SR03 - Printemps 2017
Communication par sockets TCP

Retour page d'accueil sr03.

TD sockets TCP - Présentation

Nous allons construire un dispositif clients-serveur en utilisant, cette fois-ci comme "système de transport", entre les clients et le serveur, des sockets TCP.

Les sockets TCP ont une sémantique de **flux d'octets**. On va donc rencontrer des problèmes similaires à ceux rencontrés lors d'échanges de données via un pipe Unix et vus au TD 01.

Pour simplifier et ne pas prendre trop de temps, on ne va pas réimplémenter complètement l'application client-serveur précédente, mais seulement les parties importantes qui permettent de comprendre le fonctionnement du moyen de transport utilisé.

TD sockets TCP - A - Serveur et client TCP simples

On va écrire les programmes **clio.c et sero.c**, le serveur TCP étant de **type concourant** : à chaque connexion d'un client, le serveur fork() un sous-process fils qui va traiter tout le dialogue avec ce client.

Cette première étape sera simplifiée.

Définir dans un fichier **defobj.h** un **typedef struct { ...} obj;** qui décrit la structure suivante :

```
+---------------+
| char [max 12] |
|               |
+---------------+
| char [max 24] |
|               |
|               |
+---------------+
| int ii        |
+---------------+
| int jj        |
+---------------+
```

```
| double dd     |
|               |
+---------------+
```

Construire un fichier **iniobj.h** qui initialise un tableau d'objets (longueur du tableau définie par #define tablen n) de type "obj" tels que décrits ci-dessus.

```
"ident_o1", "description_o1", 11, 12, 10.2345
"ident_o2", "description_o2", 21, 22, 20.2345
"ident_o3", "description_o3", 31, 32, 30.2345
```

Ecrire les programmes **clio.c** et **sero.c** suivants :

```
clio.c
    initialise le socket TCP
    fait une demande de connexion au serveur
        boucle :
             envoi de "n" objets "obj" (obtenus de iniobj.h)
             le dernier objet envoyé contient un marqueur de fin
             obj.iqt = -1
    fermer la connexion
    fin

sero.c
    initialise le socket TCP
    boucle :
        attente des connexions clients
        quand connexion client: fork()
            fils : traiterclient()
                    boucle : lire data client sur socket
                             jusqu'à objet contenant "fin"
            père : waitpid() attends fin du fils
                    si statut fin fils et retour sur attente des connexions clients
```

TD sockets TCP - B - Le serveur TCP simple devient un vrai serveur concourant

---

On va écrire une version **serv.c** du serveur TCP qui soit vraiment concourante.

En effet, en **(A)**, pour simplifier l'écriture du premier serveur TCP en ne mélangeant pas tous les problèmes, on a mis le serveur en attente pendant que le fils gère la connexion d'un client.

Ceci ne permet pas de recevoir une nouvelle connexion pendant le traitement de la première.

Un vrai serveur TCP concourant doit, aussitôt après le fork(), aller se remettre en attente sur le accept() pour créer un nouveau fils pour une nouvelle connexion qui pourrait arriver juste après la première, alors que le premier fils n'a pas encore terminé son dialogue avec son client.

Se pose alors le problème de la gestion de la terminaison de tous ces fils. Quand un sous-process fils meurt, il envoie un signal SIGCHLD à son process parent et celui-ci récupère le status de fin du fils dans le paramètre ad-hoc de l'appel wait() ou waitpid().

Que se passe-t-il si le serveur (le père) ne fait pas de wait() pour récupérer la fin des fils ? On peut écrire une version **sernow.c** de serv.c sans le waitpid() et observer le résultat :

```
 1929 pts/2    00:00:00 sernow
 1931 pts/2    00:00:00 sernow <defunct>
 1933 pts/2    00:00:00 sernow <defunct>
```

Comme le père ne partage pas l'espace virtuel du fils, et ne fait pas de wait, il n'est pas informé de la fin de chaque fils. Ceux-ci restent <defunct> (process en état "zombie") dans le système jusqu'à la fin du serveur.

Si on construit un serveur de ce type, après la création d'un certain nombre de fils, le fork() renverra une erreur : il y a une limite système au nombre de sous-process qu'un même process peut créer (il y aussi une limite au nombre total de process dans la machine).

Pour un serveur destiné à rester actif très longtemps (même parfois en permanence), ce comportement est inacceptable.

Il va donc falloir traiter la terminaison des fils tout en allant AUSSI le plus vite possible dans le accept().

Une façon (ce n'est pas la seule : on peut en inventer d'autres à base de données partagées avec les fils et testées par le serveur), sera de mettre en place un "handler" de traitement du signal SIGCHLD envoyé au parent quand un fils se termine.

Dans le handler, on fera un waitpid() pour récupérer le status du fils et permettre ainsi à celui-ci de disparaître du système en libérant ses ressources.

Nota : on ajoutera une instruction **sleep(1);** dans la routine "traiterclient" du serveur fils, afin de simuler un dialogue un peu long entre client et serveur. Ceci est nécessaire pour vérifier que le traitement du signal SIGCHLD a été correctement programmé. Sinon, le fils termine avant que le serveur père ait repris la main et se soit remis dans le accept(), ce qui fait que l'on peut passer à côté du problème et avoir un programme qui "a l'air" de marcher, mais qui se plantera dans une situation réelle.

