Server Project (Chat)

0) Structure du projet

Database

Nous avons une database SQLite simple avec une seule table Client avec comme attributs le username (clé primaire) et le password. Le username est l'attribut qui va nous permettre de différencier chaque client entre eux lors du login sur le serveur.

De plus, SQLite nous a permis de travailler beaucoup plus facilement en groupe avec git car lorsque l'on se transfert le programme, la base de données SQLite se transfert aussi. Cependant, les données que chaque membre crée sont propres à eux et ne se transmettent pas. Cela signifie que chaque membre a dû se constituer sa propre base de clients.

ClientClass.py et RoomClass.py

Afin de faciliter la compréhension du code et l'usage de certaines fonctionnalités, nous avons créé deux classes :

- Une classe Client qui prend comme attributs le username, l'IP, le port, la socket et la room. Cela permet d'avoir toutes ses informations directement dans un objet client.
- Une classe Room qui prend comme attribut un nom, un client admin créateur, une liste de clients la composant. Cette classe est indispensable à notre feature de création de chat room privée.

Server_functions.py, client_functions.py et room_functions.py

Le serveur comme le client ont accès à des fonctions avec la commande *#Fonction*. Afin de mieux séparer et répartir le code, nous avons créé trois fichiers :

- « Server_functions.py » qui comprend toutes les fonctions relatives au serveur c'est à dire toutes les fonctions accessibles depuis le terminal ayant lancé « server.py »
- « Client_functions.py » qui comprend toutes les fonctions relatives au client c'est à dire toutes les fonctions accessibles depuis les terminaux ayant lancé « client.py »
- « Room_functions.py » qui comprend toutes les fonctions relatives aux rooms et disponibles aux clients (ayant lancé « client.py »)

Cyphering.py

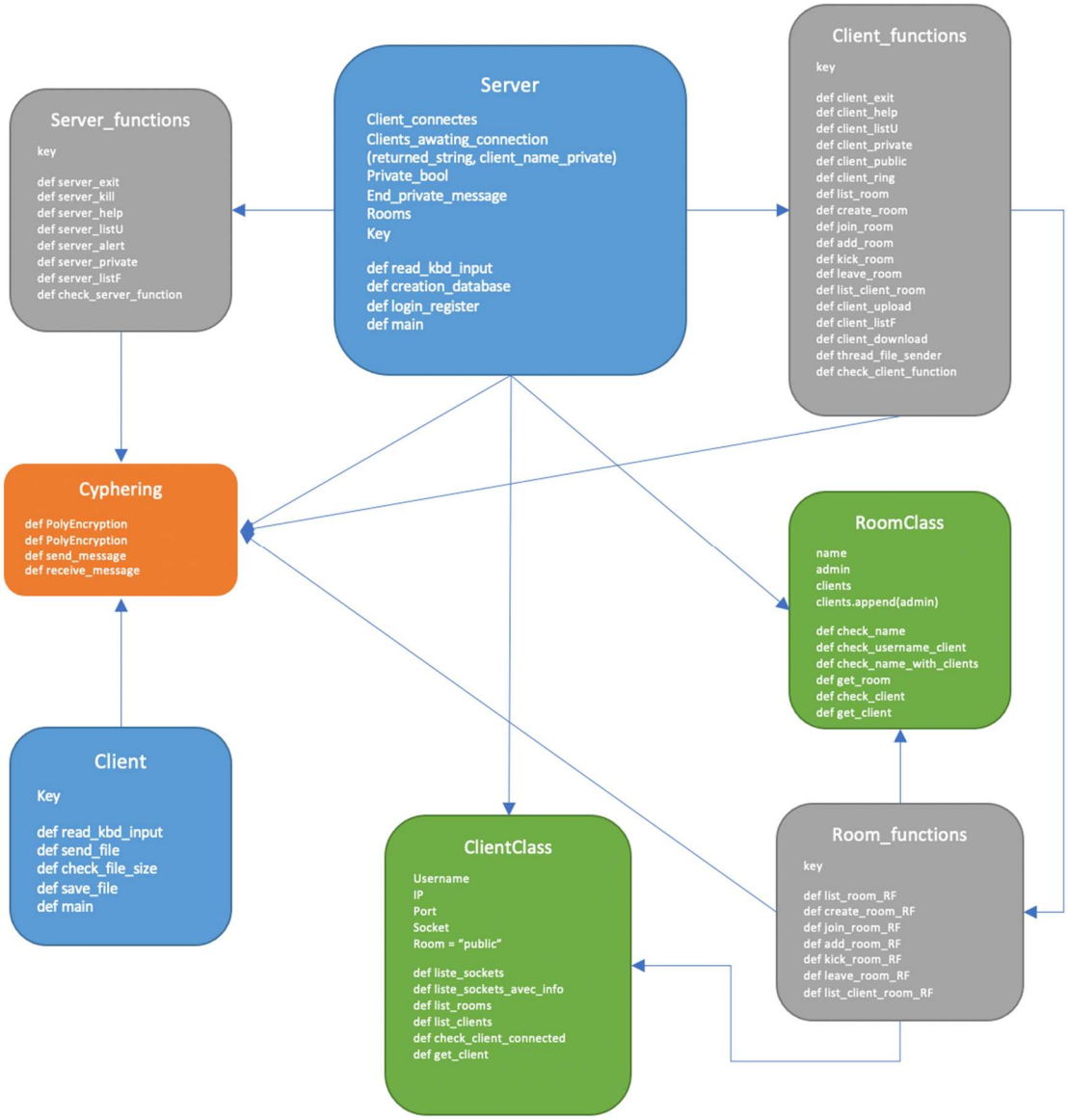
Ce fichier permet de crypter et décrypter les messages envoyés par les clients. Il comprend deux fonctions relatives au cryptage/décryptage poly alphabétique et deux fonctions relatives à l'envoi et la réception de messages.

