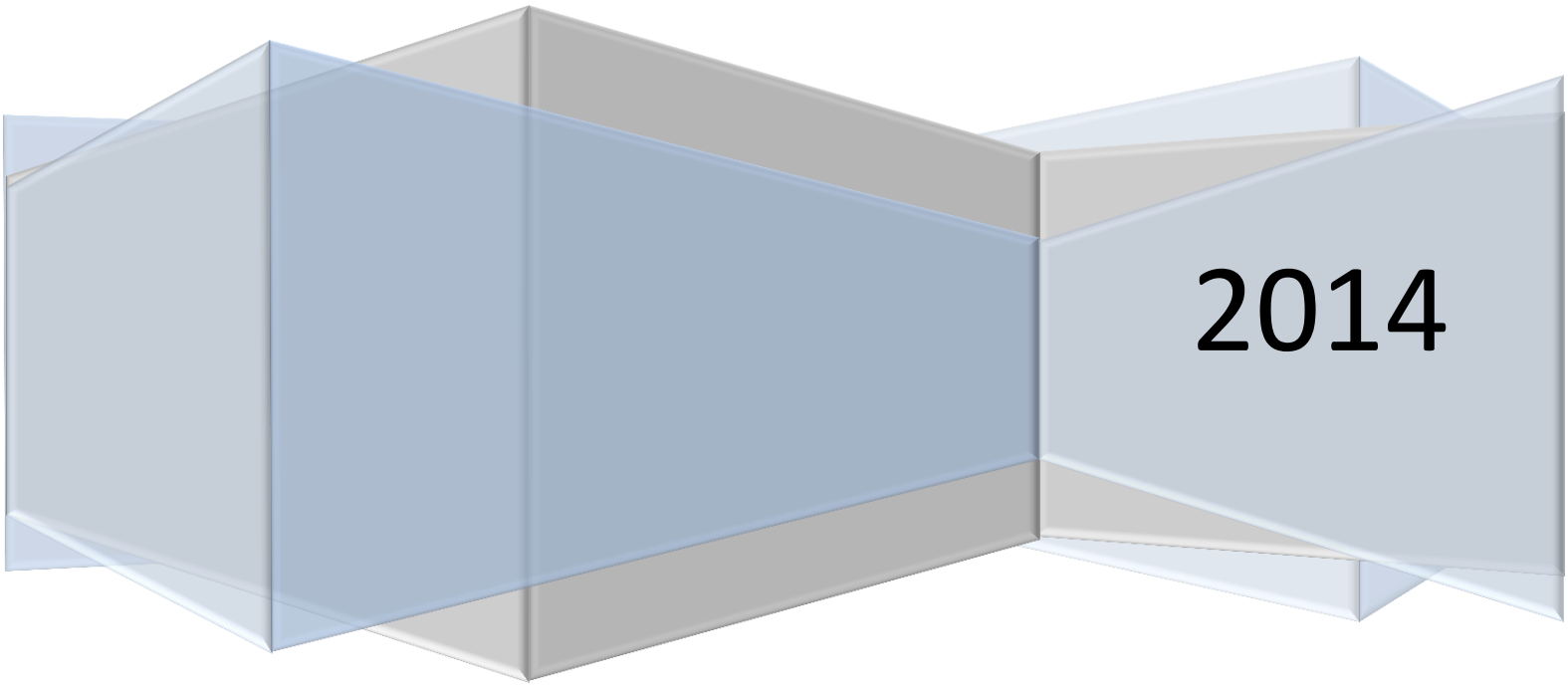
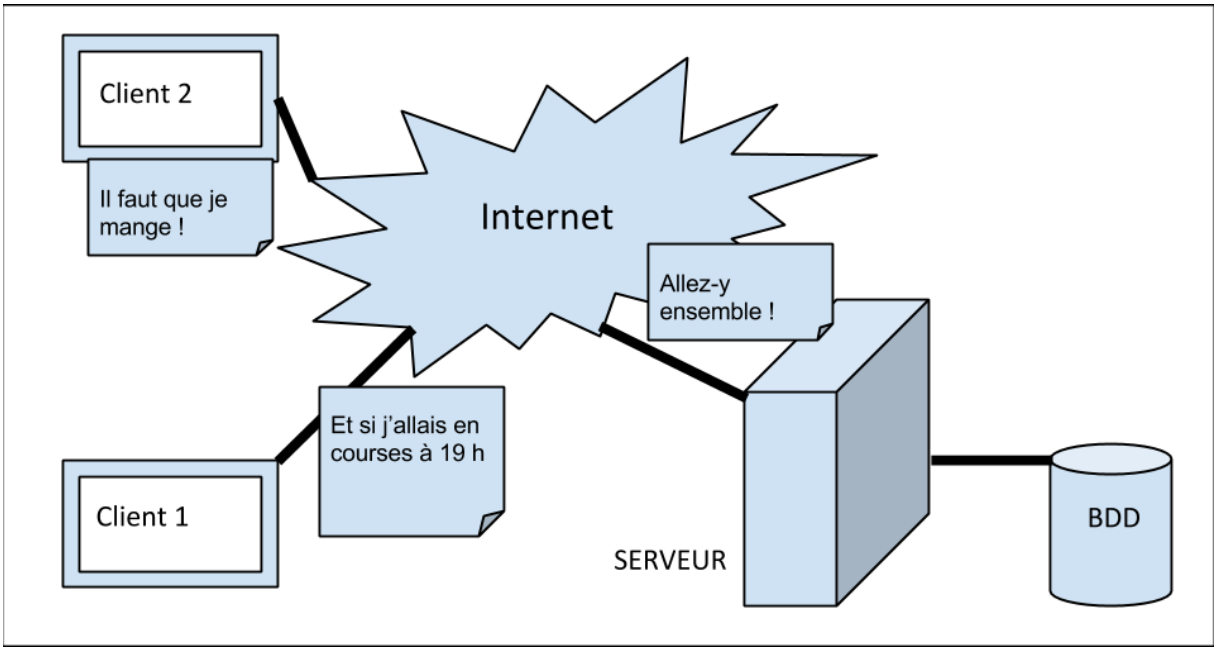
Rapport projet programmation de système d'exploitation

