Projet Unix n°4 :
Création d'un serveur de messagerie (threads)

Introduction

Il s'agit ici de créer un serveur de messagerie mettant en relation les clients par paire. De plus les échanges de message entre deux clients se font au tour par tour : le serveur invite donc successivement le client soit à saisir un message soit à attendre le message de son correspondant.

Le protocole utilisé ici est TCP, et les aspects de synchronisation et de concurrence sont gérés via les threads UNIX.

Guide d'utilisation

Exécuter la commande make dans le répertoire du projet.
Pour exécuter le serveur, exécuter le binaire serveurchat:
Ex: `./serveurchat`

Pour exécuter le client, exécuter le binaire clientchat
Ex: `./clientchat <ip_du_serveur> 33016`

Architecture du projet

Le projet ne contient que deux fichiers : le fichier serveur et le fichier client, qui gèrent entièrement toutes les fonctionnalités.

I. Fichier Serveur

Le fichier serveur contient l'intégralité des fonctions relatives au serveur.

On décrit une structure `client_t` stockant l'identité et le statut d'un client connecté au serveur.
On stockera dans une variable globale `tab_clients` les clients tels que décrits par cette structure.

A l'exécution du binaire, le serveur affiche la listes des interfaces réseaux exploitant TCP sur la machine ainsi que le PID du processus. Cela nous est utile par la suite pour connaître l'adresse IP à utiliser côté client pour se connecter au serveur. L'affichage du PID est utile pour faciliter l'envoi de signaux. Puis le serveur entame sa boucle principale qui consiste simplement à attendre des connexions.

a/ Traitement d'une connexion

Dès qu'une connexion est faite sur le serveur, ce dernier souhaite la bienvenue au nouveau client puis crée un thread dédié au traitement dudit client avec la fonction routine `traiter_client`.
Dans un premier temps, le serveur affiche chez le client son message de bienvenue et lui demande de choisir un pseudo (qui sera validé, ou non, par la fonction `valider_pseudo_alphanum`).
Dès lors, il enregistre l'identité du client dans une structure `client_t` avec un statut « en attente » dans `tab_clients` via la fonction `ajouter_client`. On exécute alors la fonction

`deux_clients_en_attente` qui renvoie 0 ou 1 respectivement si deux clients sont présentement en attente de correspondant ou non.

Deux scénarios sont donc possibles:

- Soit un client est déjà en attente sur le serveur, et en ce cas l'appel à `deux_clients_en_attente` aura stocké dans une variable `fd_clients_en_attente` les descripteurs de fichiers correspondant aux deux clients en attente. Le serveur lance alors une discussion entre ces derniers à l'aide de la fonction `chat_prive`.

- Soit il n'y a aucun client en attente et rien ne se produit. Le thread jusqu'ici associé au nouveau client est fermé.

b/ Chat Privé

A l'initialisation d'un chat privé entre deux clients, on passe leur statut respectif à « en chat ». On annonce ensuite à chacun le pseudo de son correspondant. La gestion du tour par tour s'effectue à l'aide de la variable entière `numero_tour` qui s'incréments à chaque message échangé. Alors si `numero_tour` est pair alors c'est au premier client d'écrire son message tandis que le second client est invité à attendre, et inversement dans le cas impair.

Lorsque le message saisi par un client est « exit » ce dernier quitte le chat et ferme sa connexion au serveur par un appel à la fonction. Dans ce cas, la fonction supprimer_client est appelée afin de supprimer l'identité du client de `tab_clients`. Son correspondant est alors informé du départ par un message et son statut repasse à « en attente ». La fonction `deux_clients_en_attente` est ensuite de nouveau appelée, afin de vérifier si un autre client n'est pas déjà en attente afin de les mettre en discussion. Sinon, on sort de la boucle du tour par tour et le thread est fermé.

c/ Signaux

L'envoi du signal SIGUSR1 envoie le message « On ferme ! » à chacun des clients connectés, avant de fermer leur socket respective puis la socket du serveur.
Cette fonctionnalité est réalisée par l'affectation de la fonction handler `on_ferme` sur le signal USR1. Cette dernière parcourt `tab_clients` et, pour chaque case non nulle, envoie le message « On ferme ! » au client correspondant avant d'appeler la fonction `supprimer_client` sur ce dernier.

