Systèmes d'exploitation et réseaux Projet de programmation réseaux version 1 : Messagerie TCP

Introduction

Mode d'emploi. Il s'agit d'un projet très encadré qui sera fait en partie pendant 4 derniers TP : vous développerez en faisant une série d'exercices détaillés une messagerie réseau, permettant à plusieurs utilisateurs d'échanger des messages.

Le projet se fait normalement en solo. Les binômes sont tolérés mais doivent être déclarés, on attend un peu plus de matière dans le travail de binôme.

On vous demande de rendre chaque exercice dès qu'il est terminé (ou un blocage important arrive). Le rythme normal est de 2 rendus par semaine. Démarrez rapidement, pour poser le maximum de questions en TP. L'enseignant est aussi à votre écoute (mais moins réactif) par mail asarin@irif.fr (n'hésitez pas d'attacher vos fichiers.c), et sur le forum moodle.

Plan de travail Vous développerez successivement (dans les exercices 1–5)

- 1. un serveur TCP "echo" qui permet à un client de se connecter et lui retourne ses messages ;
- 2. sa version multiclient;
- 3. sa version qui gère quelques commandes simples, et non seulement l'écho;
- 4. une variante améliorée ou le serveur connait la liste de ses connexions;
- 5. et finalement un serveur messagerie TCP.

Ensuite:

6. vous mettrez vraiment votre application en réseau.

Dans la partie optionnelle (elle n'est pas plus difficile) vous pourrez développer

- 7. un client TCP qui pourra communiquer avec le serveur;
- 8. la résolution DNS pour le client...
- 9. et pour le serveur

Le développement et les tests peuvent être faits sur une seule machine, mais il est très conseillé d'essayer de communiquer avec vos serveurs à travers un réseau local ou Internet (exercice 6). Le reste (client et DNS) est vraiment optionnel

Références Avant de commencer, on vous conseille de (re-)lire les slides et les exemples de cours 12, et le poly réseaux de Juliusz Chroboczek [JCH-R] (surtout chapitres 1 et 3), tout ceci est disponible dans l'espace Moodle de cours. On donnera des conseils plus précis pour chaque exercice.

Pour tester une application réseaux sur une seule machine. Vous le ferez tout le temps :

- ouvrir deux (ou plus) fenêtres de terminal
- dans l'une vous lancez votre serveur (en configurant son port). Il se met à attendre les clients.
- dans l'autre fenêtre vous lancez un client : telnet, votre client à vous, ou nc (par exemple, si vous êtes sous MacOs et n'avez pas installé telnet). Il faudra indiquer l'adresse IP de la machine locale (localhost, 127.0.0.1, ::1 ou bien ::ffff:127.0.0.1 sur une installation capricieuse) et le port du serveur;

 normalement une connexion doit se faire et un dialogue commencer. Distinguez entre ce que le client envoie, et ce qu'il reçoit du serveur!

- comme d'habitude, pour déboggage vos programmes peuvent afficher des nombreux messages dans leur terminaux. Surveillez les deux fenêtres.
- rapidement, vous aurez aussi plusieurs fenêtres terminaux clients et une avec le serveur!

Pour avoir du recul Essayez de distinguer les deux aspects de la programmation réseau en C :

La simplicité conceptuelle. Ainsi un serveur configure un port, ouvre une socket, se met en attente, si un client arrive le serveur crée une connexion représentée par une autre socket. Toutes les échanges sont des simples read et write (les sockets se comportent comme des descripteurs de fichiers);

le bidouillage structures de données lourdes, conversion entre plusieurs formats des entiers, nombreux formats d'adresses IP, gestion des signaux et des erreurs, les octets \n,\r,\0 ajoutés par vous et par telnet...

Il y aura aussi dans ce projet des révisions et des approfondissement des sujets déjà vus : pointeurs et mémoire dynamique, tampons, concurrence et threads.

Exercices préparatoires

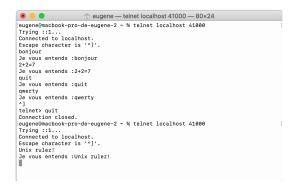
Exercice 1 – Premier programme — serveur écho simple serverEcho.c

Références : Slides du cours 12 ; poly [JCH-R]. N'oubliez pas le man.

Descriptif de l'application. Il s'agit d'un serveur echo. L'appel ./server port doit lancer un serveur TCP sur le port passé en paramètre et attendre qu'un client se connecte. Lorsqu'un client arrive, le serveur doit écouter les messages que lui envoie le client et lui renvoyer aussitôt, précédés d'un message supplémentaire tel que "j'ai reçu :". Lorsque le client part, le serveur retourne attendre qu'un client se connecte. On vous suggère d'utiliser un numéro de port entre 40000 et 44000.

```
reseau — server 41000 — 80×24

[eugene@macbook-pro-de-eugene-2 reseau % gcc -Wall server1.c -o server | |
| eugene@macbook-pro-de-eugene-2 reseau % ./server 41900 | |
| client connecté | |
| client connecté | |
```



Architecture suggérée de l'application :

- la procédure client_arrived(int sock) implémente la fonction d'écho du serveur :
 elle écoute sur la socket du client et lui renvoie tous ses messages, et retourne lorsque le client part. On peut détecter que le client part lorsque read/write échoue.
- la procédure listen_port(int port_num) fait tout le reste : elle
 - crée une socket s avec le constructeur s=socket(PF_INET6,SOCK_STREAM, 0)

— crée une structure pour l'adresse, struct sockaddr_in6 sin;, la remplit de 0 avec memset, et affecte les valeurs à ses deux champs :

```
sin.sin6_family=PF_INET6;
sin.sin6_port=htons(port_num);
```

(attention, on doit changer le codage d'entier pour le numéro du port avec htons). La socket serveur n'a pas besoin d'autres paramètres, telles que address IP!

