Yann Blanck RT21

SAE3.02

Documentation Développeur

Sommaire

I. code Serveur

Dans cette partie nous allons parlé du code permettant le fonctionnement du serveur

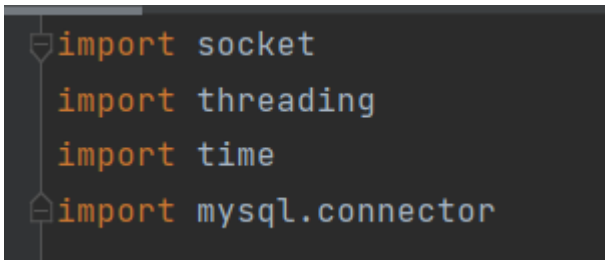
1. Initialisation

Dans un premier temps, le code comprend les imports nécessaires à son fonctionnement. On retrouve le package ***socket***, qui permet la connexion entre deux appareils grâce à une adresse IP et un port de connexion. Ce package utilise le protocole TCP/IP.

Ensuite, nous avons le package ***threading***, qui permet la gestion de plusieurs fonctions ou programmes de manière simultanée. Ici, il est utilisé afin de gérer plusieurs connexions de clients au sein du même serveur.

Puis, le package ***time*** est inclus. Ce package permet de gérer le temps et peut être utilisé, par exemple, pour gérer le temps de bannissement d'un client.

Enfin, le package ***mysql-connector*** est utilisé. Il permet au serveur de se connecter à une base de données MySQL et d'insérer des données dans des tables, ou de créer ces tables si elles n'existent pas déjà.



Dans la suite du code, nous trouvons un ensemble de listes utilisées dans diverses fonctions du programme :

- La liste « clients » sera utilisée pour gérer les clients qui se connectent.

- La liste « ban » servira à recenser les clients bannis.

- La liste « kick » sera dédiée aux clients ayant été expulsés du serveur.

```
# Liste pour stocker les connexions des clients et leurs identifiants
clients = {}
ban = {}
kick = {}
admin = "toto"
```

2. Base de donnée

Notre serveur est connecté à une base de données servant à enregistrer les différents utilisateurs ainsi que les messages envoyés par ces derniers, avec l'heure et la date d'envoi. Dans un premier temps, la connexion à la base de données est initialisée en spécifiant l'adresse IP de la machine contenant la base de données. Ensuite, les informations d'identification de l'utilisateur ayant accès à la base de données, telles que son nom et son mot de passe, sont fournies. Enfin, le nom de la base de données est indiqué, laquelle contiendra les tables où seront enregistrés les différents utilisateurs et les messages envoyés par les clients.

```
# Initialiser la connexion à la base de données MySQL
db_connection = mysql.connector.connect(
    host="127.0.0.1",
    user="root",
    password="toto",
    database="serveurchat",
    #auth_plugin = "mysql_native_password"
)
db_cursor = db_connection.cursor()
```

Dans un second temps, nous allons créer les tables qui serviront d'enregistrement pour les messages et les différents identifiants d'utilisateurs. Le package ***mysql.connector*** va exécuter des lignes de commande utilisées dans MySQL pour créer les tables si elles n'existent pas déjà et ajouter les différentes colonnes nécessaires.

Dans la table des messages, on retrouvera un identifiant (id), le nom d'utilisateur correspondant au login (user), le contenu du message, ainsi que le timestamp indiquant la date et l'heure d'émission du message.

```
# Créer la table messages si elle n'existe pas
db_cursor.execute('''
    CREATE TABLE IF NOT EXISTS messages (
        id INT AUTO_INCREMENT PRIMARY KEY,
        user VARCHAR(255),
        message TEXT,
        timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    )
''')
```

Dans la table utilisateurs on vas retrouvé un id et le nom correspondant au login que l'utilisateur a inscrit

```
# Créer la table messages si elle n'existe pas
db_cursor.execute('''
    CREATE TABLE IF NOT EXISTS utilisateurs (
        id INT AUTO_INCREMENT PRIMARY KEY,
        nom VARCHAR(255)
    )
''')
db_connection.commit()
```

