Rapport de projet : Chat201 – édition thread & réseau

Daniel DEFOING (`ddef0003`) Belinda ÖZNUR (`bozn0002`)
Haluk YILMAZ (`hyil0002`)

21 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 plus-value 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 librairie standard `std::thread` [3], qui permet d'abstraire certaines opérations de la librairie en C (par exemple, accéder au thread courant avec `std::this_thread` [3]).
Utiliser C++ permet donc l'usage de certaines librairies standard absentes en C qui permettent d'abstraire des opérations de librairies *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).
Dans la suite, on verra en profondeur la façon dont les sommets de ce graphe communiquent.

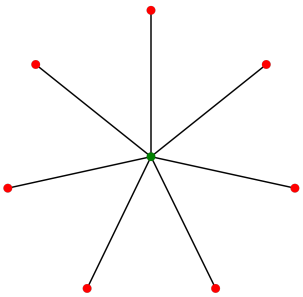

FIGURE 1 – *Graphe Étoile typique.*

3.2 Choix d'implémentation communs aux clients et au serveur

3.2.1 Du protocole de communication

L'utilisation du *Transmission Control Protocol* [6] comme moyen de communication client-serveur n'était pas un "choix" à proprement parler, mais il convient de mentionner que sa mise en place a été faite en suivant globalement la méthode décrite dans la séance de Travaux Pratiques associée [7]. Pour ce qui est de protocole de réseau, ce sont les adresses de famille IPv4 qui sont utilisés.

3.2.2 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` [8] 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 Capacités d'envoi et de réception : adaptation à la taille des données transmises

Tout client possède un socket pour communiquer avec le serveur. Ces sockets possèdent 2 buffers par défaut dont l'un pour l'écriture et l'autre pour la lecture. La taille d'un buffer change d'un OS ou d'une machine à l'autre. Cependant ces tailles vont des dizaines à des centaines de Ko. Étant donné que chaque message a au plus 1 Ko et que chaque message est traité très rapidement dés l'envoi ou la réception, ces tailles sont largement suffisants dans le cadre de ce projet. Si toutefois on arrivait à une situation de buffer plein, alors le serveur se chargera de déconnecter l'utilisateur en question.

Il serait utile de rajouter que la taille par défaut donnée par la machine pour ces buffers de socket IPv4 peut être augmenté. On peut multiplier jusqu'à des dizaines de fois ceux-ci en choisissant la valeur maximale. Nous ne l'avons pas fait dans le cadre de ce projet que les valeurs offertes par défaut sont suffisantes.

3.3.2 Gestion de l'aspect temporel de la communication via deux threads, un socket par client

La section précédente se focalise sur l'aspect taille du problème. Pour ce qui est de l'aspect temporel dans la gestion de ces 2 buffers, nous utilisons deux threads. Le thread noyau qui est créé en même temps que le processus est dédié à l'envoi de messages (qui doit donc en quelque sorte remplir le buffer des messages à envoyer, puis la vider progressivement) tandis que le second thread créé manuellement est dédié à la réception des messages (qui doit donc vider le buffer et afficher le contenu de messages entrants). Ainsi les deux buffers ne sont pas limités par l'éxécution de l'autre.
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. Si le client n'arrive pas à se connecter au serveur, il ré-envoie des demandes de connexion

périodiquement.
La déconnexion du serveur par un client suit ce processus :

1. Attente de la fin des opérations en cours.
2. Le client ferme ses sockets.
3. Le thread "non-noyau", servant à la réception des messages, est fermé en premier
4. Le thread principal est terminé à son tour, achevant le processus de déconnexion.

En cas d'erreur, un exit est provoqué avec le code d'erreur correspondant. Ceci ne pose pas de problème car il n'y a pas de ressources à gérer dynamiquement. Les données de l'instance sont supprimés, les sockets sont fermés par le système, les threads sont terminés automatiquement.

3.4 Conception du serveur : De l'utilisation de `poll`

`poll` [10] est l'outil qui a été choisi côté serveur pour gérer les connexions et envoi de messages par les clients. Il fonctionne ainsi [10] :

1. Initialiser un `std::vector` de `pollfd` (descripteurs de fichier) pour gérer les communications. Chaque entrée dans ce vecteur correspond à une socket connecté au serveur, y compris le socket principal qui gère les nouvelles connexions.
2. Pour chaque client qui se connecte au serveur, une nouvelle entrée est ajoutée au `std::vector`. Symétriquement, ors d'une déconnexion, il faut parcourir ce vecteur pour localiser et supprimer l'entrée correspondante.
3. `poll` est *level-triggered*, ce qui signifie que tant qu'un descripteur de fichier est prêt à être traité, l'entrée correspondante dans le vecteur seras signalée. Le serveur est donc "notifié" tant que les données sont disponibles, [11] [12] et il doit se charger de traiter explicitement les évenements signalés pour éviter que ces notifications ne persistent indéfiniment.
   En pratique, il *boucle* sur le vecteur pendant toute sa durée de vie.
4. Lorsqu'un événement de lecture est détecté sur une entrée du vecteur, le serveur lis le message du client. Il traite ensuite ce message, vérifie sa validité (taille, format..), puis il recherche à qui le client expéditeur a voulu transmettre son message [1] et, s'il existe, écrit dans le `pollfd` du client récepteur.

