

**Networks & Architectures**

**Projet Serveur Chat**

**Membres du groupe :**

GUO Huaiyuan

GRARD Vincent

GRIL Florian

DEFFORGE Julien

ESILV 2020-2021

IOS 2 – A4

Sommaire

Table des matières

[Introduction 3](#_Toc59136350)

[Le client 3](#_Toc59136351)

[Le serveur 6](#_Toc59136352)

[La base de données 12](#_Toc59136353)

# Introduction

Notre projet concerne le projet d'un serveur implémentant un chat avec de nombreuses fonctionnalités. Ce chat doit permettre l’interaction, l’échange de données entre plusieurs personnes grâce à l’intermédiaire du serveur. Ces personnes doivent en effet s’authentifier pour accéder au serveur et au chat via notre système de connexion basé sur SQLite. En cas de problème (enquête judiciaire), notre serveur doit aussi pouvoir nous offrir l’accès aux logs et au chat pour le résoudre.

Les fonctionnalités introduites sont donc implémentées côté client ou serveur. En lançant notre serveur, il est alors possible de se connecter via la base de données déjà créée ou alors de s’inscrire directement, pour cela un pseudo et un mot de passe sont nécessaires. Ensuite, le client a accès à de nombreuses fonctionnalités comme l’envoi de message dans le chat global, privé ou en message privé avec d’autres utilisateurs. Il peut aussi transférer des fichiers sur le serveur, en télécharger depuis le serveur et faire des échanges avec les autres utilisateurs. L’utilisateur peut avoir accès à la liste des personnes connectés et des fichiers du serveur.

Il y a aussi des fonctionnalités coté serveur permettant d’alerter les utilisateurs (via un message envoyé à tous les utilisateurs), de bannir ou plutôt kick des personnes du serveur et enfin de stopper le serveur (et donc kick tous les clients).

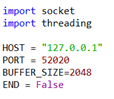
Dans le rapport suivant nous allons alors expliquer les différentes parties de notre code et comment nous avons implémenté les fonctionnalités présentées plus tôt. Tout d’abord avec la partie client comprenant le threading et l’envoi de certains messages. Ensuite, avec la partie serveur qui est utile pour toutes les fonctionnalités du client et du serveur. Enfin, avec la partie SQLite permettant le lien avec la base de données permettant de s’inscrire et de s’authentifier.

Nous livrons alors trois fichiers pythons comprenant chacun respectivement le code des trois parties (client, serveur, base de données), le fichier de base de données comprenant la base de données (le fichier .db) et enfin le fichier ReadMe.txt qui contient les informations pour installer et faire fonctionner notre projet server de chat.

# Le client

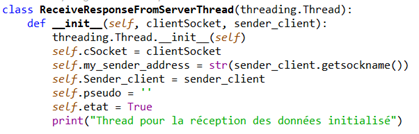
On importe d’abord socket et threading qui permettent de créer et gérer entre autres les sockets.

D’abord, on définit l’host ici « 127.0.0.1 » pour une utilisation locale et le numéro de port libre 52020, le buffer size qui permet de définir la taille utilisée en fonction du flux qui arrive, l’END = False est lui utile lors que l’utilisation ou fermeture du Client.



*Figure 1 : variables globales*

On crée ensuite la classe ReceiveResponseFromServerThread() (Figure 2) prenant en entrée le Thread, cette classe gère la réception des données du serveur. Pour initialiser cette classe on utilise les différents socket client, le pseudo utilisé, l’état du client (initialisé à True). Lorsque le Thread est lancé, la méthode run est exécutée.





