Projet de programmation Multitâches

Application de messagerie instantanée

Adam Belghith, Tunui Franken, Maroua Ombaya

Dernière compilation : 6 novembre 2022 à 20:04

Table des matières

1 Description du projet, objectifs **3**

2 Choix du langage **3**

3 Quickstart **3**

4 Architecture fonctionnelle **4**

5 Protocole de communication **4**
5.1 Symbole de fin de transmission . . . 4
5.2 Initialisation de la connexion . . . 5
5.3 Fonctionnement des échanges . . . 5
5.4 Quelques exemples de transmission . . . 6

6 Architecture technique **6**
6.1 Les clients . . . 6
6.2 Le serveur . . . 9
6.3 Les exclusions mutuelles . . . 14
6.4 Les messages de debug/log . . . 15

7 Conclusion **15**

1 Description du projet, objectifs

Cette application multitâches, codée en C, est une illustration sur un cas concret des problématiques liées à :

— la synchronisation d'une application client/serveur
— l'exclusion mutuelle
— l'interblocage de processus

Il s'agit d'une application de messagerie instantanée qui permet à plusieurs utilisateurs de communiquer par l'intermédiaire d'un serveur.

Le code source peut être trouvé ici :

`https://git.tunuifranken.info/efrei/projet-multitaches`.

2 Choix du langage

Nous avions le choix entre C et Java pour le code de cette application.

Le langage C, plus bas niveau, nous a paru intéressant pour deux raisons :

1. Il permet de savoir précisément ce que l'on fait et donc de contrôler d'avantage de choses. Il est également plus dangereux, dans le sens ou il ne pardonne pas les erreurs de gestion de mémoire ou d'implémentation. La gestion des erreurs a été une fin en soi.
2. D'un point de vue pédagogique, l'absence d'objets et de fonctions de très haut niveau nous a obligés à prendre en considération un grand nombre de détails nécessaires à l'implémentation d'une telle application.

3 Quickstart

La compilation du programme se fait grâce au `Makefile` fourni. Il n'a été testé qu'en environnement GNU/Linux. Il est recommandé d'utiliser un compilateur compatible `C99` à cause de la syntaxe des macros de debug et de log (voir 6.4).

En plus de compiler le programme, le `Makefile` compile le présent rapport avec `latexmk`. Ne pas hésiter à lancer les recettes séparément si LaTeX n'est pas installé.

Une configuration du programme est possible avant compilation dans le fichier header `src/config.h`. Des explications y sont sous forme de commentaires.

Tous les logs écrivent sur `stderr`. Ainsi, si l'on veut supprimer les logs de la sortie standard et les rediriger vers un fichier pour n'avoir que le déroulement du programme :

```
$ ./server 2> server.log
$ ./client localhost 2> client.log
```

Le socket utilisé est un socket IPv4. Le client et le serveur ne sont donc pas obligatoirement sur la même machine. Le port d'écoute par défaut est 9999, et le client intègre la résolution du nom de domaine.