   Si le destinataire spécifié du message ne fait référence à aucun client connecté, un message d'erreur est renvoyé au client. Si le message à envoyer n'est pas dans le bon format, (au minimum 2 mot séparé d'un espace) le serveur ignore la requête. Si le message à envoyer est plus grand que 1024 octets, le client n'est pas déconnecté : l'énoncé du projet spécifiait que le client devait être déconnecté *si le serveur* ***recevait*** *un message excédant 1024 octets*, mais notre implémentation *prévient* cette éventualité en n'envoyant tout simplement pas les messages trop massifs. À la place, le client est averti que son message est trop long.

4 Améliorations non réalisées de l'implémentation actuelle

4.1 `epoll` comme alternative à `poll`

`epoll` [13] 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 *à l'instant* où des données sont disponibles) ou *level-triggered* (des "notifications" sont envoyées en continu tant que des données sont disponibles). [11]
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`). [14]

En clair, `epoll` est cité comme une alternative plus *performante* 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`. L'utilisation d'un mécanisme tel que `epoll` aurait certainement été indispensable dans le cadre d'un serveur qui devrait accueillir plusieurs milliers d'utilisateurs en simultané ou qui devrait transmettre des données plus volumineuses que du simple texte.

Cependant, l'implémentation d'`epoll` aurait tout à fait pu être fait en quelques jours supplémentaires.

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.

4.2 `boost::asio` comme alternative à `poll`

La première itération du programme visait à utiliser `boost::asio` [15], une bibliothèque puissante basée sur `epoll` qui fournissait une automatisation de certains traitements (comme la mise en place d'un *thread pool* côté serveur par exemple).
Cependant, nous avons constaté que son utilisation représentait une solution qui dépasse le cadre de ce cours pour un serveur ayant du être réalisé en 2 semaines, ayant du gérer un nombre restreint de connexions simultanées (1000), et traiter des messages contenant uniquement du texte brut. De plus, bien que très instructif, nous n'étions pas assez à l'aise avec tout les aspects et la richesse de cette bibliothèque dans le temps imparti. Cette situation nous a permis d'explorer d'autres solutions, comme `poll`, qui étaient plus adaptées dans le cadre de ce cours.
Malgré l'abandon de cette solution, l'utilisation de `boost::asio` nous a tout de même permis d'approfondir une multitudes de concepts, tels que le fonctionnement et l'optimisation de grand serveurs, les mécanismes asynchrones, `epoll`.

4.3 Utilisation théorique des FIFOs

On l'a vu en sections 3.1 et 3.2, tout client est séparé en deux *threads* dont l'un gère un buffer pour les messages entrants et l'autre un buffer pour les messages sortants. Cependant, l'implémentation actuelle de ce projet fait que le nombre de messages en attente d'être envoyés ou reçus sont limités par la taille de ces buffers. Pour se débarasser de cette limitation, on peut avoir recours aux `std::queue` ou FIFOs [1].
On peut esquisser le fonctionnement théorique d'un tel système (qui aurait certainement pu, comme pour `epoll`, être implémenté en quelques jours) :

1. Initialement, la FIFO des messages entrants serait vide. Dès qu'un message serait envoyé au client, celui-ci serait notifié et pourrait `pop()` un élément de la FIFO pour le traiter.
2. Chaque nouveau message entrant correspondrait à une nouvelle "notification" pour le client, à une nouvelle tâche à faire : on aurait donc facilement pu avoir un "compteur de tâches à faire" qui indiquerait combien de fois le client devrait extraire un message de la FIFO. Ce compteur serait naturellement décrémenté dès que le client aurait fini une tâche.

   Par exemple, si un client recevait 3 messages en même temps, il serait censé être notifié 3 fois et donc `pop()` le premier élément de la FIFO 3 fois (ce qui viderait exactement la FIFO, en principe).
3. Lorsque la FIFO serait vide, le client aurait donc en théorie réalisé toutes les tâches qui lui étaient assignées et n'aurait par conséquent pas besoin de vérifier son contenu en permanence.

5 Difficultés rencontrées et solutions trouvées

5.1 Problèmes de synchronisation côté serveur

Pendant le développement, l'hypothèse selon laquelle on aurait pu faire face à des problèmes de synchronisation côté serveur (typiquement, en recevant deux messages en même temps) était bien présente.