Server.py et client.py

Ces deux fonctions forment le corps du programme. Ce sont ces fonctions que l'on lance lorsque l'on veut faire tourner notre programme. L'ordre a son importance : il faut d'abord up le serveur (en lançant server.py) avant de pouvoir y connecter n'importe quel client (en lançant client.py).

Elles vont se charger de gérer toutes les commandes/messages tapés par les clients et serveur.

Voici le diagramme qui montre le fonctionnement de notre projet. Les flèches indiquent quel fichier fait référence à quel fichier.



Nota Bene

Dans la suite du rapport, nous avons répondu à toutes les recommandations et contraintes du cahier des charges. La seule partie que nous nous sommes permis d'éluder est celle concernant la description du fonctionnement de chaque fonction du programme. En effet avec quelques 2000 lignes de code et pas moins de 50 fonctions, notre rapport atteindrait très vite une taille trop importante. Nous nous sommes donc contentés d'effectuer un topo rapide du fonctionnement des fonctions clients et serveurs. Pour plus de détails, nous vous invitons à vous référer au programme python qui commente de manière très précise chaque fonction.

1) Connexion des clients au serveur en UDP ou TCP

On utilise le protocole de transmission TCP car on veut que les paquets soient surs d'arriver à leur destination. Afin de procéder à nos échanges, nous utilisons la librairie socket.

Tout d'abord on prépare le serveur en exécutant server.py. La connexion principale se crée en localhost sur le port 12800 (choisi arbitrairement) avec les paramètres d'échanges TCP classiques. On procède ensuite au bind et au listen.

Du côté client, c'est client.py qui est exécuté pour également créer une socket TCP afin de se connecter au serveur. Une fois la communication acceptée côté serveur, les informations clients sont récupérées et un thread d'authentification est lancé.

En lançant chaque authentification de client dans un nouveau thread, on évite ainsi que les authentifications ne soient autorisées qu'une par une ralentissant ainsi le serveur si de nombreux utilisateurs veulent se connecter en même temps.

A l'issue d'une authentification réussie (connexion ou création de compte), une nouvelle entité Client est créée avec les informations de l'utilisateur venant de se connecter au chat. Cette entité est ensuite ajoutée à la liste des clients connectés de la session publique actuelle.