Contamain Kilian-Humbert Cedric



2014

Contenu



1. Présentation du projet TOMM (Tag Oriented Market Maker) :

Ce logiciel a pour but de mettre en relation les membres d'une communauté qui souhaitent échanger des services ou des biens en accédant aux marchés proposés sous la forme de Tag. Y sont différents sujet de conversation comme par exemple le Sport ou l'Ecole. L'utilisateur a donc l'accès à tous les posts des threads comme par exemple dans le cas rugby où il va pouvoir consulter les postes comme « A quelle heure est l'entrainement demain ? ». Chaque utilisateur qui a été enregistré peut souscrire à un thread et voir les postes en relation avec le Tag souscrit.

L'utilisateur peut bien sûr créer des postes sur les Threads et même créer un nouveau sujet de discussion.

En outre, il sera question dans un deuxième temps d'introduire une gestion du temps de sorte à ne pas "spammer" les utilisateurs avec des annonces ne correspondant pas à leur attentes temporelles.

2. Fonctionnement du système :

Au lancement du programme, l'utilisateur peut voir une sélection de commande pour se connecter :

-la commande login Nomd'utilisateur Motdepasse qui permet de s'authentifier auprès du serveur.

-la commande create Nomd'utilisateur Motdepasse qui permet d'enregistrer son compte dans la base de donner du serveur.

-la commande help qui permet d'avoir des détails sur toutes les commandes et leurs arguments.

Chaque utilisateur devra donc se connecter avant de pouvoir utiliser toutes les fonctionnalités de TOMM.

Ensuite si l'utilisateur est authentifié par le serveur il arriver sur un deuxième menus où il devra rentrer d'autres commandes qui sont :

- la commande *see* Nomduthread qui permet d'affichers les posts qui sont dans le Thread nomé.

-la commande edit Nomduthread Message qui trouve le thread que l'utilisateur veut éditer et ensuite écris un poste dans le thread.

-la commande new Nomduthread permet de créer un nouveau thread.

-la commande help Nomdelacommande permet d'avoir des détails sur la commande entrée.

-la commande souscrire NomDuTag permet à l'utilisateur de suivre les nouveaux messages sur le thread.Elle n'est cependant pas implémentée ici.

3. Le Développement :

Le projet est par définition coupé en deux parties, un client et un serveur, nous avons donc mis en place une procédure formelle pour nommer les fonctions et les fichiers. On distingue trois groupes dans les fonctions qui ont été créées.

La partie TOMM est similaire pour le client et le serveur est joue essentiellement le rôle de gestion des listes chaînées contenant les données. Nous y avons cependant intégré la partie « IHM » car les tâches de l'interface se résument en grande partie à afficher les données.

