# Documentation programmeur

Table des matières:



## Introduction

Voici le document expliquant le fonctionnement des codes du projet Seleenix. Les codes concernés sont répartis dans trois scripts : client.py, gui.py, et server.py. Les deux premiers scripts permettent aux clients de se connecter à un serveur de tchat et d'interagir avec celui-ci. Le premier script, client.py, permet une interaction avec le serveur en mode terminal, tandis que le second offre la même fonctionnalité avec une interface graphique. Quant au script server.py, il s'agit du script permettant de faire fonctionner le serveur de tchat. Il est important de noter que les codes sont déjà documentés donc une lecture croisée entre cette documentation et le code est conseillée pour mieux comprendre le projet.

## Architecture des codes

Dans cette partie, nous allons tout d'abord expliquer comment sont structurés les différents codes autant du côté client que du côté serveur.

### Côté client

D'une façon générale, les codes clients sont structurés de la même manière, une des différences est liée à l'utilisation de classes pour les fenêtres de l'application graphique. Nous allons détailler l'architecture et le comportement des deux versions.

#### Non-graphique (client.py)

Dans un premier temps des variables sont initialisées au début du programme:

- connected : variable déterminant l'état de la connexion (défaut : false)
- key : la clé de chiffrage AES
- IV : le vecteur d'initialisation AES
- cipher : l'objet permettant le chiffrage et déchiffrage (unique).

Puis sont implémentées des classes permettant de créer des erreurs personnalisées:

```
class ChallengeRefused(Exception): # erreur customisée en lien avec le challenge
    def __init__(self, message):
        super().__init__(message)

class ConnectionClosedByServer(Exception):
    def __init__(self, message):
        super().__init__(message)

class KickedFromRoom(Exception):
    def __init__(self, message):
        super().__init__(message)

class BannedFromServer(Exception):
    def __init__(self, message):
        super().__init__(message)
```

Le point d'entrée du programme en tant que script se fait à partir de la ligne de test : 'if **name** == "**main**":'. Ici s'y plusieurs mécanismes:

- appel fonction arg_parse() (afin récupérer les arguments)
- Vérifier la présence du drapeau - -search → vérification de l'environnement sur lequel le script est exécuté, si c'est une machine virtuelle il renvoie une erreur, sinon il lance une recherche d'un paquet UDP sur le réseau contenant les informations du serveur.
- Une fois le paquet trouvé il lance la fonction main avec comme arguments positionnels l'hôte et le port du serveur.
- Dans le cas où le drapeau de recherche n'est pas spécifié il récupère les valeurs des arguments - - port et - -host. Si ceux-ci sont vides il prendra comme valeurs par défaut le couple hôte:port suivante : 127.0.0.1:1234. Puis exécute la fonction main() avec ces deux arguments positionnels.

La fonction main() prenant deux arguments positionnels permet d'initier une connexion synchrone avec le serveur et de commencer à discuter avec celui-ci afin d'obtenir une autorisation :

- Il demande un user (optionnel car le serveur a une liste de pseudonymes) et un mot de passes (obligatoire).
- Puis génère un défi (challenge), une liste de 16 nombres respectant une logique donnée, connue par le serveur et les clients uniquement. Cette génération passe une compréhension de listes et utilise la bibliothèque random pour arriver à ses fins.
- Ce challenge se trouve concaténé aux identifiants dans une variable bytes nommée payload.

```
challenge = ",".join([str(random.randint(1,65535) * 2) for i in range(16)]) # un challenge
payload = (challenge + ";" + user + "," + password).encode()
```

- Il initie une connexion avec le serveur et envoie la payload chiffrée par la fonction de chiffrage AES encrypt().
- Selon la réponse il détermine si il est autorisé ou pas à interagir, si il reçoit banni ou autre chose que "synced" dans le message il comprend qu'il a été refusé.
- la réponse de synchronisation contient des informations contextes utilisateurs (user, salles).
- La fonction interactive se lance avec comme argument l'hôte.

La fonction interactive() permet de lancer les fonctions d'interactions (send, receive), elle débute par l'attribution de la valeur True à la variable globale connected. Puis lance dans un nouveau thread la fonction receive() avec les arguments socket du client et hôte. Dans son thread principal elle exécute la fonction send() prenant les mêmes arguments postionnels.