2) Capacité d'envoyer et recevoir des messages (chat) (280 caractères par message)

Une fois que l'utilisateur est connecté, il peut recevoir et envoyer des messages dans le chat public jusqu'à ce qu'il décide de quitter le chat serveur en faisant #Exit. Il existe 3 différents états de chat pour un client :

- Public: l'état par défaut dans lequel, lorsqu'un message est envoyé par un client du public, tous les autres clients connectés, qu'ils soient sur le public ou non, reçoivent le message
- Private: l'état dans lequel deux clients s'envoient des messages de manière privée, c'est à dire que seuls eux deux (et le serveur) peuvent voir les messages échangés. Pour rentrer dans cet état, l'utilisateur doit taper la commande #Private + le nom de l'utilisateur avec lequel il veut communiquer. Après cela, l'utilisateur mentionné est automatiquement transféré sur un canal de discussion privé (on ne lui demande pas son autorisation).
- Chat Room (feature) : l'état dans lequel, lorsqu'un client du groupe envoie un message sur le groupe, seuls les autres membres du groupe reçoivent le message (qu'ils soient dans la chat room ou non).

Pour retourner dans le chat public, il faut taper la commande #Public.

Notre programme n'inclut pas de GUI ce qui veut dire que nous avons un affichage console. Cette contrainte fait que lorsqu'un client A écrit un message, et que pendant ce temps, il reçoit un message d'un autre client B, alors ce message va s'insérer juste après le message tapé par client A. Cela ne fait pas planter le programme, et le client A peut continuer de taper son message. Lorsqu'il l'envoie, le message sera tel que A l'a écrit et non comme il est affiché dans la console c'est à dire : début_message_A message_B fin_message_A. Il s'agit donc uniquement d'une coquille de présentation due à l'utilisation de terminaux.

3) Transférer des fichiers (jpeg ou autres) (client à client, client à serveur)

Afin de permettre l'échange de fichiers entre clients et entre un client et le serveur, nous nous appuyons sur 3 méthodes dont le fonctionnement sera détaillé plus bas. Ces méthodes sont #TrfU, #ListF et #TrfD.

Si un client veut envoyer en fichier au serveur, il lui suffit d'utiliser la commande #TrfU. S'il veut en télécharger un, il devra d'abord utiliser la commande #ListF afin d'avoir accès à la liste des fichiers hébergés par le serveur, puis la commande #TrfD suivi du nom d'un de ces fichiers.

Nous avons fait le choix de ne pas permettre l'échange direct de fichiers entre les clients. En effet, il aurait alors fallu composer :

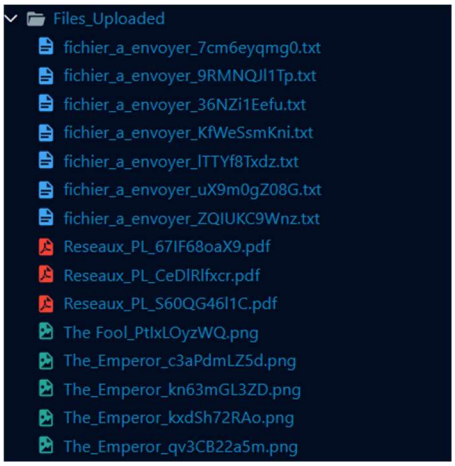
- Soit avec un système où des fichiers peuvent être envoyés à un autre clients et se retrouver sur sa machine sans son accord
- Soit créer un système d'autorisation de téléchargement afin d'autoriser un client extérieur à nous envoyer un fichier

Afin de simplifier l'utilisation de notre application, nous avons plutôt choisi de faire transiter tous les fichiers sur le serveur. Si un client souhaite partager un fichier avec un autre client, il lui suffit de l'uploader sous un certain nom puis de communiquer ce nom avec son ami afin que celui-ci le télécharge. Cela nous permet également de surveiller les contenus partagés afin d'éviter que l'application serve au transfert de fichiers indésirables. Dans le cadre d'un projet étudiant nous nous contentons d'afficher dans le log server qui envoie/télécharge quel fichier, mais dans le cadre d'une réelle mise en circulation d'une application de chat nous pourrions également sauvegarder dans une autre table de la database quels fichiers ont été uploadés/téléchargés et surtout par qui afin de garder un œil sur tous les échanges.

Afin de permettre une vérification facile du bon fonctionnement de notre application, un code de dix caractères aléatoires est ajouté à la fin de chaque fichier uploadé. Cela permet de se rendre compte du bon fonctionnement de la fonction d'Upload en cas d'utilisation répétée sans avoir à la tester avec une dizaine de fichiers de tests. Par exemple dans le cas de 5 uploads du fichier 'fichier_a_envoyer.txt', on obtient le résultat suivant :

Bien évidemment cette modification du nom des fichiers serait retirée dans le cadre d'une mise en circulation réelle de l'application. Il suffirait alors de sauvegarder les anciennes versions des fichiers en archive ailleurs sur la machine et conserver uniquement les versions les plus récentes dans ce dossier.

