Whiskr

WHISKR
DOCUMENTATION DEVELOPPEUR

#SAE302 #Security #Network

https://github.com/Ykari68/SAE302/

Cette fiche technique a pour but de présenter le fonctionnement technique du produit Whiskr.

Table des matières

Table des matières 2
I. Serveur 3
Le serveur est la passerelle entre chaque client qui souhaite communiquer entre eux. Il va gérer toutes les connexions et assurer un lien sécurisée entre les clients. 3
1/ Threads 3
a) Thread de communication 3
b) Thread console 5
2/ Hébergement des clients 5
3/ Connexion à la base de données 7
II. Client 8
1/ Classe Login 8
2/ Classe Compte 9
3/ Classe Main 9
4/ Classe Canal 10
5/ Classe SocketThread 11

I. Serveur

Le serveur est la passerelle entre chaque client qui souhaite communiquer entre eux. Il va gérer toutes les connexions et assurer un lien sécurisé entre les clients.

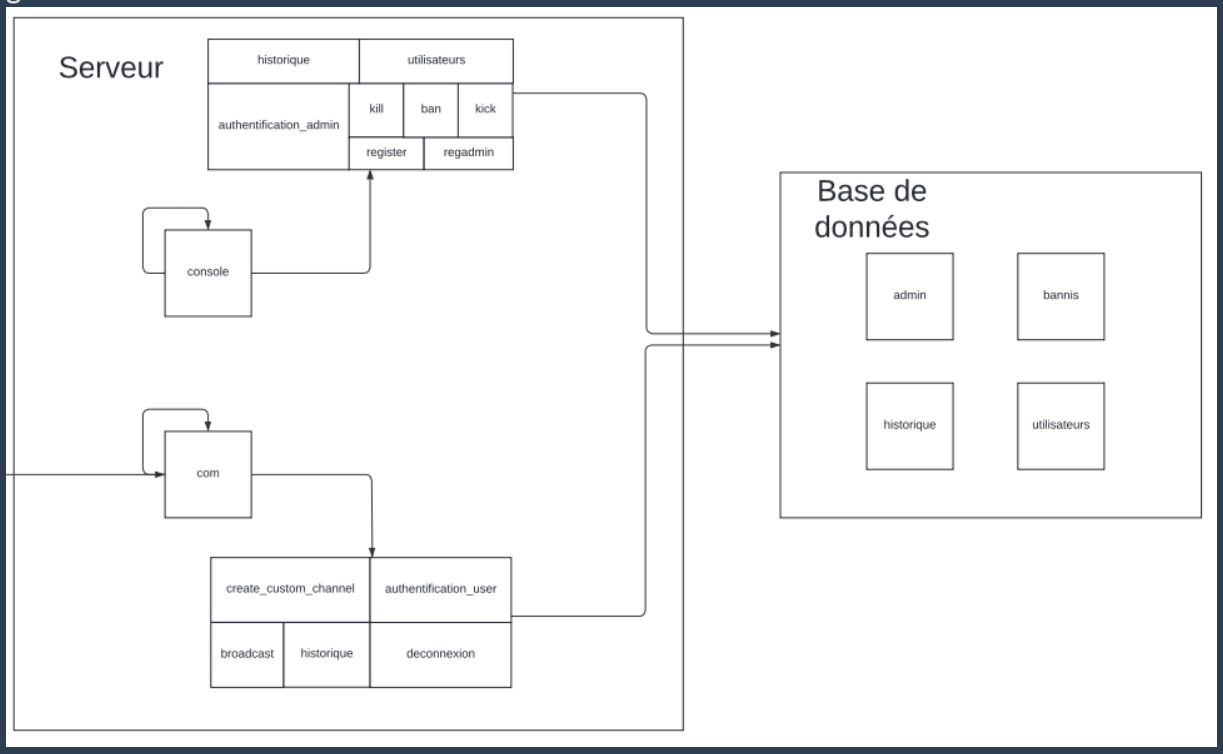

Schéma explicatif du fonctionnement du serveur

1/ Threads

Le serveur dispose de deux threads, qui tournent constamment dans le programme. Il y a un thread de communication, qui gère les clients connectés. Ce thread se lance dès l'arrivé d'un client dans le serveur. Enfin, il y a un thread pour la console. Il permet à l'administrateur du serveur d'envoyer des commandes et de gérer le comportement du serveur manuellement.