La partie d'échange de donnée, les envois, réceptions de fichiers et le protocole qui régissent cet échange.

La partie instruction qui dicte le comportement des deux parties précédentes. Dans le code elle est commentée plus abondamment que les deux autres ce pour expliquer au mieux les caractéristiques de notre programme.

Dans la mesure du possible nous avons donc tenté de regrouper ces fonctions dans un même fichier et de décomposer en fonctions simples l'exécution des tâches.

Voici une présentation des fonctions qui composent le projet :

Le fichier login_serv.c :

Contient toutes les fonctions nécessaires à la partie gestion de l'utilisateur.

La fonction void Create_Comptes(Utilisateur *New_Us) :

Cette fonction permet de créer le compte de l'utilisateur en recevant en argument la structure Utilisateur. Tout d'abords on ouvre le fichier comptes.txt en mode protégé. Ensuite on lit la structure En tête qui est au début du fichier et qui permet de connaître le nombre d'utilisateur dans la base de données. Si l'en tête n'existe pas on verrouille le fichier grâce à un

mutex pour écrire le fichier en tête et éviter tout problème de concurrence. Ensuite on déverrouille le mutex. Ensuite on vérifie qu'il n'existe pas un utilisateur qui possède le même login et s'il y en a un on ferme le fichier et on arrête la fonction. Ensuite on se positionne après le dernier fichier. On verrouille le fichier avec un mutex pendant l'écriture, on écrit la structure, on déverrouille le fichier puis on le ferme.

La fonction void Encrire_En_Tete_Comptes(En_tete *Cur_Te)

Cette fonction écrit une en_tête pour les fichiers. Les en_tête sont très utiles pour permettre de connaitre ici le nombre de personne enregistré dans le fichier. Pour écrire l'en_tête, on se positionne au début du fichier et après on écrit avec fwrite().

int Validate_User(Utilisateur *New_Us) :

Cette fonction parcourt tous le fichier comptes.log. Il utilise l'en_tête pour savoir le nombre de structure qu'il doit lire, il compare le nom et le mot de passe de l'utilisateur pour chaque compte présent dans le fichier. Ensuite s'il trouve un nom et un mot de passe qui correspond on retourne vrai sinon on retourne faux.

La fonction void envoyerFichier(int sock,char * Buff) :

Cette fonction permet d'envoyer un fichier avec le nom contenu dans la variable Buff vers le client. Pour cela on ouvre le fichier et on test l'existence du fichier,

Le fichier TOMM_serv.c :

Contient toutes les fonctions nécessaire à la gestion des threads,des postes et de l'affichage.

La fonction void read_Message_TOMM(Message_TOMM *Mes_Lu, char *Nomfichier) :

La fonction lit le message qui est dans le fichier rentré en paramètre.

La fonction void write_Message_TOMM(Message_TOMM *Mes_Ec) :

Créer un fichier echanges.txt se met au début du fichier avec fseek puis ensuite il écrit la structure Mes_ec dans le fichier avant de le fermer.

La fonction Ecrire_Thread_Envoi(Thread *Cur_Th, Utilisateur *Cur_User) :

On créer un fichier nommée en fonction du nom de l'utilisateur pour ensuite écrire tous le thread avec tous les postes dans le fichier.

La fonction Thread *Lire_Thread_Reception(Utilisateur *Cur_User):

La fonction consiste à lire un thread reçu dans un fichier qui dépend du nom de l'utilisateur. On lit les postes dans les fichier puis on utilise la fonction Ajout_Post() pour rajouter le poste à la liste chainé comportant tous les poste.

La fonction void Ajout_Thread_BDD(Thread *New_Th, Thread *Root_Th) :

La fonction rajoute le thread en entrée à la suite de la liste chainée.

La fonction void Save_Data(Thread *Root_Th) :

La fonction permet d'enregistrer les threads et les postes correspondant dans un fichier. Le but est d'enregistrer la structure thread à la base du fichier puis écrire tous les postes des threads dans le fichier pour pouvoir les récupérer quand le serveur s'arrête.

La fonction Thread *Lire_Thread_BDD(Thread *Cur_Th) :

La fonction permet de lire les threads du fichier.

La fonction void Lire_BDD(Thread *Root_Th) :

La fonction permet de récupérer tous les threads du fichier après les avoir lu avec la fonction Lire_Thread_BDD. On créer une liste chainée de thread, cela permet d'avoir la base de données dans la ram.