La fonction receive() s'occupe du traitement des messages envoyés par le serveur, mais aussi du positionnement du terminal pour éviter certains artefacts visuels (avec les codes ANSI)

Ex :

```
print("\r\033[2K",end="")  # carriage return + clear line
```

Selon l'entête de la réponse la fonction a ses propres logiques qui sont customisables à volonté puisque du côté client, elle supporte les entêtes suivants :

- "cmd: " :  Résultat de la commande /rooms.
- "users:" : Résultat commande /users.
- "jn:" : Résultat d'un join de salons.
- "query:" : Résultat requête administrateurs.
- "us:" : Résultat suite à une commande unsubscribe.
- "fwd:" : Messages d'un autre utilisateur(dans le même salon) transité par le serveur.
- "old:" : Permet de récupérer partiellement les messages du salon actuel.

Dans cette fonction se trouve une exception du nom KickedFromRoom qui s'active lorsqu'un utilisateur est kick d'une room dans laquelle il se trouve actuellement.

Dès qu'elle reçoit bye ou arret, elle lance la fermeture du programme. (en mettant la variable connected à False)

La fonction send() prend les mêmes arguments positionnels que la fonction receive() et permet aussi de positionner le terminal avec les codes ANSI.

- Elle demande l'entrée du l'utilisateur
- Si elle est vide elle fait rien.
- Si elle est comporte des caractères, elle l'affiche dans le fils de discussion avec le préfixe "you:" avant de l'envoyer au serveur.
- Si le message commence par des entêtes de commande /query, /accept, /unsubscribe, elle retravaille la forme des payloads et l'envoie au serveur.

Encore une fois si elle reçoit bye ou arrêt, elle lance la fermeture du programme. (en mettant la variable connected à False)

## Graphique (gui.py)

Dans un premier temps des variables sont initialisées au début du programme:

- connected : variable déterminant l'état de la connexion (défaut : false)

- key : la clé de chiffrage AES
- IV : le vecteur d'initialisation AES
- cipher : l'objet permettant le chiffrage et déchiffrage (unique).

Puis sont implémentées des classes permettant de créer des erreurs personnalisées:

```
class ChallengeRefused(Exception): # erreur customisée en lien avec le challenge
    def __init__(self, message):
        super().__init__(message)

class ConnectionClosedByServer(Exception):
    def __init__(self, message):
        super().__init__(message)

class KickedFromRoom(Exception):
    def __init__(self, message):
        super().__init__(message)

class BannedFromServer(Exception):
    def __init__(self, message):
        super().__init__(message)
```

Le point d'entrée du programme en tant que script se fait à partir de la ligne de test : 'if **name** == "**main**":'. Ici s'y plusieurs mécanismes:

- appel fonction arg_parse() (afin récupérer les arguments)
- Vérifier la présence du drapeau - -search → vérification de l'environnement sur lequel le script est exécuté, si c'est une machine virtuelle il renvoie une erreur, sinon il lance une recherche d'un paquet UDP sur le réseau contenant les informations du serveur.
- Une fois le paquet trouvé, il y a attribution des valeurs trouvées à la variable port et host.
- Dans le cas où le drapeau de recherche n'est pas spécifié il récupère les valeurs des arguments - - port et - -host. Si ceux-ci sont vides il prendra comme valeurs par défaut le couple hôte:port suivante : 127.0.0.1:1234. Puis il instancie le widget PyQT Login avant d'exécuter l'application.

La classe Login hérite des spécificités de la classe parente QWidget et selon l'interaction de l'utilisateur exécute la méthodes suivante: Login()

Celle-ci récupère les attributs d'instance username et password, et effectue globalement les mêmes actions que le client non-graphique en vu d'obtenir une autorisation côté serveur:

- il créé un objet socket à l'aide la fonction client_socket_create()
- Créé un challenge de la même logique le client non-graphique.
- Concatène le challenge aux identifiants dans une variable bytes nommée Payload qui possède un suffixe GUI (indiquant au serveur un contexte supplémentaire).
- Attends la réponse de synchronisation (comme pour client non-graphique).

