Systèmes d'exploitation et réseaux
Projet de programmation réseaux version 4 : 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 (avec résolution DNS) qui pourra communiquer avec le serveur ;
8. et/ou la résolution DNS 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 dans le dossier `Code utile`, 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 :

- ouvrez 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. . .

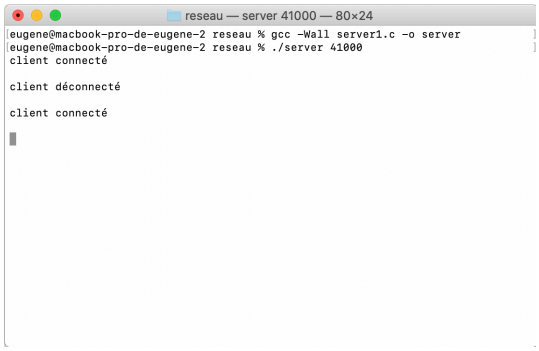
Il y aura aussi dans ce projet des révisions et des approfondissement des sujets déjà vus : pointeurs et mémoire dynamique, tampons, concurrence et threads.

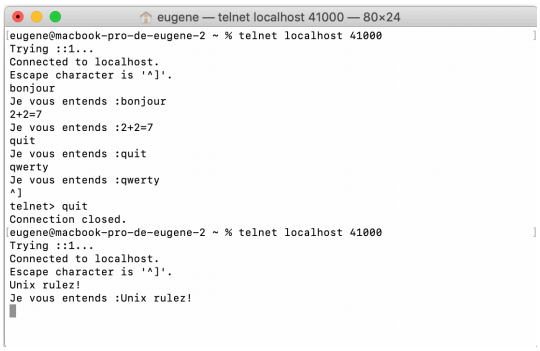
Exercices préparatoires

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.





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

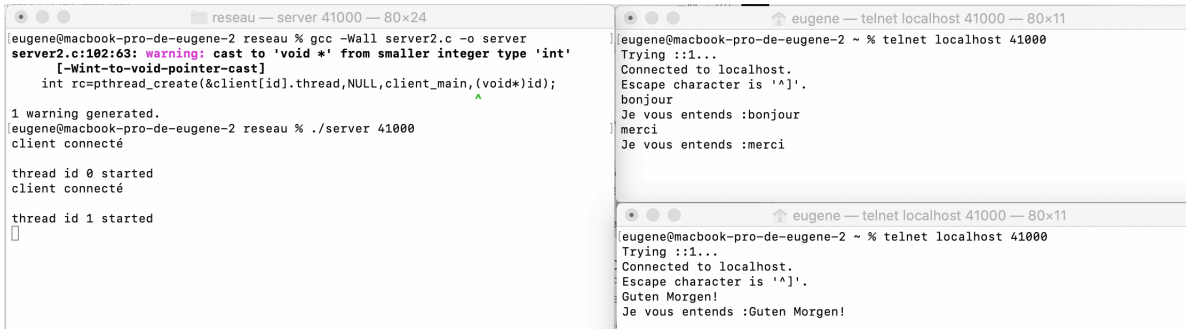
```
signal(SIGPIPE, SIG_IGN);
```

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`).

Exercice 3 – *Serveur de commandes simples — `serverCom.c`*

Descriptif de l'application. Le serveur précédent se contentait de répéter à chaque client les messages qu'il recevait de sa part. Dans cet exercice le serveur exécutera certaines commandes demandées par le client.

Nous allons mettre en place un protocole de message entre les clients et le serveur :
- chaque message (du client et du serveur) est terminé par `\n` ;
- le client doit envoyer des messages de la forme : `[commande] [arguments] \n` qui indique l'action à effectuer et les arguments éventuels ;
- Le serveur doit ensuite répondre par : `ok [reponse] \n` en cas de succès ou par `fail [message] \n` en cas d'erreur.

Dans cet exercice, le serveur ne supportera que trois ou quatre commandes :
- **`echo`** renvoie `ok` suivi des arguments qu'il a reçus.
- `rand` sans argument renvoie un nombre aléatoire ; sa variante `rand n` retourne un nombre aléatoire entre 0 et `n` (exclu).
- `time` (optionnel) sans argument renvoie l'heure et le date.
- `quit` déconnecte le client (et ne renvoie rien au client).
- toute autre commande doit échouer avec `fail`.

