Système et Réseaux : Projet « *freescord* »

avril 2025 — Pierre Rousselin

1 Introduction et consignes générales

Le but du projet *freescord* est de permettre à des utilisateurs du monde entier de discuter et d'échanger des fichiers en se connectant à un serveur.

Ce travail doit être fait seul.

Le point de départ doit être l'ensemble des fichiers fournis avec ce sujet sur `moodle`.

Vous rendrez votre travail sur le dépôt moodle avant le 17 mai 23h59.

— Vous devez fournir votre code complet avec un Makefile mis à jour de façon à ce que que les correcteurs puissent compiler avec la commande

```
$ make
```

et c'est tout. J'insiste lourdement : à part un court rapport, **aucun fichier non nécessaire à la compilation** ne doit être transmis, en particulier aucun exécutable, ni fichier `.o` ne doit être dans votre dépôt.

— Chaque fichier `.c` ou `.h` commencera par

```
/* prénom nom numéro_étudiant
   Je déclare qu'il s'agit de mon propre travail.
   Ce travail a été réalisé intégralement par un être humain. */
```

— Il est interdit d'utiliser des IA génératives comme *chatGPT*, *copilot*, ... Vous devez apprendre à programmer un client et un serveur sans piller le travail collectif de l'humanité et sans vider les rivières.

— Fournir aussi un rapport concis et clair au format texte ou pdf qui contient :

- La description détaillée et claire de vos ajouts au protocole *freescord* s'il y a lieu.
- La description des fonctionnalités mises en œuvre avec quelques éléments permettant de comprendre rapidement votre implémentation.
- Le cas échéant, ce qui ne fonctionne pas et vos tentatives pour résoudre les problèmes.
- Si vous avez utilisé une ressource externe au cours, le mentionner, donner la référence (l'URL si c'est sur le web).
- Indiquer si un autre étudiant vous a aidé.

— Une bibliothèque permettant d'utiliser des listes doublement chaînées est fournie (fichiers `list.c`, `list.h` et fichier de tests `test_list.c`). Elle permet de manipuler des listes de pointeurs `void *` sans (trop) s'embêter. Sa documentation est dans le fichier `list.h`. Vous pouvez ajouter des fonctions à cette bibliothèque ou en modifier certaines, mais il faut le documenter, le tester et le mentionner dans le rapport.

— Derniers conseils :

- Le contenu du module permet déjà de faire énormément de choses, sans avoir à aller chercher des choses sur le web.
- Attention à la gestion de votre temps, équilibrez vos efforts avec les autres matières. Si vous êtes à jour, il est possible de mener ce projet à bien assez rapidement.
- Les 2 prochaines sections sont des exercices assez guidés à faire pas à pas, comme en TP. Il y a quelques petites difficultés mais elles ne sont pas insurmontables du tout. Si vous rendez des programmes clairs, propres et rigoureux pour ces 2 sections, vous aurez au moins 12 (et probablement une bonne note au Partiel 2 aussi...)
- La complexité augmente au fur et à mesure du sujet, lorsque nous ajoutons des fonctionnalités.
- La dernière section est plus à la carte, avec des suggestions très recommandées mais aussi des parties beaucoup plus libres. Vous êtes invités, si vous avez de bonnes idées, à faire évoluer *freescord* et à implémenter ces nouveautés.

2 Faire communiquer tout ce petit monde

On décide que le protocole *freescord* utilise le protocole transport TCP sur le port 4321.

Dans cette partie, on ne fixe rien d'autre sur le protocole applicatif, on va déjà faire en sorte que plusieurs utilisateurs puissent se connecter au serveur et que les lignes entrées dans les terminaux des clients soient recopiées sur tous les terminaux des clients.

C'est déjà assez technique et intéressant.

Exercice 1 : Un serveur et un client

On commence par le minimum vital.