a) Thread de communication

Le thread de communication (com) se lance pour chaque arrivé d'un client dans le serveur. Cela veut dire que chaque client créé un thread de communication pour gérer sa connexion.

La première étape que cette Thread exécute est l'authentification de l'utilisateur. S'il n'est pas authentifié, le thread se fermera et terminera la connexion entre le client et le serveur.

```
if authentification_user(username, password) == True:
        historique(conn)
        conn.send("Authentification réussie.".encode())
```

```
else:
    conn.send("Authentication non réussie.".encode())
    deconnexion(username, conn, clients)
```

Ensuite, on créer une boucle qui attend constamment l'arrivé d'un message de la part du client avec qui on a établi une connexion.

```
while True:
    """
    La boucle ici sert à perpétuellement recevoir des messages de la part de n'importe quel client.
    """
    try:
        #On reçoit les messages de la part de l'utilisateur.
        message = conn.recv(1024).decode()
```

Ce message peut être un message lambda, auquel cas on appel la fonction broadcast qui saura envoyer ce message à tous les clients connectés, à l'exception de l'envoyeur.

```
broadcast(username, message, clients)
```

```
#Les deux prochaines fonctions servent à gérer les clients.
def broadcast(sender, message, clients):
    """
    Cette fonction permet d'envoyer un message à tous les utilisateurs connectés, sauf l'utilisateu
    """
    for client_username, client_conn in clients.items():
        """
        On créer une boucle afin d'obtenir tous les clients connectés lors de l'envoie du message.
        """
        if client_username != sender:
            """
            Si le client n'est pas l'envoyeur, on envoie le message.
            """
            try:
                client_conn.send(f"{sender}: {message}".encode())
                """
                Et on envoie.
                """
                enregistrer_message(sender, message)
            except:
                deconnexion(client_username, client_conn, clients)
```

Le thread sait également gérer des messages codés.
En effet, il se peut que le client ait des demandes spécifiques, tel que la liste des utilisateurs connectés.

Lorsqu'un code est reconnu, au lieu d'envoyer le message du client aux autres, on envoie ce qui a été spécifiquement demandé.

Exemple d'une demande d'envoi des utilisateurs connectés au client :

```
if message == "A6rafZ5qz66SS0wTHgu3MKQJuJfbzCdu":
    #On génère la liste des utilisateurs connectés.
    utilisateurs_connectes = ", ".join(clients.keys())
    #On concatène cette liste dans un message.
    message = f"{utilisateurs_connectes}"
    #On envoie ce message avec un code spécifique afin que le client sac
    conn.send(("9AlaoKX1XBgF4PpOouj5M7ULgShcN0HF " + message).encode())
```

b) Thread console

Le second thread se lance qu'une seule fois lors du démarrage du serveur. Il se ferme que lorsque le serveur s'éteint.

Il consiste également en une boucle, qui demande d'abord les identifiants et mot de passe administrateur avant de démarrer la console.

Il a été intentionnellement fait de sorte que la vérification d'identité soit mise à l'intérieur de la boucle afin que l'administrateur soit régulièrement déconnecté. Cela est plus sécurisé.

