



PROJET - SYSTÈME D'EXPLOITATION

---

## Messagerie instantanée

---

*Élèves :*

Hamza LAOUNI

Aïssa PANSAN

*Enseignant :*

Ralph BOU NADER

*16 avril 2023*

*Send you our best regards,*

# Sommaire

<b>1</b>	<b>Introduction générale</b>	<b>3</b>
1.1	Explication rapide de fonctionnement de l'application . . . . .	3
<b>2</b>	<b>Communication inter-processus</b>	<b>4</b>
2.1	Structure principale du programme . . . . .	4
2.1.1	Structure globale . . . . .	4
2.1.2	Liste chaînée . . . . .	6
2.1.3	Fonction associé . . . . .	6
2.1.4	Difficultés rencontrées : Le nommage des mkfifo . . . . .	8
2.1.5	Fonction du serveur . . . . .	9
2.1.6	Problèmes rencontrés . . . . .	12
2.1.7	Exécution coté client . . . . .	13
2.2	Terminaison de la communication . . . . .	15
<b>3</b>	<b>Communication inter-machines</b>	<b>16</b>
3.1	Introduction aux sockets . . . . .	16
3.2	Implémentation en C . . . . .	16
3.3	Paramétrage du socket . . . . .	17
3.3.1	Procédure pour une communication fonctionnelle . . . . .	18
3.4	Communication client/serveur et interface utilisateurs en console . . . . .	19
3.5	Modification de la struct FIFO . . . . .	19
3.6	Suppression client . . . . .	19
3.6.1	Principe . . . . .	19
3.7	Utilisation de mutex . . . . .	21
3.7.1	Principe . . . . .	21
3.8	Interface utilisateur backend . . . . .	22
3.8.1	Création d'un compte . . . . .	22
3.8.2	Connexion à un compte . . . . .	23

3.9	Chiffrement de mots de passe . . . . .	23
3.10	Enregistrement des conversations . . . . .	23
<b>4</b>	<b>Interface graphique</b>	<b>26</b>
4.1	Principe . . . . .	26
4.2	Implémentation . . . . .	27
4.3	Essai . . . . .	28
4.4	Modification du code principal . . . . .	30
<b>5</b>	<b>Conclusion générale</b>	<b>31</b>
<b>6</b>	<b>Point d'améliorations</b>	<b>31</b>
<b>7</b>	<b>Annexe</b>	<b>32</b>
7.1	Définition des fonctions utilisées pour les sockets . . . . .	32
7.2	Fonction communication client/server sur socket . . . . .	34
7.3	Fonction pour la connexion . . . . .	36
7.4	Fonction pour l'inscription . . . . .	37
7.5	Définition des fonctions Python . . . . .	38
7.6	Codage de l'interface graphique . . . . .	43

# 1 Introduction générale

Nous avons l'honneur de vous présenter le rapport de notre projet de Système d'exploitation, réalisé en tant qu'étudiants en première année d'ingénierie en informatique.

Ce projet s'inscrit dans le cadre de notre formation en ingénieur en informatique, qui comprend l'utilisation du langage C et des notions vues en système d'exploitation telles que les processus, les pipes, les threads et les sémaphores. Notre but a été de développer une application de messagerie instantanée qui permettait dans un premier temps de faire communiquer au sein d'une même machine plusieurs processus. Dans un second temps, nous avons modifié le code existant pour faire communiquer plusieurs machines entre elle à l'aide de sockets. Toutes ces communications incorporent des fonctionnalités supplémentaires comme l'enregistrement des conversations, l'authentification des utilisateurs et bien d'autres.

Nous sommes fiers de partager avec vous les résultats de notre travail et espérons qu'il démontrera notre compréhension des concepts que l'on a étudiés durant ce semestre.

## 1.1 Explication rapide de fonctionnement de l'application

Notre application permet de faire communiquer plusieurs processus entre eux ou plusieurs ordinateurs connectés au sein du même réseau local. Un processus à part va se charger de la communication, chaque client est aussi représenté par un processus qui gère les communications avec le processus d'échange. Les utilisateurs après s'être connecté ou inscrit peuvent librement discuter avec les clients déjà connectés dans un salon et se déconnecter à leur guise.

## 2 Communication inter-processus

### 2.1 Structure principale du programme

#### 2.1.1 Structure globale

La communication inter-processus s'articule autour d'un processus "Père". Ce dernier va exécuter le code du fichier nommé **SERVEUR.c** qui gère les communications. C'est ce processus qui va s'occuper d'ajouter les clients dans notre structure de données, générer les terminaux de communication pour les clients et écouter et distribuer les messages de n'importe quels utilisateurs via un thread. Les clients sont des processus fils générés par le processus père, qui vont exécuter le code **exec2.c** tout en étant associés à un émulateur de terminal **xterm**. Ces processus vont pouvoir envoyer des messages et les recevoir à l'aide de deux pipes nommées de type mkfifo, générées et ouvertes par le processus père, et de deux threads associés à la tâche d'écrire et de recevoir. Ainsi pour chaque client, on a :

- Un processus dit "Serveur" qui est toujours existant.
- Un fichier historique appelée **Conversation**.
- Un processus fils par client.
- Deux mkfifo de réception et d'envoi associés lue par le thread de distribution du serveur, par client.
- Un thread d'envoi qui écrit dans le mkfifo **réception** par client.
- Un thread de réception qui lit dans le mkfifo **distribution** par client.

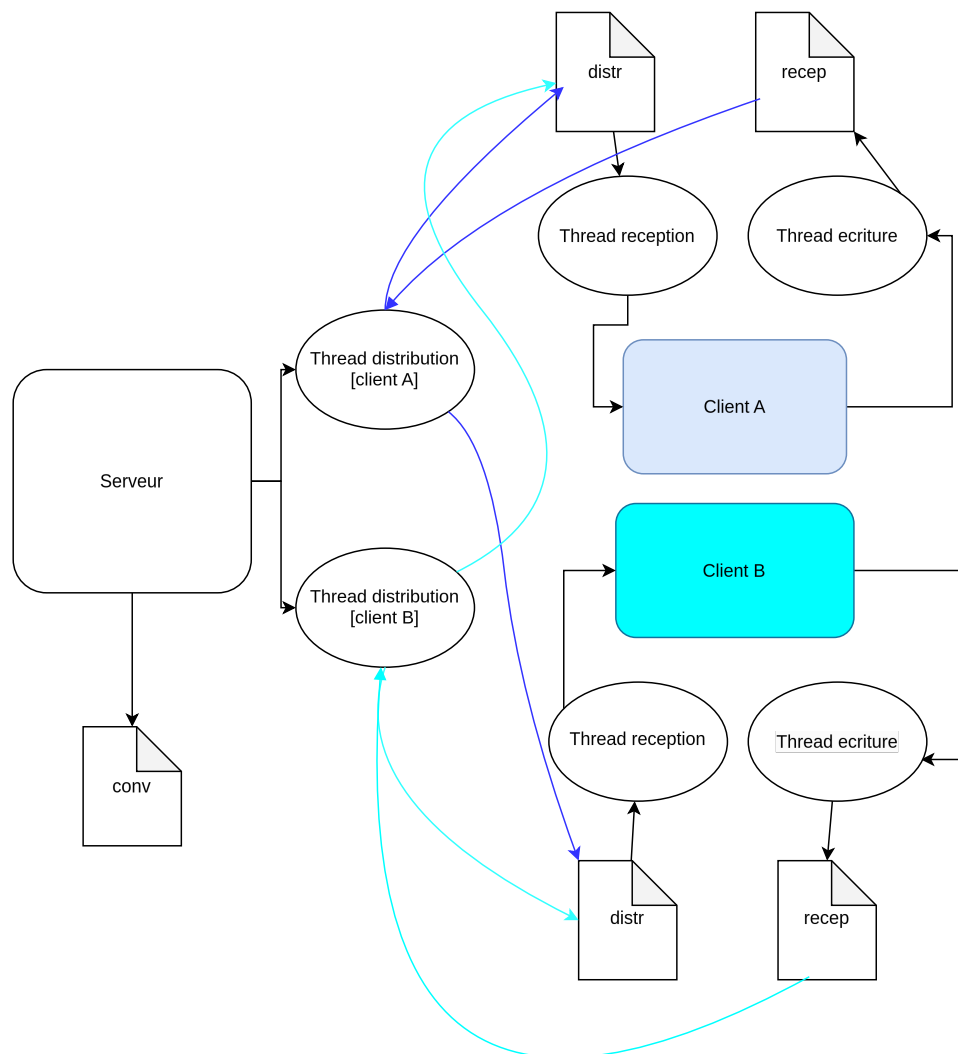


FIGURE 1 – Schéma de la structure de la communication client/serveur

### 2.1.2 Liste chaînée

Afin de gérer efficacement tout un ensemble d'utilisateurs, il a été décidé d'utiliser une structure de données de type liste chaînées afin de conserver une trace des clients au cours de la communication.

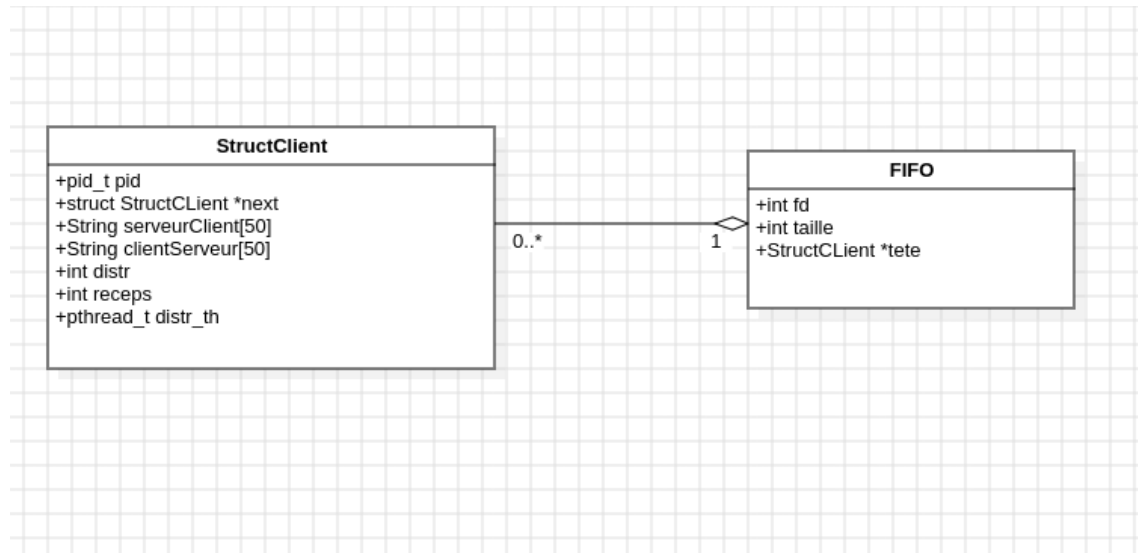


FIGURE 2 – Liste chaînée des clients

La liste **FIFO** contient un descripteur de fichier qui point vers le fichier historique **Conversation**. La structure **FIFO** va pointer vers le premier client connecté. Les maillons de cette structure sont représentés par des **structClient**. Les chaînes de caractères **serveurClient** vont servir à stocker le chemin du mkfifo d'envoi, il en est de même pour le mkfifo de réception avec la chaîne **clientServeur**. Pour rendre le code plus générique les noms des fichiers mkfifo ne sont pas connus à l'avance, la méthode pour générer les noms des mkfifo sera détaillé plus tard. Ensuite, les champs **distr** et **receps** permettent de stocker les descripteurs des mkfifo de distribution et de réception.