- lie la socket s et la structure sin avec bind() et lance l'écoute avec listen(s,1024);
- lance une boucle infinie pour accepter les clients avec s2=accept(s,NULL,NULL), Lorsqu'un client arrive, on appelle client_arrived avec la socket renvoyée par accept.
- finalement la fonction main est presque triviale, elle
 - analyse la ligne de commande;
 - lance la procédure listen_port.
- Détectez et gérez toutes les erreurs (et affichez le perror), ceci après socket, bind, listen, accept, read/write etc. Gérez-les doucement, en faisant exit seulement si on ne peut plus continuer. Vérifiez que vous avez appelé close pour toutes les sockets ouvertes.

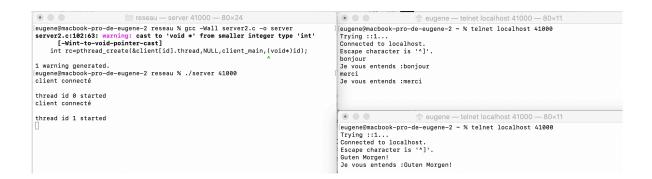
Conseils pour tester Lancez d'abord votre serveur dans une fenêtre terminal (avec « ./server port » en choisissant un numéro de port ≈40000). Dans une autre fenêtre lancez le client « telnet localhost port » (avec le même numéro de port) pour tester votre application : lorsque vous entrez un message suivi d'un retour à la ligne, vous devez recevoir le même message. Pour arrêter le terminal, appuyez ctrl-], puis tapez quit. À la place de telnet sous MacOS utilisez nc, sous Windows (interdit) téléchargez putty.

```
Un peu de bidouillage    Pour débogguer le programme plus facilement, notamment si
vous recevez l'erreur Address already in use lors de l'appel à bind, on vous suggère
d'utiliser l'option suivante sur la socket serveur, avant l'appel à bind :
int optval = 1;
setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval));
On vous suggère aussi d'utiliser (dans main) le code
signal(SIGPIPE, SIG_IGN);
afin d'éviter que le serveur se fasse tuer lorsqu'un client quitte sans prévenir.
```

Exercice 2 – Serveur echo multi-client serverMulti.c

Avant de commencer Révisez un peu les threads, en particulier les exercices 2 et 3 du TP8.

Descriptif de l'application. Votre précédent serveur était très limité puisqu'il ne pouvait gérer qu'un seul client à la fois (essayez d'en connecter deux). Vous devez maintenant le modifier pour gérer plusieurs clients en créant un *thread* par client qui mènera un dialogue avec lui (il renverra au client ses messages).



Comment programmer. Copiez votre programme de l'exercice 1.

Vous utiliserez le type suivant pour stocker les paramètres de chaque client ¹:

```
typedef struct client_data_s {
   int sock;
   pthread_t thread;
}client_data;
```

On vous propose l'architecture suivante de l'application :

- Ajoutez une variable globale int nr clients pour compter le nombre de clients.
- Ajoutez une procédure init_clients() qui met le compteur nr_clients=0.
- Ajoutez une fonction client_data* alloc_client(int sock) un constructeur qui
 - alloue la mémoire pour une structure client_data et renvoie NULL en cas d'échec;
 - initialise sont champ sock par le numéro passé en paramètre;
 - incrémente le compteur nr_clients.
 - renvoie ensuite le pointeur de client_data alloué);
- Ajoutez une fonction free_client(client_data* cli) qui libère la mémoire, et décrémente le compteur.
- Pour éviter les data races protégez la manipulation du compteur de clients nr_clients :
 - ajoutez en variable globale un verrou mutex pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER.
 - dans alloc_client() et free_client() prenez le verrou avant de toucher au compteur et libérez-le après.
- Modifiez client_arrived(int sock) qui doit maintenant
 - créer une donnée client avec client_data* cli=alloc_client(sock);
 - créer un thread en lui passant comme paramètre le pointeur du client. Indication :

```
rc=pthread_create(&cli->thread,NULL,worker,(void*)cli)
```

- afficher le nombre de clients actifs;
- retourner au main sans attendre, le thread travailleur se débrouillera tout seul.
- Ajoutez une fonction worker (un peu plus haut pour que ça compile sans warning), la fonction lancée dans chaque thread par client_arrived. Elle récupère les données du client et puis en boucle elle écoute les messages du client et les lui renvoie (vous avez écrit quelques lignes de code pour cela dans l'exercice 1). Lorsque le client se déconnecte, elle clot la socket, appelle free_client et sort.
- La procédure listen_port ne change pas.
- La fonction main doit appeler init_clients avant de se mettre à l'écoute.
- N'oubliez pas de gérer les erreurs (je ne le répéterai plus).

^{1.} On l'enrichira dans les exercices suivants. Pour l'instant même le champ thread n'est pas très utile...

Testez votre application en s'y connectant avec 2 ou 3 clients, essayez d'en fermer certains (avec ctrl-] et puis quit, ou d'une manière plus violente, par exemple avec un kill).

à suivre