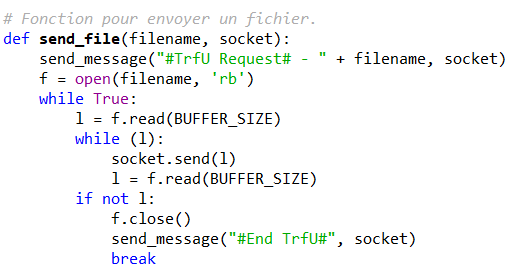
*Figure 2 : Classe ReceiveResponseFromServerThread côté Client*

Cette méthode attend d’abord un message du serveur et tant qu’il n’y en a pas, on ne passe pas à la méthode suivante et donc la première instruction attendant le message est bloquante. Ce message est décodé et s’il reçoit #Response TrfD#, il écrira le document demandé à être téléchargé coté Client. S’il reçoit #Kill, le thread envoie un message au client pour qu’il se ferme automatiquement et fermera le Thread par la suite en passant son état à False ensuite. Si le try ne fonctionne pas le thread se fermera.

Nous avons ensuite créé plusieurs fonctions utiles pour connecter les sockets, envoyer des messages ou des fichiers (en public ou privé) (Figure 3).

La première fonction permet de se connecter au serveur via socket.socket et .connect() qui connecte la socket au bon host et port défini. On peut alors aussi envoyer un message ou un fichier (en privé ou en public), ces méthodes utilisent le .send() pour permettre l’envoi. De plus, pour envoyer un message, il a besoin d’être encodé (via UTF-8) et pour utiliser l’envoi de fichier nous utilisons des requêtes de messages pour permettre l’échange entre serveur et client.





*Figure 3 : fonctions de connexion et d’envoi de messages et de fichiers*

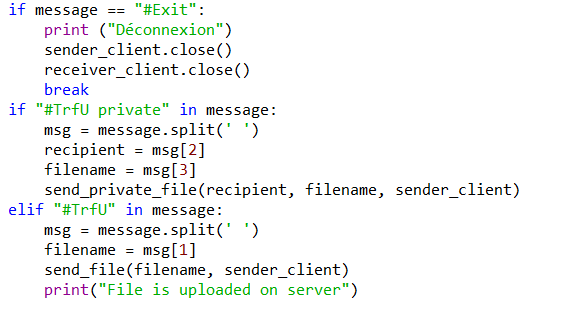
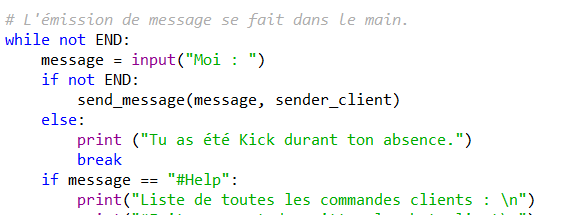
Dans notre main (Figure4), on crée tout d’abord les sockets de création pour l’envoi et la réception de données, on crée ensuite le thread avec les deux sockets créés et on le start.



*Figure 4 : main côté Client (part 1)*

Tant qu’END n’est pas vrai, (Figure 5) alors on peut envoyer des messages depuis le thread et recevoir des réponses, avec #Help on obtient les infos sur les fonctions utilisables (implémentés dans la partie serveur) et il y a aussi les instructions permettant de transférer un fichier vers le serveur ou en privé en utilisant les fonctions send\_file ou send\_private\_file (fait en splittant les infos sur message).

Pour un message d’exit, celui-ci ferme les sockets de création pour l’envoi et la réception de données ce qui permet de quitter le serveur (de déconnecter l’utilisateur).



*Figure 5 : main côté Client (part 2)*

# Le serveur

La partie serveur de notre projet est la pièce centrale. En effet notre structure est de type « centralisée » et fonctionne sur le même principe que la topologie en étoile. Chaque client va se connecter au serveur et lui envoyer des messages. Le serveur s’occupe ensuite de redistribuer les messages aux autres clients. Le programme python qui « reçoit » et gèrent les connexions entrantes venant des clients s’appelle « server\_multi\_thread » (Figure 6). Cette gestion des connexions entrantes se fait dans le « main » à la fin du fichier python dans le « While SERVER\_SWITCH » où SERVER\_SWITCH est défini à True (cf. Figure 7) au début du programme et tourne à False que lorsque nous écrierons : #Exit (cf. commandes Admin).