### 2.1.3 Fonction associé

Après avoir défini la structure d'une liste chaînée, il est nécessaire de définir un ensemble de fonction permettant d'initialiser et de manipuler cette structure.



**newFIFO :**

**Entrées :** `char conv[ ]`

**Sortie :** `FIFO*`

**Fonctionnement :** Génère une nouvelle liste et associe le fichier de conversation à la file

**newStructclient :**

**Entrées :** `pid_t pid`, `char serveurClient[ ]`, `char clientServeur[ ]`

**Sortie :** `Structclient *`

**Fonctionnement :** Alloue un maillon de type **StructClient** et initialise les champs du maillon tout en copiant le **PATH** des descripteurs de fichiers d'envoi et de réception

**add :**

**Entrées :** `pid_t pid`, `char serveurClient[ ]`, `char clientServeur[ ]`, `FIFO* f`

**Sortie :** `void`

**Fonctionnement :** La fonction `add` ajoute un maillon à la structure de file. Elle appelle **newStructclient** pour générer un maillon.

**pop :**

**Entrées :** `FIFO* f`

**Sortie :** `Structclient*`

**Fonctionnement :** La fonction retire un maillon de tête et le renvoie

**freeFIFO :**

**Entrées :** `FIFO* f`

**Sortie :** `tvoid`

**Fonctionnement :** La fonction va tour à tour free le pointeur renvoyé par `pop` jusqu'à que la liste est vide.

#### 2.1.4 Difficultés rencontrées : Le nommage des mkfifo

Lors d'un chat de groupe, le serveur ne peut connaître à l'avance le nombre d'utilisateurs qui vont communiquer, la seule limite étant qu'il au maximum 5 utilisateurs, on ne peut donc pas simplement écrire :

---

```
mode_t mode = 0660;
if (mkfifo("reception", mode) == -1)
{
    perror("mkfifo");
    exit(50);
}
if (mkfifo("envoi", mode) == -1)
{
    perror("mkfifo");
    exit(50);
}
```

---

Un tel code provoquerait une erreur, lorsqu'un second client voudra se connecter, car le mkfifo va tenter d'ouvrir la même pipe.

Pour s'affranchir de cette difficulté, on peut décider de nommer les fichiers mkfifo et aussi **Conversation** dans le même temps, de manière aléatoire.

---

```
// Dans le main
srand(getpid());
char conv[25];
sprintf(conv, "./conversations/%d.txt", rand());
Clients = newFIFO(conv);
```

---

Chaque appel de la fonction **rand()** va en réalité aussi appeler la fonction **srand()** qui fait office de graine, c'est-à-dire le nombre de départs par lequel on va par la suite

déterminer un autre nombre via du pseudo-aléatoire. Notre graine est le pid du père, il sera toujours, dans la majorité des cas, différent de l'exécution précédente.

On utilise le même procédé pour générer le nom des mkfifo, on va par la suite les stocker dans **clientServeur** et **serveurClient**

---

```
// Dans le main
creerClient(name, Clients, rand() % BIGNUMBER, rand() % BIGNUMBER);
```

---

### 2.1.5 Fonction du serveur

La structure de données étant bien établie, il nous est alors impératif de définir les fonctions qui vont pouvoir régir la communication entre les clients.

**creerFork :**

**Entrées :** **FIFO\*** f, **char** serveurClient[ ], **char** clientServeur[ ], **char** name

**Sortie :** **int**

**Fonctionnement :** La fonction va générer un processus fils qui va exécuter via xterm le fichier exec2

**creerClient :**

**Entrées :** **char** name[ ], **FIFO\*** f, **int** tube1, **int** tube2

**Sortie :** **void**

**Fonctionnement :** Fonction qui va permettre la génération d'un nouveau client via **creerFork**, ouvrir les pipes nommés via **creerPipes** puis les associer aux clients via la file, ajouter ce client dans la liste chaînée via add et générer le thread de distribution du serveur (qui s'occupera de distribuer les messages envoyés entre clients)

**creerPipes :**

**Entrées :** **char** serveurClient[ ], **char** clientServeur[ ]

**Sortie :** `void`

**Fonctionnement :** La fonction va ouvrir les mkfifo d'écoute et d'envoi en fonction des PATH renseigné via `serveurClient` et `clientServeur`

**distr :**

**Entrées :** `void*` arg

**Sortie :** `void*`

**Fonctionnement :** Routine exécutée par les threads générée via **`créerClient`**, ils vont exécuter la fonction `distribuer` à chaque fois qu'une des caractères seront détectés sur le mkfifo (`clientServeur`) de réception via `read`.

**distribuer :**

**Entrées :** `char` message[ ], `size_t` sizemsg, `Structclient*` distributeur

**Sortie :** `void`

**Fonctionnement :** Fonction exécuté uniquement si des char sont détectés sur le mkfifo réception du client. La fonction va se charger d'envoyer le message à tous les clients (Sauf celui qui l'a envoyé) via le mkfifo de `distr`. On va aussi écrire dans le fichier conversation.

On évite d'envoyer deux fois le même message en vérifiant si `temp != destributeur` ou

```
Structclient *temp = Clients->tete;
```

L'enchaînement des différentes actions peut être représenté via un diagramme d'activité.

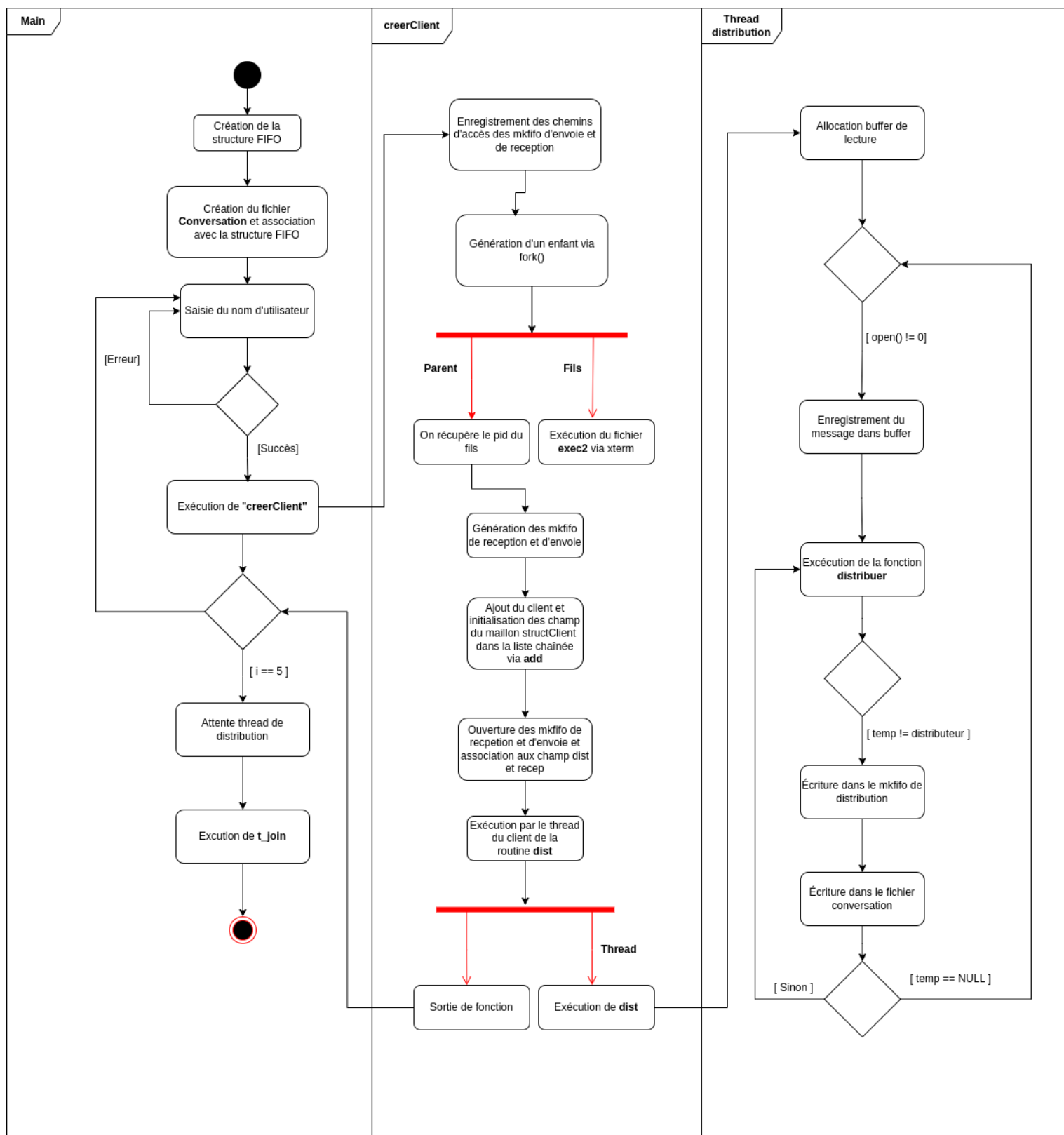


FIGURE 3 – Diagramme d'activité des communications

### 2.1.6 Problèmes rencontrés

Les threads de distributions sont attentifs à n'importe quel message envoyé par un client, dès qu'il le réceptionne il le transmet à tous les utilisateurs. Cependant, un problème se pose dans le cas où il y a beaucoup de client qui converse. Dans le cas exceptionnel où deux clients envoient un message en simultanées, leurs threads respectifs de distribution vont tenter d'écrire dans le même mkfifo de réception du client cible , ce qui peut occasionner des erreurs de synchronisation.

