Rapport de projet : Chat201 – édition thread & réseau

Daniel DEFOING (`ddef0003`) Belinda ÖZNUR (`bozn0003`)
Haluk YILMAZ (`hyil0003`)

19 décembre 2024

Table des matières

1 Introduction

Ce rapport décrit globalement la conception du second projet dans le cadre du cours de Systèmes d'Exploitation (INFO-F201). Il présente les choix de conception qui ont guidé notre développement, et les difficultés qui ont pu survenir durant celui-ci. Pour voir de façon précise les changements ayant eu lieu tout au long du projet et la contribution de chacun, veuillez consulter le repository GitHub de notre projet.

2 Choix du langage :pourquoi C++ plutôt que C ?

Des outils qui facilitent le développement en général

Une raison fondamentale qui a guidé cette décision est que C++ possède des fonctionnalités absentes en C tels que les références, les chaînes de caractères (ou *strings*, qu'on a souvent subsitués aux `char*[]`) ou encore les classes.
On peut aussi parler des conteneurs STL comme `std::vector` ou `std::queue` [1], très utilisés dans cette implémentation du projet.
Or, dans le cadre du projet présenté ici, ces éléments apportent une pluvalue non négligeable en permettant de structurer un code de façon plus fine qu'en C ou de simplifier grandement certaines opérations. L'exemple le plus trivial qu'on peut donner de ceci est le passage de paramètres par référence plutôt que par pointeur dans certaines fonctions, qui permet une gestion plus sûre de la mémoire.

Des librairies qui fournissent des abstractions utiles pour ce projet

On va illustrer ce point en parlant des *threads*. En langage C, on utilisera `pthread.h` [2] pour les gérer, alors qu'en C++ on a par exemple accès à la librarie standard `std::thread` [3], qui permet d'abstraire certaines opérations de la librarie en C (par exemple, accéder au thread courant avec `std::this_thread` [3]).
Utiliser C++ permet donc l'usage de certaines libraries standard absentes en C qui permettent d'abstraire des opérations de libraries *correspondantes* en C.

Pas de perte notable à ne pas utiliser le langage C

C++ reste évidemment compatible avec C : il n'existe pas d'opération en C infaisable exactement de la même façon en C++ [4]. De plus, les deux langages étant connus pour leur rapidité d'exécution, la performance du C++ n'est pas dégradée de façon significative par rapport au C dans le cadre de ce projet.
L'utiliser permet donc de tirer parti ses abstractions discutées plus haut (voir 2 sections précédentes) tout en gardant une très bonne efficacité.
Tout ceci montre que les avantages majeurs ont été trouvés à utiliser C++ plutôt que C, alors qu'aucun avantage n'a été identifié en faveur de l'utilisation de C par rapport à C++. Ce choix du langage s'est donc naturellement imposé comme le plus adapté.

3 Visualisation de l'architecture du programme

Cette section discute des choix de conception du programme final, des considérations qui permettront au lecteur de mieux comprendre la nature de ces choix et des problèmes rencontrés qui en ont suivi.

3.1 Le serveur en tant que *point relais* des clients

La structure d'un programme client-serveur tel que celui présenté ici peut être visualisée très simplement sous la forme d'un Graphe Étoile [5], avec le serveur au centre et les clients aux extrémités de chaque branche.
Cette représentation montre bien qu'un seul serveur se charge de *relayer* les informations à passer d'un client à un autre ou d'un client vers lui-même (confirmation de (dé)connexion notamment).

TODO : discuter du TCP/IP brièvement [6]

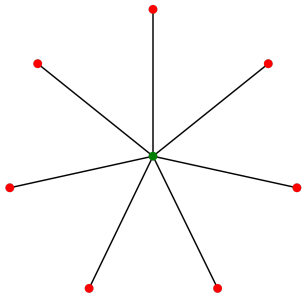

FIGURE 1 – *Graphe Étoile typique.*

Dans la suite, on verra en profondeur la façon dont les sommets de ce graphe communiquent.

3.2 Choix d'implémentation communs aux clients et au serveur

3.2.1 De la méthode d'envoi des messages

Il a été décidé d'abstraire l'échange d'informations entre les clients et le serveur principalement par une classe `Message`. Un point très important qui découle de ce choix d'implémentation est qu'on a utilisé des `std::string` [7] pour garantir une gestion dynamique de la mémoire et simplifier les opérations de manipulation des chaînes de caractère.