Enfin, le code comprend deux fonctions permettant l'enregistrement des données. La fonction save_message permet d'enregistrer les messages en prenant comme valeurs l'utilisateur (user) et le message. Ces derniers seront utilisés dans une ligne de commande SQL pour les insérer dans une table.

```
def save_message(user, message):
    # Sauvegarder le message dans la base de données
    db_cursor.execute("INSERT INTO messages (user, message) VALUES (%s, %s)", (user, message))
    db_connection.commit()
```

La fonction d'inscription prend en compte deux valeurs, `lient et client_id. Dans un premier temps, la fonction vérifie si le nom d'utilisateur existe déjà. Si c'est le cas, l'utilisateur devra choisir un autre nom. Si ce n'est pas le cas, le nom du client est enregistré dans la base de données en utilisant une ligne de commande SQL permettant d'insérer le login dans une table.

```python
def inscription(client, client_id):
    global db_cursor
    global db_connection

    try:
        # Vérifier si l'utilisateur existe déjà dans la base de données
        db_cursor.execute("SELECT id FROM utilisateurs WHERE nom = %s", (client_id,))
        result = db_cursor.fetchone()

        if result:
            client.send("Cet identifiant est déjà utilisé. Veuillez en choisir un autre.".encode())
            return

        # Insérer l'utilisateur dans la base de données
        db_cursor.execute("INSERT INTO utilisateurs (nom) VALUES (%s)", (client_id,))
        db_connection.commit()

        client.send("Inscription réussie! Vous pouvez maintenant envoyer des messages.".encode())
    except Exception as e:
        print(f"Erreur lors de l'inscription : {e}")
        client.send("Une erreur s'est produite lors de l'inscription.".encode())
```

3. Les commandes

Le serveur dispose de plusieurs commandes (kill, ban et kick). Ces commandes ne sont utilisables que sur le serveur et uniquement après avoir saisi le mot de passe administrateur. En l'absence de saisie du mot de passe, aucune commande ne pourra être exécutée.

```
def commande(serv):
    global clients
    global ban
    global serveur_connecte  # Ajouter cette ligne

    password = input("Entrez le mot de passe du serveur: ")  # Définir un mot de passe
    if password == "toto":
        serveur_connecte = True
        print("Serveur connecté.")
    else:
        print("Mot de passe incorrect. Le serveur ne sera pas connecté.")
        return

    while True:
        command = input()

        # Vérifier si le serveur est connecté avant d'autoriser les commandes
        if not serveur_connecte:
            print("Le serveur n'est pas connecté. Connectez-vous d'abord.")
            continue
```

Le code va ensuite décomposé le commande entré et vérifier qu’elle commande a été rentré.

Ensuite en fonction de la commande entré le serveur va l’exécuter.

```
comm = command[:3]
com = command[:4]
user = ""
```

- Kill va permettre de fermé le serveur et de coupé tout les conversations

```
if command.lower() == "kill":
    for client_socket in clients.keys():
        client_socket.close()
    serv.close()
    break
```

- ban vas bannir le l’utilisateur désigné après la commande (exemple « ban pierre »)

```
        elif comm.lower() == "ban":
            user = command[4:]
            for client_id, client_socket in clients.items():
                if client_id == user:
                    ban[client_socket] = client_id
                    client_socket.close()
                    break
```

- kick va permettre d’empêcher un utilisateur de communiquer pendant 2 minutes (exemple « kick michel »

```
    elif com.lower() == "kick":
        user = command[5:]
        for client_socket, id in clients.items():
            if id == user:
                kick[client_socket] = time.time() + 120  # Bloquer le client pour 120 secondes
                break
    else:
```

si le commande entré n’est pas reconnue alors le serveur indiquerale message « commande inconnue » suivie de la commande que vous avez rentré.

```
    else:
        print(f"Commande inconnue: {command}")
```

/!\ ces commandes sont présente dans le code mais ne sont pas entièrement foctionnel

4. Gestion d’envoi de message

Pour gérer l’envoie et la réception de message le code utilise plusieurs fonction.

a. Broadcast

Cette fonction utilise deux arguments, message et sender. Elle permet de gérer l'envoi de messages à tous les clients connectés au serveur de chat. Elle parcourt la liste des clients pour trouver tous les clients connectés.