Afin de palier à ce problème, on peut utiliser les mutex. Les mutex, nous permettrons de contrôler l'écriture sur le pipe distribué à un seul thread à la fois. Quand un thread va exécuter la zone critique, il décrémentera `fd_mutex`, si un autre thread tente d'exécuter la portion critique. Il devra vérifier si le résultat de `pthread_mutex_lock(&fd_mutex);` est supérieur à 0. Si ce n'est pas le cas, cela veut dire qu'un autre thread d'un autre client écrit actuellement, il devra attendre que le thread ait fini de rendre le mutex en l'incrémentant.

---

```
pthread_mutex_t fd_mutex = PTHREAD_MUTEX_INITIALIZER; //Le mutex utilisé pour
    bloquer l'écriture sur le fd distr
...
// Dans la fonction distribuer
Structclient *temp = Clients->tete;
while (temp != NULL)
{
    if (temp != destributeur)
    {
        pthread_mutex_lock(&fd_mutex); //Debut de la portion critique
        if (write(temp->distr, message, sizemsg) <= 0)
        {

            printf("probleme d'écriture ~ mutex\n");
            t_join(Clients);
        }
    }
}
```

---

```
        pthread_mutex_unlock(&fd_mutex); // Fin de la portion critique
    }

    temp = temp->next;
}
```

---

On a le même procédé pour contrôler l'écriture du fichier **Conversation**

### 2.1.7 Exécution coté client

Des que le processus fils va exécuter via **execv** un terminal xterm avec les chemins d'accès des différents tubes de réception et d'envoi et de lecture. Le client va :

1. Stocker le chemin d'accès du mkfifo de réception et le nom d'utilisateurs **path-Name**. La structure fait office de tableau.
2. Lancer deux threads, l'un va se charger d'envoyer des messages, l'autre les réceptionne et l'affiche dans le terminal xterm du client

**writeth :**

**Entrées :** **void\*** arg(temp)

**Sortie :** **void**

**Fonctionnement :** La fonction prend en argument un pointeur vers la structure pathName et va enregistrer la saisie du client via fgets dans un buffer. La fonction va par la suite lancer clearPreviousLine et va écrire dans le fichier dans le mkfifo de réception (Qui est à l'écoute par le thread de distribution du processus père du fils) le nom de l'utilisateur suivit du message.

**readth :**

**Entrées :** **void\*** arg (argv[1])

**Sortie :** **void**

**Fonctionnement :** La fonction va continuellement lire le mkfifo de distribution (En attente de l'envoi du serveur) par le chemin d'accès renseigné via `arg[1]`. La fonction va par la suite afficher à l'utilisateur le nom de l'utilisateur ainsi que le contenu du message.

**clearPreviousLine :**

**Entrées :** `void`

**Sortie :** `void`

**Fonctionnement :** La fonction supprime ce que l'utilisateur a écrit, quand ce dernier veut écrire un message, afin d'éviter un doublon.

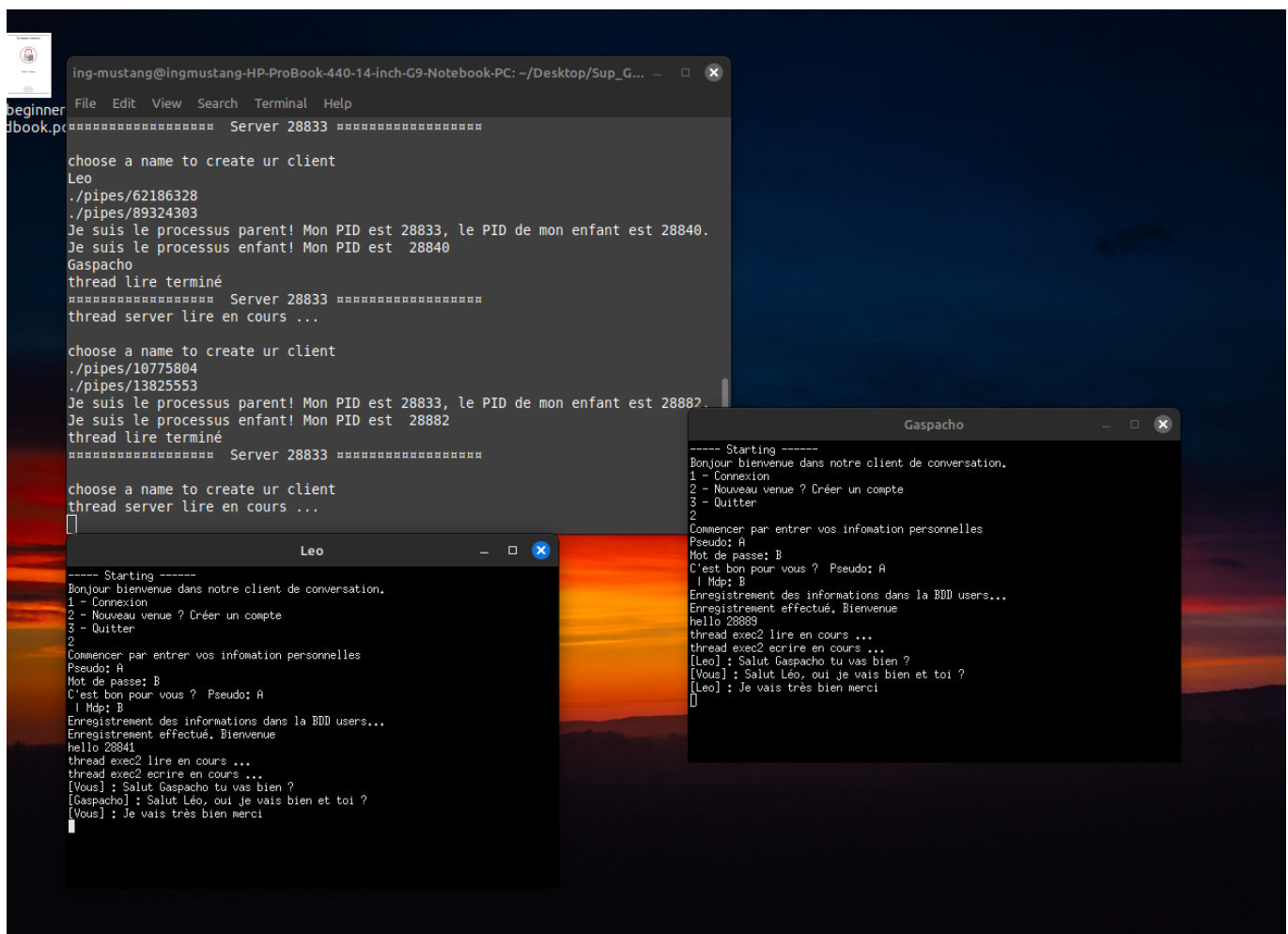


FIGURE 4 – Exemple de communication entre deux utilisateurs



## 2.2 Terminaison de la communication

Pour terminer la communication, coté serveur, il suffit de faire la combinaison Crtl+C, cela envoie un signal **SIGSTOP** qui sera capturé par un **signal** qui fera exécuter la fonction **sighandler**. Cette fonction va supprimer les mkfifo des clients via **unlink**, tuer les processus fils en évitant l'apparition de zombie via **waitpid** et le nettoyage de la structure de file via **freeFIFO**. À la fin, on exécute la fonction **t\_join** pour fermer les descripteurs de fichiers du serveur et supprimer la structure, le struct **FIFO**

## 3 Communication inter-machines

Notre première version du client de communication nous permet d'engager des discussions avec plusieurs clients au sein d'un même ordinateur. Cela dit, les cas d'applications de notre client de messagerie, dans la vie, sont quasi-nulles à cause de la communication qui n'est limitée qu'au sein d'un même ordinateur. L'idée nous est alors venue d'améliorer notre version actuelle du client de communication pour permettre la communication entre deux machines connecté via le même réseau. C'est ainsi qu'a démarré la seconde phase de notre projet, la communication inter machines via des sockets.

### 3.1 Introduction aux sockets

Pour réussir à faire communiquer deux machines connectées sur le même réseau, il a été décidé d'utiliser la technologie des sockets. Simple d'utilisation et multifonctionnelles, il était très bien adapté à notre objectif de communication inter-machines.

Les sockets sont une composante essentielle dans les réseaux informatique, il joue le rôle d'interface entre les processus applicatif tels que des navigateurs web ou des clients de messagerie électronique pour communiquer avec des services réseau responsable du transport de l'information. L'autre fonctionnalité incontournable des sockets et de transmettre des informations, de la même manière qu'un tube, à d'autres processus. Les sockets servent donc principalement à transmettre des informations entre des processus ou un serveur.

### 3.2 Implémentation en C

Leur utilisation en C, se fait grâce à l'importation de plusieurs bibliothèques qui active aussi des services réseaux qui seront nécessaires pour la communication via un réseau local.

---

```
#include <sys/socket.h> // Definition des fonctions lié aux sockets
#include <arpa/inet.h> // Fonction de definition de variables et structures
                        essentielle à la communication reseau
```

---

Pour que la communication se fasse bien, il est nécessaire d'avoir deux fichiers sources

---

en C bien distincts. L'exécution de ces fichiers dépend du rôle de la machine dans la communication. L'ordinateur qui gèrera toutes les communications exécutera le code du fichier **SERVER.c**, il fera office de serveur et permettra à l'aide de socket de faire transiter les messages de tous les clients. Toutes les autres machines qui voudront discuter exécuteront le code du fichier **CLIENT.c**, le socket de cette exécution établira la communication entre le socket server et le client et pourra faire transiter les messages du client.

Par la suite, on a accès à un large panel de fonction pour utiliser des sockets qui sont définis dans [l'annexe](#) de ce rapport.

### 3.3 Paramétrage du socket

Après avoir initialisé le socket, il est nécessaire de le configurer via sa structure **struct sockaddr\_in** pour pouvoir l'utiliser correctement. La structure est défini comme ci-contre.

---

```
struct sockaddr_in
{
    short    sin_family;
    unsigned short sin_port;
    struct  in_addr  sin_addr;
    char    sin_zero[8];
};
```

---

La 1re étape cruciale est le choix du port de communication. Nous avons décidé que les communications passeraient par le port 7777, il est donc impératif d'ouvrir ce port du côté client pour que ce dernier puisse recevoir des paquets du serveur.

Le numéro de port est passé à la variable **sin\_port** de **sockaddr\_in** en faisant une conversion de bit au préalable via la fonction **htons**.

Enfin, l'adresse IP du serveur est passée en argument aux champs **sin\_addr** via la fonction **inet\_pton** qui convertit une adresse IPv4/IPv6 en binaire.