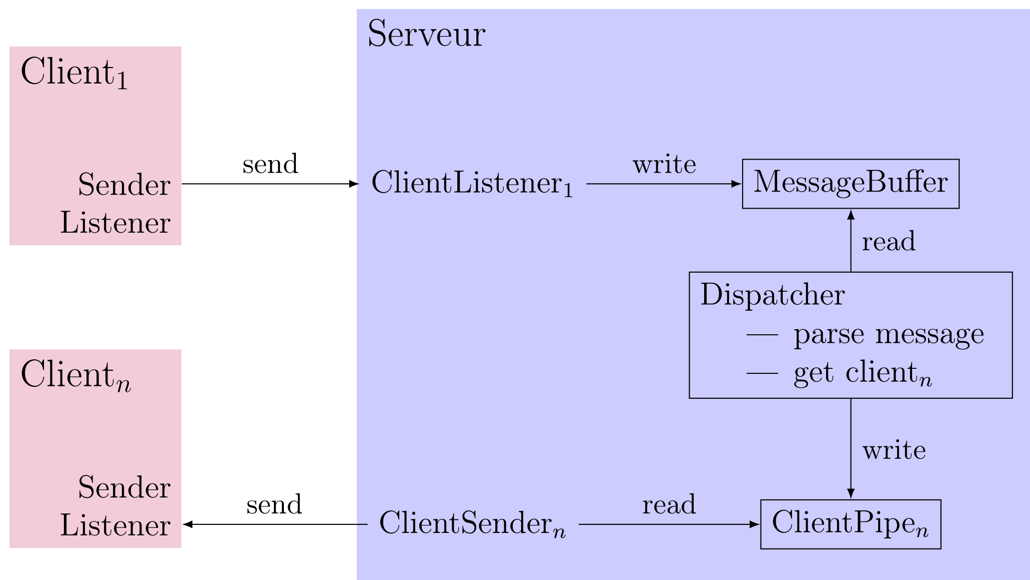
4 Architecture fonctionnelle

Le projet est composé de deux exécutables :

1. `server`, qui utilise plusieurs threads :
   — `Dispatcher` : gère le parsing des messages et le routage vers les bonnes fonctions et les bons clients.
   — `ClientListener` : est à l'écoute des messages entrants.
   — `ClientSender` : envoie les messages au client.

   Le serveur crée une instance de `ClientListener` et une instance de `ClientSender` par client connecté.
2. `client`, composé de deux processus :
   — `Listener` : est à l'écoute des messages en provenance du serveur, les imprime sur la sortie standard.
   — `Sender` : lit les messages sur l'entrée standard et les envoie au serveur.

Un envoi de message peut être schématisé de la manière suivante :



5 Protocole de communication

Un protocole de communcation a été défini pour que le client et le serveur puissent correctement échanger les données. Ainsi, le fichier header `common.h` contient des définitions de commandes qui correspondent chacune à un aspect fonctionnel du programme.

5.1 Symbole de fin de transmission

Pendant l'exécution du programme, le client lit sur la sortie standard jusqu'au premier retour à la ligne ('`\n`'). Le serveur attend donc le '`\n`' comme signe d'une fin de transmission.

Lors de l'initialisation de la session, le client et le serveur vont s'échanger quelques messages avec '`\n`' comme caractère de fin de transmission. Mais pour la suite des échanges, le '`\n`' ne pourra pas être utilisé comme symbole de fin de transmission par le serveur, car le message à envoyer peut contenir des retours à la ligne (par exemple `cmd_help`).

Le caractère `ETX` (*end of text*), qui correspond au '`\03`', sera alors utilisé pour les messages en provenance du serveur.

5.2 Initialisation de la connexion

Lors de la connexion du client, celui-ci utilise deux commandes avant de poursuivre son exécution :

— `cmd_hello` : envoie le caractère `CMD_HELLO` (H), suivi d'un retour à la ligne.

client ——— '`H\n`' ———→ serveur

Le serveur répond par l'identifiant du client :

client ←——— '`23\n`' ——— serveur

— `cmd_get_name` : envoie le caractère `CMD_GET_NAME` (N), suivi d'un retour à la ligne.

client ——— '`N\n`' ———→ serveur

Le serveur répond par le nom du client :

client ←——— '`alice\n`' ——— serveur

5.3 Fonctionnement des échanges

Maintenant que la session est initialisée, le client va simplement envoyer tel quel ce que l'utilisateur aura tapé. C'est à l'utilisateur de taper les bonnes commandes. Pour cela il y a la commande `CMD_HELP` (h).

Les messages reçus par le serveur sont découpés grâces aux espaces, puis le premier caractère de la commande est analysé. Ceci autorise l'utilisateur à faire certaines fautes de frappes : `h`, `help` ou `hippopotame` donneront tous trois la commande `cmd_help`.

Côté serveur, le `ClientListener` va transmettre le message au `MessageBuffer` après l'avoir préfixé de l'identifiant du client suivi d'un espace.