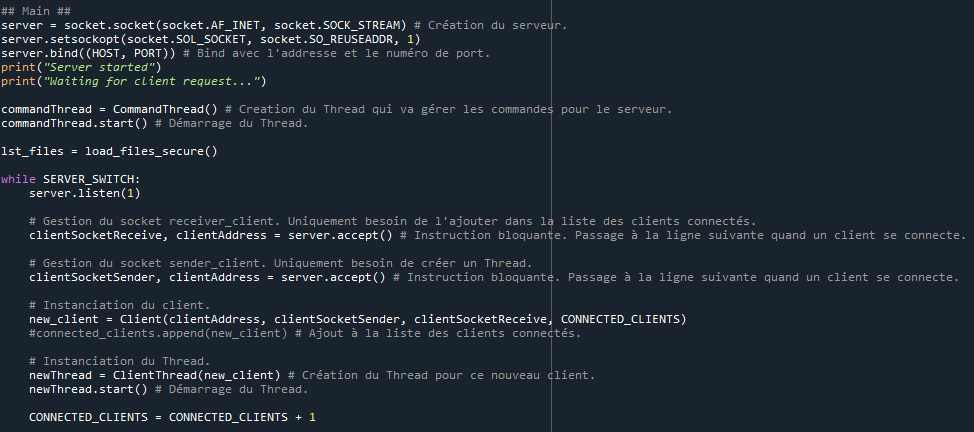
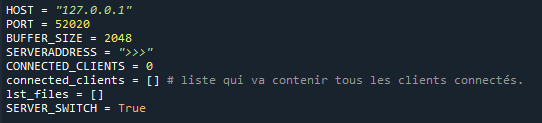


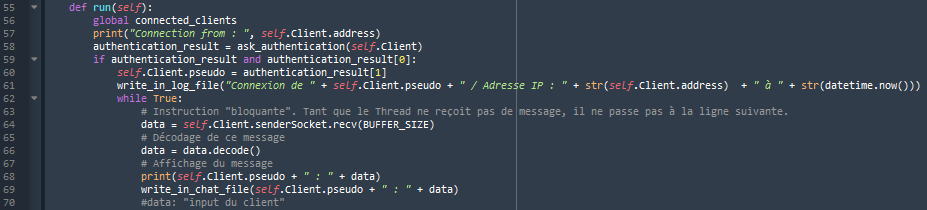
Figure 6 : Le main du serer\_multi\_thread.py

Figure 7 : Les variables globales



Au préalable le serveur a été créé et lié à l’adresse localhost et au numéro de PORT 52020 (cf. Figure 2). Le serveur est ensuite mis à l’écoute et attend la connexion d’un client (cf. Figure 8 ligne 64). Lorsqu’un client se connecte, le processus s’enclenche.

Figure 8 : Début ClientThread



Dans ce script il y a deux classes héritant de « threading.Thread » : ClientThread et CommandThread. Le ClientThread est la classe permettant de gérer tous les clients en parallèle. Lorsqu’un client va se connecter au serveur, un nouveau « ClientThread » sera créé et lui sera associé. La classe « Client » (Figure 9) permet de faire cette association entre le client et le thread.

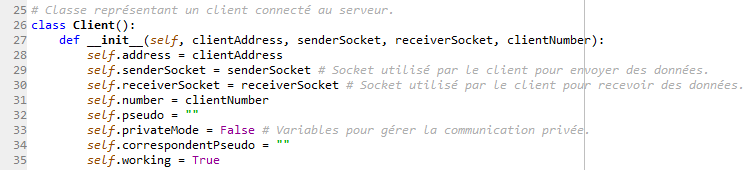


Figure 9 : class Client côté serveur

Afin de pouvoir envoyer et recevoir des messages de façon simultanée le client ouvre deux canaux de communications (voir la partie sur le programme « Client »), donc deux sockets. On retrouve ces deux sockets dans la classe présentée ci-dessus :

* senderSocket correspond au socket utilisé par le client pour envoyer des données. Donc lorsque le serveur recevra un message du client il devra « attendre » (recv) sur la « senderSocket ».
* receiverSocket correspond au socket utilisé par le client pour recevoir des données. Donc lorsque le serveur voudra envoyer un message au client il devra se servir de ce socket.