On associe ensuite les informations du type struct aux sockets via **bind**. Ces étapes

sont les mêmes du côté serveur ou client, la seule différence étant que le serveur généré automatiquement son adresse IP via la constante **INADDR\_ANY**. Du côté du client, il faut associer l'adresse IP du serveur, c'est-à-dire l'adresse IP de la machine qui héberge le serveur. La communication s'établit lorsque le serveur écoute sur le port 7777, via **listen** et que le client tente d'établir une communication via la commande **connect**.

Enfin, il est nécessaire de paramétrer aussi un type **sockaddr\_in** pour le client juste après que ce dernier est contacté le serveur, auquel cas le serveur ne saurait pas comment recontacter le client.

### 3.3.1 Procédure pour une communication fonctionnelle

.

Pour pouvoir communiquer avec une autre machine connectée sur le même réseau, il faut pouvoir suivre quelques étapes.

1. Ouvrir les ports d'entrée et de sortie du port choisis dans le serveur
2. On change l'adresse IP **SERVER\_IP** du fichier **CLIENT.c** avec l'adresse IP de la machine serveur dans le réseau
3. Coté serveur on compilera de cette façon : `gcc -Wall -pthread SERVER.c -o [serv]`
4. Coté client, il faut rajouter le drapeau **-lcrypto** après `-o`
5. On exécute le programme serveur
6. Suivit des programmes client dans la limite de 3 clients

Pour pouvoir communiquer de la même manière qu'avec la version 1 du client, il suffit de changer l'adresse IP **SERVER\_IP** avec celle du localhost.

### 3.4 Communication client/server et interface utilisateurs en console

### 3.5 Modification de la struct FIFO

L'implémentation des sockets change la manière de communiquer avec un utilisateur ne serait ce que pour l'implémentation de la struct FIFO. Les autres fonctions impliquées dans le processus sont assez similaire à la version FIFO avec cependant, quelque changement la définition des fonctions est faite dans la partie [Annexe](#) de ce rapport.

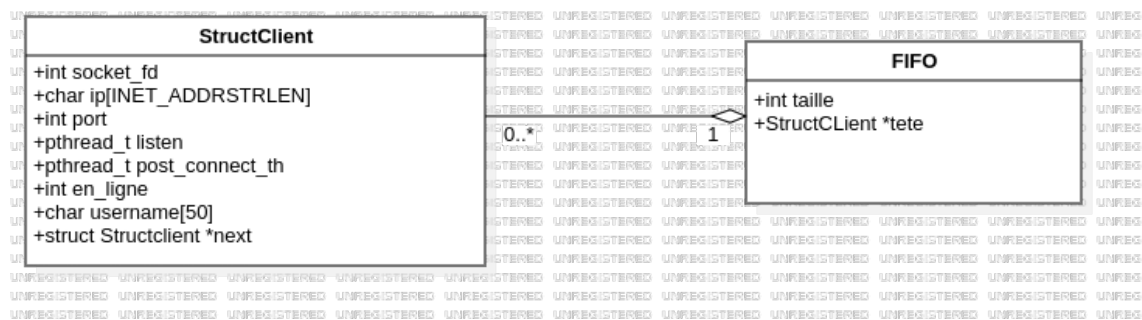


FIGURE 5 – Image

### 3.6 Suppression client

Dans notre précédente version de client de communication, si un utilisateur quittait la conversation, alors il était impossible de continuer la communication entre les autres utilisateurs. C'est pourquoi, nous avons implémenté une fonction capable de supprimer un client de la conversation et de la structure liste.

#### 3.6.1 Principe

Quand un client quitte la conversation, il doit entrer la combinaison Crtl+C, après un nettoyage coté client. **La fonction recv va recevoir un dernier message du client qui ne contient aucun caractère.** Ce cas est géré dans la fonction `post_connexion`, cette dernière va alors lancer la fonction `remove_client`

---

```
void remove_client(Structclient *c)
```

---

```
{
    if (!Clients || !c)
    {
        printf("Liste vide ou client NULL.\n");
        return;
    }

    // Retrait du client de la liste
    if (Clients->tete == c)
    {
        // Le client est la tête de la liste
        Clients->tete = c->next;
    }
    else
    {
        // Recherche et retrait du client dans la chaîne
        Structclient *prev = Clients->tete;
        while (prev != NULL && prev->next != c)
        {
            prev = prev->next;
        }
        if (prev != NULL)
        {
            prev->next = c->next;
        }
    }

    strcpy(c->username, "");
    free(c); // Libérer la mémoire du client
    c = NULL;
    Clients->taille--;
    printf("Client supprime\n");
}
```

}

---

La fonction est analogue à une fonction de suppression d'un maillon à une position particulière, la recherche se faisant sur l'utilisateur déconnecté.

L'avantage de cette méthode est que l'utilisateur déconnecté peut à tout moment se reconnecter sans que l'accès à la conversation lui soit bloqué.

## 3.7 Utilisation de mutex

De la même manière, qu'à la partie 1, nous avons besoin d'utiliser des mutex pour sécuriser des portions critiques du code. Cela dit, dans cette version, nous allons recourir à un nouveau type de mutex appelé mutex read/write (ou mutex rwlock), en plus des mutex vu dans la partie 1.

### 3.7.1 Principe

Les mutex rwlock se distingue des mutex binaires par l'existence de deux verrous, l'un pour l'écriture via **pthread\_rwlock\_wrlock**, l'autre pour la lecture via **pthread\_rwlock\_rdlock**.

Lorsqu'un thread obtient un verrou de lecture, il peut lire une ressource partagée tant qu'aucun thread n'est en train de modifier ladite ressource partagé. Un thread qui modifie une section critique acquérir un verrou d'écriture et va ainsi bloquer tous les autres threads, jusqu'à que ce dernier libère son verrou.

L'avantage de cette méthode est que seul l'écriture est un élément bloquant pour les threads et non pas la lecture.

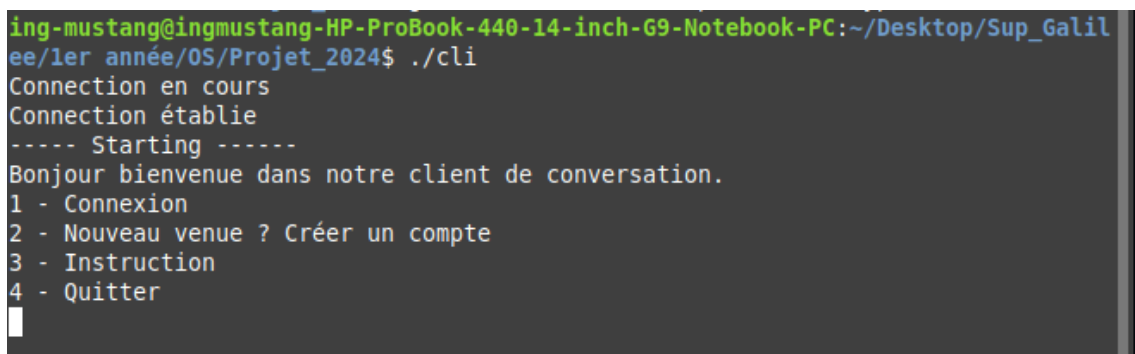
Plusieurs portions critiques du code utilisent ce type de mutex :

- Lors du nettoyage de toute la structure **FIFO**, quand on associe le pointeur **temp** à la tête de la liste
- Lors de la distribution d'un message
- Lors de l'envoi d'un message, via des mutex binaires
- Lors de l'étape de la suppression d'un utilisateur

- Dans la fonction `checkinClientsenligne`
- Lors de l'ajout d'un nouveau client dans la structure **FIFO**, dans `accept_thread`

### 3.8 Interface utilisateur backend

Après avoir tenté d'implémenter une ébauche d'interface utilisateur dans la partie 1, nous avons décidé de finaliser l'interface utilisateur cotée backend. Pour ce faire, il fallait réaliser une interface cohérente pour deux options incontournables, la connexion et la création d'un nouveau compte.



```
ing-mustang@ingmustang-HP-ProBook-440-14-inch-G9-Notebook-PC:~/Desktop/Sup_Galil  
ee/1er année/OS/Projet_2024$ ./cli  
Connection en cours  
Connection établie  
----- Starting -----  
Bonjour bienvenue dans notre client de conversation.  
1 - Connexion  
2 - Nouveau venue ? Créer un compte  
3 - Instruction  
4 - Quitter  
█
```

FIGURE 6 – Affichage du menu en console

Un utilisateur est identifié au moyen d'un **pseudo** et d'un **Mot de passe**. Ses logins sont enregistrés dans un fichier texte nommé **DATA**. Les échanges entre utilisateur et client sont soumis à un principe de contrôle sur ce qui a été envoyé, il existe ainsi plusieurs codes d'erreurs (ou "flags") qui sont envoyés pour continuer ou recommencer certaines étapes.

- `"||"` => Utilisateurs déjà connectés
- `"0"` => Utilisateur inconnu dans la base de données
- `"1"` => Succès

#### 3.8.1 Création d'un compte

Quand un client entre son pseudo, il faut pouvoir vérifier deux éléments lors de la réception du message par le serveur :



- Si le pseudo correspond à un utilisateur déjà connecté, via **checkinClientsenligne**
- Si le pseudo correspond à un utilisateur déjà présent inscrit dans le fichier **DATA** via **askforvaliduser**.

Si ces tests sont passés avec succès, le client choisi son mot de passe, ce dernier sera haché puis le couple (pseudo,mot de passe) sera enregistré dans un type struct **info\_client**. On va envoyer une instance de ce type au serveur qui se chargera ensuite d'écrire les informations dans le fichier texte, puis d'initier la communication via **post\_connect**.

### 3.8.2 Connexion à un compte

L'utilisateur envoie son pseudo au serveur, ce dernier contrôle de la même manière qu'auparavant si l'utilisateur associé à ce pseudo est déjà connecté ou s'il existe bien dans la base de donnée. Ensuite, l'utilisateur envoie un mot de passe, encore une fois haché coté client, qui sera comparé à celui inscrit dans le fichier. S'il est bon, l'utilisateur pourra engager la conversation

## 3.9 Chiffrage de mots de passe

Afin de garantir davantage de sécurité à l'utilisateur, le mot de passe qu'on envoie lors de la création d'un compte ou lors du processus de connexion est systématiquement haché. Pour ce faire, nous avons fait appel à la bibliothèque

Une même séquence de chaînes de caractère sera haché de la même manière, ce qui nous permet de comparer plus facilement les mots de passe (Sans forcément utiliser une clé de décryptage). Ce sont les fonctions **get\_password\_and\_hach** et **get\_password** qui permettent de hacher un mot de passe donnée en entrée, l'implémentation de ces fonctions s'est fait à l'aide de **ChatGPT**.