La fonction vérifie également que le nom du client auquel elle envoie le message n'est pas le même que celui qui envoie le message, en comparant les noms de la liste avec celui de sender, qui représente l'expéditeur.

```python
def broadcast(message, sender):
    for client, client_id in clients.items():
        # Ne pas envoyer le message à l'expéditeur d'origine
        if client != sender:
            try:
                client_name = clients[sender]
                client.send(f"{client_name}: {message}".encode())
            except:
                # En cas d'erreur, supprimer le client de la liste
                remove_client(client)
```

b. message_inscription

Cette fonction utilise deux arguments, client et address. Elle décoder le message reçu par un client et le décompose. Si le message contient le mot "inscription", alors la fonction d'inscription, qui ajoute le login à la base de données, est appelée pour traiter la deuxième partie du message contenant le login.

```
def message_inscription(client, address):
    client_id = client.recv(1024).decode()
    clientident = client_id.strip()[12:]
    inscript = client_id[:11]
    #print(client_id)
    #print(clientident)
    #print(inscript)

    # Vérifier si le message est une demande d'inscription
    if inscript.lower() == "inscription":
        inscription(client, clientident)
        return
```

c. message_connexion

Cette fonction prend deux arguments, client et address. Dans un premier temps, elle lit le message reçu et le décompose. Ensuite, la fonction vérifie dans la base de données si l'utilisateur est déjà inscrit. Si ce n'est pas le cas, le serveur renverra le message : "Vous n'êtes pas inscrit. Veuillez vous inscrire pour accéder au chat."

```
def message_connexion(client, address):
    client_id = client.recv(1024).decode()
    client_name = client_id.strip()[10:]
    connexion = client_id[:9]

    # Vérifier si l'utilisateur est inscrit dans la base de données
    db_cursor.execute("SELECT id FROM utilisateurs WHERE nom = %s", (client_name,))
    result = db_cursor.fetchone()
    #print(result)

    if not result:
        client.send("Vous n'êtes pas inscrit. Veuillez vous inscrire pour accéder au chat.".encode())
        return
```

Si l'utilisateur existe, le serveur va marquer « Nouveau client connecté: » suivie du login de l'utilisateur, la fonction vas ensuite rentrer dans une boucle qui va permettre de recevoir les message de les sauvegarder dans la base de donnée et de les envoyer a tout les autre utilisateur via la fonction broadcast.

```
    try:
        if result:
            clients[client] = client_name
            print(f"Nouveau client connecté: {client_name}")

            while True:
            # Recevoir un message du client
                message = client.recv(1024)
                print(f"{client_name}, {message.decode()}")
                if not message:
                    break

                save_message(client_name, message.decode())

                # Diffuser le message à tous les clients connectés
                broadcast(message.decode(), client)
                #print(message.decode())
    except ConnectionResetError:
        # Le client s'est déconnecté de manière inattendue
        print("Client déconnecté de manière inattendue.")
    finally:
        # Gérer la déconnexion du client
        handle_disconnection(client)
```

5. Gestion des clients

a. remove_client

Cette fonction prend comme argument client. Elle permet de supprimer le client de la liste des clients connectés en comparant le client désigné dans l'argument avec les clients présents dans la liste. Ensuite, la fonction ferme la connexion avec le client..

```
def remove_client(client):
    if client in clients:
        client_id = clients.pop(client)
        print(f"Client {client_id} déconnecté.")
        client.close()
```

b. handle_disconnection

Cette fonction prend comme argument client_socket. Elle récupère le socket du client, puis appelle la fonction `remove_client` pour déconnecter le client.

```
def handle_disconnection(client_socket):
    global clients

    # Récupérer le nom du client déconnecté
    disconnected_client = clients.get(client_socket)

    if disconnected_client:
        print(f"Client déconnecté: {disconnected_client}")
        remove_client(client_socket)
    else:
        print("Erreur lors de la déconnexion : client non trouvé.")
```

c. handle_client

Cette fonction prend en argument client et address.