- Si elle comporte synced elle attribut True à la variable d'instance connected, et récupère username, current_room, allowed_room que sont le contexte de l'utilisateur authentifié.
- Puis exécute open_chat() qui créé une instance ChatApp() et connecte les signaux avec cette classe.
- Et la fonction updateNewWindowsUi() permet de rafraichir le style de la page ChatApp selon le contexte de l'utilisateur authentifié.

La classe ChatApp hérite des spécificités de la classe parente QWidget et se trouve être la fenêtre graphique principale permettant l'interaction avec le serveur.

Voici ses attributs:

```
connected (bool): Un booléen pour savoir si l'utilisateur est connecté.
allowed_room (str): Les rooms autorisées.
room (str): La room actuelle.
username (str): Le nom d'utilisateur.
last_message (str): Le dernier message envoyé.
showMessageSignal (str, str): Un signal pour afficher les messages.
showErrorMessageSignal (str, str): Un signal pour afficher les messages d'erreur.
stopThreadSignal (obj): Un signal pour arrêter le thread.
```

Voici ses méthodes:

```
EditConnected (func): Modifier la variable globale connected.
IsConnected (func): Récupérer la variable globale connected.
join (func): Rejoindre une room.
close (func): Fermer la fenêtre.
closeEvent (func): Fermer la fenêtre.
sendMessage (func): Envoyer un message.
restore_old_messages (func): Restaurer les anciens messages d'un salon.
handleReceivedMessage (func): Traiter les messages reçus.
showInfoMessage (func): Afficher un message.
showCriticalMessage (func): Afficher un message d'erreur.
```

Les méthodes surlignées en Rouge sont les méthodes nécessaires à minimum d'interaction avec le serveur c'est pourquoi nous allons les détailler.

Mais pour rajouter une logique faut reprendre depuis la méthode de classe, __ init __ qui permet de construire la classe, grâce à celle-ci on sait que chaque fonction est liée au moins à un bouton ou plus. Excepté de handleReceivedMessage qui tourne dans un nouveau thread. On sait que un panel admin est ouvert il sera expliqué plus tard.

Ce thread n'est pas créé à l'aide de la bibliothèque threading mais plutôt de Qthread offrant une intégration plus forte avec l'application graphique PyQT puisque celle-ci permet de faire transiter les signaux entre plusieurs threads et plusieurs instances de classes.

```
self.recvthread = ReceiveThread(self)
self.recvthread.messageReceived.connect(self.handleReceivedMessage)
connectSignals(self)
```

Cette classe Qthread contient comme attributs des signaux, une méthode run et stop. La première intercepte les messages du serveur et les envoie à la fonction handleReceivedMessage à l'aide signaux. Et la seconde permet d'arrêter le thread.

Pour aller plus dans les détails, la fonction handleReceivedMessage s'occupe de l'affichage des messages et des effets de ceux-ci sur l'interface graphique, elle supporte les mêmes entêtes que le client non-graphique mais dispose d'un de plus:

- "scb:" : réponse à une souscription de salon → elle rafraichie les rooms autorisées avec la commande /rooms.

Les salons autorisés sont en Vert et ceux dont l'accès est interdit sont en Rouge, lorsqu'on clique sur un des boutons des salons on exécute la fonction join() avec comme argument positionnel le texte du bouton.

La méthode join() envoie une requête /join si le bouton est vert et /subscribe si il est rouge.

La méthode sendMessage() permet d'envoyer des messages au serveur si la connexion avec celui-ci est toujours active et que le contenu du message ne soit pas vide. Le contenu est récupéré à partir de la box message_input et est vidée à chaque envoi de messages. Si le message n'est pas vide il rajoute à la liste le message envoyé au serveur avec comme préfixe "Vous:".

## Côté serveur

Le code serveur est plus ou moins structuré de la même façon que le client non graphique puisque celui-ci a été pensé pour être en mode terminal. C'est-à-dire qu'il présente également du positionnement de terminal (codes ANSI) pour amoindrir les artéfacts visuels.