Revenons au ClientThread. La méthode run (héritée de threading.Thread) est la méthode exécutée lorsque le Thread est démarré (newThread.start()). On utilise un « while True » pour laisser le Thread s’exécuter. La toute première action entre le client et le serveur est la gestion de l’authentification. Si le client à un compte dans la BDD il peut accéder au serveur, sinon il peut en créer un. Si l’utilisateur saisi un mauvais mot de passe il ne pourra pas accéder au serveur et devra se déconnecter puis se reconnecter.

Une fois le client authentifié, lorsque celui-ci enverra un message au serveur, le message sera analysé. La ligne surlignée (dans la capture suivante) correspond à une instruction « bloquante », c’est ici que le thread est en attente de réception d’un message venant du client.

Le message est ensuite décodé (converti en string) et analysé. S’il s’agit d’une commande celle-ci sera analysée et l’opération correspondante sera exécutée. S’il s’agit d’un message celui-ci sera redistribué à tout le monde ou seulement au « correspondent » du client si ce dernier est en mode « private ».

Dans ce programme il y a aussi de nombreuses fonctions permettant par exemple de lister les fichiers (cf. figure11), d’écrire dans le fichier de log (cf. figure 10), d’écrire dans le fichier de sauvegarde des chats, d’envoyer des messages, des fichiers, de recevoir des fichiers, tout cela de façon publique ou privée.

Figure 11 : lister les fichiers

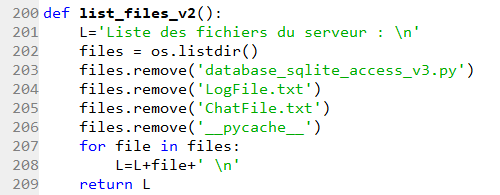
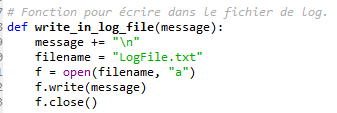
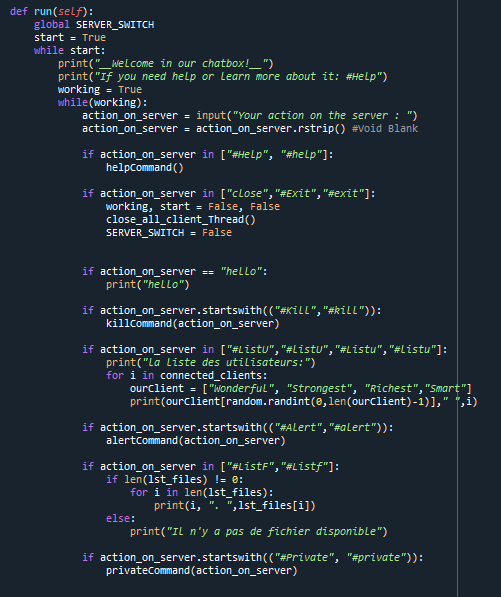


Figure 10 : Fonction d'écriture dans le fichier de log



Il y a aussi des fonctions permettant de gérer les commandes du serveur (Kill, lister les fichiers, lister les utilisateurs connectés, afficher l’aide et alerter les clients). Ces commandes sont d’ailleurs traitées via le CommandThread(cf. figure 12). Grâce à ce thread, le serveur peut envoyer des messages, recevoir des messages, etc. tout en laissant à un administrateur la possibilité d’agir sur le serveur :