Cette fonction cette fonction appelle les fonctions message_connexion et message_inscription.

```
def handle_client(client, address):

    message_connexion(client, address)

    message_inscription(client, address)
```

6. Fonction main

La fonction main vas crée la connexion aux socket sur le port 12345. Elle vas ensuite démarrer un thread pour gérer les différentes commande du serveur.

```python
def main():
    global clients

    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind(('0.0.0.0', 12345))
    server_socket.listen(5)
    #print(server_socket)

    commande_thread = threading.Thread(target=commande, args=(server_socket,))
    commande_thread.start()

    print("Serveur activé")
```

La fonction vas ensuite rentrer dans une boucle qui vas vérifier si l'utilisateur est présent dans la liste des personne qui sont kick. Elle vas aussi gérer un thread qui va permettre de gérer la connexion de plusieurs client en même temps.

```python
try:
    while True:
        # Accepter une connexion client
        client_socket, address = server_socket.accept()

        if client_socket in kick and time.time() < kick[client_socket]:
            print("Client kické. Attendez un moment avant de vous reconnecter.")
            client_socket.close()
            continue

        # Gérer le client dans un thread distinct
        client_thread = threading.Thread(target=handle_client, args=(client_socket,address))
        client_thread.start()

except KeyboardInterrupt:
    print("Arrêt du serveur.")
finally:
    # Fermer le socket du serveur
    server_socket.close()

    db_cursor.close()
    db_connection.close()
```

II. Code Client

Dans cette partie nous allons parlé du code du client permettant a l'utilisateur, de chatté

1. Initialisation

Dans un premier temps le programme appelé les paquet socket, ***threading***, PYQT5 et ***sys***

- ***socket*** permettre d'établir la liaison via avec le serveur via connexion IP
- ***threading*** permettre d'effectuer plusieurs tache de manière simultané
- ***sys*** permet de gérer des action qui nécessiterais le des actions lié au système

```
import socket
import threading
from PyQt5 import QtCore, QtWidgets
import sys
```

2. fenêtre de Login

La class Login permet de gérer a l’aide de PYQT l’ouverture de la fenêtre permettant la connexion d’un utilisateur.

a. send_login

Cette fonction gère l'envoi du login au serveur après l'appui sur le bouton de connexion. Elle prend la ligne dans laquelle l'utilisateur saisit son login et la ligne dans laquelle l'utilisateur saisit l'adresse IP du serveur ciblé. Ensuite, elle émet un message contenant le login et l'adresse IP au serveur.

```
class Login(QtWidgets.QWidget):
    switch_window = QtCore.pyqtSignal(str, str)

    def sending_login(self):
        username = self.lineEdit.text()
        ip_address = self.ipLineEdit.text()
        self.switch_window.emit(username, ip_address)
        login_data = f"connexion {username}"
        self.send_message(login_data)
```

b. open_inscription

Cette fonction vas appeler la fonction showinsc ce trouvant dans la class ChatApplication.

```
def open_inscription(self):
    self.fenisnc = Inscription(send_message_func=self.send_message)
    self.fenisnc.inscription_signal.connect(self.switch_window)
    self.fenisnc.show()
```

c. Fonction Init

La fonction Init on initialise les fonctions permettant de crée l'application pour gérer la connexion de l'utilisateur.

