive shutdown)
Destruction de la socket (primitive close)
```

1. Pourquoi est-il nécessaire de traiter le signal `SIGCHLD` dans une telle application ?
2. Implantez serveur et client.

3 Entraînement : exercice corrigé

3.1 Énoncé : 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'implantation que les opérations de base sur les piles sont disponibles dans une bibliothèque.

3.2 Correction (essayez d'abord ! ! !)

L'implantation 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 connection 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-tete classiques */
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>

#include <sys/socket.h>         /* fichiers d'en-tete "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],   /* Commande a envoyer */
       message[BUFFER_SIZE];    /* Message recu */
  struct sockaddr_in saddr;     /* Adresse du serveur */

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

  /* 4.1. On prepare l'adresse du serveur. */
  memset((char *)&saddr,'0',sizeof(struct sockaddr_in));
  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 connection au serveur, tant qu'on y arrive pas. */
  while (connect(sclient,(struct sockaddr *)&saddr,sizeof(saddr)) == -1) ;

  /* 6. On communique. */
  commande[0] = 'i';
  while(commande[0] != 'f') {
    printf("Entrez un nombre, un operateur (+ ou -) ou f pour sortir.\n");
    scanf(" %s",commande);                      /* Lecture commande clavier */
    write(sclient,commande,strlen(commande)+1); /* Envoi de la commande */
    read(sclient,message,BUFFER_SIZE);          /* Reception du message */
    printf("Message recu : %s\n",message);      /* Affichage */
  }
  /* 7 et 8. On termine et libere les ressources. */
  shutdown(sclient,2);
  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-tete classiques */
#include <stdlib.h>
#include <sys/types.h>
#include <string.h>
#include "pile.h"                   /* Bibliotheque de pile */
#include <sys/socket.h>             /* fichiers d'en-tete "reseau" */
#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'ecoute */
       sservice,                    /* Descripteur socket de service */
       caddrlen;                    /* Longueur de l'adresse du client  */
  float resultat;                   /* Resultat pour les calculs */
  char message[BUFFER_SIZE];        /* Message recu ou a envoyer */
  struct sockaddr_in saddr,         /* Mon adresse serveur */
                     caddr;         /* Adresse du client */
  pile p;                           /* La pile */

  /* 1. On cree la socket d'ecoute. */
  secoute = socket(AF_INET,SOCK_STREAM,0);
  /* 2.1 On prepare l'adresse du serveur. */
  memset((char *)&saddr,'0',sizeof(struct sockaddr_in));
  saddr.sin_family = AF_INET;
  saddr.sin_addr.s_addr = 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. */
    message[0] = 'i';
    init_pile(&p);                  /* On initialise la pile */
    while (message[0] != 'f') {
      bzero(message,BUFFER_SIZE);   /* Mise a 0 du message */
      read(sservice,message,BUFFER_SIZE);
      switch(message[0]) {          /* Traitement du message (tres basique !) */
        case '+' : resultat = depiler(&p)+depiler(&p); /* Cas operateur + */
                   empiler(&p,resultat);
                   sprintf(message,"%f",resultat);
                   write(sservice,&message,strlen(message)+1);
                   break;
        case '-' : resultat = depiler(&p)-depiler(&p); /* Cas operateur - */
                   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 liberation des ressources. */
    shutdown(sservice,2);
    close(sservice);
  }
  close(secoute);
  return 0;
}
```