Le transfert de fichiers fonctionne aussi bien pour des images, du texte ou des fichiers PDF



4) Multi-thread

En l'absence de réponse par rapport à nos questions sur le multi-thread et étant donné que rien n'empêche notre serveur d'accueillir une cinquantaine de clients à la fois tel qu'il est actuellement conçu, nous avons interprété cette partie comme « Implémentez du multi-thread dans votre application afin de réduire la charge sur le thread principal si de nombreux utilisateurs sont connectés en même temps »

Dans cette optique, nous avons créé plusieurs thread côtés serveur et client afin d'éviter une gestion trop mécanique de notre application.

Tout d'abord nous avons fait le choix de créer un nouveau thread pour chaque émission de fichier. Il s'agit d'une tâche qui peut prendre un temps indéfini suivant la taille du fichier à envoyer, et dans l'hypothèse que l'envoi soit relativement long nous ne voulons pas que notre thread principal de serveur soit bloqué pendant ce temps.

Nous avons néanmoins fait le choix de ne pas utiliser de thread pour la sauvegarde de fichier. En effet, utiliser un thread pour la sauvegarde de fichiers nous aurait peut-être permis de recevoir des fichiers en même temps de recevoir des chats mais cela nous aurait contraint d'ajouter un système d'identification des messages afin de déterminer si un message contient la data d'un chat ou d'un fichier. En effet dans le cas contraire l'application aurait mélangé les contenus et peut-être affiché dans le chat public la data de fichiers à enregistrer.

Néanmoins afin de ne pas ignorer cette problématique des réceptions de fichiers trop gourmandes en temps, nous avons défini une taille de fichier maximale au-dessus de laquelle les fichiers ne seront pas uploadés. Cette taille maximum est définie dans le fichier client_functions.py et est bien à modifier en fonction des caractéristiques de notre serveur (par défaut nous l'avons mise à 999999999). Si l'utilisateur essaie d'envoyer un fichier plus gros, il reçoit un message d'erreur lui indiquant la raison de l'erreur ainsi que la taille maximum autorisée

5) Ligne de commande sur le serveur

Afin de pouvoir exécuter un code différent en fonction de la commande entrée, nous avions besoin d'un moyen de vérifier la nature de la commande (commande exit, commande help, etc). Toutefois nous ne voulions pas utiliser une succession de blocs if sur le premier mot de la commande afin d'exécuter le code adapté. Nous avons alors pensé à utiliser un switch, mais avons appris que ceux-ci n'existaient pas en Python. Nous avons alors trouvé une alternative que nous allons vous présenter ici. Pour ce faire nous allons prendre l'exemple de client_functions.py, le fonctionnement est similaire pour server_functions.py.

Nous associons d'abord chaque # de commande à une variable afin de pouvoir facilement ajouter des commandes :

```
#! Commandes clients
EXIT_CLIENT = "#Exit" #Commande utilisée par les clients pour quitter son terminal
HELP_CLIENT = "#Help" #Commande utilisée par les clients pour obtenir de l'aide
LISTU_CLIENT = "#ListU" #Commande utilisée par les clients pour obtenir la liste des autres
PRIVATE_CLIENT = "#Private" #Commande utilisée par les clients pour discuter en privé les u
PUBLIC_CLIENT = "#Public" #Commande utilisée par les clients pour revenir au chat public ap
UPLOAD_CLIENT = "#TrfU" #Commande utilisée par les clients pour télécharger des fichiers
RING_USER = "#Ring" #Commande utilisée par les clients pour ring un utilisateur s'il est co
LISTF_CLIENT = "#ListF" #Commande utilisée par les clients pour voir tous les fichiers
DOWNLOAD_CLIENT = "#TrfD" #Commande utilisée par les clients pour télécharger des fichiers

CREATE_CHATROOM_CLIENT= "#CreateRoom" #Commande utilisée par les clients pour créer des dis
JOIN_CHATROOM_CLIENT="#JoinRoom" #Commande utilisée par les clients pour rejoindre une room
LIST_CHATROOM_CLIENT="#ListRoom" #Commande utilisée par les clients pour lister toutes les
ADD_CLIENT_CHATROOM_CLIENT="#AddRoom" #Commande utilisée par les clients pour ajouter un ut
KICK_CLIENT_CHATROOM_CLIENT="#KickRoom" #Commande utilisée par les clients pour kick un uti
LEAVE_CLIENT_CHATROOM_CLIENT="#LeaveRoom" #Commande utilisée par les clients pour quitter u
LIST_CLIENT_CHATROOM_CLIENT="#ListClientRoom" #Commande utilisée par les clients pour liste
```