## 3.10 Enregistrement des conversations

Les conversations de tous les utilisateurs sont enregistrée dans un fichier localisé dans le dossier **conv**. À la différence de la version 1, les fichiers sont nommés en fonction de

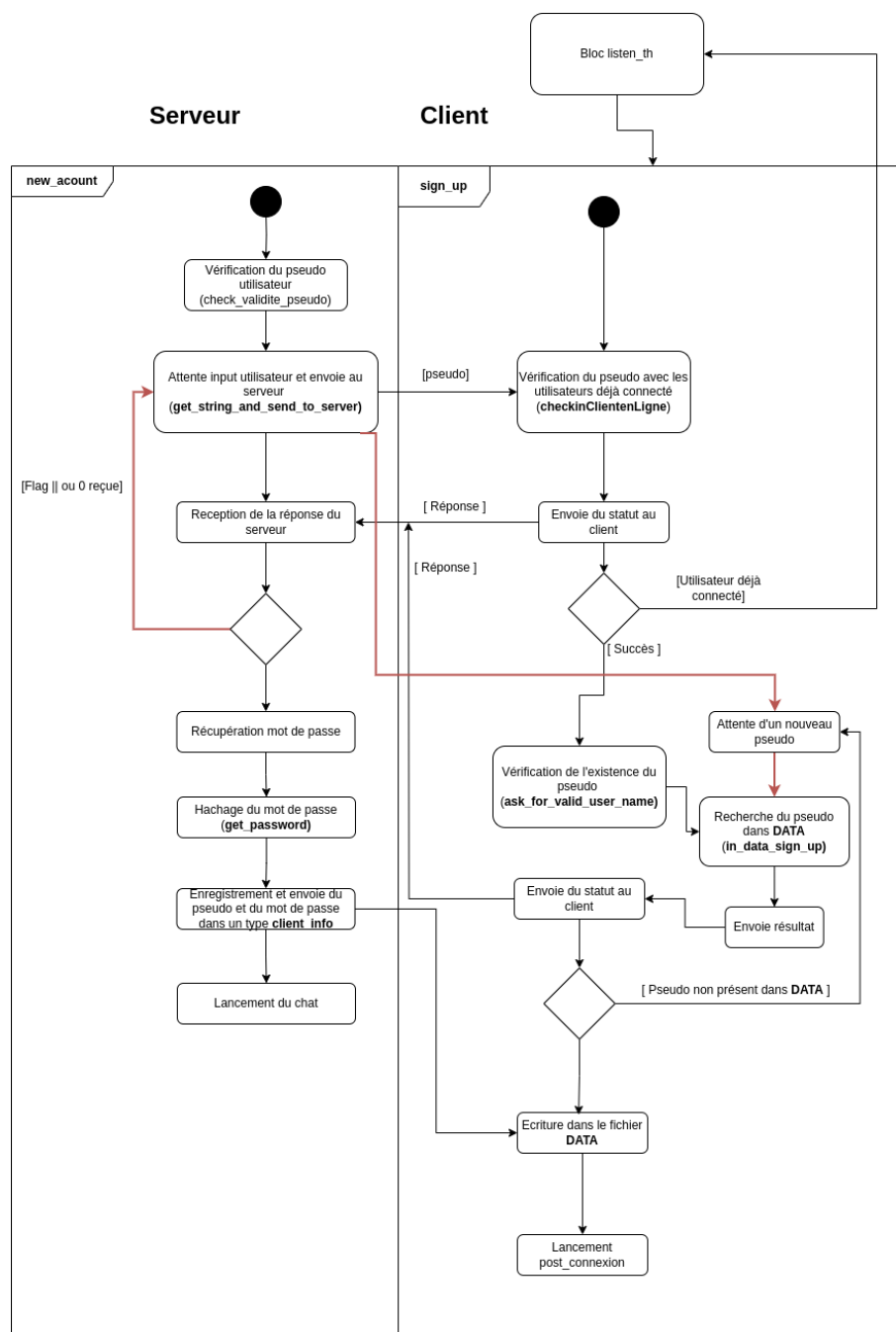


FIGURE 7 – Diagramme d'activité des nouvelles connexions

la date grâce à la fonction **new\_nom** qui fait appel à la bibliothèque **time.h**. À chaque appel de **distribuer**, on écrit dans ce fichier et on évite d'écrire des caractères inconnus via **fflush**.

## 4 Interface graphique

Pour affiner davantage l'expérience utilisateurs, un client de messagerie classique dispose généralement d'une interface graphique pour enrichir l'expérience utilisateur. C'est pourquoi nous avons décidé d'en implémenter une.

### 4.1 Principe

Notre interface graphique a été écrite en **Python**, c'est un langage très versatile qui permet d'écrire rapidement des applications fonctionnelles, d'où le choix de ce langage.

L'implémentation d'une interface graphique devait se faire en minimisant les changements du code principal et ceux afin d'éviter un trop grand nombre d'incompatibilités. Dans notre dernière version, l'interface graphique se comporte comme une interface qui communique directement avec le client. Le client, c'est-à-dire, l'exécution du fichier `./cli` va servir d'intermédiaire entre le serveur et l'interface graphique, comme le montre le schéma suivant.

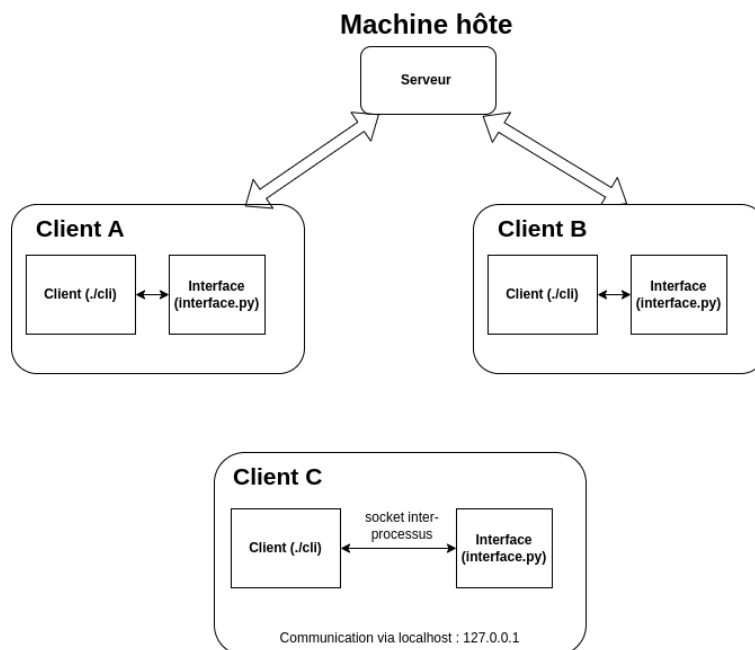


FIGURE 8 – Schéma des communications après l'ajout de l'interface graphique

Le client et l'interface graphique communiquent à travers localhost. Un socket est donc

utilisé entre le programme client et le programme d'interface graphique pour échanger des informations.

## 4.2 Implémentation

Le programme **Python** doit faire appel à un certain nombre de bibliothèques pour fonctionner :

- **tkinter** pour générer l'interface graphique
- **ttkbootstrap** une extension qui permet d'étoffer la partie graphique de l'interface
- **socket** pour l'utilisation des sockets sur **Python**
- **threading** pour l'utilisation de threads dans **Python**

L'interface graphique est réalisée en plusieurs phases :

1. Chargement d'une image de fond d'écran via **PIL** et définition de la résolution de l'image
2. Génération des boutons (Sign up, Sign in,...)
3. Génération des champs, user name et mot de passe pour l'inscription et la connexion
4. Génération du chat box

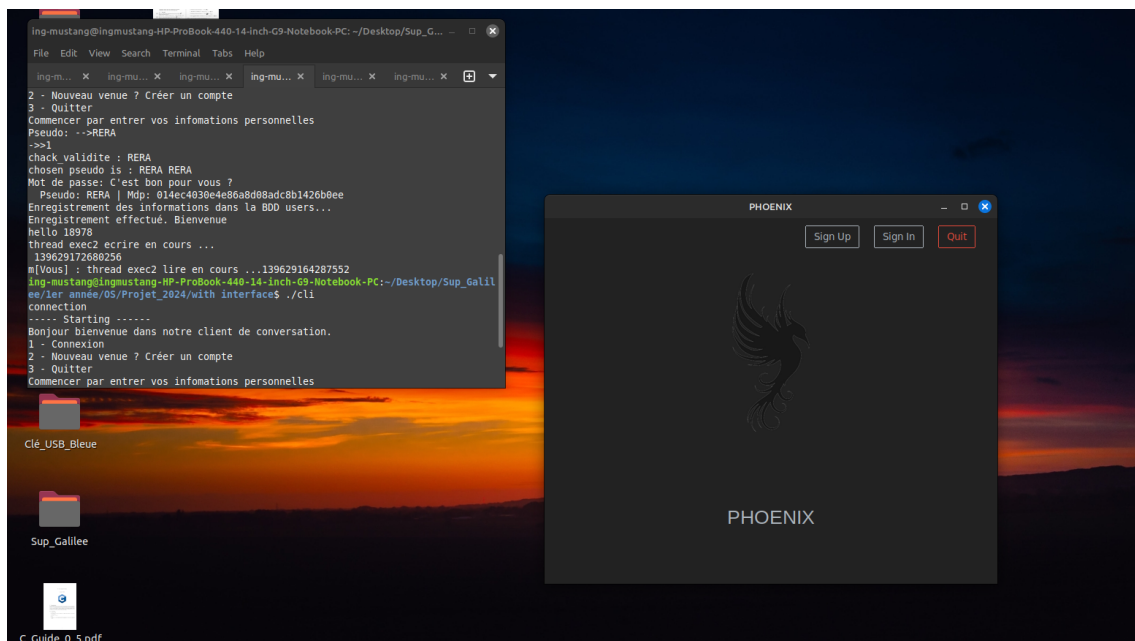


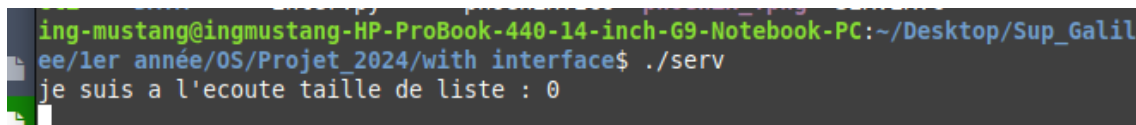
FIGURE 9 – Illustration de l'interface graphique

Le détailler du code se retrouve en Annexe.

