Utilisation de l'interface *socket* pour des communications TCP/UDP de type Client/Serveur*

Hector Briceño, Yves Caniou
Université Lyon 1

9 novembre 2006

Version 0.02

Objectifs.

- Utilisation de l'interface sockets,
- Développement d'un client HTTP,
- Développement et études d'une application Client/Serveur en mode connecté,
- Développement et études d'application Client/Serveur en mode non connecté,
- Développement d'un serveur traitant des requêtes de plusieurs clients (mode concurrent)

Pré-requis.

Programmation en langage C, Système, Notions sur TCP/UDP

Remarques.

- Tous les programmes seront écrits en langage C.
- Ce TP se déroule sur 2 séances de 3 heures.
- Ce document est accessible à l'URL `http://graal.ens-lyon.fr/~ycaniou/teaching/CCIR5/TP_1_2.pdf`.

1 Introduction et rappels (pour vous aider !)

1.1 Conseils

Parmis les conseils que nous pouvons vous donner et les choses que vous pouvez lire, on peut noter :

- À propos des structures en C,
- Consultez l'interface des sockets présentée en annexe,
- Utilisez les `manpages` ! → `man ip`, `man socket`, `man string` (`man strcpy`...)
- Le site `http://www.commentcamarche.net` pour un rappel sur le protocole HTTP,

*Merci à Olivier Glück pour les sources de ses TPs donnés en M2 SIR

– Bibliographie : J.M. Rifflet - « La communication sous UNIX »,
– **PENSEZ à sauvegarder vos fichiers entre chaque scéance**.

1.2 Rappels

1.2.1 Les pointeurs

Pour résumer, on parle de pointeur à propos d'une variable qui désigne l'adresse par exemple d'une autre variable. Ainsi, si on considère le programme suivant[1],

```
#include <stdio.h>

int main(int argc, char** argv)
{

  int a ;  /* Déclaration de la variable a */
  int *b ; /* Déclaration de la variable b */

  a=5 ;    /* Affectation de a */
  b=&a ;   /* Affectation de b*/

  printf("a=%d ; b=%x ; b=%d\n",a,b,*b) ;

  *b=9 ;   /* Changement du contenu de b */

  printf("a=%d ; b=%x ; b=%d\n",a,b,*b) ;
}
```

On voit que `a` est un entier, que `b` est une adresse sur un entier ou sur un tableau d'entier. Lors de l'affectation de `b`, `b` prends pour valeur l'adresse de `a`. On dit que `b` *pointe* vers `a`. Ceci est schématisé à la figure 1.

Un exemple d'exécution du programme donne :

```
FouDuBassan:~/Cours/0506/CCIR5/TP_1_2/Source>./a.out
a=5 ; b=bfd3c270 ; b=5
a=9 ; b=bfd3c270 ; b=9
```

Questions :

☞ Décrivez et expliquez ces résultats

1.3 Les chaînes de caractères

1.3.1 Définition

Une chaîne de caractère est un vecteur de caractères. Puisque nous devons pouvoir avoir accès au caractère 2 en prenant la position du caractère 1 en lui ajoutant une «case», la variable «contenant» la chaîne de caractère doit donc être la position du premier caractère, c'est-à-dire son adresse ! Donc, le type d'une chaîne est un pointeur sur caractère.

[1] Disponible à l'URL http://graal.ens-lyon.fr/~ycaniou/teaching/CCIR5/exemplePointeurs.c

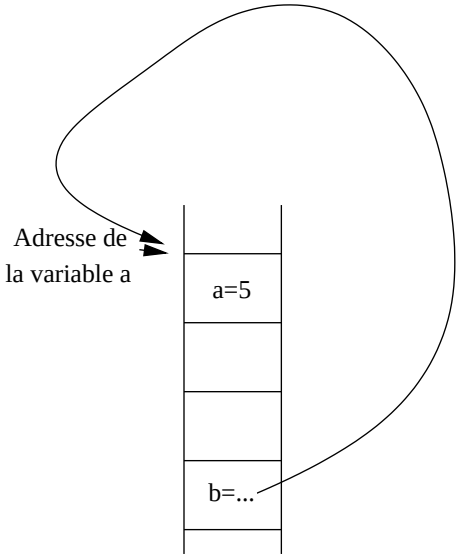


Figure 1: Comme b contient la valeur de l'adresse de a, on dit que b «pointe» vers a

Remarque :

- Notons que le premier caractère est le caractère 0, puisqu'on prend la position du premier caractère auquel on ajoute le numéro du caractère (ici 0) pour avoir la position du caractère désiré...