Le `Dispatcher` pourra alors parser le message qui doit être de la forme suivante :

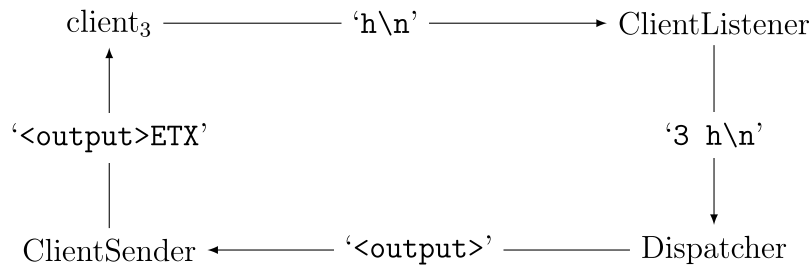
`'<id> <commande> [<sous-commande>] [<suite>]\n'`

Le message envoyé au client par le serveur sera alors de la forme :

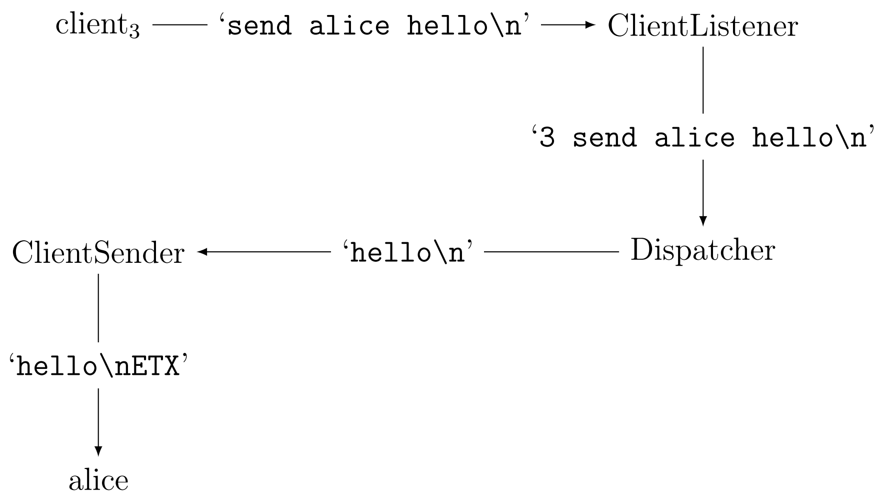
`'<output>ETX'`

5.4 Quelques exemples de transmission

`CMD_HELP` (h)



`CMD_SEND_MESSAGE` (s)



6 Architecture technique

6.1 Les clients

Un client doit essentiellement gérer deux tâches parallèles : le `Sender` et le `Listener`. Le client est donc composé de quelques fonctions nécessaires pour cela :

`main`

Point d'entrée de tout programme C, la fonction `main` commence par vérifier les arguments passés en ligne de commande. Ensuite, elle ignore les signaux d'interruption, puisqu'ils seront gérés dans les processus fils. Puis un socket est créé pour la communication avec le serveur.

Le prompt est instancié dans un espace de mémoire partagée (voir 6.1.1).

Le `main` crée alors un sémaphore nommé (voir 6.3.1). Pour éviter que différents clients éventuellement lancés sur la même machine partagent le même sémaphore, l'identifiant du client est alors récupéré pour être utilisé dans ce nom.

C'est aussi le moment de récupérer le nom du client (qui par défaut vaut l'ID), pour l'affecter au prompt.

Il ne reste plus qu'à faire deux `fork` successifs, pour le `Listener` et pour le `Sender`. Le `main` pourra alors terminer son exécution quand les deux processus auront terminé. Pour cela, un premier `waitpid` est lancé et va récupérer le numéro du processus ayant retourné en premier. En fonction du processus retourné, l'autre est alors tué.

Avant de terminer l'exécution du `main`, une fonction est lancée pour quitter proprement le programme.

`Listener`

Cette fonction regroupe toutes les actions réalisées par le processus fils correspondant au `Listener`. On commence par la gestion des signaux d'interruption `SIGINT` et `SIGTERM`. Le handler appelé à la réception de ces signaux est décrit par la fonction `interrupt` (voir plus bas).