### 4.3 Essai

Dans cet essai, deux machines vont communiquer, la machine A fait office de serveur et de client et la machine B sera uniquement un client. Le client de la machine A est **aissa\_pansan** tandis que le client de la machine B est **hamza**.

Pour commencer la conversation, la machine A lance le serveur via `./serv`, il est désormais prêt à accepter les connexions de n'importe quels clients.



```
ing-mustang@ingmustang-HP-ProBook-440-14-inch-G9-Notebook-PC:~/Desktop/Sup_Galilee/1er année/OS/Projet_2024/with interface$ ./serv
je suis a l'ecoute taille de liste : 0
```

FIGURE 10 – Démarrage du serveur

Le client *aissa pansan* va lancer l'interface Python, puis lancée le programme `./cli` pour qu'il puisse se connecter au serveur. Une fois cela fait, il peut se connecter et va ainsi accéder au client de messagerie. (Figure 11)

Le client *Hamza* va lui aussi suivre la même procédure que le client A. Il sera alors en mesure de parler avec le client A. (Figure 12)

Quand le client *Hamza* décide de quitter la communication (au moyen de Ctrl+C), le serveur en est notifié et il l'enlève de la liste des clients connectés. Le processus de communication peut toujours continuer (Figure 13)

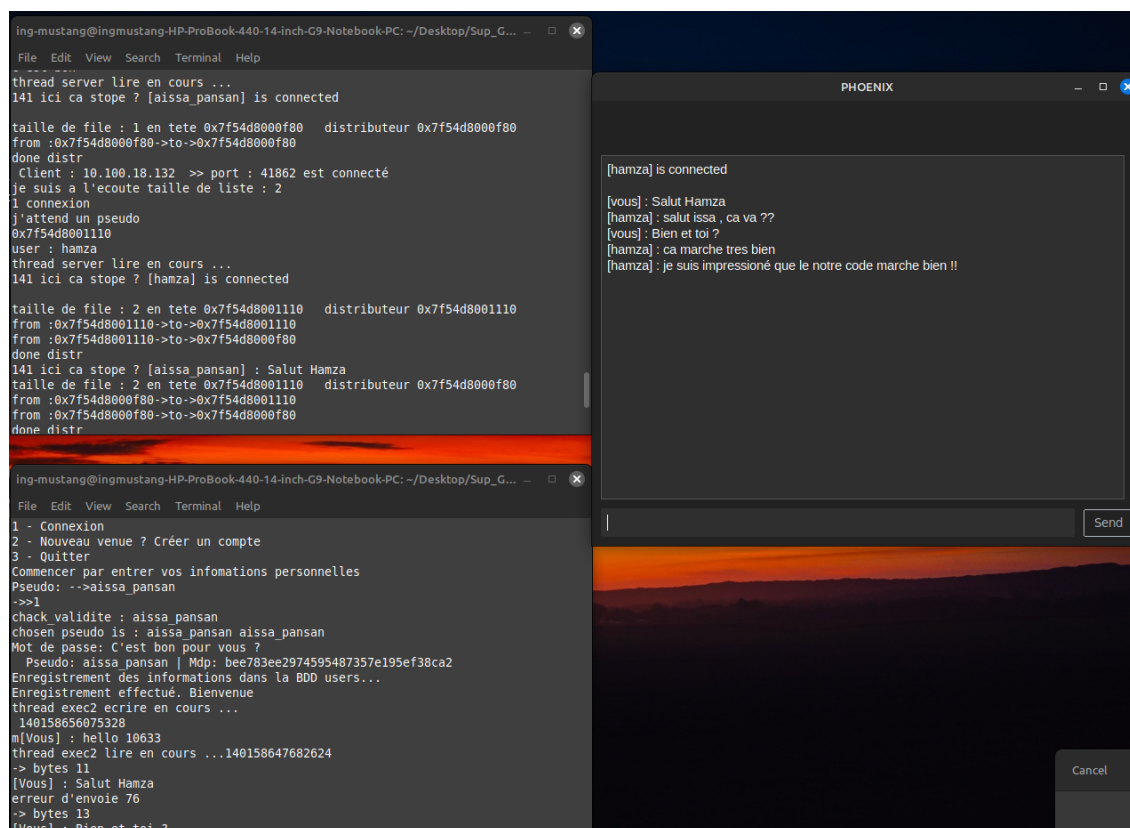


FIGURE 11 – Discussion entre le client A et B, à travers la machine A

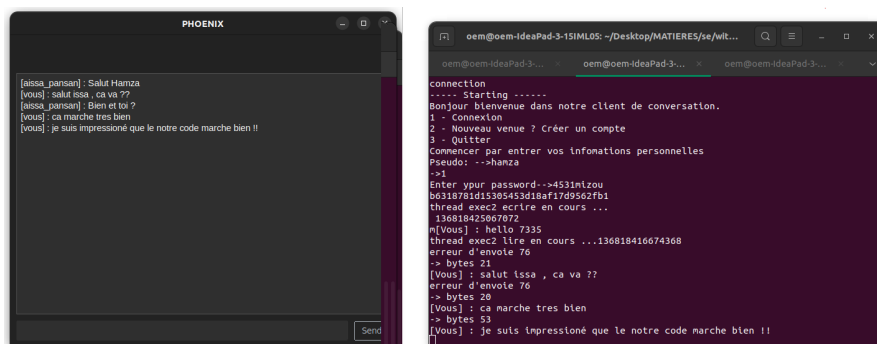


FIGURE 12 – Discussion entre le client A et B, à travers la machine B

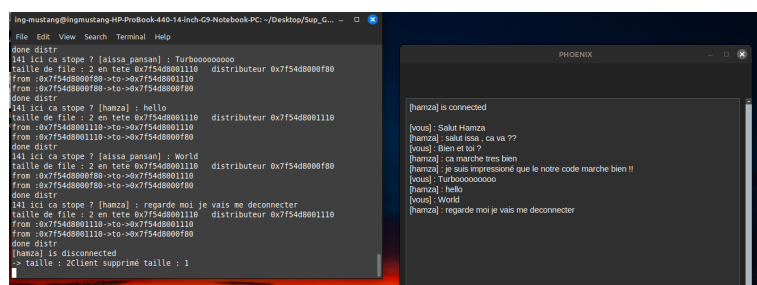


FIGURE 13 – Déconnexion du client Hamza

## 4.4 Modification du code principal

Pour que la version 2 du code puisse fonctionner avec l'interface graphique, il a fallu modifier certains points du fichier **CLIENT.c**. La principale modification est l'ajout d'un nouveau socket qui va directement communiquer avec l'interface graphique. Du côté de l'interface, le code Python va aussi générer un thread qui va communiquer avec le code client (exécutable `./cli`). Ainsi, par exemple, les saisies des logins ne sont plus capturées au moyen de **fgets**, mais plutôt par un appel à **recv** en destination du socket Python. En conséquence, la majeure partie des fonctions ont désormais en paramètres le descripteur du socket Python, comme avec la fonction **get\_string\_sendtoserver**.

---

```
void get_string_send_toserver(char pseudo[], int fd, int py)
{
    char buffer[256];
    size_t bytes;
    bytes = recv(py, buffer, sizeof(buffer), 0);
    if (bytes <= 0)
    {
        printf("erreur ligne 135\n");
    }
    buffer[bytes] = '\0';
    strcpy(pseudo, buffer);
    fprintf(stdout, "-->%s\n", buffer);
}
```

---

Pour pouvoir faire le tout, il est nécessaire de lancer l'exécutable python (`interface.py`) via `python3 interface.py` (L'utilisation de Python3 est fortement recommandée) puis l'exécutable client (`./cli`)



## 5 Conclusion générale

Ce projet a constitué une expérience enrichissante à de nombreux égards. Il a permis d'avoir une compréhension plus fine du cours de Système d'exploitation, notamment en ce qui concerne l'utilisation des threads, des tubes nommés et des sémaphores. Ce projet nous a également aidé à découvrir de nouvelles notions telles que les sockets et l'utilisation de python pour réaliser une interface graphique. Cette expérience a été très formatrice pour nous et nous sommes reconnaissants d'avoir eu cette opportunité de développement.

## 6 Point d'améliorations

Malgré la présence de nombreuses fonctionnalités dans notre projet, certains points restent encore à améliorer. Nous n'avons pas eu le temps de mettre en place une communication fermé entre deux utilisateurs. Actuellement, le chat ne se fait qu'en groupe.

Pour améliorer davantage l'expérience utilisateurs, il aurait aussi pertinents de mettre en place un système de commande pour pouvoir quitter plus facilement une fonction. Pour ce faire, il aurait fallu interpréter chacun des messages afin de remarquer des commandes qui commenceraient avec un point d'interrogation ( "!quit" pour quitter).

## 7 Annexe

### 7.1 Définition des fonctions utilisée pour les sockets

**socket :**

**Entrées :** `int` domain , `int` type, `int` protocol

**Sortie :** `int`

**Fonctionnement :** La fonction `socket` va générer un socket en fonction du domain, du type et du protocole choisis. Dans le cadre de ce projet, nous avons utilisé les paramètres `AF_INET` pour *domain* et le type `SOCK_STREAM` pour spécifier l'utilisation du protocole TCP\_IP et le paramètre *protocol* à 0. La fonction renvoie un entier, de la même manière qu'un descripteur de fichier, qu'on utilisera pour manipuler le socket.

**bind :**

**Entrées :** `int` sockfd, `const struct sockaddr *addr`,

**Sortie :** `int`

**Fonctionnement :** La fonction `bind` est utilisée pour lier un nom à un socket. Elle prend en entrée le descripteur de fichier du socket (*sockfd*), une structure (*addr*) contenant l'adresse à laquelle le socket sera lié, et la longueur de cette structure (*addrlen*). Cette fonction permet essentiellement d'associer une adresse IP et un numéro de port à un socket non nommé avant. La valeur de retour est un entier, 0 en cas de succès et -1 en cas d'erreur.

**listen :**

**Entrées :** `int` sockfd, `int` backlog

**Sortie :** `int`

**Fonctionnement :** La fonction `listen` marque le socket référencé par *sockfd* comme un socket passif, prêt à accepter les connexions entrantes. Le paramètre *backlog* spécifie la longueur maximale de la file d'attente des connexions en attente d'acceptation. Ici, on

a 3 connexions maximales. La valeur de retour est un entier, 0 en cas de succès et -1 en cas d'erreur.