On associe ensuite chaque variable à une fonction :

```
#! Dictionnaire utilisé dans la fonction principale de ce fichier à savoir Check_client_functions (ci-dessous)
#! Il est utilisé comme un switch case
#! A gauche des ":" c'est la key (que l'on a défini tout en haut du fichier)
#! A droite des ":" c'est la value qui est ici la fonction qui sera executé en fonction de la key que l'on saisi
options = {
        EXIT_CLIENT : Client_Exit,
        HELP_CLIENT : Client_Help,
        LISTU_CLIENT : Client_ListU,
        PRIVATE_CLIENT : Client_Private,
        PUBLIC_CLIENT : Client_Public,
        CREATE_CHATROOM_CLIENT : Create_Room,
        JOIN_CHATROOM_CLIENT: Join_Room,
        LIST_CHATROOM_CLIENT: List_Room,
        ADD_CLIENT_CHATROOM_CLIENT: Add_Room,
        KICK_CLIENT_CHATROOM_CLIENT: Kick_Room,
        LEAVE_CLIENT_CHATROOM_CLIENT: Leave_Room,
        LIST_CLIENT_CHATROOM_CLIENT: List_Client_Room,
        UPLOAD_CLIENT : Client_Upload,
        RING_USER : Client_Ring,
        LISTF_CLIENT : Client_ListF,
        DOWNLOAD_CLIENT : Client_Download
    }       You, 13 days ago • Création de l'architecture de commande pour client_functions.py
```

Depuis nos codes principaux, il nous suffit alors de faire appel à la fonction suivante afin d'être redirigé vers la méthode de commande correspondante :

```python
def Check_client_functions(msg_recu, client, clients_connectes,  Rooms):
    commande = msg_recu.split(' ')[0]

    try:
        return options[commande](msg_recu,client, clients_connectes,  Rooms)
    except :
        Send_Message("Command not found, try using #Help",key,client.socket)
        You, 13 days ago • Création de l'architecture de commande pour client_fu
```

Commande va contenir le #... et le return va récupérer le nom de fonction associé à cette clé pour ensuite y faire appel avec les paramètres indiqués entre parenthèses.

Cela nous permet d'automatiquement effectuer une redirection vers la sous-fonction adéquate en fonction de l'input client/serveur. Le seul inconvénient est que nous ne pouvons pas faire varier le nombre ou la nature des paramètres pour chaque sous-fonction étant donné que leur appel à toutes ce fait ici. Cela n'est néanmoins pas un problème, il nous suffit de fournir à chaque sous-fonction tous les paramètres dont une voisine pourrait avoir besoin, quitte à ne pas tous les utiliser à chaque fois.

1) Fonctions du serveur :

Afin d'alléger le projet, nous avons décidé de placer toutes les fonctions relatives au serveur dans un fichier à part entière : server_functions.py.

1) #Help (list command)

Fonction qui permet de lister toutes les fonctions disponibles pour le serveur avec le nom de la commande et les paramètres nécessaires à son fonctionnement. Le message est préétabli et print à la demande.

2) #Exit (server shutdown)

Fonction qui permet à l'admin server de shutdown le chat. Afin d'éviter des crashs du côté des clients, ceux-ci voient également leurs sessions être fermées qu'ils soient connectés ou en cours de connexion.

3) #Kill

Fonction qui permet au serveur d'enlever un client de la liste des clients connectés et ainsi que de le kick du chat. Le client en question en sera informé avant d'être renvoyé de la session et son renvoi apparaîtra également sur le chat public.

4) #ListU (list of users in a server)

Fonction qui permet au serveur de lister tous les clients connectés au serveur. Pour ce faire on regarde tous les éléments de la liste des clients connectés.

5) #ListF (list of files in a server)

Fonction qui permet au serveur de lister tous les fichiers upload sur son serveur.