Voici un exemple de session (>>> représente l'invite du client) :

```
>>> echo
ok
>>> echo je suis amaury
ok je suis amaury
>>> rand
ok 42736
>>> rand 36
ok 30
>>> hello
fail unknown command 'hello'
>>>
fail empty command
>>> quit
[server disconnected]
```

L'architecture suggérée On part du code de l'exercice 2. Inspirez-vous aussi du code des fichiers `serverTCPtamponModular.c` (pour la récption) et `clientTCPtampon.c` (pour l'envoi) sur Moodle.
- Ajoutez une macro pour définir la taille maximale d'un message *`#define BUFSIZE 1024`*
- Modifiez la boucle de `worker` pour qu'elle ressemble au code suivant :

```
char question[BUFSIZE], response[BUFSIZE],buf[BUFSIZE];
...
while(1) {
    if(get_question(cli->sock,buf, &len, question) < 0)
        break; /* erreur: on deconnecte le client */
    if(eval_quest(question, response) < 0)
```

```
            break; /* erreur: on deconnecte le client */
        if(write_full(cli->sock, response, strlen(response)) < 0) {
            perror("could not send message");
            break; /* erreur: on deconnecte le client */
    }}
```

Ici la fonction `get_question()` utilise le tampon `buf` pour lire les données qui arrivent du client, et récupérer sa requête dans le tampon `question` ; la fonction `eval_quest` exécute la commande stockée dans le tampon `question`, et prépare la réponse du serveur dans le tampon `response`. La fonction `write_full` envoie via la socket tout le contenu du tampon (en réessayant en cas d'écritures partielles). Ces fonctions renvoient -1 en cas d'échec.

– Adapteer la fonction `get_question(int sock, char *buffer, int *len, char* question)`. Elle part du `buf` avec `*len` lettres, lit encore quelque lettres si nécessaire, et récupère une question (délimitée par `\n` ou `\r\n`). Faites en sorte que la question se termine par `\0`, en écrasant le `\n`[2]. Renvoyez `-1` dans le cas d'une erreur.

> **Pour comprendre la commande du client** Étant donné une chaîne `str` constituée de plusieurs mots séparés par des espaces, pour découper le premier mot on peut faire
>
> ```
> char *first_word = strsep(&str, " ");
> char *everysthing_else=str;
> ```
>
> Attention, le pointeur `str` pointe maintenant sur la deuxième partie de la chaine, ou bien est `NULL` si la chaîne contenait initialement un seul mot. Ne pas utiliser `strtok` qui ne fonctionne pas avec plusieurs threads

> **Pour fabriquer la réponse** La fonction `snprintf(char* str, int size, char* format, ...)`, très similaire à **`printf`** que vous utilisez tous les jours, écrit une sortie formatée dans la chaine `str`, sans dépasser la taille `size`, c'est exactement ce qu'il nous faut pour fabriquer les réponses du serveur.

– Ajoutez une fonction `eval_quest` qui va analyzer le message du client, effectuer l'action demandée et écrire la réponse dans le tampon passé en argument :
`int eval_quest(char *msg, char *resp, int resp_len)`. Cette fonction retourne `-1` s'il faut déconnecter le client (soit parce qu'il le demande ou en cas d'erreur critique). Afin de rendre le code lisible et de faciliter l'ajout de commande dans le futur, on vous conseille une structure telle que celle-ci :

```
int eval_quest(char *question, char *resp) {
    char *cmd, *args;
    ... // analyse question et separe la commande des arguments
    if(strcmp(cmd, "echo") == 0)
        return do_echo(args, resp);
    else if(strcmp(cmd, "rand") == 0)
        return do_rand(args, resp);
    else if(strcmp(cmd, "quit") == 0)
        return do_quit(args, resp);
    else {
        snprintf(resp, resp_len, "fail unknown command %s\n", cmd);
        return 0;}}
```

2. si le `telnet` a inséré le caractère invisible `\r` (retour-chariot) avant le `\n`, il faut aussi l'écraser

Vous devez ensuite implémenter chaque commande dans une fonction à part (`do_rand`, `do_echo` et `do_quit`). Ces fonctions doivent respecter la taille `BUFSIZ` de la chaine consruite `resp`

Exercice 4 – *Détail technique à régler : connaitre l'ensemble des clients — `serverComList`*

Avant de commencer Révisez les listes doublement chainées. Téléchargez et consultez mon programme `exempleList.c` dans la section `Fichiers Utiles`.

Ce qui nous manque Dans le programme de l'exercice précédent, pour chaque client connecté il y a un thread qui s'en occupe. On connait aussi le nombre total de clients `nr_clients`. Mais il n'existe nul part l'information sur l'ensemble des clients connectés (et leurs threads). Ainsi il est impossible de lister tous les clients connectés, un thread ne peut pas transmettre l'information pour un autre client etc. Or, c'est indispensable pour une messagerie.