On initialise les différentes partie de l'interface graphique en leur donnant un nom, une taille et et l'emplacement.

```
    def __init__(self, send_message_func):
        QtWidgets.QWidget.__init__(self)

        self.feninsc = Inscription(send_message_func=send_message_func)

        self.setObjectName("log")
        self.resize(428, 167)

        self.lineEdit = QtWidgets.QLineEdit(self)
        self.lineEdit.setGeometry(QtCore.QRect(30, 110, 241, 31))
        self.lineEdit.setObjectName("lineEdit")

        self.pushButton = QtWidgets.QPushButton(self)
        self.pushButton.setGeometry(QtCore.QRect(280, 110, 75, 31))
        self.pushButton.setObjectName("pushButton")
        self.pushButton.clicked.connect(self.sending_login)

        self.inscbutton = QtWidgets.QPushButton(self)
        self.inscbutton.setGeometry(QtCore.QRect(280, 69, 75, 31))
        self.inscbutton.setObjectName("inscbutton")
        self.inscbutton.clicked.connect(self.open_inscription)

        self.label = QtWidgets.QLabel(self)
        self.label.setGeometry(QtCore.QRect(150, 10, 121, 16))
        self.label.setObjectName("label")
```

```
self.label_2 = QtWidgets.QLabel(self)
self.label_2.setGeometry(QtCore.QRect(30, 90, 121, 16))
self.label_2.setObjectName("label_2")

# Add a QLineEdit for IP address input
self.ipLineEdit = QtWidgets.QLineEdit(self)
self.ipLineEdit.setGeometry(QtCore.QRect(30, 30, 241, 31))
self.ipLineEdit.setObjectName("ipLineEdit")
self.ipLineEdit.setPlaceholderText("Enter Server IP")

# Utilisation d'un layout vertical
layout = QtWidgets.QVBoxLayout(self)

# Ajout des widgets au layout
layout.addWidget(self.label)
layout.addWidget(self.label_2)
layout.addWidget(self.ipLineEdit)
layout.addWidget(self.lineEdit)
layout.addWidget(self.pushButton)
layout.addWidget(self.inscbutton)
```

```
self.send_message = send_message_func

self.setWindowTitle("Form")
self.pushButton.setText("Connexion")
self.label.setText("Connexion au serveur")
self.label_2.setText("Entre votre pseudo")
self.inscbutton.setText("Inscription")
```

3. Fenêtre de chat

La class Chat permet de gérer a l'aide de PYQT l'ouverture de la fenêtre permettant d'envoyer des message d'un utilisateur a un autre.

a. sending_text

Cette fonction permet de gérer l'envoie de message après avoir appuie sur le bouton « Envoyer ».

```
def sending_text(self):
    text = self.line.text()
    self.message_signal.emit(text)
    self.send_message(text)
```

b. disconnect

Cette fonction envoie un signal lorsque l'utilisateur appuie sur le bouton déconnexion,

```
def disconnect(self):
    self.disconnect_signal.emit()
```

c. Fonction Init

La fonction Init on initialise les fonctions permettant de crée l’application pour gérer l’envoie de message aux serveur.

On initialise les différentes partie de l’interface graphique en leur donnant un nom, une taille et et l’emplacement.

```
    def __init__(self, send_message_func):
        QtWidgets.QWidget.__init__(self)

        self.setObjectName("Form")
        self.resize(1128, 630)
        self.send_message = send_message_func

        self.pushButton = QtWidgets.QPushButton(self)
        self.pushButton.setGeometry(QtCore.QRect(840, 580, 81, 31))
        self.pushButton.setObjectName("pushButton")
        self.pushButton.setText("Envoyer")
        self.pushButton.clicked.connect(self.sending_text)

        self.line = QtWidgets.QLineEdit(self)
        self.line.setGeometry(QtCore.QRect(302, 580, 541, 31))
        self.line.setObjectName("lineEdit")

        self.textEdit = QtWidgets.QTextEdit(self)
        self.textEdit.setGeometry(QtCore.QRect(170, 10, 801, 551))
        self.textEdit.setObjectName("textEdit")

        self.disconnect_button = QtWidgets.QPushButton(self)
        self.disconnect_button.setGeometry(QtCore.QRect(10, 580, 81, 31))
        self.disconnect_button.setObjectName("disconnect_button")
        self.disconnect_button.setText("Déconnexion")
        self.disconnect_button.clicked.connect(self.disconnect)
```

```
        # Utilisation d'un layout vertical
        layout = QtWidgets.QVBoxLayout(self)

        # Ajout des widgets au layout
        layout.addWidget(self.textEdit)
        #layout.addWidget(self.channel_combobox)
        layout.addWidget(self.line)
        layout.addWidget(self.pushButton)
        layout.addWidget(self.disconnect_button)

        self.setWindowTitle("CHAT SAE")
```