On voit donc que, si l'utilisation de TCP simplifie d'une certaine façon la gestion du dialogue client<-->serveur, elle amène d'autres contraintes.

```
Avantages de TCP :
   - garantit la fiabilité des échanges client<-->serveur
   - établit une connexion client<-->serveur pendant laquelle
     le fils traitant sait qu'il dialogue toujours avec le même client
   - gère de façon naturelle "n" dialogues simultanés avec des
     clients différents.
```

```
Inconvénients de TCP :
   - l'échange par "flux d'octet" impose l'implantation dans ce flux
     d'un protocole applicatif (reconstituer des messages pour
     construire un dialogue)
   - la gestion du serveur "maître" est plus complexe
```

TD sockets TCP - C - Serveur TCP concourant et clients TCP

On va écrire **client.c et server.c** qui font plusieurs échanges d'objets à chaque connexion, server.c étant capable de gérer plusieurs connexions simultanées.

server.c utilisera la même méthode que serv.c (capture signal) pour gérer les terminaisons des fils et accepter plusieurs clients simultanés

client.c va envoyer à server.c un message de requête (on utilisera pour simplifier l'envoi d'un objet "obj"). Le serveur, de son côté, modifie l'objet et le renvoie au client. Le client, qui boucle sur la lecture de data en réponse à sa requête, reconstitue l'objet et l'affiche.

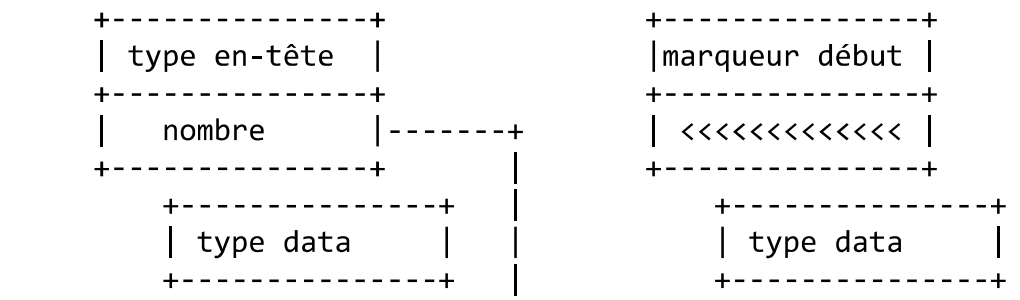
** Attention ** : client.c NE SAIT PAS combien d'octets devront être lus pour avoir la totalité de la réponse et ne devra pas se mettre en situation bloquante en essayant de lire trop de données.

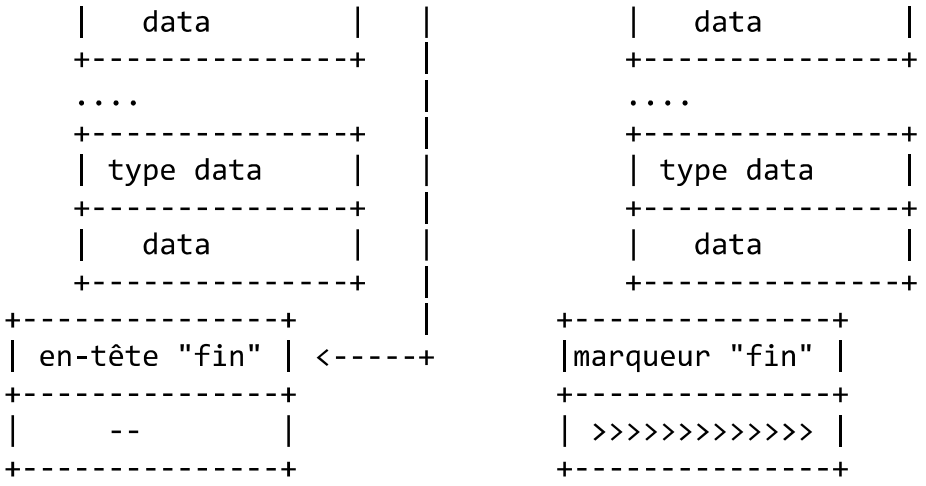
Il faudra donc utiliser un protocole "applicatif" qui décrit la façon dont la réponse du serveur est envoyée au client.

Après lecture de la réponse, le client envoie une requête "fin" indiquant au serveur que cette connexion peut être fermée.

Problématique :

```
On a deux grands choix :
 - travailler sur le contenant : encapsulation
   des données utiles dans des données de gestion;
 - travailler sur le contenu : utiliser des
   marqueurs de début et/ou de fin de messages. Ceci
   suppose que les marqueurs ne sont jamais présents
   dans les données,  ce qu'on peut assurer par des
   techniques d'échappement.
```





L'important est de pouvoir assurer une extraction fiable, depuis le flux, d'une série de messages de longueur différentes, ces longueurs n'étant pas connues à l'avance, mais découvertes au fur et à mesure de la lecture du flux.