Descriptif de l'application. On ajoutera à l'application précédente une liste de clients connectés. Pour pouvoir la tester, on ajoutera une nouvelle commande `list`, le serveur doit répondre `ok` et lister les sockets des clients connectés

Comment programmer On part du code de l'exercice précédent (et on s'inspire largement de programme `exempleList.c`). Le changement principal est qu'on maintiendra une liste doublement chainée des `client_data`.

- Ajoutez les champs `prev` et `next` dans la structure `client_data`. Bien sûr, ce sont les pointeurs qui pointent sur les données du client précédent et suivant.
- Ajoutez les variables globales `client_data *first,*last;` pour les deux extrémités de la liste ;
- Complétez les fonctions déjà présentes de votre programme :
  - `init_clients` doit en plus mettre à `NULL` les variable `first` et `last`.
  - `alloc_client` doit en plus accrocher la structure nouvellement créée à la fin de la liste.
  - `free_client` doit sortir la structure de la liste, recoudre le trou, et libérer la mémoire.
  - Toutes les manipulations de la liste chainée par plusieurs chaines sont potentiellement dangereuses, protégez les données avec vos verrous mutex.
- Testez que la liste fonctionne en l'affichant côté serveur dans chaque appel de `client_arrived`.
- Ajoutez une nouvelle commande `list` à votre serveur — en insérant un nouveau cas dans `eval_quest` et une nouvelle fonction `do_list`. Provisoirement, le serveur renverra au client la liste de toutes les sockets connectées.

Testez en connectant et déconnectant les clients et en appelant `list`.

La messagerie

Exercice 5 – *On arrive au but : messagerie TCP — `serverMess.c`*

Avant de commencer Révisez en détail (et finalisez si ce n'est pas encore fait) l'application de TP 5 (exercices 1 et 2, en dernier recours le corrigé du prof est fourni sur moodle).

Descriptif de l'application. Maintenant que votre serveur gère les commandes, nous allons pouvoir définir des commandes pour une messagerie. Chaque client va devoir envoyer un pseudonyme au serveur afin de participer. Le serveur doit pouvoir fournir une liste des autres clients connectés. Le client peut ensuite demander au serveur d'envoyer un message à un autre client, et vérifier s'il a reçu des messages. On aura donc besoin d'ajouter les commandes suivantes :

- `nick pseudo` indique au serveur que le client s'appelle `pseudo`, le serveur doit vérifier que ce pseudonyme n'est pas déjà utilisé et contient entre 1 et 8 caractères, tous alphanumériques.
- `list` demande au serveur de retourner la liste des clients connectés, séparés par des espaces (`ok pseudo1 pseudo2 ...`).
- `send pseudo texte` envoie le `texte` passé en argument au client `pseudo`. Le serveur devra rejeter les messages vides et les messages pour des clients n'existant pas.
- `recv` demande au serveur si le client a reçu un message, le serveur retourne alors soit un message vide (indiquant qu'il n'y a aucun message en attente) ou bien le premier message en attente avec le pseudo de l'expéditeur (`ok pseudo ...`). S'il y a plusieurs messages en attente, le serveur n'envoie que le premier (et le client doit appeler plusieurs fois `recv`).

On ajoutera la contrainte qu'un client ne peut pas appeler `list`, `send` ou `recv` tant qu'il n'a pas donné son pseudonyme avec `nick`. Voici un exemple de session avec trois clients :

```
>>> nick amaury
ok
>>> list
ok amaury
[eugene se connecte]
>>> list
ok amaury eugene
[mihaela se connecte]
[mihaela se deconnecte]
[eugene se deconnecte]
>>> list
ok amaury
```

```
>>> nick amaury
fail nickname unavailable
>>> nick eugene
ok
>>> list
ok amaury eugene
>>> quit
```

```
>>> nick mihaela
ok
>>> list
ok amaury eugene mihaela
>>> quit
```

Comment programmer On part du code de l'exercice précédent.

- Ajoutez les structures de données `msg` et `mbox` et les fonctions associées du TP4 dans votre programme du serveur.
- Ajoutez deux nouveaux champs à la structure `client_data_s`

```
char nick[9];\\ pseudo jusqu'a 8 lettres
mbox box; \\ boite aux lettres
```

- Complétez le constructeur `alloc_client` qui (entre autre) mettra la chaine vide dans le pseudo du nouveau client, et initialisera sa boite aux lettres.
- Compléter le destructeur `free_client` en nettoyant la boite.
- Ajoutez les fonctions `int valid_nick(char* nick)` qui vérifie que le pseudo `nick` satisfait les contraintes ci-dessus (et renvoie 1 si c'est le cas, utilisez `isalnum` si vous voulez) ; et `client_data* search_client(char* nick)` qui cherche dans la liste des clients et renvoie le `client_data` avec un pseudo donné (ou `NULL` s'il n'y en a pas).