La fonction void Generate_Data(Thread *Root_Th) :

Permet de générer une base de données dès le lancement.

La fonction void Get_Param(Message_TOMM *Mes, char *arg1, char *arg2) :

Cette fonction permet de récupérer les paramètres.

La fonction Post *Get_Post(Message_TOMM *Mes, Utilisateur *Cur_Us) :

Cette fonction récupère le poste selon le contenu du message et de l'utilisateur.

Le fichier Net_Serv.c :

La fonction void Start_up_TOMM(int argc, char *argv[]) :

Cette fonction est la fonction de base du serveur, cette fonction permet de se connecter avec le client.

La fonction void Interpret_Request(Message_TOMM *Mes, Utilisateur *Cur_Us) :

Cette fonction reçoit une donnée dans la structure message et selon la donnée la fonction lance d'autre fonction. Elle interprète le message en entrée.Pour cela on utilise un switch case.

La fonction void See_Cli(Message_TOMM *Mes, char *NomFichier) :

Cette fonction cherche le thread qui intéresse l'utilisateur pour ensuite l'écrire dans un fichier avec tous les postes qui sont liés au thread.

La fonction void Edit_Cli(Message_TOMM *Mes) :

Cette fonction permet de rajouter le message dans le thread à la suite de la liste chainé des postes.

Le fichier serveur1_main.c :

Lance juste les fonctions contenues dans les fichiers Login_serv.c et TOMM_serv.c et Net_Serv.c.

le fichier Net_Serv.c :

Contient tous ce qui est gestion du réseau, avec notamment la création de la socket, la gestion de la connexion et la gestion des workers pour le multi-utilisateur.

Le fichier Net_Cli1.c :

Le fichier Net_cli1.c contient toutes les fonctions nécessaire au client pour la communication avec le serveur

La fonction int Create_Sock();

Cette fonction permet de créer une socket à travers laquelle passe les fichiers qui viennent du serveur.

la fontion Connection_TOMM(int sock, int argc, char *argv[]);

Permet de se connecter avec le serveur, par défaut sur le localhost 20000 comme dans le cours.

la fonction Authentification_Locale(Message_TOMM *Mes, Utilisateur *Cur_User);

Cette fonction permet de générer les paramètres de l'utilisateur courant. Dans un premier temps, si l'utilisateur rentre correctement son login avec son pseudo et son mot de passe, on cherche a savoir si il est dans la base de donnée avec la fonction Get_User et affiche le login. Si l'utilisateur rentre seulement Login, on utilise la fonction Auto_Log. Enfin si l'utilisateur rentre create on v=créer un compte en utilisant la fonction Created_user.

void Write_Auto_Log(Utilisateur *Cur_User);

Cette fonction génère le fichier "log" qui permet une authentification automatique sans saisir son Identifiant ou mot de passe, pour cela on écrit seulement la structure de l'utilisateur courant dans un fichier log.

Utilisateur *Auto_Log();

La fonction utilise le fichier "log" présent dans le répertoire pour authentifier le client en lisant la structure utilisateur dans le fichier.

void RecevoirFich(int sock , char * Nomfichier);

Cette fonction receptionne le fichier qui est envoyé dans "Nomfichier" à partir de la socket. Utilise la structure sFich. On créer un fichier et si il est ouvert on alloue la mémoire avec un malloc(sizeof(sFich)). Ensuite on reçoit avec un recv() et l'on met les données dans un buffer que l'on écris ensuite dans le fichier avec fwrite(). Enfin on libère la mémoire alloué au buffer.

void envoyerFichier(int sock,char * Nomfichier);

Cette fonction envoi le fichier "Nomfichier" dans la socket en utilisant la structure sFich. On ouvre un fichier, si le fichier est ouvert on libère la mémoire de la taille de la structure sFich et on lit tous les caractères que l'on

met dans un buffer en attendant que le flag finFich soit à 1. Ensuite on envoi tous le buffer grâce à a fonction send(). Après on libère la mémoire prise par le buffer avec free() avant de fermer le fichier.

Le fichier TOMM_Cli.c :

Contient toutes les fonctions nécessaire à la gestion des threads, des postes et de l'affichage pour le client.

void read_Message_TOMM(Message_TOMM *Mes_Lu, char *Nomfichier);

Cette fonction permet de lire le message contenu dans un fichier qui a été envoyé par le serveur. On cherche donc le message que l'on veut dans le fichier avec fseek(),fread() .

void write_Message_TOMM(Message_TOMM *Mes_Ec, char *Nomfichier);

Cette fonction permet d'écrire la structure Message_TOMM dans le fichier.