**accept :**

**Entrées :** `int sockfd, struct sockaddr *addr, socklen_t *addrlen`

**Sortie :** `int`

**Fonctionnement :** La fonction `accept` est utilisée avec les sockets de type `SOCK_STREAM` pour accepter une nouvelle connexion sur un socket en attente. Elle extrait la première connexion en attente sur la file d'attente des connexions du socket référencé par *sockfd* et crée un nouveau socket avec le même descripteur de fichier pour cette connexion. L'adresse du client connecté est stockée dans *addr* si celui-ci n'est pas nul. La longueur de l'adresse est stockée dans *addrlen*. La valeur de retour est un entier représentant le descripteur de fichier du nouveau socket, ou -1 en cas d'erreur.

**connect :**

**Entrées :** `int sockfd, const struct sockaddr *addr, socklen_t addrlen`

**Sortie :** `int`

**Fonctionnement :** La fonction `connect` est utilisée par un client pour établir une connexion avec un serveur. Elle prend en entrée le descripteur de fichier du socket (*sockfd*), une structure (*addr*) contenant l'adresse et le numéro de port du serveur auquel se connecter, ainsi que la longueur de cette structure (*addrlen*). La valeur de retour est un entier, 0 en cas de succès et -1 en cas d'erreur. Il est à noter que la fonction fait un appel implicite à la fonction `bind`.

**recv :**

**Entrées :** `int sockfd, void *buf, size_t len, int flags`

**Sortie :** `ssize_t`

**Fonctionnement :** La fonction `recv` est utilisée pour recevoir des données sur un socket de type `SOCK_STREAM`. Elle prend en entrée le descripteur de fichier du socket

(*sockfd*), un tampon (*buf*) où stocker les données reçues, la longueur de ce tampon (*len*), et des indicateurs (*flags*) pour contrôler le comportement de la fonction. La valeur de retour est le nombre d'octets reçus, ou -1 en cas d'erreur.

**send :**

**Entrées :** `int sockfd`, `const void *buf`, `size_t len`, `int flags`

**Sortie :** `ssize_t`

**Fonctionnement :** La fonction `send` est utilisée pour envoyer des données sur un socket de type `SOCK_STREAM`. Elle prend en entrée le descripteur de fichier du socket (*sockfd*), un tampon (*buf*) contenant les données à envoyer, la longueur de ce tampon (*len*), et des indicateurs (*flags*) pour contrôler le comportement de la fonction. La valeur de retour est le nombre d'octets envoyés, ou -1 en cas d'erreur.

**shutdown :**

**Entrée :** `int sockfd`, `int how`

**Sortie :** `int`

**Fonctionnement :** La fonction `shutdown` est utilisée pour fermer une partie d'une connexion TCP/IP. Elle prend en entrée le descripteur de fichier du socket (*sockfd*) et un indicateur (*how*) indiquant quelle partie de la connexion fermer (0 pour la lecture, 1 pour l'écriture, 2 pour les deux). La valeur de retour est un entier, 0 en cas de succès et -1 en cas d'erreur.

## 7.2 Fonction communication client/server sur socket

**accept\_thread :**

**Entrées :** `void* arg(Clients->tete)` **Sortie :** `void*`

**Fonctionnement :** La fonction va paramétrer les paramètres de socket client et va générer un thread qui va exécuter la fonction `listenth`.

Il est important de remarquer que si la génération du thread a réussi, le thread parent

va se détacher de son thread fils et va ré-exécuter la boucle en attente d'une connexion client

**listenth :**

**Entrées :** `void*` `arg(Clients->tete)` **Sortie :** `void*`

**Fonctionnement :** La fonction va enregistrer les choix utilisateur lié au menu et va en fonction exécuter une routine en fonction.

**post\_connexion :**

**Entrées :** `Structclient*` `temp` **Sortie :** `void`

**Fonctionnement :** La fonction va générer un thread qui va exécuter la fonction `post_connect`

**post\_connect :**

**Entrée :** `void*` `arg(Structclient)`

**Sortie :** `void*`

**Fonctionnement :** La fonction est en écoute perpétuelle d'un message provenant d'un client, quand le serveur le réceptionne, il lance exécute la fonction **distribuer**

**distribuer :**

**Entrée :** `char` `message[]`, `size_t` `sizemsg`, `Structclient*` `destributeur`

**Sortie :** `void`

**Fonctionnement :** La fonction envoie à tous les utilisateurs connectés, le message envoyé par **post\_connexion** au moyen de la struct **FIFO** via des mutex classique et mutex `rwlock`

### 7.3 Fonction pour la connexion

**sign\_in :**

**Entrée :** `Structclient *` arg

**Sortie :** `int`

**Fonctionnement :** La fonction gère le processus de connexion d'un client. Elle reçoit le pseudo, contrôle s'il n'est pas déjà associé à un utilisateur déjà connecté via **checkinClientsenligne** et si le pseudo existe dans **DATA** via **ask\_for\_valid\_username**. Ensuite, elle compare si le mot de passe reçu et le même avec celui stocké dans **DATA**. Si les informations sont correctes, elle appelle **post\_connexion** et retourne **EXIT\_SUCCESS**, sinon elle retourne **EXIT\_FAILURE**.

**comparerChaine :**

**Entrées :** `char *` char1, `const char *` chaineComparaison

**Sortie :** `int`

**Fonctionnement :** La fonction **comparerChaine** compare deux chaînes de caractères à l'aide de la fonction **strcmp**. Elle retourne 1 si les chaînes sont identiques, et 0 sinon. Elle a été utilisée pour inverser la réponse de **strcmp**

**checkinClientsenligne :**

**Entrées :** `char []` pseudo, `int` fd

**Sortie :** `int`

**Fonctionnement :** La fonction vérifie si un pseudo est déjà connecté en parcourant une liste de clients en ligne. Si le pseudo est trouvé, elle envoie un message d'erreur au client et retourne **EXIT\_FAILURE**. Sinon, elle retourne **EXIT\_SUCCESS**.

**in\_data\_sign\_in :**

**Entrées :** `char []` pseudo, `char []` password

**Sortie :** `int`

**Fonctionnement** : La fonction `in_data_sign_in` vérifie si un pseudo existe dans le fichier de données `DATA`. Si le pseudo est trouvé, elle copie le mot de passe correspondant dans le tampon `password` et retourne 1. Si le pseudo n'est pas trouvé, elle copie "&&" dans `password` et retourne 0. Le mot de passe sera utiliser ensuite par `check_for_user_name_validity`

`check_for_user_name_validity` :

**Entrées** : `char [] pseudo`, `int fd`, `char [] Password`

**Sortie** : `void`

**Fonctionnement** : La fonction vérifie si un pseudo existe dans le fichier de données `DATA` en appelant `in_data_sign_in` et en vérifiant le champ **Password**. Si ce dernier correspond à "&&", alors cela signifiera que le pseudo n'existe pas dans `DATA`. Le champ **Password** est initialisé avec "&&" qui sera vérifié une dernière fois dans `sign_in`. Si tous, c'est bien passé, le champ **Password** est vide.

`new_infos_client` :

**Entrée** : `void`

**Sortie** : `client_info *`

**Fonctionnement** : La fonction alloue dynamiquement de la mémoire pour une nouvelle structure `client_info` et retourne un pointeur vers celle-ci. Si l'allocation échoue, elle affiche un message d'erreur et retourne `NULL`.

## 7.4 Fonction pour l'inscription

`sign_up` :

**Entrée** : `Structclient * arg`

**Sortie** : `int`

**Fonctionnement** : La fonction gère le processus d'inscription d'un client. Elle vérifie si le pseudo est déjà en ligne et s'il existe dans **DATA** enregistre les informations du client

dans DATA via le struct **client\_info** qui est reçue. Si l'inscription réussit, elle appelle **post\_connexion** et retourne EXIT\_SUCCESS, sinon elle retourne EXIT\_FAILURE.

**in\_data\_sign\_up :**

**Entrée :** `char []` pseudo

**Sortie :** `int`

**Fonctionnement :** La fonction vérifie si un pseudo est déjà présent dans le fichier de données DATA. Elle ouvre le fichier en mode ajout et lecture, puis parcourt chaque ligne pour comparer les pseudos. Si le pseudo est trouvé, elle retourne 1, sinon elle retourne 0.

**ask\_for\_valid\_user\_name :**

**Entrées :** `char *` pseudo, `int` fd

**Sortie :** `void`

**Fonctionnement :** La fonction `ask_for_valid_user_name` contrôle le résultat de **in\_data\_sign\_in** dans une boucle while. Ainsi, tant que l'utilisateur n'envoie pas de pseudo valide à la connexion, il devra recommencer la saisie. L'utilisateur en est informé par l'envoi du flag 0. Si tout ce passe bien, on envoie le flag 1 au client.

## 7.5 Définition des fonctions Python

**threaded\_network\_connection :**

**Entrées :** Aucun

**Sortie :** Aucun

**Fonctionnement :** La fonction *threaded\_network\_connection* démarre un nouveau thread qui exécute la fonction *start\_server*. Cela permet de gérer les opérations du serveur en arrière-plan.

**send\_message :**

**Entrées :** `int` action\_number



**Sortie :** Aucun

**Fonctionnement :** La fonction *send\_message* envoie un numéro d'action au serveur via le socket client. Elle encode l'entier en chaîne de caractères avant de l'envoyer.

**send\_message\_th :**

**Entrées :** *str* message

**Sortie :** Aucun

**Fonctionnement :** La fonction *send\_message\_th* encode un message texte en UTF-8 et l'envoie au serveur via le socket client.

**start\_server :**

**Entrées :** Aucun

**Sortie :** Aucun

**Fonctionnement :** La fonction *start\_server* initialise un socket serveur qui écoute sur l'adresse IP locale **127.0.0.1** et le port **6667**. Elle attend les connexions des clients et accepte les connexions entrantes.

**display\_content :**

**Entrées :** *event* event (optionnel)

**Sortie :** Aucun

**Fonctionnement :** La fonction *display\_content* récupère le texte saisi par l'utilisateur, l'affiche dans la zone de texte du chat, et envoie ce texte au serveur.

**display\_content\_th :**

**Entrées :** *str* message

**Sortie :** Aucun

**Fonctionnement :** La fonction *display\_content\_th* affiche un message dans la zone de texte du chat.

#### **threaded\_reception :**

**Entrées :** Aucun

**Sortie :** Aucun

**Fonctionnement :** La fonction *threaded\_reception* démarre un nouveau thread qui exécute la fonction *receive\_messages*. Cela permet d'écouter les messages entrants du serveur en arrière-plan.

#### **receive\_messages :**

**Entrées :** Aucun

**Sortie :** Aucun

**Fonctionnement :** La fonction *receive\_messages* écoute les messages entrants du serveur et les affiche dans la zone de texte du chat.

#### **chat :**

**Entrées :** Aucun

**Sortie :** Aucun

**Fonctionnement :** La fonction *chat* met à jour l'interface utilisateur pour afficher l'interface de chat et démarre la réception de messages.