4. Fenêtre de d’inscription

La class Inscription permet de gérer a l’aide de PYQT l’ouverture de la fenêtre permettant l’inscription des utilisateur.

a. sending_inscription

Cette fonction vas récupérer les ligne dans laquelle l’utilisateur rentre sont login et la ligne dans laquelle l’utilisateur rentre l’adresse IP du serveur ciblé.

La fonction vas par après envoyer un signal contenant le username et l’adresse IP.

```
class Inscription(QtWidgets.QWidget):
    inscription_signal = QtCore.pyqtSignal(str, str)

    def sending_inscription(self):
        username = self.lineEdit.text()
        ip_address = self.ipLineEdit.text()

        registration_data = f"inscription {username}"
        print(f"envoyer {registration_data}")
        self.inscription_signal.emit(registration_data, ip_address)
        self.send_message(registration_data)
        self.close()
```

b. send_inscription_data

Cette fonction appelle la fonction connect_to_server en renseignant les argument username et ip_address.

```
    def send_inscription_data(self, username, ip_address):
        self.connect_to_server(username, ip_address)
```

c. Fonction Init

La fonction Init on initialise les fonctions permettant de crée l'application pour gérer l'envoie de message aux serveur.

On initialise les différentes partie de l'interface graphique en leur donnant un nom, une taille et et l'emplacement.

```
    def __init__(self, send_message_func):
        QtWidgets.QWidget.__init__(self)

        self.send_message = send_message_func
        self.setObjectName("log")
        self.resize(428, 167)

        self.lineEdit = QtWidgets.QLineEdit(self)
        self.lineEdit.setGeometry(QtCore.QRect(30, 110, 241, 31))
        self.lineEdit.setObjectName("lineEdit")

        self.ipLineEdit = QtWidgets.QLineEdit(self)
        self.ipLineEdit.setGeometry(QtCore.QRect(30, 30, 241, 31))
        self.ipLineEdit.setObjectName("ipLineEdit")
        self.ipLineEdit.setPlaceholderText("Enter Server IP")

        self.pushButton = QtWidgets.QPushButton(self)
        self.pushButton.setGeometry(QtCore.QRect(280, 110, 75, 31))
        self.pushButton.setObjectName("pushButton")
        self.pushButton.clicked.connect(self.sending_inscription)
```

```
        self.label = QtWidgets.QLabel(self)
        self.label.setGeometry(QtCore.QRect(150, 10, 121, 16))
        self.label.setObjectName("label")

        self.label_2 = QtWidgets.QLabel(self)
        self.label_2.setGeometry(QtCore.QRect(30, 90, 121, 16))
        self.label_2.setObjectName("label_2")

        layout = QtWidgets.QVBoxLayout(self)

        # Ajout des widgets au layout
        layout.addWidget(self.label)
        layout.addWidget(self.label_2)
        layout.addWidget(self.ipLineEdit)
        layout.addWidget(self.lineEdit)
        layout.addWidget(self.pushButton)

        self.send_message = send_message_func

        self.setWindowTitle("Form")
        self.pushButton.setText("Inscription")
        self.label.setText("Inscription au serveur")
        self.label_2.setText("Entrez votre pseudo")
```

5. Class ChatApplication

La class ChatApplication permet la gestion des différentes fenetre ainsi que de la connexion, la déconnexion, la réception et l'envoie de message.

Dans un un premier temps la fonction showfen1 va être appeler afin d'ouvrir la fenêtre de connexion.

```
class ChatApplication(QtWidgets.QWidget):
    def __init__(self):
        super().__init__()

        self.showfen1()
```

a. Connect_to_server

Cette fonction prend deux arguments, TEXT et ip_address. Elle commence par vérifier si une adresse IP a été fournie. Si c'est le cas, un socket client est créé, puis il se connecte au serveur de chat en utilisant l'adresse et le port 12345.