- Modifiez la fonction `do_list` pour qu'elle affiche la liste de pseudos (et non les numéros de sockets).
- Ajoutez les cas `nick`, `send`, `rcv` à `eval_quest` et les nouvelles fonctions `do_nick`, `do_send` et `do_recv` (par ailleurs, elles doivent prendre un argument de plus : le `client_data` du celui qui fait la demande).
- Implémentez les fonctions `do_send` et `do_recv` (elles ressemblent un peu aux fonctions `send_mess` and `recieve_mess` TP4).
- Ajoutez l'interdiction de faire `send`, `rcv`, `list` pour les utilisateurs anonymes.

Exercice 6 – *Mise en réseau*

✍ Essayez de vous connecter à votre serveur (ou à celui d'un camarade de promo) à partir d'une autre machine, en utilisant telnet/putty/nc ou un autre client, dans le réseau local ou à travers L'Internet. Demandez conseil au prof.

Extensions optionnelles

Exercice 7 – *Deuxième programme — client telnet simple*

Vous devez maintenant implémenter un client telnet simple qui se connecte à un serveur, attend que l'utilisateur entre un message au clavier, l'envoie au serveur puis écoute la réponse, et ainsi de suite. L'appel `./client address port` doit lancer un client TCP qui se connecte à l'hôte dont l'adresse (numérique ou symbolique) et port sont spécifiés sur la ligne de commande, et attend que l'utilisateur entre un message au clavier. Une fois fait, le client envoie le message au serveur et attend la réponse du serveur. Utilisez la convention suivante : **tout message (du client ou du serveur) doit être terminé par "\n"**. Afin de permettre à l'utilisateur de quitter proprement le programme, le client doit se déconnecter du serveur lorsque l'utilisateur entre le message "q" ou "quit".

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

- La fonction `connect_server(const char *host, const char *port)` se connecte au serveur et renvoie la socket permettant de connecter avec celui-ci (ou `-1` en cas d'échec). Inspirez-vous de l'exemple `clientDNS.c`
- Les fonctions `get_response()` et `write_all` qui récupère/envoie un message au serveur, en utilisant les tampons.
- La procédure `speak_to_server(int sock)` affiche l'invite `>>>`, lit un message de l'utilisateur avec `fgets` (ou `scanf`), l'envoie au serveur et affiche la réponse reçue avec `receive_message`, et ce jusqu'à ce que l'utilisateur quitte.
- `main` analyse la ligne de commande et se connecte au serveur grâce à la fonction `connect_server`. Enfin elle lance le dialogue avec le serveur en utilisant la procédure `speak_to_server`.
- la procédure `speak_to_server` devra envoyer les requêtes saisies par l'utilisateur au serveur, et afficher ses réponses. Vous pouvez utiliser un tampon comme dans l'exemple `clientTCPtampon.c`

Vous pouvez tester votre client en vous connectant au serveur de l'exercice précédant avec `./client ::1 42000`, ou `./client localhost 42000` ou même `./client 127.0.0.1 42000`, saura-t-il comprendre qu'il s'agit toujours de la machine locale ? Essayez maintenant de vous connecter au serveur de votre voisin : si son ordinateur s'appelle "i10" alors vous pouvez entrer `./client i10 42000`.

Exercice 8 – *Améliorations du serveur : reverse DNS*

Maintenant que vous pouvez vous connecter facilement à d'autres ordinateurs, il peut être utile pour le serveur de savoir d'où vient la connexion. Il est possible d'obtenir l'adresse IP d'un client en donnant un argument supplémentaire à `accept` :

```
struct sockaddr_in6 addr;
socklen_t addr_len = sizeof(addr);
int client_sock = accept(srv_sock, (struct sockaddr *)&addr, &addr_len);
```

Vous pouvez afficher l'adresse IP grâce à la fonction `inet_ntop`, toutefois ce n'est pas très lisible. Étant donné que vos machines ont des noms DNS, on peut faire l'inverse du client : obtenir le nom DNS à partir de son IP (reverse DNS) en utilisant la `getnameinfo` :

```
char buf[NI_MAXHOST];
if(getnameinfo((struct sockaddr *)&addr, addr_len, buf, sizeof(buf), NULL,
       0, 0) == 0)
   printf("connection from %s\n", buf);
```

Modifiez le code du `main` pour récupérer l'adresse avec `accept` et affichez son reverse DNS. Vérifiez que cela marche en demandant à votre voisin de se connecter sur votre serveur.