Questions :

- ☞ Faites un schéma de ce qu'il se passe en mémoire pour la chaîne de caractères "Tipi" et une variable a qui pointe dessus. Quel est le dernier caractère ?
- ☞ Quelle(s) est/sont la/les différence(s) entre un pointeur sur un caractère et un pointeur sur une chaîne de caractères ?

1.3.2 Utilisation

Voilà un exemple d'utilisation[2] où l'on voit deux façons d'utiliser les chaînes de caractères. Pour la première, la réservation mémoire est statique, pour la seconde, la réservation de la mémoire est dynamique.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char** argv)
{
  char chaine1[500] ;
  char *chaine2 ;

  strcpy(chaine1,"Bonjour !\n\nComment vas ?\n") ;
```

[2] Disponible à l'URL http://graal.ens-lyon.fr/~ycaniou/teaching/CCIR5/exempleChainesCaracteres.c

```
  chaine2=(char*)(malloc(sizeof(char)*100)) ;
  sprintf(chaine2,"%s : pas mal\n",chaine1) ;

  printf("La première chaine vaut\n%s",chaine1) ;

  printf("--------------- Césure --------------\n") ;

  printf("La deuxième chaine vaut\n%s",chaine2) ;

  printf("--------------- Césure --------------\n") ;

  printf("Pour chaine1\n"
         "Valeur retournée par sizeof : %d\n"
         "Valeur retournée par strlen : %d\n"
         ,sizeof(chaine1)
         ,strlen(chaine1)) ;

  printf("--------------- Césure --------------\n") ;

  printf("Le caractère 0 est %c, et aussi %c, "
         "le caractère 14 est %c ou encore %c\n\n"
         ,chaine1[0]
         ,*chaine1
         ,chaine1[14]
         ,*(chaine1+14)) ;
}
```

Questions :

- ☞ Expliquez ce que fait le programme (notamment combien de mémoire est réservée pour la chaine de caractère `chaine2`) et exécutez le.
- ☞ Comment expliquez-vous la fonction `sizeof()` ?
- ☞ Expliquez à l'aide d'un petit schéma le dernier `printf()` effectué.
- ☞ Peut-on accéder au caractère 874 de `chaine1` ou `chaine2` en lecture ? En écriture ? Expliquez.

1.4 Les structures

Il est possible de regrouper des types (entiers, flottants, caractères ou types complexes comme les structures que nous présentons maintenant) dans des structures. L'un des exemples les plus parlants est le type d'un couple d'entiers. On peut définir le type `couple_entier` et l'utiliser de la façon suivanteDisponible à l'URL http://graal.ens-lyon.fr/~ycaniou/teaching/CCIR5/exempleStructures.c :

```
#include <stdio.h>

typedef struct couple_entier
{
  int a ;
  int b ;
} couple_entier ;

int main(int argc, char** argv)
```

```
{
  float a ;
  int b ;
  int c ;
  couple_entier couple1 ;

  a=5.5 ;
  b=9 ;

  couple1.a=843 ;
  couple1.b=-65 ;

  printf("Les valeurs de a et b sont %.2f, %d\n\n",a,b) ;

  printf("Les champs 1 et 2 du couple sont : %d et %d\n\n"
         ,couple1.a
         ,couple1.b) ;

  printf("La taille d'une structure est de %d octets\n",sizeof(couple1)) ;
}
```

Questions :

- ☞ Que remarquez vous ? Où est définie la structure ?
- ☞ Comment a-t-on accès aux différents *champs* d'une structure ?

1.5 Un peu plus loin avec les pointeurs et les structures

Nous avons vu dans la section précédente comment accéder aux champs d'une structure. Regardons maintenant[3] comment on peut y accéder via l'adresse de la structure.