6) # Private (private chat with another user)

Fonction qui permet au serveur d'envoyer un chat privé à un utilisateur en particulier (le client ne pouvant pas lui répondre en privé)

7) #Alert

Fonction qui permet au serveur d'envoyer un message à tous les clients connectés, ce message commence par MESSAGE FROM SERVER

2) Fonctions du client :

De même que pour les fonctions serveur, nous avons placé toutes les fonctions clients dans un fichier client_functions.py.

1) #Help (list command)

Fonction qui permet de lister toutes les fonctions disponibles pour le client avec le nom de la commande et les paramètres nécessaires à son fonctionnement.

2) #Exit (client exit)

Fonction qui permet à l'utilisateur de quitter le serveur chat. Après avoir rentré cette commande le programme client.py se termine et le client est retiré de la liste des clients connectés. Nous nous sommes également assuré que si jamais un client venait à ne pas utiliser cette fonction et se contentait de fermer sa fenêtre de terminal, le terminal du server ne crasherait pas. Quand une telle situation se produit le server se contente de notifier une fermeture intempestive de session client.

3) #ListU (list of users in a server)

Fonction qui permet de lister tous les utilisateurs connectés au chat. Cette fonction est notamment utile lorsque l'on veut rentrer dans une conversation privée avec un autre utilisateur ou que l'on veut créer une chat room car on a besoin des noms des utilisateurs pour ces fonctions.

4) #ListF (list of files in a server)

Fonction qui permet de lister tous les fichiers uploadés sur le serveur.

5) #Ring (notification if the user is logged in)

Fonction qui permet à un utilisateur de ping un autre utilisateur de manière privée pour lui faire savoir qu'il veut communiquer avec lui.

6) #TrfU (Upload file transfer to a server)

Fonction qui permet à un client de télécharger un fichier sur le serveur. Ce fichier a une taille limite (voir rubrique Multi Thread) et peut être défini par son nom s'il est dans le répertoire courant, ou par son chemin absolu.

Son principe de fonctionnement est plus complexe qu'il n'y paraît. Le client exécute la commande #TrfU en indiquant le nom/chemin du fichier qu'il veut envoyer. En réalité, il envoie au serveur un message composé de ces deux éléments mais aussi de la taille du fichier qu'il souhaite envoyer.

Une fois le message envoyé, le client se verrouille dans une boucle while en attendant que le serveur lui indique qu'il est prêt à recevoir le fichier (ou qu'il refuse l'envoi si le fichier est trop volumineux).

Pendant ce temps le serveur reçoit le message contenant les informations fichiers, se prépare à le recevoir en créant un fichier à remplir avec les datas reçues puis envoie un message OK UPLOAD au client. Une fois ce message reçu, le client sort de sa boucle et commence l'envoi du fichier à l'aide d'un thread.

Nous avons utilisé cette méthode car sans ce processus de 'handshake' entre le client et le serveur, le client envoyait la data du fichier avant que le serveur n'ait le temps de s'y préparer et le message contenant les informations sur le fichier contenait également à sa fin le début des datas du fichier.

7) #TrfD (transfer Download file to a server)

Fonction qui permet à un client de télécharger un fichier depuis le serveur.

Son fonctionnement est similaire à la fonction #TrfU mais c'est cette fois ci le client qui demande un fichier, le serveur qui lui renvoie ses informations afin que le client se prépare à la réception, le client qui envoie un message de 'handshake' (en l'occurrence OK DOWNLOAD) puis le serveur qui envoie le fichier à l'aide d'un thread.

8) # Private (private chat with another user)

Fonction qui permet à un client A d'entamer une conversation privée avec un autre client B connecté de son choix. Comme sur zoom, une fois la commande rentrée par A, A et B se retrouvent dans un canal de discussion privé où seuls eux peuvent voir les messages qu'ils s'envoient. Si A ou B décide de retourner dans le chat public ou dans une room, l'autre utilisateur n'est pas automatiquement transféré vers le public, il reste sur le canal privé jusqu'à ce qu'il décide de changer. (comme zoom)

9) #Public (back to the public)

Fonction qui permet au client de revenir sur le chat public lorsqu'il est soit dans une conversation privée, soit dans une chat room.