#### **affiche\_signIn :**

**Entrées :** Aucun

**Sortie :** Aucun

**Fonctionnement :** La fonction *affiche\_signIn* met à jour l'interface utilisateur pour afficher le formulaire de connexion et envoie un message au serveur pour indiquer l'action de connexion.

#### **affiche\_signUp :**

**Entrées :** Aucun

**Sortie :** Aucun

**Fonctionnement :** La fonction *affiche\_signUp* met à jour l'interface utilisateur pour afficher le formulaire d'inscription et envoie un message au serveur pour indiquer l'action d'inscription.

**submitin :**

**Entrées :** Aucun

**Sortie :** Aucun

**Fonctionnement :** La fonction *submitin* gère le processus de connexion de l'utilisateur. Elle envoie les informations d'identification (nom d'utilisateur et mot de passe) au serveur et affiche les messages appropriés en fonction de la réponse du serveur.

**submitup :**

**Entrées :** Aucun

**Sortie :** Aucun

**Fonctionnement :** La fonction *submitup* gère le processus d'inscription de l'utilisateur. Elle envoie les informations d'inscription (nom d'utilisateur et mot de passe) au serveur et affiche les messages appropriés en fonction de la réponse du serveur.

**afficher\_message\_labelin :**

**Entrées :** Aucun

**Sortie :** Aucun

**Fonctionnement :** La fonction *afficher\_message\_labelin* affiche un message d'erreur indiquant qu'une tentative de connexion a échoué.

**afficher\_vide\_labelin :**

**Entrées :** Aucun

**Sortie :** Aucun

**Fonctionnement :** La fonction *afficher\_vide\_labelin* affiche un message d'erreur indiquant que les champs de connexion sont vides.

**afficher\_notexist\_labelin :**

**Entrées :** Aucun

**Sortie :** Aucun

**Fonctionnement :** La fonction *afficher\_notexist\_labelin* affiche un message d'erreur indiquant que le compte utilisateur n'existe pas.

**afficher\_enligne\_labelin :**

**Entrées :** Aucun

**Sortie :** Aucun

**Fonctionnement :** La fonction *afficher\_enligne\_labelin* affiche un message d'erreur indiquant que l'utilisateur est déjà en ligne.

**afficher\_success\_labelin :**

**Entrées :** Aucun

**Sortie :** Aucun

**Fonctionnement :** La fonction *afficher\_success\_labelin* affiche un message indiquant que la connexion a réussi.

**afficher\_message\_labelup :**

**Entrées :** Aucun

**Sortie :** Aucun

**Fonctionnement :** La fonction *afficher\_message\_labelup* affiche un message d'erreur indiquant qu'une tentative d'inscription a échoué.

**afficher\_vide\_labelup :**

**Entrées :** Aucun

**Sortie :** Aucun

**Fonctionnement :** La fonction *afficher\_vide\_labelup* affiche un message d'erreur indiquant que les champs d'inscription sont vides.

**afficher\_exist\_labelup :**

**Entrées :** Aucun

**Sortie :** Aucun

**Fonctionnement :** La fonction *afficher\_exist\_labelup* affiche un message d'erreur indiquant que le compte utilisateur existe déjà.

**quit\_func :**

**Entrées :** Aucun

**Sortie :** Aucun

**Fonctionnement :** La fonction *quit\_func* ferme le socket client et quitte l'application.

## 7.6 Codage de l'interface graphique

```
# -----Main-----

window = tk.Window(themename="darkly")
window.title("PHOENIX")

try:
    # Chargement de l'image avec PIL
    icon = Image.open('phoenix.png')
    photo_icon = ImageTk.PhotoImage(icon)
    window.iconphoto(False, photo_icon)
except Exception as e:
    print(f"Erreur lors de la définition de l'icône: {e}")

    # Chargement de l'image avec PIL
window.geometry('620x500')
front_img = Image.open('phoenix_.png')
target_width = 230
```

```
ratio = target_width / front_img.width
target_height = int(front_img.height * ratio)
resized_img = front_img.resize((target_width, target_height), Image.Resampling.LANCZOS)
front_photo = ImageTk.PhotoImage(resized_img)
threaded_network_connection()

# Création et placement des boutons dans un cadre
button_frame = tb.Frame(window)
button_frame.grid(row=0, column=2, sticky='en', padx=(10, 20), pady=(10, 0))
#placement de l'image
image_label=Label(window,image=front_photo)
image_label.grid(row=1, column=0, columnspan=3, pady=(10, 0), sticky='ew')
# Bouton 'Sign Up'
signup_btn = tb.Button(button_frame, text="Sign Up", bootstyle="light outline",command=affiche_signUp)
signup_btn.grid(row=0, column=0, padx=10)

#Bouton 'Sign In'
signin_btn = tb.Button(button_frame, text="Sign In", bootstyle="light outline",command=affiche_signIn)
signin_btn.grid(row=0, column=1, padx=10)

# Bouton 'Quit'
quit_btn = tb.Button(button_frame, text="Quit", bootstyle="danger outline", command=quit_func)
quit_btn.grid(row=0, column=2, padx=10)

# creation de la label frame
message_label=tb.Label(window,text="PHOENIX",font=("Arial",20),bootstyle="light")
message_label.grid(row=3,columnspan=3,pady=20)

# creation et placement de l'entry de username et password et le submit btn de sign in
signIn_frame=tb.Frame(window)
signIn_frame.grid(row=2,columnspan=3,pady=20)
user_labelin=tb.Label(signIn_frame,text="Username",bootstyle="light")
user_labelin.grid(row=0,column=0,padx=10,pady=10)
user_entryin=tb.Entry(signIn_frame,bootstyle="darkly")
user_entryin.grid(row=0,column=1,padx=10,pady=10)

pass_labelin=tb.Label(signIn_frame,text="Passeword",bootstyle="light")
pass_labelin.grid(row=1,column=0,padx=10,pady=(0,10))
pass_entryin=tb.Entry(signIn_frame,bootstyle="darkly",show="*")
pass_entryin.grid(row=1,column=1,padx=10,pady=(0,10))
vide_msgin=tb.Label(signIn_frame,text="Veuillez remplir tout les champs",bootstyle="danger" ,font=('Arial', 10))
```

```
messagein=tb.Label(signIn_frame,text="Ce username exist deja ou bien il contient une ;",bootstyle="danger"
,font=('Arial', 10))
not_existin=tb.Label(signIn_frame,text="Ce username n'exist pas",bootstyle="danger" ,font=('Arial', 10))
en_lignein=tb.Label(signIn_frame,text="Ce username est deja en ligne",bootstyle="danger" ,font=('Arial', 10))
successin=tb.Label(signIn_frame,text="success",bootstyle="danger" ,font=('Arial', 10))
submit_btnin = tb.Button(signIn_frame, text="Submit in", bootstyle="light outline",command=submitin)
submit_btnin.grid(row=3, columnspan=2,pady=(10))

# creation et placement de l'entry de username et password et le submit btn de sign up
signUp_frame=tb.Frame(window)
#signUp_frame.grid(row=2,columnspan=3,pady=20)
user_labelup=tb.Label(signUp_frame,text="Username",bootstyle="light")
user_labelup.grid(row=0,column=0,padx=10,pady=10)
user_entryup=tb.Entry(signUp_frame,bootstyle="darkly")
user_entryup.grid(row=0,column=1,padx=10,pady=10)
pass_labelup=tb.Label(signUp_frame,text="Passeword",bootstyle="light")
pass_labelup.grid(row=1,column=0,padx=10,pady=(0,10))
pass_entryup=tb.Entry(signUp_frame,bootstyle="darkly")
pass_entryup.grid(row=1,column=1,padx=10,pady=(0,10))
vide_msgup=tb.Label(signIn_frame,text="Veuillez remplir tout les champs",bootstyle="danger" ,font=('Arial', 10))
messageup=tb.Label(signIn_frame,text="Ce username contient une ;",bootstyle="danger" ,font=('Arial', 10))
existup=tb.Label(signIn_frame,text="Ce username existe deja",bootstyle="danger" ,font=('Arial', 10))
en_ligneup=tb.Label(signIn_frame,text="Ce username est deja en ligne",bootstyle="danger" ,font=('Arial', 10))
successup=tb.Label(signIn_frame,text="success",bootstyle="danger" ,font=('Arial', 10))

submit_btnup = tb.Button(signUp_frame, text="Submit up", bootstyle="light outline",command=submitup)
submit_btnup.grid(row=2, columnspan=2,pady=(20))

# chat box
chat_frame=tb.Frame(window)
chat_frame.grid_rowconfigure(0, weight=1) # Permet à la ligne du chat_text de s'étendre
chat_frame.grid_rowconfigure(1, weight=0) # Garde la ligne du entry_button_frame sans expansion
chat_frame.grid_columnconfigure(0, weight=1)
#chat_frame.grid(row=4,column=0,columnspan=3,pady=10,sticky="sew")
chat_scrol=tb.Scrollbar(chat_frame,orient="vertical",bootstyle="light")
chat_scrol.grid(row=0,column=3,pady=10,sticky="nse")
chat_text=tb.Text(chat_frame,yscrollcommand=chat_scrol.set,wrap="none",font=("Arial",11))
chat_text.grid(row=0,column=0,columnspan=2,pady=10,padx=10,sticky='ew')

entry_button_frame=tb.Frame(chat_frame)
entry_button_frame.grid_rowconfigure(0, weight=1) # Permet à la ligne du chat_text de s'étendre
entry_button_frame.grid_columnconfigure(0, weight=1)
```

```
entry_button_frame.grid_columnconfigure(2, weight=0)
entry_button_frame.grid(row=1,columnspan=3,sticky='ew')
chat_entry=tb.Entry(entry_button_frame,bootstyle="darkly",width=60,font=("Arial",11))
chat_entry.bind("<Return>", display_content)
chat_button=tb.Button(entry_button_frame, text="Send", bootstyle="light outline",command=display_content)#,command=send
chat_button.grid(row=0, column=2,sticky='es')
chat_entry.grid(row=0,column=0,padx=10,sticky="sew")
```

```
# Configuration des poids de colonnes pour permettre l'expansion centrée de l'image
window.grid_columnconfigure(0, weight=1)
window.grid_columnconfigure(1, weight=1)
window.grid_columnconfigure(2, weight=1)
window.grid_rowconfigure(0, weight=0)
window.grid_rowconfigure(1, weight=1)
window.grid_rowconfigure(2, weight=1)
window.grid_rowconfigure(3, weight=1)
window.grid_rowconfigure(4, weight=1)
```

```
window.mainloop()
```