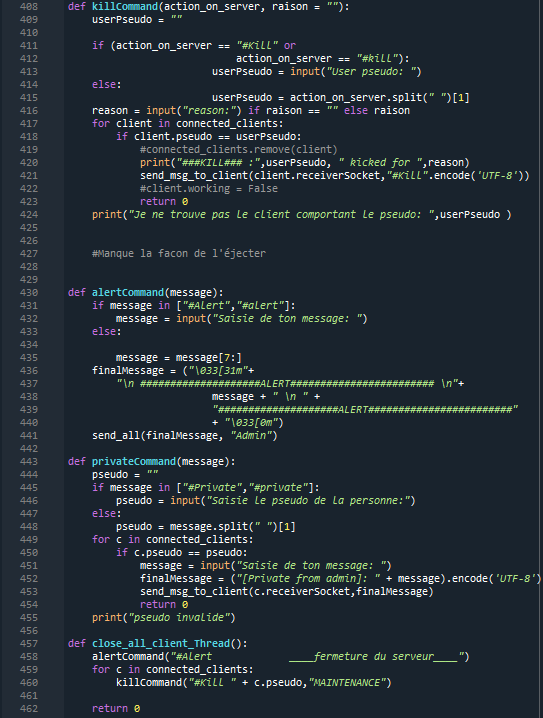
Figure 12 : Les commandes du Thread serveur



Avec les commandes en question :

Figure 12 : Les commandes du Thread serveur

Figure 13 : les commandes liant le Thread Serveur



La fonction pour gérer la réception d’un fichier est la suivante :

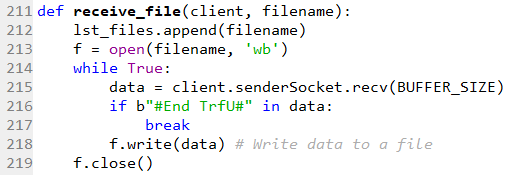


Figure 14 : fonction de réception de fichier

On ajoute dans la liste des fichiers le nom du fichier reçu, on l’ouvre, et tant qu’il y a réception des données on écrit dans le fichier. La fonction se stoppe lorsque le client envoie un « tag » particulier (« #End TrfU# »).

Pour envoyer un fichier nous avons développé cette fonction :

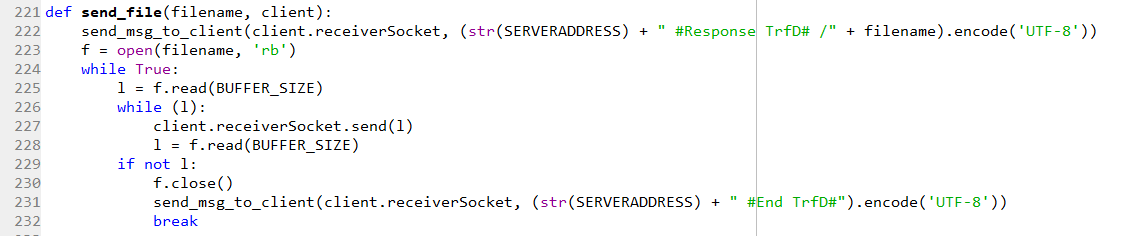


Figure 15 : fonction d'envoi de fichier

Pour commencer, le serveur indique au client qu’il va envoyer un fichier avec le tag « #Response TfD# » avec le nom du fichier séparé par un slash. Ainsi le client pourra se préparer et ouvrir un fichier avec ce nom. Ensuite le serveur lit le fichier et l’envoie au client. Lorsqu’il n’y a plus de données à lire, le fichier se ferme et le serveur indique au client que l’envoi de fichier est terminé.

Afin de lister les fichiers déjà présents dans le dossier du serveur lors d’un démarrage du programme, nous chargeons les fichiers préexistants. C’est la fonction « load\_files » qui s’en charge.