Le reste de la fonction consiste en une boucle infinie. On lit caractère par caractère ce qui arrive sur le socket créé dans `main`. Pour cela, on commence par une lecture d'un caractère, sachant que la fonction est bloquante. Dès que quelque chose est reçu, le sémaphore est bloqué (`sem_wait`) et on démarre une boucle de lecture jusqu'à un caractère de fin. Chaque caractère lu est analysé puis envoyé sur `stdout`. Le caractère `EOT` entraîne une fermeture de session, et le caractère `ETX` indique une fin de flux, ce qui déclenche la mise à jour du prompt et son impression. Le sémaphore est alors libéré (`sem_post`) et la boucle infinie redémarre.

L'utilisation de deux appels distints à `recv` a été rendu nécessaire par l'utilisation du sémaphore. Le premier, bloquant, doit être fait avant le `sem_wait`, sinon le `Listener` s'approprie le sémaphore aussi longtemps qu'il en a l'occasion. Le second appel à `recv` est rendu non bloquant pour permettre de libérer le sémaphore en fin de flux.

`Sender`

Tout comme pour le `Listener`, la fonction `Sender` est exécutée par le processus fils correspondant au `Sender`.

Nous avons le même système de gestion des signaux qui permettra de quitter proprement l'application.

La boucle infinie est ici beaucoup plus simple que celle du `Listener`. On commence par bloquer le sémaphore pour pouvoir écrire le prompt, puis le libérer tout de suite après. Cela peut paraître court, mais il s'agit essentiellement de ne pas interrompre l'écriture du `Listener`.

La suite consiste simplement à lire une ligne complète (terminant par '`\n`') grâce à `fgets`, puis de l'envoyer au serveur grâce au socket.

`handle_args`

Cette fonction permet d'arrêter très tôt l'exécution de notre programme si les arguments passés sont incorrects. Dans ce cas, une simple explication de l'invocation en ligne de commande est affichée :

```
usage: ./client <server_hostname|server_ip> [<server_port>]
```

La fonction `handle_args` permet également de renvoyer à `main` le port à utiliser pour le socket. En effet, il est possible de fournir ce numéro de port en `CLI` ou de l'omettre pour utiliser le port par défaut.

`clean_and_close`

Cette fonction est simplement appelée à la toute fin de la fonction `main` afin de permettre une fermeture propre de la session. La gestion des signaux (ignorés par le processus père et gérés seulement dans les fils) permet de s'assurer de l'exécution de cette fonction de nettoyage dans tous les cas.

Elle ferme le sémaphore, informe le serveur de la déconnexion (`cmd_bye`), fait un `munmap` pour libérer la mémoire partagée pour le prompt, puis ferme le socket.

`interrupt`

À part afficher un message de debug lors de la réception des signaux `SIGINT` et `SIGTERM`, cette fonction ne fait pas grand chose. Elle est surtout utile dans la mesure ou la structure `sigaction` permet de n'appeler ce handler que dans le processus fils, le père ignorant les signaux pour éviter de mettre fin à la session trop tôt.

`cmd_hello`

Les fonctions préfixées par `cmd_` correspondent à des commandes, ici des commandes internes (mais qui peuvent très bien être appelées par l'utilisateur).

La commande `cmd_hello` est utilisée pour deux choses :

1. Elle permet au client de recevoir son identifiant, utile pour la création du sémaphore.
2. Elle informe le serveur de l'arrivée du client, lui permettant ainsi de l'enregistrer dans sa base.

`cmd_bye`

Cette commande consiste simplement à envoyer le caractère correspondant à la commande `CMD_BYE` au serveur, lui permettant de libérer les informations de ce client dans sa base.

`cmd_get_name`

La commande `cmd_get_name` demande au serveur de renvoyer le nom d'utilisateur du client. Ce nom, qui peut être amené à changer pendant la session, est stocké et géré par le serveur.

Le client en a simplement besoin pour l'afficher dans le prompt.

6.1.1 Mémoire partagée

Le prompt utilisé par le client est modifié par le processus `Listener` et accédé par le processus `Sender`. Pas besoin d'exclusion mutuelle donc, mais l'espace mémoire pour cette chaîne de caractères doit être partagée.

