Projet de programmation système en C/C++
Salon de discussion

Ben-David Malyane 471086, Anton Romanova 521935

19 décembre 2021

1 Introduction

Dans le cadre du projet "Salon de Chat" du cours INFO-F-201, nous avons dû implémenter un salon de chat, avec un client et un serveur pour le système d'exploitation Linux, bien que notre implémentation fonctionne tout aussi bien sur d'autres systèmes d'exploitation *nix. Notre programme permet à plusieurs clients d'échanger des messages dans un salon de chat par l'intermédiaire d'un serveur.

2 Choix d'implémentation

2.1 Serveur

Le serveur porte deux rôles.

- Gérer les connexions et déconnexions des clients
- Transmettre les messages envoyés par les clients à tous les clients connectés.

Le serveur a été, en grande partie conçu en orienté-objet afin de faciliter l'implémention. La classe ChatRoom, singleton, y représente toute la partie logique du salon de chat.

Après avoir appelé le constructeur de cette classe, les signaux SIGPIPE seront ignorés. Nous sommes conscients que cette implémentation n'est pas très propre, plus particulièrement si c'était un grand projet. Il aurait été préférable de ne pas ignorer tous les signaux `SIGPIPE` du programme, mais plutôt appeler `pthread_sigmask` avant l'appel système `write`. Tout de même, suite à la petite taille du projet, ainsi que la certitude que ce code, ayant été réalisé dans un cadre purement académique, n'évoluera pas, nous avons décidé de laisser ceci tel quel.

En plus du signal `SIGPIPE`, la classe ChatRoom gère le signal `SIGINT`. Lorsque celui-ci est envoyé au programme, le handler `void ChatRoom::SigHandler(int sig)` est appelé. Celui-ci met la variable d'instance volatile et atomique `should_stop_` à `true`. Ainsi, le serveur s'arrêtera après avoir traité tous les messages et avoir envoyé un message avertissant tous les clients de sa future mort.

Pour gérer la communication simultanée avec plus d'un client sans créer plusieurs threads, nous avons utilisé la fonction de multiplexing d'entrée-sortie, select.

Il n'est pas interdit pour deux (ou plusieurs) clients d'avoir un nom d'utilisateur identique. Cette interdiction aurait pu très simplement être implémentée en utilisant la structure `std::unordered_set` plutôt que `std::vector` dans la classe `ClientSet`, ainsi qu'en ajustant les opérateurs d'égalité et la fonction de hachage de la structure `Client`.

message(Packet)	
champ	type
longueur du message	size_t
Timestamp	time_t
Message	char*

Table 1 – Structure d'un message

Bien que ceci n'ait pas été demandé, nous avons trouvé important que les noms d'utilisateurs soient filtrés. Ainsi, lorsque le client envoie son nom d'utilisateur, tous les caractères non-alphanumériques sont ignorés. Ce filtrage est effectué à l'aide d'un appel à `std::isalnum`. Le résultat du filtrage dépend donc de la locale C actuelle.

Le format des messages redistribués aux clients par le serveur est une simple chaîne de caractères telle que "username sent a message at Sun Dec 19 22 :21 :41 2021
> message"

2.2 Client

Le client est l'application que chaque utilisateur va utiliser pour discuter avec d'autres personnes. Le client se présente sous forme d'un programme en ligne de commande.

Dans la logique d'une application client-serveur, notre client devait être capable d'envoyer des messages au serveur (parler) et simultanément pouvoir recevoir des messages du serveur (écouter).

Pour ce faire nous avons décide de simuler premièrement le parallélisme du processus d'envoi et de réception du client en créant un thread d'écoute en plus d'un thread d'envoi.

Un message complet envoyé au serveur devait être formatté dans un certain ordre comme représenter dans le TABLE1.