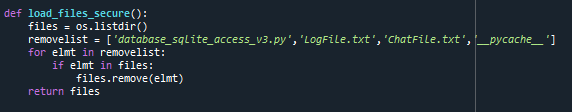


Figure 16 : fonction de chargement des fichiers

# La base de données

La base de données SQLite permet au client de s’inscrire et de s’authentifier.

Le code python sqlite nommé « database\_sqlite\_access .py » a pour but d’exploiter la database « server\_chat\_database.db » pour vérifier l’existence ou le mot de passe d’un utilisateur.

Pour s’y connecter, la fonction initialisation() utilise la commande : conn = sqlite3.connect(db\_file) avec db\_file le chemin d’accès à la database.

Ensuite, lors de la connexion d’un client, il lui est demandé s’il possède ou non un compte. S’il n’en a pas, il peut répondre « non » et s’inscrire en écrivant « username/password ». La fonction add\_user intervient et envoie une requête sql : conn.execute("INSERT INTO users (username, password) VALUES(?, ?)", (username, password,)). Ainsi, le nouveau client est inscrit.

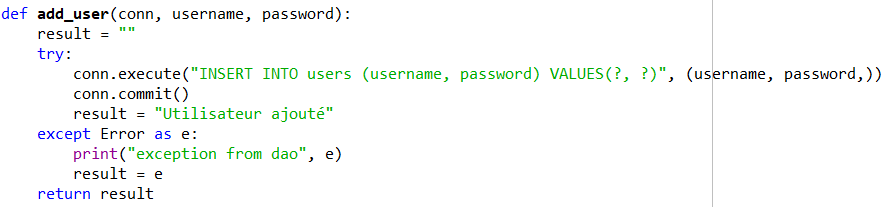


Figure 17 : les commandes liant le Thread Serveur

Lors d’une nouvelle connexion, il pourra s’identifier. Ceci à l’aide de la fonction authentification (conn, username, password). Celle-ci se connecte à la database et va sélectionner l’utilisateur correspondant. A l’aide des fonctions cursor() et fetchall(), on accède à une liste de tuples qui ont donc le même utilisateur (unique ici car la database renvoie une erreur lors de l’inscription si l’utilisateur existe déjà, une contrainte d’unicité sur la colonne « username » a été ajoutée lors de la création de la table). Dans ce tuple, la 3eme donnée étant celle du mot de passe correspondant, on peut donc valider ou non l’authentification.

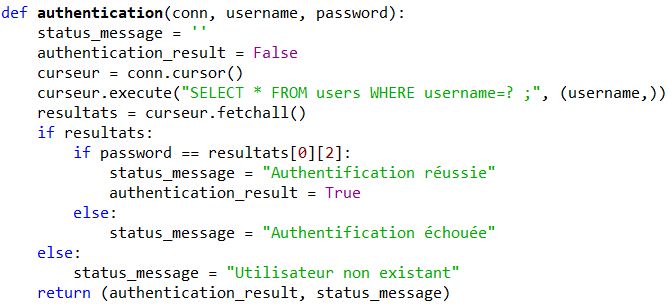


Figure 18 : les commandes liant le Thread Serveur

Par le même procédé, on peut également vérifier lors de la connexion si l’identifiant existe avant d’essayer de vérifier son mot de passe. La fonction check\_if\_user\_exist(conn, username) regarde simplement si le .fetchall() a bien un résultat.

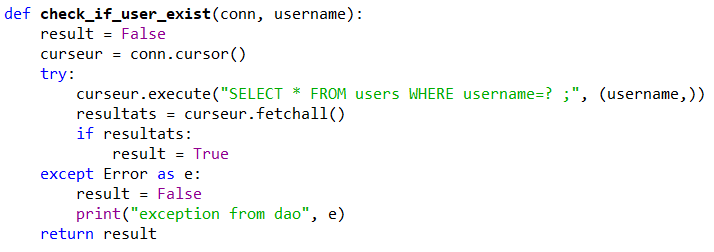


Figure 19 : les commandes liant le Thread Serveur