```
#include <stdio.h>

typedef struct couple_entier
{
  int a ;
  int b ;
} couple_entier ;

int main(int argc, char** argv)
{
  couple_entier couple1 ;
  couple_entier *ptr_couple ;

  ptr_couple=&couple1 ;

  couple1.a=843 ;
  couple1.b=-65 ;

  printf("Les valeurs de a et b sont %d, %d\n\n"
         ,(*ptr_couple).a
         ,(*ptr_couple).b) ;
```

[3] Code Disponible à l'URL http://graal.ens-lyon.fr/~ycaniou/teaching/CCIR5/exempleStructuresPointeurs.c

```
  printf("Les valeurs de a et b sont %d, %d\n\n"
         ,ptr_couple->a
         ,ptr_couple->b) ;

}
```

Remarques :

– Notez que la seconde façon d'accéder aux champs est la plus couramment utilisée.

1.6 On discute des paramètres de main() ?

Tout simplement, `argc` est un entier et contient le nombre de paramètres donnés en ligne de commande. `argv` est un pointeur sur un pointeur sur caractère.

☞ Quelles sont les différentes possibilités pour `argv` ?

Procédons par ordre : nous avons déjà vu qu'un `char*` est l'adresse d'une succession de caractères (et éventuellement d'un seul). C'est donc un pointeur sur un caractère ou sur une chaîne de caractères.

`argv` est l'adresse d'un `char*`. C'est donc l'adresse d'une succession d'adresses (éventuellement une seule). `argv` est un vecteur d'adresses, où chaque case (au nombre de `argc`) contient l'adresse d'une succession de caractères (éventuellement un seul). `argv` est donc un vecteur de chaînes de caractères qui sont les arguments donnés au programme !

Au passage, on note que les arguments donnés en ligne de commande sont donc stockés en tant que chaînes de caractères.

Histoire de vous forcer à relire ce qui précède, une chaîne de caractères étant un vecteur de caractères, `argv` est donc un vecteur de vecteurs :)

☞ Faites un programme qui affiche les arguments, de 0 à `argc`, passés en ligne de commande. Que remarquez-vous ?

1.7 À propos de la compilation

On se bornera ici à la compilation de programmes compris dans un seul document source. Pour compiler un code `nomDuCode.c`, taper simplement dans une fenêtre Xterm, alors que vous êtes dans le répertoire contenant le source : `gcc nomDuCode.c`. Ceci génèrera le fichier exécutable `a.out`. Pour l'exécuter avec 3 arguments par exemple, taper `./a.out arg1 arg2 arg3`, `arg`X prenant la valeur désirée.

2 Développement d'un client HTTP

2.1 Notions utiles sur : Proxy et telnet

Petits rappels pendant la scéance.

Application : Pour le récupérez des salles TPs, vous pouvez utiliser votre navigateur ou opérer avec les fonctions suivantes :

```
$>export http_proxy="http://proxy710.univ-lyon1.fr"
$>wget http://graal.ens-lyon.fr/~ycaniou/teaching/CCIR5/TP_1_2.pdf
```

☞ Expliquez ce que ces instructions font, et pourquoi les utiliser de cette façon, plutôt que d'utiliser votre navigateur préféré.

2.2 Vérification de la présence du serveur HTTP

Le format d'une requête HTTP est le suivant[4] :

- La première ligne est composée d'une commande HTTP (GET/POST/HEAD/ ?), d'une URL qui identifie la ressource demandée puis de la version du protocole HTTP utilisée,
- Les lignes suivantes constituent l'en-tête de la requête HTTP,
- **La fin de l'entête est signalée par une ligne vide**,
- Les lignes suivantes constituent le contenu de la requête qui bien souvent est vide (sauf dans le cas d'une requête POST).

Voici un exemple de requête qui demande des informations sur la page d'accueil du serveur HTTP à l'aide de la commande HEAD. La réponse du serveur se trouve à la suite de la requête.