Dans un premier temps des variables sont initialisées au début du programme:

- connected : variable déterminant l'état de la connexion (défaut : True)
- key : la clé de chiffrage AES
- IV : le vecteur d'initialisation AES
- cipher : l'objet permettant le chiffrage et déchiffrage (unique).
- host : l'hôte sur lequel bind les connexions (défaut : "0.0.0.0")
- port : récupéré depuis arg_parse() (défaut : 1234)
- socket_list : dictionnaire content les sockets des clients (la clé est un nom d'utilisateur)
- user_caps : dictionnaire contenant les instances user_capabilities des clients (la clé est un socket)
- mysql_passwd : mot de passe mysql (variable d'environnement)
- default_pseudo : liste de pseudonymes par défaut (si le champs utilisateur envoyé par le client est "default")

Mais également sont présentes des classes d'exceptions pour des erreurs personnalisées :

```
class LogoutBroadcast(Exception):
    def __init__(self, message):
        super().__init__(message)

class IncorrectPassword(Exception):
    def __init__(self, message):
        super().__init__(message)

class BannedUser(Exception):
    def __init__(self, message):
        super().__init__(message)

class LimitConnectionIP(Exception):
    def __init__(self, message):
        super().__init__(message)
```

Il y aussi une autre classe, cette classe est extrêmement importante puisqu'elle contient le contexte d'un utilisateur dans une instance. Elle se nomme user_capabilities et permet de gérer localement les droits des utilisateurs sans devoir interroger constamment la base de donnée.

Ces attributs sont les suivants:

```
username: le nom d'utilisateur
allowed_room: les salons autorisés
is_admin: les droits d'administrateur
all_rooms: tous les salons disponibles
current_room: le salon actuel
```

Ils sont tous privés donc ils ont des getter et des setter, voici globalement les méthodes :

```
__init__: constructeur de la classe
is_admin: méthode pour vérifier si l'utilisateur est administrateur
username: méthode pour récupérer le nom d'utilisateur
allowed_room: méthode pour récupérer les salons autorisés
```

Le point d'entrée du programme en tant que script se fait à partir de la ligne de test : 'if **name** == "**main**":'. Ici s'y plusieurs mécanismes:

- il créé la variable server_socket et y bind les adresses acceptées et son port d'écoute.
- Il met scapy en mode non verbeux.
- Il lance dans un nouveau thread la fonction send_announcement() qui envoie un message UDP dans le réseau (contenant les informations utiles à la connexion au serveur).
- Il rentre dans une boucle d'acception qui boucle si connected est vrai :
    - une fois une connexion acceptée dans un bloc try : il attend un message (contenant la payload) et essaie de la déchiffrer avec la fonction decrypt(), si il y a une erreur de déchiffrement il envoie "garbage" au client qui a pour effet de fermer sa connexion et il ferme le socket.