6.2 Le serveur

Le serveur est beaucoup plus complexe que le client. Commençons par une définition des différents éléments qui entrent en jeu.

`MessageBuffer`

Le `MessageBuffer` est un pipe global, utilisé par le `Dispatcher` et tous les `ClientListener`. Il permet d'implémenter un buffer géré par le kernel. Ce buffer contient les messages lus par le `ClientListener` et qui seront récupérés par le Dispatcher pour traiter les messages un à un.

`ClientPipe`

Le `ClientPipe` est un pipe qui permet la communication entre le `Dispatcher` et les différents `ClientSender`. Chaque instance de `ClientSender` a accès en lecture au pipe affecté à son client. Le `Dispatcher`, quant à lui, utilise la liste globale des clients pour écrire dans le pipe du bon client.

Liste des clients (`struct client_details`)

La liste des clients est une liste globale qui contient pour chaque client les informations le concernant. Pour cela, un `struct` contient les éléments suivants pour représenter un client :
- `id` : L'identifiant du client, qui doit être égal à l'index de la structure dans la liste. Ceci permet d'accéder à un client particulier en indexant simplement son `id` et d'affecter rapidement un nouveau client à l'index libre le plus bas. Ce champ vaut -1 quand la place est libre.
- `name` : Le nom d'usage du client, vaut l'`id` du client par défaut.
- `sockid` : Le socket de communication avec le client.
- `pipe` : Instance de `ClientPipe` pour la communication entre le `Dispatcher` et le `ClientListener` de ce client.
- `listener_thread` : Le numéro du thread créé pour le `ClientListener`. Il est utilisé pour permettre la déconnexion d'un client en particulier en mettant fin au thread.
- `sender_thread` : Le numéro du thread créé pour le `ClientSender`. Il est utilisé pour permettre la déconnexion d'un client en particulier en mettant fin au thread.

La longueur de cette liste est prédéfinie grâce à la constante `MAX_CLIENTS`.

Liste des groupes (`struct group_details`)

La liste des groupes et aussi une liste globale du même type que la liste des clients. Sa longueur est définie à `MAX_GROUPS` et chaque `struct` la composant contient les éléments suivants :
- `id` : L'identifiant du groupe, qui fonctionne de la même façon que pour la liste des clients.
- `name` : Le nom affecté au groupe lors de sa création.

— `clients` : Une liste de longueur `MAX_CLIENTS`, qui est une table de vérité. Chaque index dans cette liste identifie un client, et le booléen indique la présence ou non du client dans le groupe. Ceci permet de ne garder aucune information de groupe dans la liste des clients, et de vérifier très rapidement si un client fait partie d'un groupe, sans parcourir de liste.

Mutex des `ClientListener`

Comme les `ClientListener` de chaque client peuvent à tout moment écrire dans le `MessageBuffer`, un mutex global est créé.

`ClientListener`

Le `ClientListener` est un thread qui gère la réception des messages en provenance du client associé. Chaque message reçu est transmis au `Dispatcher` par l'intermédiaire du `MessageBuffer`.

`ClientSender`

Le `ClientSender` est un thread qui lit les messages dans le `ClientPipe` et les envoie au client qui lui est associé.

`Dispatcher`

Ce thread a la plus grosse part de travail. Il lit un à un les messages du `MessageBuffer`. Chaque message est parsé puis routé vers la bonne fonction pour un traitement dépendant de ce qui est reçu.

6.2.1 Les fonctions utilisées

`main`

De la même façon que le client, la fonction `main` récupère le port d'écoute via la fonction `handle_args`. Ensuite, la liste des clients est initialisée avec des valeurs par défaut, qui permettront d'établir quel index est libre. De même, la liste des groupes est initialisée.

Puis, le socket est créé pour la communication, et le pipe du `MessageBuffer` est défini. Le mutex global est initialisé, il sera utilisé par chaque instance de `ClientListener`.

Le thread du `Dispatcher` est alors lancé avant la gestion des signaux, pour éviter que le Dispatcher en hérite.