1. Côté serveur, écrire la fonction

```c
/** Créer et configurer une socket d'écoute sur le port donné en argument
 * retourne le descripteur de cette socket, ou -1 en cas d'erreur */
int create_listening_sock(uint16_t port);
```

2. Puis, toujours pour le serveur, dans une fonction `main`, ajouter le code permettant, successivement :
   a) d'accepter les demandes de connexion des clients
   b) de lire les octets envoyés par le client et de les lui renvoyer
   c) jusqu'à ce que le client ait fermé sa socket.
3. Côté client, écrire la fonction

```c
/** se connecter au serveur TCP d'adresse donnée en argument sous forme de
 * chaîne de caractère et au port donné en argument
 * retourne le descripteur de fichier de la socket obtenue ou -1 en cas
 * d'erreur. */
int connect_serveur_tcp(char *adresse, uint16_t port);
```

4. Dans la fonction `main` du client, se connecter au serveur dont l'adresse IP est donnée comme argument (dont on vérifiera l'existence).
5. Puis dans une boucle :
   — lire une ligne d'entrée du terminal ;
   — envoyer cette ligne au serveur ;
   — lire la réponse du serveur (qui devrait être la même chose) ;
   — et la recopier sur le terminal.
6. Lorsque l'utilisateur met fin à l'entrée avec `Ctrl-D`, le client sort de la boucle, ferme sa `socket` et prend fin.
7. Tester, vérifier que tout fonctionne bien, en particulier que le serveur peut servir un deuxième client une fois que le premier a terminé.

--- * ---

Exercice 2 : Utilisateurs et *threads*

1. Dans un fichier `user.h`, créer une structure `struct user` qui contiendra, en particulier, la `sockaddr` du client (disons pour l'instant, seulement pour IPv4), sa taille, et le descripteur de fichier de la socket associée au client.
2. Dans un fichier `user.c`, écrire les deux fonctions suivantes (déclarées et documentées dans `user.h`). Noter que la fonction `user_accept` contient l'appel système `accept`.

```c
/** accepter une connection TCP depuis la socket d'écoute sl et retourner un
 * pointeur vers un struct user, dynamiquement alloué et convenablement
 * initialisé */
struct user *user_accept(int sl);
/** libérer toute la mémoire associée à user */
void user_free(struct user *user)
```

3. Modifier le serveur de façon à utiliser ces deux fonctions et vérifier que tout fonctionne convenablement.
4. On passe à un serveur multithreadé. Définir la fonction

```c
/** Gérer toutes les communications avec le client renseigné dans
 * user, qui doit être l'adresse d'une struct user */
void *handle_client(void *user);
```

   Puis faire en sorte que, après avoir accepté un nouvel utilisateur, le serveur laisse un nouveau *thread* le prendre en charge. Ce *thread* est détaché avec `pthread_detach`, pour que ses ressources soient libérées lorsqu'il se termine. Ce *thread* prendra en argument l'adresse retournée par `user_accept`. Il doit libérer la mémoire associée à `user` avant de se terminer.
5. Tester avec plusieurs clients, vérifier que l'echo fourni par le serveur fonctionne bien pour tout le monde, sans qu'il y ait de blocage dans une des lectures.
6. À ce stade, vous avez un serveur qui fait de l'echo et des clients qui peuvent se connecter au serveur simultanément. Veillez à bien polir votre code et à vérifier qu'il n'y a aucune zone d'ombre. Si vous utilisez un système de contrôle de version, c'est un bon moment pour créer un tag.

--- * ---

Exercice 3 : echo vers tous les clients

On va modifier serveur et clients de façon à ce que chaque message de chaque client soit renvoyé à tous les clients (et non pas, seulement son expéditeur).