3.3 Conception des clients

3.3.1 Deux FIFOs (ou *files*) à vider en permanence

Tout client possède deux `std::queue` [1] qu'il observe en permanence : une pour les messages qu'il envoie, et une pour ceux qu'il reçoit.
Utiliser des `std::queue` n'ayant pas de taille limitée ainsi est une partie de la solution aux problèmes de synchronisation qui peuvent être rencontrés dans le cadre de ce projet. En effet, si un client reçoit plusieurs messages en même temps, les messages peuvent être *push* à l'arrière de la FIFO servant et traités séquentiellement par le client qui la videra progressivement.
Mais n'y a-t-il pas un risque que, si on envoie plusieurs messages simultanément à un même client, les deux messages risquent de corrompre la FIFO de réception du client (qui est un objet critique) en voulant s'insérer dedans en même temps ?
En principe, oui, c'est un problème qui peut survenir si on ne prend pas de précaution contre lui. C'est pour cela qu'on a fait usage de mutex (ou *exclusions mutelles*) [8] pour protéger la FIFO de réception côté client (éviter qu'elle reçoive plusieurs messages simultanément).

3.3.2 Deux threads, deux FIFOs par client

Ce point fait écho à la discussion de la section précédente sur l'usage de FIFOs côté client.
Il faut préciser que le processus côté client se divise en deux *threads* : un dédié à la réception des messages (qui doit donc vider et afficher le contenu de la FIFO de messages entrants) et un autre dédié à l'envoi de messages (qui doit donc en quelque sorte remplir la FIFO des messages à envoyer, puis la vider progressivement).
Cette division du processus client en deux threads est due à des contraintes évidentes de performance et autorise le client à envoyer et recevoir des messages simultanément (gestion asynchrone des communications).

3.3.3 Gestion de la (dé)connexion au serveur

Tout client doit suivre le protocole TCP [6] pour initialiser sa connexion et doit s'identifier auprès du serveur. (incomplet)

Gestion de la déconnexion (quel thread tue l'autre, comment on se déconnecte) TODO :

1. Signal de terminaison envoyé aux deux threads

2. Attente de la fin des opérations en cours
3. Libération des ressources
4. Fermeture propre de la connexion

3.4 Conception du serveur

3.4.1 De l'utilisation de `poll`

`poll` [9] est l'outil qui a été choisi côté serveur pour gérer les connexions et envoi de messages par les clients. Il fonctionne ainsi [9] :

1. Initialiser un `std::vector` de `pollfd` (autrement dit, un vecteur de descripteurs de fichier pour pouvoir établir les communications).
2. Chaque client qui se connecte au serveur lui fait ajouter une nouvelle entrée au `std::vector`. Symétriquement, lors d'une déconnexion, il faut le parcourir pour supprimer l'entrée dans le vecteur du client qui s'est déconnecté.
3. `poll` est *edge-triggered*, ce qui signifie que le serveur est "notifié" en permanence quand des données sont disponibles. [10] [11] Le serveur doit donc se charger de vider le vecteur de `pollfd` dès qu'il est notifié que des données à transmettre sont disponibles : en pratique, il *boucle* sur le vecteur pendant toute sa durée de vie.
4. S'il observe que, dans un `pollfd`, il y a un message à transmettre, alors il recherche à qui le client envoyeur a voulu transmettre son message [1] et, s'il existe, écrit dans le `pollfd` du client récepteur.

   Si le message à envoyer ne contient pas de nom de destinataire ou que le nom de destinataire spécifié ne fait référence à aucun client connecté, un message d'erreur est renvoyé.

4 Améliorations non réalisées de l'implémentation actuelle

4.1 `epoll` comme alternative à `poll`

`epoll` [12] fonctionne de façon assez similaire à `poll`, mais a deux différences majeures :

1. Il ne fonctionne que sur les systèmes d'exploitation Linux (ou basés sur Linux) ;
2. Il peut être *edge-triggered* (une "notification" est envoyée seulement lorsque des données sont disponibles) ou *level-triggered* (une "notification" est envoyée tant que des données sont disponibles). [10]
3. On peut aller chercher les données uniquement des *file desciptors* actifs (donc : ceux qui ont reçu un message) plutôt que de devoir itérer sur l'ensemble des *file descriptors* disponibles (ce qui est malheureusement nécessaire pour `poll`). [13]

En clair, `epoll` est cité comme une alternative plus *rapide* que `poll`, mais n'a pas été implémenté dans le cadre de ce projet car l'API est plus complexe à utiliser que celle de `poll`. Cela aurait tout à fait pu être fait en quelques jours supplémentaires cependant.