Date *Obtenir_Date();

Cette fonction permet d'obtenir la date en utilisant localtime.

void Affiche_Date(Date *Cur_Date);

Fonction d'affiche de la date.

Utilisateur *New_Utilisateur(char *Nom, char *Pw);

Permet de génerer un nouvelle utilisateur en récupérant le nom et le mot de passe que l'on recopie dans la structure Utilisateur. Sachant que tous les utilisateurs créer ne sont pas administrateur.

void Affiche_Utilisateur(Utilisateur *Cur_Us);

Affiche l'utilisateur qui utilise la session courante. La fonction affiche le nom, le mot de passe et aussi tous les tags souscrits par l'utilisateur.

Thread *New_Thread(Date *End_Date, char *Titre);

Cette fonction permet de créer un thread qui est composé d'un Identifiant, une date de création qui correspond à l'appel de cette fonction.

void Affiche_Thread(Thread *Cur_Th);

Affiche simplement le thread contenu dans la structure Thread donnée en argument.

void Affiche_Th_Po(Thread *Cur_Th);

Affiche tous les posts contenus dans un Thread. Pour cela on utilise l'adresse du premier poste. Puis ensuite vu que les postes sont des listes chaînées on affiche les postes qui suivent.

void Affiche_Al_Th(Thread *Root_Th);

Affiche tous les Threads qui suivent celui passé en argument.Pour cela on récupère l'adresse du thread qui suit vu que c'est une liste chaînée.

void Ajout_Thread(Thread *New_Th, Thread *Root_Th);

Cette fonction ajoute un thread à la suite chaînée si le premier a l'adresse du suivant qui est nul, on lui donne l'adresse du nouveau thread. Sinon on parcours la liste à la recherche du dernier thread et on lui donne l'adresse du nouveau thread.

Thread *Rech_Thread(Thread *Root, char *txt);

Cette fonction permet de parcourir la liste chaînée à la recherche d'un Thread à partir de son nom.

Thread *Match_Th_Ta(int i, Thread *Root_Th, Tag *Root_Ta);

Renvoi le thread correspondant au Tag passé en argument.

void Affiche_Post(Post *Cur_Po);

Cette fonction affiche le poste donné en argument.

void Ajout_Post(Post *New_Post, Thread *Cur_Thread);

Cette fonction ajoute un post dans le thread correspondant.

void Ajout_Post_BDD(Post *New_Post, Thread *Cur_Thread);

Ajoute un post dans le thread Correspondant sans mettre à jour le nombre de post qui est déjà remplit.

Tag *Generate_Tags();

Genere les Tags :"ROOT","Voiture", "Courses", "Sport", "Ecole", "Assos" .

void Souscrire_Th(Utilisateur *Cur_User, Tag *Root_Ta, char Tab_Sub[NOMBRE_TAGS][TAILLE_NOM]);

Ajoute des tags aux Utilisateurs en inscrivant l'identifiant du Thread dans leur description

void Help_Menu();

Affiche l'explication et la syntaxe de toutes les commandes.

Message_TOMM *Interpret_Cli();

Cette fonction est l'interface de commande avec l'utilisateur, on récupère la commande avec fgets les trois arguments que l'on peut avoir dans la commande. Selon la commande avec un strcmp() et selon la commande on donne un numéro pour savoir à qu'elle commande correspond le premier argument. Et on sauvegarde les autres arguments dans la structure Message_TOMM.

Utilisateur *Get_User(Message_TOMM *Mes);

Renvoi l'utilisateur que se trouve dans le Message.

void Write_Success(Message_TOMM *Mes);

Génère un message pour le succès de la requête.

void Write_Fail(Message_TOMM *Mes);

Génère un message pour le non succès de la requête.

void Lire_See(FILE *new_File, Message_TOMM *Mes_Lu);

La fonction lit le fichier qui a été envoyé par le serveur pour ensuite afficher tous les postes du thread donnée.

void Traitement_Answer(char *Nomfichier);

Traite la reception des réponses, supprime le fichier de réponse.