3) Fonctions de chat room:

Nous avons placé toutes les fonctions relatives à la création, modification ou information de room dans un fichier à part entière room_functions.py. Ces fonctions restent néanmoins accessibles uniquement pour les clients (et non le serveur).

1) #CreateRoom (creation of room)

Fonction qui permet à un client de créer une room avec au moins deux autres clients exceptés lui-même. Tous les noms (de la room ou des utilisateurs) doivent être écrits sans espaces car si on met un espace dans le nom de la room alors le programme va comprendre que le nom de la room est le premier mot avant l'espace et que les autres mots après sont des noms d'utilisateurs alors qu'en fait pas du tout.

Les clients ajoutés à la room ne peuvent être que des clients connectés et non seulement des clients appartenant à la base de données, dans ce cas la fonction renvoie une erreur spécifique.

Après une création de room réussie, le créateur ainsi que tous les membres de cette room ne sont pas redirigés automatiquement vers cette room. D'où la nécessité de la fonction suivante.

2) #JoinRoom (join specified room (only members))

Fonction qui permet à un client de rejoindre une room à laquelle il appartient. S'il n'appartient pas à la room ou si la room n'existe pas, alors la fonction lui renvoie un message d'erreur adapté.

3) #LeaveRoom (leave room definitely, not just back to public)

Fonction qui permet à un client de quitter une room à laquelle il appartient. Attention il ne s'agit pas juste de retourner dans le chat public (pour cela il faut faire #Public), cette commande retire définitivement le client de la liste des clients appartenant à la room. En cas de réussite, le client qui quitte le groupe est redirigé vers le chat public et les autres membres sont notifiés de son départ.

Si le membre qui quitte le groupe est l'admin, alors le premier client de la liste des clients de la room devient admin. Il en est alors notifié par un message.

Si après son départ, le nombre de membres est inférieur à 2, alors la room n'a plus de raison d'être et est dissoute. Chaque membre en est alors notifié est redirigé vers le chat public.

4) #AddRoom (add client to room (only admin))

Fonction qui permet à l'admin d'une room d'ajouter un client connecté qui n'appartient pas déjà à la room. Si le client n'existe pas, n'est pas connecté ou alors appartient déjà à la room, alors la fonction renvoie une erreur adaptée au problème.

Si le client qui essaye d'ajouter un autre client n'est pas l'admin ou alors n'appartient même pas à la room mentionné, alors la fonction renvoie un message d'erreur adapté.

5) #KickRoom (kick client from room (only admin))

Fonction qui permet à l'admin d'une room de retirer un client de la room. Si l'admin essaye de se retirer alors la fonction renvoie un message d'erreur (car cela revient à faire #LeaveRoom).

Si après avoir retiré le client, le nombre de membres est inférieur à 2, alors la room n'a plus de raison d'être et est dissoute. Chaque membre en est alors notifié et est redirigé vers le chat public.

Si le client qui essaye d'ajouter un autre client n'est pas l'admin ou alors n'appartient même pas à la room mentionné, alors la fonction renvoie un message d'erreur adapté.

6) #ListRoom (list rooms client belong to)

Fonction qui permet à un client de lister toutes les rooms auxquelles le client appartient. Si le client n'appartient à aucune room, alors on lui affiche un message personnalisé.

Le fait de pouvoir lister les rooms auxquelles un client appartient est utile pour avoir le nom exact de ces dernières et ainsi pouvoir les rejoindre, lister ses membres et savoir si on en est l'admin.

7) #ListClientRoom (list client in specified room (only members))

Fonction qui permet à un client appartenant à une room de lister tous les clients de cette même room. Si le client n'appartient pas à la room mentionné, alors on lui affiche un message d'erreur.

Pour quitter une room, le client peut soit rentrer dans un chat privé avec un autre utilisateur (#Private <user>) ou bien simplement revenir dans le chat public (#Public).

6) Le serveur nécessite un username et un mot de passe pour se connecter en tant que client. (SQLite pour traiter les password) la database doit être sur le serveur.

Lorsque l'on lance le programme client.py (après avoir lancé server.py sinon la connexion ne se fait pas), on est directement redirigé vers le formulaire de login. On propose à l'utilisateur soit de se connecter avec un compte existant (username et password) soit de se créer un compte.