4.2 Attente active pour les FIFOs côté client

On l'a vu en section 3.3 (**??**), tout client est séparé en deux threads dont l'un gère une FIFO pour les messages entrants et l'autre les messages sortants. Cependant, l'implémentation actuelle de ce projet fait que le contenu des FIFOs est vérifié en permanence tant que le client est connecté [2]
Par manque de temps, un système plus propre n'a pas pu être mis en place, mais on peut en esquisser le fonctionnement théorique (qui aurait certainement pu, comme pour `epoll`, être implémenté en quelques jours) :

1. Initialement, la FIFO des messages entrants est vide. Dès qu'un message est envoyé au client, celui-ci commence à vider la FIFO.

1. En pratique, on recherche le nom du client récepteur dans un `std::unordered_map` avec le nom du client comme clé et son `pollfd` comme valeur.
2. On pourrait en quelque sorte parler d'*attente active* de la part du client. [14]

2. Chaque nouveau message entrant correspond à une nouvelle "notification" pour le client, à une nouvelle tâche à faire : on peut donc facilement avoir un "compteur de tâches à faire" qui indique combien de fois le client doit extraire un message de la FIFO. Par exemple, si un client reçoit 3 messages en même temps, il est censé être notifié 3 fois et donc `pop()` le premier élément de la FIFO 3 fois (ce qui vide exactement la FIFO, en principe).
3. Lorsque la FIFO est vide, le client n'a pas besoin de vérifier son contenu en permanence.

5 Difficultés rencontrées et solutions trouvées

5.1 Problèmes de synchronisation

— côté serveur, accès concurrents (problème producteur-consommateur)
— Signaux asynchrones (mentionner la conception du SignalManager dans l'explication de la solution)
— ...

5.2 Garantie de l'intégrité du contenu partagé par les clients

— Gestion des tailles limites (longueur de pseudos et de messages)
— Comment s'est-on assurés que le messages étaient bien transmis sans perte ?

Références

[1] `std::queue` - cppreference (dernière modification : 2024, 2 août)
URL : `https://en.cppreference.com/w/cpp/container/queue`

[2] pthreads(7) — Linux manual page (dernière modification : 2024, 15 juin).
URL : `https://www.man7.org/linux/man-pages/man7/pthreads.7.html`

[3] `std::thread` - cppreference (dernière modification : 2023, 24 octobre).
URL : `https://en.cppreference.com/w/cpp/thread/thread`

[4] StackOverflow - *Is there anything that can be done in C and not in C++ and the opposite way ? [closed]* (dernière modification : 2010, 9 décembre)
URL : `https://stackoverflow.com/questions/4403328/is-there-anything-that-can-be-done-in-c-and-n`

[5] Wikipedia : *Graphe étoile* (dernière modification : 2019, 21 janvier)
URL : `https://fr.wikipedia.org/wiki/Graphe_%C3%A9toile`

[6] Wikipedia : *Transmission Control Protocol* (dernière modification : 2024, 17 décembre)
URL : `https://en.wikipedia.org/wiki/Transmission_Control_Protocol`

[7] C++ Programming Language : `std::string` (dernière modification : inconnu)
URL : `https://cpp-lang.net/docs/std/containers/strings/string/`

[8] `std::mutex` - cppreference (dernière modification : 2024, 6 mars)
URL : `https://en.cppreference.com/w/cpp/thread/mutex`

[9] poll(2) - Linux manual page (dernière modification : 2024, 15 juin).
URL : `https://www.man7.org/linux/man-pages/man2/poll.2.html`

[10] StackOverflow - *Level vs Edge Trigger Network Event Mechanisms* (dernière modification : 2022, 28 août).
URL : `https://stackoverflow.com/questions/1966863/level-vs-edge-trigger-network-event-mechanisms`

[11] StackOverflow - *Is poll() an edge triggered function ?* (dernière modification : 2013, 25 février).
URL : `https://stackoverflow.com/questions/15072165/is-poll-an-edge-triggered-function?rq=3`

[12] epoll(7) — Linux manual page (dernière modification : 2024, 12 juin).
URL : `https://www.man7.org/linux/man-pages/man7/epoll.7.html`

[13] StackOverflow - *What is the purpose of epoll's edge triggered option ?* (dernière modification : 2022, 25 septembre).
URL : `https://stackoverflow.com/questions/9162712/what-is-the-purpose-of-epolls-edge-triggered-`
`rq=3`

[14] Wikipedia : *Busy Waiting* (dernière modification : 2024, 2 novembre).
URL : `https://en.wikipedia.org/wiki/Busy_waiting`

Toutes sources consultées pour la dernière fois le 21/12/24, 16h.