On entre alors dans la boucle d'acceptation des connexions entrantes. Lors d'une connexion, on vérifie qu'il y a bien une place libre dans la liste des clients. On crée alors un `ClientPipe` et on l'affecte au client lors de l'enregistrement du client dans la liste globale.

Il ne reste alors plus qu'à démarrer les threads pour `ClientSender` et `ClientListener`, et les ajouter à l'enregistrement du client.

`get_next_free_client_id` et `get_next_free_group_id`

Respectivement pour la liste des clients et la liste des groupes, ces fonctions parcourent la liste et retournent le premier index libre trouvé. Si la fonction retourne `MAX_CLIENTS` (ou `MAX_GROUPS`), cela veut dire qu'aucune place n'est libre.

`get_client_id_from_name` et `get_group_id_from_name`

Permettent de retrouver l'identifiant d'un client (respectivement d'un groupe) grâce à un nom. Ces fonctions sont notamment utilisées par la commande `cmd_send_message` pour l'envoi via un nom.

`create_group`

Cette fonction reçoit un nom pour créer un groupe de ce nom. Elle vérifie d'abord si la liste des groupes n'est pas pleine, puis qu'un tel groupe n'existe pas déjà. Si le groupe peut être créé, la structure correspondant au premier index libre est remplie avec les valeurs adéquates.

La fonction retourne l'identifiant du groupe créé. Ceci permet de signifier que le groupe n'a pas pu être créé en renvoyant -1.

`delete_group_if_empty`

Inverse de `create_group`, la fonction `delete_group` reçoit un identifiant de groupe pour le retirer de la liste des groupes. La suppression du groupe a lieu seulement si le groupe est vide.

`disconnect_client`

Lors de la déconnexion d'un client, le serveur a du ménage à faire. Cette fonction commence par vérifier que le client existe bien, puis le retire de tous ses groupes. Pour cela, la liste des groupes est à parcourir une fois, en mettant le booléen du client à 0 dans la liste des clients de chaque groupe. De plus, si un groupe contient le client à déconnecter, la fonction appelle `delete_group_if_empty` pour supprimer le groupe s'il est nouvellement vide.

Puis, les threads `ClientListener` et `ClientSender` sont terminés.

Enfin, les champs du `struct` du client dans la liste des clients sont remis à leur valeur par défaut, après avoir fermé les file descriptors du socket et du `ClientPipe`.

`Dispatcher`

La fonction `Dispatcher` est appelée par le thread `Dispatcher`.

Dans une boucle, les messages sont lus depuis le `MessageBuffer`. Chaque message finit par '`\n`', puisqu'il a été envoyé depuis la ligne de commande du client.

Commence alors le parsing. Nous avons décidé de ne pas créer de fonction séparée pour parser les messages, car le message peut être parsé en plusieurs fois. Cela permet au `Dispatcher` d'avancer dans le routage au fur et à mesure de ce qui a déjà été parsé.

On commence donc par extraire l'identifiant du client émetteur. Vient ensuite la commande. Les commandes ont été définies par un seul caractère à chaque fois. Le message arrivant étant découpé par espaces, le `Dispatcher` prend en compte seulement le premier caractère et ignore les autres. Ceci permet d'être résilient aux fautes de frappes.

Le reste de la fonction consiste en un `switch`. En fonction de la commande reçue, la fonction correspondante est appelé avec, si besoin, un traitement préliminaire.

Un exemple de parsing en plusieurs fois peut être trouvé pour la commande `cmd_list`. Une sous-commande `cmd_list_groups` existe, mais n'est évidemment pas vérifiée si l'on ne se situe pas dans la commande `cmd_list`.

Le parsing pour la commande `cmd_send_message` est un peu plus complexe. Il faut extraire le nom du destinataire, obtenir son identifiant, obtenir un groupe portant ce nom si aucun client n'a été trouvé, puis envoyer le message (au client ou au groupe, dépendant de ce qui a été trouvé).

Les commandes `cmd_group_join` et `cmd_group_quit` sont en fait des sous-commandes de `cmd_group`. C'est donc un autre exemple de parsing en plusieurs fois.