Si le début de TEXT contient "inscription", alors la variable TEXT est envoyée directement au serveur. Si ce n'est pas le cas, la fonction appelle la fonction showfen2.

```
def connect_to_server(self, TEXT, ip_address):
    if ip_address:
        print(ip_address)

        self.client_socket = socket.socket()
        try:
            self.client_socket.connect((ip_address, 12345))
            if TEXT[:11] == "inscription":
                self.send_message(TEXT)

            self.showfen2(TEXT)
        except Exception as e:
            print(f"Error connecting to the server: {e}")
```

b. Showfen1

La fonction showfen1 permet d'ouvrir la fenêtre graphique de la class Login et d'utiliser la fonction send_message avec cette dernière ainsi que la fonction connect_to_server.

```
def showfen1(self):
    self.fen1 = Login(send_message_func=self.send_message)
    self.fen1.switch_window.connect(self.connect_to_server)
    self.fen1.show()
```

c. Showinsc

Le fonction showinsc permet d'ouvrir la fenêtre graphique correspondant a la class Inscription et d'utilisé la fonction send_message avec cette dernière ainsi que la fonction connect_to_server.

```
def showinsc(self):
    self.fenisnc = Inscription(send_message_func=self.send_message)
    self.fenisnc.inscription_signal.connect(self.connect_to_server)
    self.fenisnc.show()
```

d. Showfen2

La fonction showfen2 vas d'abord fermer les autre fenêtre graphique, puis as ensuite ouvrir la fenêtre graphique correspondant la class Chat.

La fonction showfen2 vas aussi crée un thread gérant l'envoie de message.

```
def showfen2(self, username):
    self.fen1.close()
    self.fenisnc.close()

    self.fen2 = Chat(send_message_func=self.send_message)
    self.fen2.message_signal.connect(self.receive_message)
    self.fen2.disconnect_signal.connect(self.disconnect_client)

    self.send_thread = threading.Thread(target=self.receive_messages)
    self.send_thread.start()

    self.fen2.show()
```

e. receive_messages

La fonction Receive_messages permet de gérer la réception de message dans un boucle infinie , c'est dernier vont ensuite être afficher dans le chat de la fenêtre graphique de chat.

```
def receive_messages(self):
    while True:
        response = self.client_socket.recv(100)
        print(f"\n{response.decode()}")
        self.fen2.textEdit.append(response.decode())
        if response.lower() == 'bye':
            break
```

f. message

Cette fonction a pour argument message. Elle permet de prendre le message que l'utilisateur envoie et de l'afficher dans la fenêtre de chat globale. Ensuite, la ligne dans laquelle l'utilisateur a saisi le message est effacée pour faciliter la saisie ultérieure.

```
def message(self, message):
    self.fen2.textEdit.append(f"{message}")
    self.fen2.line.clear()
```

g. send_message

Cette fonction a pour argument message. Elle permet d'envoyer les messages via le socket client.

```
def send_message(self, message):
    self.client_socket.send(message.encode())
```

h. disconnect_client

Cette fonction va permettre une déconnexion total du client.

Dans un premier temps la fonction va fermé la fenêtre client et arrêter l'application de PYQT, par la suite elle vas arrêter le thread d'envois de message.

```
def disconnect_client(self):
    # Ferme la fenêtre de chat
    self.fen2.close()

    # Termine le programme
    QtWidgets.QApplication.quit()
    reponses = "bye"

    # Arrête le thread
    self.send_thread.quit()
    self.send_thread.wait()
```

i. Fonction main

La fonction main appelle la class ChatApplication, elle effectue aussi l'ouverture de la fenêtre de connexion et d'inscription en appelant les fonction showfen1 et showinsc

```
def main():
    app = QtWidgets.QApplication(sys.argv)
    chat_app = ChatApplication()
    chat_app.showfen1()
    chat_app.showinsc()
    sys.exit(app.exec())
```