Pour la connexion, on va récupérer le username et le password rentré par l'utilisateur et ensuite faire une requête SQL à la base de données pour vérifier que le username existe et que le password est correct. Si toutes les vérifications sont validées, alors le client est rajouté à la liste des clients connectés et peut désormais recevoir et envoyer des messages.

Pour la création de compte, on demande un username et un mot de passe comme précédemment excepté que cette fois-ci, on va uniquement checker si le username est déjà utilisé par un autre client de la database. Si c'est le cas, alors on affiche un message d'erreur et demande à l'utilisateur de rentrer

un autre username. Dans le cas contraire, toutes les conditions sont remplies pour créer le compte et le client est rajouté à la liste des clients connectés.

Etant donné que la database devait être sur le serveur, nous ne pouvions pas utiliser MySQL. Nous nous sommes donc tournés vers SQLite qui a permis à chaque membre du groupe d'avoir sa propre base de données d'utilisateurs lorsqu'il codait de son côté.

7) Logs (login, @IP, date, etc.) sur le serveur

A chaque action qui est réalisée sur le serveur, ce dernier en est notifié. Que ce soit un message envoyé en public, privé ou dans une chat room, la connexion d'un client, toutes les actions relatives aux rooms (Création, Join, Leave, Kick...), ou encore toutes les actions relatives aux fichiers (upload, download), dès que l'on engage un processus de création, de transfert (fichiers/messages) ou de changement d'état du client, les informations suivantes sont affichées sur le serveur :

- Date (jour, heure)
- Détails client (username, @IP, port)
- Détails sur l'action réalisé (message + destinataire pour message privé par exemple)

Pour toutes les actions du type #Help, #ListRoom, #ListU...nous avons décidé de ne pas en notifier le serveur car peu informatives et possiblement spamables.

8) Notre fonctionnalité originale

Pour notre fonctionnalité originale, nous avons décidé de rajouter la possibilité à l'utilisateur de créer des chat rooms avec 3 clients ou plus à l'intérieur (lui compris). Avec cette fonction de création de chat room s'ajoute 6 autres fonctions reliées qui permettent :

- De lister tous les clients d'une room
- D'ajouter un client à une room
- De supprimer un client d'une room
- De lister toutes les rooms auxquelles appartient un client
- De quitter une room définitivement
- De rejoindre la room

Pour plus d'informations sur chaque fonction, se conférer à la partie 5.3.

Bonus

1) Le serveur tourne sur Linux et Windows

Sur les 4 membres du groupe, deux utilisaient le système d'exploitation Windows et les deux autres utilisaient MacOS/Linux. Nous avons donc été obligés dès le départ de respecter cette condition de compatibilité d'OS. Cela nous a cependant empêché de rajouter quelques features de mise en forme de la console comme par exemple rajouter des couleurs différentes au message ou encore clear la console pour la rendre plus lisible.

2) Transmission sécurisée avec un chiffrement de bout-en-bout

Au moment d'envoyer des messages au client ou au serveur avec la fonction Send_message (msg_encodé, clé, socket), on va crypter la transmission à l'aide d'un algorithme de Polyencryption. Le

principe est simple, on va se baser sur les codes décimaux ASCII des caractères. Pour chiffrer le message, on va simplement additionner les codes ASCII du caractère de la clé avec le caractère du message à crypter. Il ne faut pas oublier de moduler cette somme avec 127 car il n'y a pas de caractères au code ASCII supérieur à 127 (127 compris car =del). De plus il faut éviter les 32 premiers caractères car ils ne sont pas des caractères écrivables (eof, esc, …). Ensuite, pour déchiffrer ces messages on fait le processus inverse, c'est à dire que l'on va soustraire les codes ASCII des caractères de la clé avec ceux du message chiffré en respectant les mêmes règles que pour le chiffrement (code ASCII compris entre 32 et 126). Avec ce chiffrement, on est certain que la transmission est cryptée et sécurisée de bout en bout du client au serveur et inversement rendant l'interception des messages obsolètes sans la clé de cryptage. Cette clé de cryptage est déclarée en dur dans le programme, elle est donc la même pour tous les messages.

Pour ce qui est des fichiers ceux-ci ne sont pas cryptés pendant leur transfert. En effet notre cryptage était adaptable sur certains types de fichiers (par exemple .txt) mais pas sur d'autres comme les PDF. Cela a l'air d'être du au fait que notre cryptage ne fonctionne qu'avec des encodages UTF-8.