Le `default` du `switch` renvoie au client un message d'erreur.

`ClientListener`

Code correspondant au thread du `ClientListener`.

Nous avons un cas d'exemple de synchronisation : dans la fonction `main`, le thread du `ClientListener` est démarré avant d'être ajouté au `struct` du client. Cela veut dire qu'au démarrage de son exécution, le `ClientListener` peut ne pas encore avoir totalement accès aux éléments constituant le client. Ces éléments sont à priori enregistrés avant le démarrage du thread, mais dans le cadre de la programmation défensive, une boucle permet au `ClientListener` d'attendre l'enregistrement complet du client avant de poursuivre son exécution.

Le reste de la fonction consiste en une boucle infinie. Un premier `recv` bloquant sur le socket du client attend qu'un message arrive. Dès qu'un message commence à être reçu, le mutex des `ClientListener` est verrouillé.

Le `ClientListener` va alors pouvoir écrire le message dans le `MessageBuffer`. Il commence par envoyer l'identifiant du client, ce qui permettra au `Dispatcher` de router correctement ensuite. Puis il démarre une deuxième boucle, mais sur un `recv` non bloquant cette fois. Ceci permet de quitter la boucle dès qu'il n'y a plus rien à recevoir. Le `ClientListener` libère alors le mutex, puis redémarre un nouveau cycle de lecture.

`ClientSender`

Cette fonction est appelée par le thread du `ClientSender`.

La même boucle de synchronisation que pour le `ClientListener` a lieu. Puis ce qui est lu dans le `ClientPipe` est envoyé au client via son socket.

`clean_and_close`

Le serveur a tout comme le client une fonction pour quitter proprement le programme. Elle est appelée par le handler lors de la capture des signaux `SIGINT` et `SIGTERM`.

Cette fonction commence par déconnecter tous les clients : pour chacun d'entre eux, un message `EOT` est envoyé, puis la fonction `disconnect_client` est appelée.

Le `MessageBuffer` et le mutex des `ClientListener` sont fermés, puis le thread du `Dispatcher` est tué. Enfin, le socket du serveur est fermé pour mettre fin à la connexion.

`interrupt`

Les signaux `SIGINT` et `SIGTERM` sont capturés pour permettre d'appeler la fonction `clean_and_close`.

6.2.2 Les commandes

Certaines fonctions sont préfixées par `cmd_` : il s'agit de commandes provenant du client et qui font une action précise.

`cmd_help`

Envoie au client l'aide, c'est-à-dire les commandes disponibles et leur syntaxe.

`cmd_hello`

C'est une commande interne, envoyée par le client lors de sa connexion. Le serveur envoie simplement l'identifiant du client.

`cmd_group_join`

Ajoute un client à un groupe. Si le groupe n'existe pas, il est créé.

`cmd_group_quit`

Retire le client d'un groupe. Une fois la suppression effectuée, la fonction `delete_group_if_empty` est appelée pour faire le ménage si jamais le groupe est vide.

`cmd_set_name`

Modifie le nom d'usage du client. Si le nom est déjà pris, la modification est refusée.

`cmd_get_name`

Commande interne correspondant à la commande `cmd_get_name` du client, qui l'utilise pour modifier son prompt. Cette fonction se contente donc d'envoyer au client son nom.

`cmd_list_clients`

Envoie au client la liste des clients connectés.

Pour envoyer le message d'un coup tout en ajoutant les clients connectés au fur et à mesure du parcours de la liste des clients, une série de `malloc` est faite. Un `char*` temporaire permet d'agrandir progressivement le message en libérant le `malloc` précédent.

`cmd_list_groups`

Envoie au client la liste des groupes et les clients présents dans chaque groupe.

Cette fonction suit le même principe que la fonction `cmd_list_clients` avec la complexité supplémentaire d'ajouter une chaîne de `malloc` pour afficher chaque client au sein de chaque groupe.

`cmd_broadcast`

Pour broadcaster un message, il suffit de l'envoyer à chaque client connecté en ignorant l'émetteur.