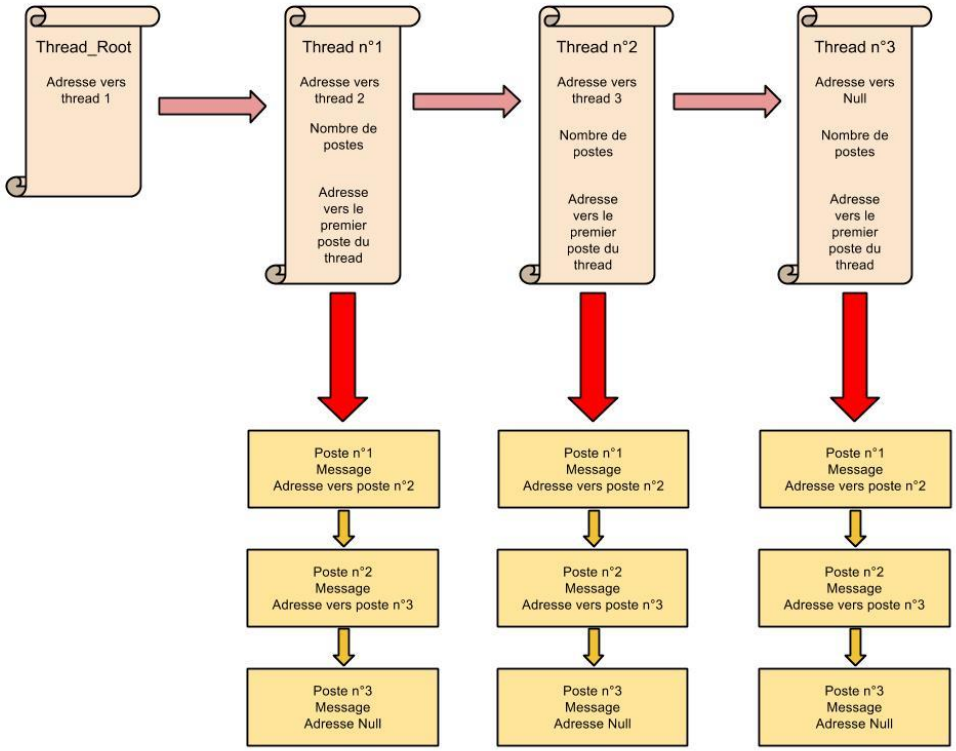
void rm_File(char *NonFichier);

Supprime le fichier passé en argument.

Le fichier client1_main.c :

Lance juste les fonctions contenu dans les fichier TOMM_cli.c et Net_cli.c.

4. La base de donnée



La base de données contient les threads et les postes des différents utilisateurs. Dans un premier temps il y a une structure Thread Root qui ne contient que l'adresse vers les autres structures Threads, Thread1,Thread2,... Les structures threads pointent vers le premier Poste de la liste chaînée. L'utilisateur va donc lorsqu'il va créer son poste ou son thread rajouter un élèment à la liste chainée. La base de donnée est donc dans la ram lors de l'écriture du poste ce qui permet de gérer la concurrence.

Cependant lorsque le client veut consulter un poste ou un thread, on va enregistrer les postes du thread dans un fichier que l'on envoie directement au client.

Lorsque le programme est finis la base de donnée est enregistré dans des fichiers, il y a un fichier Thread_Root qui contient seulement la strcture Thread_Root, il y a ensuites les fichiers Thread1, Thread2 qui contiennent la structure correspondantes mais aussi tous les postes correspondant au thread.

5. Gestion de la concurrence :

Lorsqu'un utilisateur souhaite rajouter un poste dans un thread, il va modifier la liste chainée contenant les postes. Si deux utilisateurs souhaitent créer un poste dans le même thread il y a un problème de concurrence, pour éviter cela on utilise un mutex et le deuxième utilisateur devra attendre avant de pouvoir rajouter son poste.

Ensuite lors du la l'authentification, de la création de compte ou de la suppression, l'accés au fichier comptes.txt est vérrouillé aux autres utilisateurs pour éviter tous les problèmes de concurrences.

6. Conclusion

Pour terminer, nous devons mentionner que comme expliqué dans le code, notre serveur ne répond pas à la caractéristique de multithread, bien qu'après avoir essayé durant de très longues heures et jours il demeure un bug dont nous n'arrivons pas à venir à bout. La version qui a été envoyée ne contient pas la mouture la plus avancé en termes de debug mais plutôt une version à même de fonctionner correctement.

Nous avons donc implémenté notre fonction dans un serveur moins sophistiqué mais qui permet de valider une partie du fonctionnement de notre projet, l'envoi réception de fichiers.