Dans le but de nous conformer à cette exigence fonctionnelle, nous avons fait le choix d'implémenter une structure Argument présentée comme suit :

```
struct UserParameters {
  char *pseudo;
  char *ServerIp;
  port_t ServerPort;
};
```

Il nous a également été demandé de gérer les cas de terminaison du client. Le client peut s'arrêter soit par la réception d'un `EOF` c'est-à-dire de la combinaison des touches `"Ctrl+D"` sur l'entrée standard quand le client veut se déconnecter, soit par la perte de la connexion avec le serveur.

Dans le cas :

- `EOF`
  Le client doit pouvoir se déconnecter du serveur en entrant la combinaison des touches "`Ctrl+D`". Notre choix a été de stocker dans une variable la valeur retournée par l'appel à la fonction `fgets`. Cette fonction renvoie un `nullptr` lors de la réception d'un `EOF`. Par ce mécanisme nous vérifions chaque input de l'utilisateur au moyen d'une structure conditionnelle qui lors d'une vérifications à succès permet la déconnexion du client en lui envoyant un signal `SIGINT`.

- Perte de connexion avec le serveur
  Ici nous avons supposé que la perte de connexion avec le serveur ne survenait que lorsque le serveur

s'arrêtait avec le signal `SIGING`. Avant de s'arrêter le serveur envoie un dernier message au client lui annonçant sa terminaison. Coté client, le message est réceptionné et est évalué afin de détecter s'il s'agit d'un message ordinaire ou d'un message de terminaison du serveur.

Dans chacun de ces cas, un message est affiché à l'utilisateur pour une expérience plus agréable en cas de terminaison.

3 Limitations

Nous n'avons malheureusement pas pu finir à temps l'interface terminale pour notre salon de chat utilisant la lib ncurses. La lib est bien documentée mais suite à un manque de temps, nous avons préféré donner la priorité à l'optimisation et la qualité de notre code plutôt que l'interface utilisateur.

La taille maximale d'un message est définie par la constante `MAX_MESS_SIZE`. Le struct `Message` est défini tel que `sizeof(Message) = MAX_MESS_SIZE` Lors de l'envoi du `Message` du client au Serveur, les `MAX_MESS_SIZE` bytes sont envoyés. Ceci est donc peu efficace, surtout dans le cas où des messages courts sont envoyés. De plus, si un message plus long que la limite imposée est envoyé, l'utilisateur n'en sera pas averti.

Une autre limitation est la gestion d'erreurs assez rudimentaire. En effet, pour la plus grande partie des appels système, la macro `TRY` est utilisée. Dans le cas où la valeur de retour de la fonction passée en paramètre est inférieure à 0, une description de l'erreur est affichée sur le `stderr` et le programme sort avec le code de sortie passé en paramètre de la macro. Cette manière de gérer les erreurs ne peut pas toujours être qualifiée de "user-friendly".

De plus, suite à l'utilisation de l'appel select pour le multiplexing, sur Linux, nous sommes limités à 1024 file descriptors. Bien que sur d'autres systèmes d'exploitation Unix, comme Darwin, nous pouvons pallier à ce problème en définissant `_DARWIN_UNLIMITED_SELECT`, sur Linux, il est préférable d'utiliser l'appel système `poll`.

Une autre limitation est la manière, côté serveur, d'avertir les clients de sa mort prochaine. En effet, cet avertissement se limite à l'envoi d'un message sous forme textuelle. Ainsi, du côté client, chaque message doit être comparé à l'aide de strncmp, ce qui, de plus d'être peu efficace, peut mener à des erreurs dans le cas où le code, ainsi que le protocole de communication devait évoluer. Il est tout de même impossible que ceci mène à des erreurs avec la version actuelle de notre programme car le format de messages clients redistribués par le serveur est différent.

La taille maximale d'un

4 Difficultés rencontrées

Dans l'ensemble, nous n'avons pas éprouvé des difficultés à implémenter le salon de chat si ce n'est l'ajout de l'interface terminale avec la librairie ncurses par manque de temps.

Bien que nous n'avons pas un code infaillible, nous avons essayé d'éviter et corriger le plus de failles de sécurité simples, telles que des buffers overflows liés à des messages ou noms d'utilisateurs trop longs.