Cependant, dans l'implémentation actuelle, on peut écarter cette possibilité puisque le serveur traite les événements de manière séquentielle, en *mono-thread* grâce à `poll`, éliminant ainsi à la fois les risques d'accès concurrents et à la fois les problèmes d'ordre d'envoi et réception des messages [2]. Chaque événement est ainsi traité de manière séquentielle.

5.2 Asynchronie des signaux

Le programme présenté ici gère les signaux `SIGINT` et `SIGPIPE`. Les signaux étant asynchrones, une difficulté de ce projet était de les gérer pour :

— Éviter l'interruption inopinée d'opérations cruciales (écriture/lecture notamment)
— Permettre de définir les actions à exécuter en cas de réception d'un signal
— Garantir une *communication* adéquate entre les clients et le serveur (lorsqu'un client se déconnecte, il doit pouvoir le communiquer au serveur pour que celui-ci évite de relayer des messages dans le vide)

Il a donc été décidé de centraliser la gestion des signaux dans une classe dédiée, `SignalManager`, pour laquelle on peut lister ses choix de conception les plus importants.

2. Comprenez : si 3 messages A, B, C sont lus dans cet ordre, alors ils seront aussi envoyés dans l'ordre A, B, C.

Différence entre le mode normal et le mode `manuel`

Lors de l'arrêt du programme, le signal manager prend en compte l'absence (`SignalManager::signalHandler`) ou la présence du mode `manuel` (`SignalManager::signalHandlerManuel`).

Masquage des signaux pour le thread récepteur côté client

On en a déjà discuté dans ce rapport, mais tout client possède deux threads distincts, un pour l'envoi et un autre pour la réception de messages. Ce dernier possède un *masque* qui fait qu'il ignore tous les signaux qui lui parviennent. Ce choix a été fait pour centraliser la gestion des signaux côté client dans le thread principal.

Communication de signaux entre threads

Le masquage des signaux était une solution à la répartition des gestions des signaux par les threads. Mais il est également possible d'envoyer des signaux entre les threads. Ceci est possible car chaque thread possède un ID. Si le second thread qui s'occupe de la lecture via le serveur reçoit un signal qui indique que le serveur nous a déconnecté alors il est possible de prévenir le thread noyau pour qu'il s'occupe du reste via un `pthread_kill()` [3].

5.3 Situations de concurrence

Etant donné que deux threads sont utilisés dans le cadre de ce projet, il était nécessaire de gérer explicitement les accès cas de concurrence pour :

- L'affichage des messages envoyés ou reçu sur le terminal.
- La modification de la mémoire pour y ajouter ou retirer des messages.

Ceci a été réalisé grâce à l'utilisation d'un mutex différent pour chaque cas.

Deadlocks

Dans le cadre de ce projet, diverses situations de deadlocks/d'interblocages ont également fait surface. Il a donc fallu utiliser les threads et les mutex d'une façon réflechie pour éviter cela. Voici deux cas qu'il a fallu gérer/corriger :

- L'appel d'un méthode qui sert à déconnecter le client par le second thread. Ceci pose problème car la méthode responsable de la déconnexion essaye de joindre ce même thread pour le terminer. Il faut donc éviter d'appeler cette méthode par ce thread et seulement y faire appel par le thread noyau.
- Le blocage d'un mutex avant l'appel d'une autre méthode qui utilise lui même ce mutex dans le même scope. Ceci provoque un interblocage car quand la méthode est appelé et qu'il veut opérer il doit attendre que le mutex soit libéré indéfiniment.

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

5.4.1 Gestion des tailles limites

Pour garantir l'intégrité du contenu partagé par les clients, des limites strictes sur la taille des pseudos (maximum de 30 octets) et des messages (maximum de 1024 octets) ont été imposées à la fois coté client et coté serveur. En cas de dépassement de la taille du message, et que ce message parviens coté serveur malgré la protection coté client, le message est rejeté et le serveur déconnecte le client.

5.4.2 Fiabilité de la transmission des messages

Malgré toutes les précautions prises pour avoir un programme *safe*, il est toujours possible qu'on rencontre des cas où le client rencontre des erreurs (à cause d'un matériel défectueux, par exemple).

Ce cas d'erreur *externe, indéfinie* est également géré par notre programme (du moins en partie) : si un problème de ce type est détecté, le client problématique est déconnecté et si un autre client essayait de lui envoyer des messages (qui n'arrivaient pas à destination à cause de ces fameux problèmes *indéfinis*, justement), il est notifié du problème.

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-not-in-c-and-the-opposite-way

[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] GOOSENS J., MARKOWITCH O., cours de *Systèmes d'Exploitation* (ULB), TP n°6 : Programmation réseau

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

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

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

[11] 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

[12] 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

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

[14] 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-option?rq=3

[15] KOHLHOFF C., *Boost.Asio* (dernière modification : 2024)
URL : https://www.boost.org/doc/libs/1_87_0/doc/html/boost_asio.html

[16] 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.