L'envoi du signal SIGUSR2 permet l'affichage du tableau des connexions clients, ainsi que le nombre total de connexions en cours.
Cette fonctionnalité est réalisée l'affectation de la fonction handler `nb_connexion` sur le signal USR2. Cette fonction parcourt `tab_clients` et compte les cases non nulles (dont le nombre correspond aux nombres de connexions en cours), puis appelle `afficher_tab_clients` qui affiche `tab_clients` côté serveur. Enfin, cette fonction affiche sur le serveur le nombre de connexions en cours.

II. Fichier Client

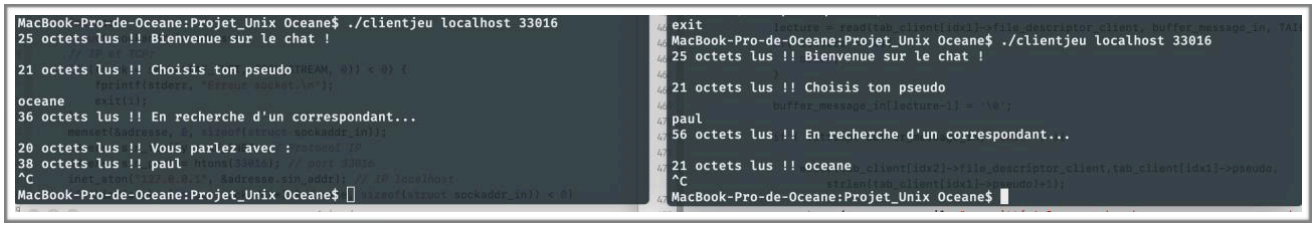
Le fichier client permet de se connecter au serveur de discussion.

Il contient donc, en premier lieu, une fonction qui crée la socket de contact : `cree_socket_tcp_ip_client`.

Il dispose d'un thread destiné à l'écriture et l'envoi de messages au serveur et d'un thread destiné à la lecture qui est à l'écoute des messages envoyés par le serveur.

Le thread principal est ici celui d'écriture, et le thread secondaire appelle la fonction `ecouter_serveur`.

Observation notée:
Lors de l'exécution de ce programme, le thread lecture affichait des messages tronqués :



On remarque bien une différence entre le nombre d'octets lus dans le buffer et la longueur du message affiché.

Le protocole TCP enverrait donc dans le buffer tous les messages d'un seul coup.
Si le serveur envoie par exemple deux messages :« Bonjour » puis « Bienvenue »

Le tableau buffer côté client contiendra « `Bonjour\0Bienvenue` » (sur les premières cases du tableau). Le nombre d'octets lus sera bien 16 (longueur de Bonjour et Bienvenue), mais la chaîne de caractères associée s'arrêtera au premier `\0`. On lira donc côté client « Bonjour » uniquement (en utilisant les fonctions `strcpy` et `print`), malgré la lecture de tous les messages (confirmée par le nombre d'octets lus).

Afin de pallier ce problème, le thread consacré à la lecture du fichier client va « dépiler » le buffer: on récupèrera au fur et à mesure le contenu du tableau buffer tant que le nombre d'octets lus ne correspondra pas à la taille de la chaîne de caractère buffer.

NB : l'appel de `length(buffer)` rendra la longueur de buffer interprété en tant que chaîne de caractères, donc de la première case à la première case contenant '`\0`'. L'appel de `size(buffer)` rendra la longueur de buffer en tant que tableau.

Voies d'amélioration

Une amélioration envisagée a été de créer une option sur le serveur afin de choisir son mode de fonctionnement :
- En chat privé (comme ici)
- En chat général, soit une salle générale de discussion instantanée. La fonction `envoyer_message_a_tous` a été créée en ce sens.

A noter que suite à l'écriture du message exit, ou suite à l'envoi du signal SIGUSR1, le client doit appuyer deux fois de plus sur la touche entrée pour que le programme s'arrête. Le comportement étant par défaut, il n'a pas été modifié ici. On aurait pu ici changer le protocole de fermeture de socket, afin que cette dernière soit faite directement dans le fichier client.