```
FouDuBassan:~/Cours/0506/CCIR5/TP_1_2>telnet www-cache.ens-lyon.fr 3128
Trying 140.77.1.58...
Connected to turing.ens-lyon.fr.
Escape character is '^]'.
HEAD http://www.google.fr HTTP/1.0

HTTP/1.0 200 OK
Cache-Control: private
Content-Type: text/html
Set-Cookie: PREF=ID=1bfe5834b86e1e33:TM=1130367517:LM=1130367517:S=Jn3Ifb1CQN4Vr46L; expires=Sun, 17
Server: GWS/2.1
Content-Length: 0
Date: Wed, 26 Oct 2005 22:58:37 GMT
Age: 0
X-Cache: MISS from www-cache.ens-lyon.fr
X-Cache-Lookup: MISS from www-cache.ens-lyon.fr:3128
Proxy-Connection: close

Connection closed by foreign host.
```

2.3 Manipulation

Une fois loggué via telnet sur le proxy du nautibus (proxy710.univ-lyon1.fr sur le port 8080), contactez un serveur HTTP (le port 80 est le port standard HTTP) de la machine serveur et effectuez une requête analogue pour vérifier que le serveur est en bon état de marche.

[4] http://www.commentcamarche.net/internet/http.php3

Questions :

- ☞ Quelles commandes utilisez-vous ?
- ☞ Essayez d'utiliser les proxies de noms respectifs `proxy710` et `miage`. En regardant «attentivement» les résultats, que remarquez vous ? Expliquez. Que pouvez-vous déduire de l'exemple donné avant ?
- ☞ Remplacez le `HEAD` par un `GET`. Que constatez-vous ?

2.4 Client HTTP en mode connecté

2.4.1 Présentation

On vous demande d'écrire un programme **client** que vous lancerez de la façon suivante : `a.out nomserveur port`. **Il fait une requête web et affiche la réponse**. Ce que nous nous proposons de réaliser constitue **le départ du commencement** d'un programme de navigation comme `mozilla` ou `konqueror`.

Question :

- ☞ À partir de ce qui a été vu en cours, donner les idées générales (création de socket, envoi, réception) qui vont vous aider à concevoir votre programme, en précisant quelles fonctions vous pensez utiliser.

Nous donnons dans ce qui suit le squelette avancé d'un programme client. Vous pouvez le trouver à l'URL http://graal.ens-lyon.fr/~ycaniou/teaching/CCIR5/skelClient.c.

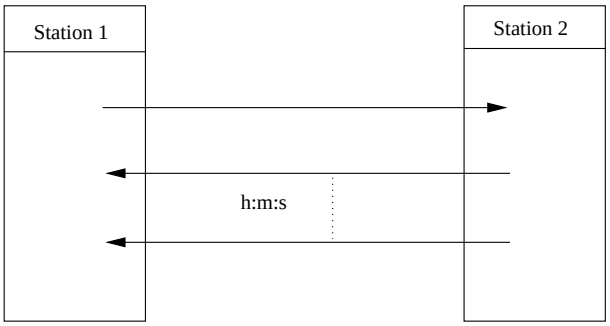
Squelette du programme client.c

```
/* pour compiler: gcc client.c qui crée l'exécutable a.out */

#include <stdio.h>
#include <stdlib.h> /* Pour exit */
#include <string.h> /* Pour bcopy */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define TAILLE 500

main(int argc, char** argv)
{
  int desc ;
  int status ;
  int n ;
  struct sockaddr_in  serveur_addr ;
  struct hostent *remotehost ;
  char chaine[TAILLE] ;

  if( argc!=3 ) {

    exit(1) ;
```

```
  }

  desc=socket(.....) ;
  if( desc == -1 ) {

    exit(2) ;
  }

  remotehost = gethostbyname(argv[1]);
  if( remotehost == NULL) {

    exit(3) ;
  }

  bcopy(remotehost->h_addr,&serveur_addr.sin_addr,remotehost->h_length) ;

  serveur_addr.sin_family = ....
  serveur_addr.sin_port = htons( .... ); /* htons sert a quoi ? */

  status = connect( .... );
  if( status == -1 ) {

    exit(4) ;
  }

  printf("Envoie requête\n") ;