`cmd_send_message`

Fonction de base pour l'envoi d'un message. Comme elle est utilisée par d'autres fonctions, la gestion de l'émetteur et du destinataire est laissée au `Dispatcher`.

La fonction `cmd_send_message` reçoit les identifiants émetteur et destinataire, ainsi que le message à envoyer. Grâce à un `malloc`, le message est préfixé de l'émetteur, ainsi que de la date d'envoi, récupérée via la bibliothèque `time.h`. Le message est également suffixé par le caractère `ETX` (*end text*), pour signifier au `Listener` du client en face que le flux est terminé.

`send_group_message`

Il ne s'agit en fait pas d'une commande à part entière, mais d'une fonction correspondant à une sous-commande de `cmd_send_message`. Ceci est expliqué dans la fonction `Dispatcher`.

Semblablement à `cmd_broadcast`, un message est envoyé à tous les clients d'un groupe.

6.3 Les exclusions mutuelles

6.3.1 Chez le client : sémaphore nommé

Pour s'assurer que le flux affiché sur la sortie standard soit atomique, il a fallu implémenter un sémaphore pour le client. Le `Sender` et le `Listener` ont en effet des fonctions bien différentes, mais écrivent tous les deux sur la sortie standard.

Le sémaphore étant partagé par des processus (et non des threads), il y avait deux possibilités d'implémentation :

1. Sémaphore non nommé, utilisant un espace de mémoire partagé.
2. Sémaphore nommé, pour laquelle la commande `sem_open` crée un fichier dans `/dev/shm/`.

Le sémaphore nommé a permis de ne pas créer l'espace de mémoire partagé à la main.

6.3.2 Chez le serveur : mutex

Le serveur utilisant des threads, l'utilisation de mutex a été faite avec la bibliothèque `pthread`.

Le seul cas où une exclusion mutuelle est possible est lors de l'écriture dans le `MessageBuffer` par les différents `ClientListener`. En effet, il a fallu s'assurer que les messages poussés dans le `MessageBuffer` (implémenté par un pipe), soient atomiques. Ceci permet au `Dispatcher` de les récupérer un par un sans devoir les reconstituer.

6.4 Les messages de debug/log

Pour plus de clarté concernant les messages imprimés à la fois côté serveur et côté client, des macros ont été définies dans le fichier de header `common.h`.

En fonction du niveau de log, (`ERROR`, `WARNING`, `INFO`, `DEBUG`), les messages sont ou ne sont pas imprimés.

```
#define warn_print(fmt, ...) \
    do { if (WARNING) fprintf(stderr, "[WARNING] %s:%d:%s(): " fmt, __FILE__,
        __LINE__, __func__); } while (0)
#define warn_printf(fmt, ...) \
    do { if (WARNING) fprintf(stderr, "[WARNING] %s:%d:%s(): " fmt, __FILE__,
        __LINE__, __func__, __VA_ARGS__); } while (0)
```

Ces macros incluent des informations utiles : le niveau du log, ainsi que le fichier, la ligne et la fonction ayant émis le log.

7 Conclusion

Ce projet nous a permis d'appréhender de multiples composantes d'un programme multitâches complet.

Avec le langage C, nous avons pu maîtriser toutes les étapes nécessaires, de la synchronisation à la gestion de la transmission de flux caractère par caractère.

La gestion d'erreur a été une partie fondamentale du développement de l'application. Nous avons en effet choisi de programmer de manière défensive : tout code de retour des fonctions doit être vérifié, toute branche conditionnelle doit être explorée. Il a également fallu choisir en fonction des cas que faire lors d'une erreur : quitter le programme, ignorer et reprendre la boucle, fermer une connexion, informer le client...

Un élément particulièrement délicat à mettre en œuvre a été la gestion du mutex lors de la réception d'un message. Lors de la boucle de lecture, la fonction `read` (ou `recv` en fonction des cas) est bloquante. Si le mutex est verrouillé juste avant, le thread garde la main de manière infinie, que ce soit lors de la lecture ou lors de son attente.

Ce type de considération à prendre montre les implications à avoir pour un programme multitâches, qui ne se contente pas de faire plusieurs choses en parallèle.