1. Côté serveur :
   a) créer un tube (*pipe*), par exemple comme variable globale (comme vous voulez)
   b) une liste d'utilisateurs actuellement connectés est maintenue tout au long du programme (utiliser la bibliothèque fournie pour les listes).
   c) Un nouveau *thread* « répéteur » est lancé au départ. Il est en lecture sur le tube, puis, chaque fois qu'il lit des octets dans le tube, les réécrit dans toutes les sockets de tous les utilisateurs. Pas de pouvoir magique ici, il faut parcourir la liste de tous les utilisateurs et écrire dans toutes les sockets de tous les utilisateurs.
   d) Dans les *threads* associés aux clients, pour chaque message reçu, le recopier dans le tube.
   e) Vous pouvez déjà voir le résultat de ces modifications, même si les clients se comporteront étrangement, les lectures dans la socket ayant lieu trop rigidement, après chaque écriture dans cette socket.
2. Pour résoudre ce problème, côté client :
   a) on utilise l'appel système `poll` qui permet, entre autre, d'être bloquant jusqu'à ce qu'un fichier (ou plus) de son tableau de descripteurs, soit prêt pour la lecture. Chaque fois que `poll` retourne, on regarde lequel des descripteurs de fichier a causé son retour :
   b) si l'entrée standard a une ligne à lire, lire cette ligne et l'envoyer ;
   c) si la socket contient des octets à lire, les lire et les écrire sur le terminal.
3. Essayer, tester, polir, ... À ce stade, normalement, on a une application très basique qui permet de chatter à plusieurs. Si vous utilisez un système de contrôle de version, c'est un bon moment pour créer un tag.

--- * ---

3 Lignes du protocole *freescord*

Exercice 4 : Lignes CRLF ou LF

Dans le protocole *freescord*, clients et serveur s'échangent des message. Un message *freescord* est une ligne :

— d'au plus 512 octets

— terminée par CRLF, c'est-à-dire les deux caractères *carriage return* (`'\r'`) et *line feed* (`'\n'`), qui font partie de cette limite.

C'est un choix très répandu dans les protocoles applicatifs (et sur Microsoft Windows). Malheureusement, ce n'est pas comme sur Unix, où la fin de la ligne est uniquement LF, c'est-à-dire le caractère *line feed* (`'\n'`).

Définir les deux fonctions `crlf_to_lf` et `lf_to_crlf` dans le fichier `utils.c`, en regardant la spécification dans `utils.h`

--- * ---

Exercice 5 : Entrées bufferisées

Il vaut mieux essayer de ne pas faire trop d'appels système `read`, chaque appel fait faire un aller-retour dans le noyaux, c'est un peu coûteux. D'un autre côté, on veut vraiment lire des lignes. On va créer une petite bibliothèque pour les entrées bufferisées (voir TD6). L'idée est d'utiliser un tableau contenant les caractères lus avec `read` et de consommer les caractères du tableau (peut-être un par un, ici pas d'importance). Quand on a lu tous ses éléments, lire un caractère dans le buffer entraîne de nouveau un appel à `read`.

1. Écrire les définitions des fonctions dans le fichier `buffer/buffer.c`.
2. Écrire un fichier de test pour, au moins essayer convenablement, ces fonctions, par exemple pour bufferiser l'entrée standard.

--- * ---

Exercice 6 : Entrées bufferisées et lignes chez le client

Utiliser les fonctions de `buffer.c` et `utils.c` chez le client pour pouvoir lire des messages dans la socket de façon bufferisée. Ce n'est pas si simple. Quelques conseils :

1. Le moindre bug dans `buffer.c` est fatal ici et dur à détecter.
2. La condition `poll` ne suffit plus. Il ne faut pas attendre s'il reste des octets à lire dans le buffer. Il y a peut-être une ligne complète à traiter déjà dans le buffer, où le début d'une ligne, auquel cas la suite arrivera nécessairement (si possible en attendant).
3. Il reste à écrire le message au serveur. Attention au nombre d'octets à écrire.
4. Tester avec plusieurs clients.

--- * ---

4 Le protocole *freescord*

TODO