  /* Envoie un "GET nom HTTP/1.0" */

  /* man string vous donnera le nom de fonctions utilisées communément sur
  **   les chaînes de caractères
  ** Vous pourrez avoir besoin des fonctions de string pour construire la
  ** requête HTTP */
  strcpy( .. . .. ) ;

  send( . ... . ) ;

  bzero(chaine,TAILLE) ;

  printf("Reçois requête\n") ;
  n = recv( . .. . ) ;

  printf("Reçu : %s\n",chaine) ;
}
```

2.4.2 Manipulation 1

- ☞ Regarder et analyser le squelette.
- ☞ Comprenez le squelette
- ☞ Ajouter suffisamment de commentaires
- ☞ Compléter le code de sorte que le programme prenne le nom de la machine distante et le port à utiliser en paramètre de ligne de commande. Son objectif est d'envoyer la requête «`GET` nom HTTP/1.0» et d'afficher la réponse du serveur à l'écran.



2.4.3 Manipulation 2

☞ Écrire un programme client en mode connecté qui lit au clavier une requête `HTTP` et affiche la réponse du serveur. Vous pouvez utiliser la fonction `scanf` pour lire une chaîne de caractères du clavier (un exemple d'utilisation est le suivant : `scanf("%s",&chaine)`, avec la variable `chaine` correctement déclarée en vecteur de caractères et pointant sur un espace mémoire alloué).

3 Transfert de messages en mode connecté

3.1 Manipulation

☞ Écrire un programme Client/Serveur qui transfère des messages en mode connecté (TCP) entre deux stations selon le schéma ci-avant. La station 2 doit envoyer 60 fois l'heure courante à la station 1.

3.2 Manipulation et questions

☞ Comptez sur le client et sur le serveur le nombre de messages échangés.

☞ Refaites le même comptage en ajoutant sur le client un délai d'environ une seconde entre la lecture de chaque message (grâce à la fonction `sleep()`). Que constatez-vous ? Y a t-il des pertes de messages ?

☞ Que se passe t-il si vous débranchez le câble réseau reliant le client et le serveur ? Même question si vous débranchez puis rebranchez rapidement le câble.

☞ Pouvez-vous avec votre implantation actuelle servir plusieurs clients ? Si oui, peuvent-ils être servis simultanément ? Lancez simultanément un nombre suffisant de clients pour remplir la file des connexions pendantes du serveur (paramètre à `listen`). Que se passe t-il si vous lancez alors un client supplémentaire alors que la file est pleine ?

4 Transfert de messages en mode non connecté

4.1 Manipulation

☞ Écrivez une nouvelle version du programme précédant en mode non connecté (UDP).

☞ Refaites les manipulations demandées et répondez de nouveau aux questions posées. Quelles sont vos conclusions ?

5 Serveur en mode concurrent

5.1 Manipulation

☞ Transformez le serveur en mode connecté écrit précédemment pour qu'il puisse répondre à plusieurs clients simultanément (serveur mono protocole, mono service en mode concurrent). Vous pourrez vous aider des exercices vus en cours. Mettez en place et décrivez un test permettant de constater que les multiples requêtes clientes sont bien traitées en parallèle.

☞ Que constatez-vous par rapport à l'écoulement du temps si le nombre de clients est important ?

5.2 Manipulation

☞ Transformez maintenant le serveur afin qu'il puisse répondre à trois services simultanément (serveur mono protocole, multi services en mode concurrent).

Le serveur pourra conserver le service initial et proposer de nouveaux services de votre choix (exécution d'une commande distante par exemple pour connaître le nombre de processus qui tournent sur le serveur, transfert de fichiers, ?).

☞ Mettez en place une procédure de test permettant de vérifier que deux requêtes clientes pour un service distinct sont bien traitées en parallèle.