- Il vérifie si le format de la variable message_1 (contenant la payload déchiffrée) est correct, dans le cas contraire il envoie “garbage” au client et ferme le socket.
- Ensuite il récupère le challenge du client et les identifiants se trouvant dans la payload. (à l’aide de split)
- Il vérifie si le challenge respecte bien une règle spécifique (ici simplement que tous les nombres soient divisibles par 2)
  - ```
    client_challenge, credentials = message_1.split(";")[0].split(","), message_1.spli
    connect_condition = lambda x: int(x) % 2 == 0
    client_condition = all(connect_condition(elem) for elem in client_challenge)
    if client_condition:
    [reste des instructions]
    ```
- Dans la variable “credentials” il prend le username et le mot de passe (si le username est “default” il prend un username aléatoire de la liste des pseudonymes)
- Puis lance la fonction user_sql_handler() avec ces informations
  -
  ```
  is_auth = user_sql_handler(unique_pseudo, credentials[1], conn, str(address))
  ```
  - Cette fonction permet l’authentification, elle est donc essentielle au fonctionnement du programme, voici sa docstring pour plus d’explications :
    -
    ```
    """
    Fonction essentielle permettant de gérer l'authentification des utilisateurs.
    Elle gère à la fois la connexion et l'inscription des utilisateurs.

    Les mots de passes ne sont pas entrés en clair dans la base de données, ils sont en
    Une limite de connexion est imposée par adresse ip (2 comptes max par ip).
    Une fois la connexion établie, les informations de l'utilisateur sont restaurées de

    Args:
        user: le nom d'utilisateur
        password: le mot de passe
        socket: le socket du client
        ip_address: l'adresse ip du client

    Returns:
        True: si l'utilisateur est authentifié
        False: si l'utilisateur n'est pas authentifié

    Raises:
        Error: si une erreur de connexion à la base de données est détectée
        IncorrectPassword: si le mot de passe est incorrect
        LimitConnectionIP: si la limite de connexion par ip est atteinte
        BannedUser: si l'utilisateur est banni

        """
    ```
    ```

- Si la réponse retournée par user_sql_handler() dans is_auth est “True”
  - La variable contenant le socket est stockée dans le dictionnaire des sockets avec comme clé le pseudonyme de l’utilisateur.
  - Puis pour finir la synchronisation il envoie au client un message contenant les informations pour la synchronisation dont le contexte de l’utilisateur.
  - Lance dans un nouveau thread la fonction interactive() avec comme arguments le nouveau socket et l’adresse de l’hôte.

La fonction interactive() du serveur fonctionne quasiment de la même manière que celle dans les autres scripts à la différence qu’il y a un appel de la fonction restore_old_messages() pour envoyer les messages du salon dans lequel se trouve l’utilisateur.  Il lance dans un nouveau thread la fonction receive() et lance dans le thread actuel la fonction Send() uniquement si c’est la première connexion (afin d’éviter d’ouvrir trop de prompt).

La fonction receive() permet de gérer la réception de messages, les commandes administrateurs et utilisateurs, mais également la redirection des messages vers les autres clients se trouvant dans le même salon. Elle enregistre chaque nouveau message reçu dans une table de la base de donnée.

Les commandes supportées dans receive sont les suivantes:

- /query (administrateurs) : afficher les requêtes d’utilisateurs
- /accept | /refuse (administrateurs) : accepter / refuser une requête
- /rooms : afficher les salons autorisés par l’utilisateur
- /users : afficher les utilisateurs connectés
- /subscribe : s’abonner à un salon
- /join : rejoindre un salon
- /unsubscribe :  se désabonner d’un salon

Toutes ces actions se font selon certaines conditions et successivement:

- Dans une boucle de réception de message,  le serveur réceptionne un message
  - Il affiche côté serveur le message précédé du pseudonyme de la personne qui a envoyé le message
  - Il vérifie si le message respecte les conditions de redirections : pas de commandes, et pas de réponses de commandes. Si c’est c’est respecté il envoie à tous les clients connectés se trouvant dans le même salon que le client qui a envoyé le message.
  - Puis il log le message dans la table channel de la base de donnée avec les informations suivantes :
    - id de la room
    - username
    - nom de la room
    - contenu du message
  - Globalement pour toutes les commandes il applique sa logique puis il réinitialise la variable reply pour la prochaine boucle.

- Certaines commandes impliquent un changement de contexte pour son utilisateur donc il faut mettre à jour ce changement dans la base de donnée d'où l'utilisation de la fonction update_userinfo_sql(), voici sa docstring pour plus d'explications:
  -

```
"""
Fonction essentielle permettant la mise à jour des informations dans la base de do
Pour aller plus loin, elle permet de mettre à jour le contexte autours des utilisa

Args:
    user: le nom d'utilisateur
    value: la valeur à mettre à jour
    reason: la raison de la mise à jour
    socket: le socket du client

Returns:
    None

Raises:
    Exception: si la raison n'est pas valide (upgrade, roomupdate, password, roomd

"""
```

La fonction Send() permet d'envoyer des messages aux utilisateurs sous le compte "Server", elle permet également à l'administrateur se trouvant derrière d'exécuter des commandes administrateurs. Il est important de noter que c'est lui qui désigne les administrateurs parmi les clients et que ces administrateurs ne peuvent grader/rétrograder le rang d'un client.

Voici les commandes supportées:

- bye [user] : déconnecter un utilisateur (sans log)
- /kick [pseudo] [rooms|ALL] 1h30 [reasons] : kick un utilisateur d'une room ou du serveur pendant une période donnée (conseillée : 1h30). Le motifs n'est pas obligatoire.
- /ban [pseudo] {reasons] : ban utilisateur:
- /kill : ferme la connexion serveur (annonce au client avant de fermer).
- /op [pseudo] : Grader un utilisateur en Administrateur.
- /deop [pseudo] : Rétrograder un utilisateur en simple client.
- /users : Afficher la liste des utilisateurs connectés.
- /query : Afficher les requêtes des utilisateurs.
- /accept | /refuse [id] : Accepter / Refuser une requête.
- /unsubscribe [user] [room] : désabonner un utilisateur d'une room.