De la même façon que le thread « com », le thread « console » attend perpétuellement un message, mais celui est interne. Lorsqu'un message est tapé, alors on vérifie la commande et on agit en fonction de cela.

```
while True:
    """
    Je créer une boucle afin de toujours po
    """
1   if authentification_admin() == True:
        print("Authentication réussie.")
2       commande = input("Serveur> ")
        #La commande kill compte 3 secondes
        if commande == "kill": 3
            kill(clients)
```

Chaque commande fait appel à une fonction qui a été conçue pour l'interface administrative.

2/ Hébergement des clients

Afin de pouvoir constamment démarrer un thread de communication, il nous faut une boucle qui reçoit les clients et qui créer le thread.

On attend donc l'arrivé un utilisateur, vérifie s'il ne demande pas de créer un nouveau compte, dans ce cas on lui créer un compte, puis on vérifie s'il n'est pas déjà connecté et s'il n'est pas banni. Ensuite, on peut enfin démarrer le thread.

```python
#Ici on créé une boucle pour constamment accepté les nouvelles connexions.
while True:
    try:
        conn, address = server_socket.accept()
        username = conn.recv(1024).decode()
        #Si le premier message envoyé commence par ce code, alors le client demande à créer un nouveau compte.
        if username.startswith("FWwCXNb9u3l0E1Ej3OqsRBGlR9zkyHIv"):
                parts = username.split()
                if len(parts) >= 3:
                    username = parts[1]
                    password = parts[2]
                    if register(username, password, conn_db) == True:
                        conn.send("Création de compte réussie!".encode())
                        conn.close()
                        continue
                    else:
                        conn.send("Compte existant!".encode())
                        continue
```

```python
                    continue
    password = conn.recv(1024).decode()
    """
    On vérifie si le nouvel utilisateur est banni ou non.
    """
    if username not in blacklist:
        """
        On vérifie si le nouvel utilisateur n'est pas banni
        """
        if username in clients:
            """
            On vérifie si le nouvel utilisateur n'est pas déjà connecté. (Utilise un pseudo déjà en cours d'utilisation)
            """
            print(f"Le username {username} est déjà connecté. Refuser la connexion.")
            conn.close()
            continue
        clients[username] = conn
        print(f"Client connecté: {username}")
        #Lancement de la fonction com pour gérer la nouvelle connexion.
        com_thread = threading.Thread(target=com, args=(conn, clients, username))
        com_thread.start()
    else:
        print(f"Le username {username} est blacklisté. Refuser la connexion.")
        conn.close()
except Exception as e:
    print(f"Fin. {e}")
    break
```

3/ Connexion à la base de données

Le serveur se connecte à une base de données afin de pouvoir notamment authentifier ses utilisateurs. Il s'y connecte localement, avec le compte root.

```
#Cette partie gère la connexion à la base de données.
try:
    conn_db = mysql.connector.connect(
        host='127.0.0.1',
        user='root',
        password='toto',
        database='serveur'
    )

    if conn_db.is_connected():
        print('Connecté à la base de données MySQL')

except Exception as e:
    print(f"Erreur de connexion à la base de données: {e}")
    print("Fermeture du serveur...")
    sys.exit()
cursor = conn_db.cursor()
```

Une fois connecté à la base de données, on peut la gérer depuis le serveur.

Exemple de la fonction enregistrer message, qui ajout à la table historique chaque message envoyé :

```
def enregistrer_message(sender, message):
    """
    Cette fonction enregistre chaque message envoyé dans la base de donner. Cela permettra ensuite d'a
    """
    try:
        cursor.execute('INSERT INTO historique (sender, message) VALUES (%s, %s)', (sender, message))
        conn_db.commit()
    except Exception as e:
        print(f"Erreur lors de l'enregistrement du message dans la base de données: {e}")
```

II. Client

Le client gère les utilisateurs directement. Il récupère les messages à envoyés et les envoie au serveur, auquel il s'est connecté avec un identifiant et un mot de passe renseigné par l'utilisateur.

Le code fonction avec des classes qui représentent chacune une fenêtre (à l'exception de la classe SocketThread). Chaque classe appelle à une autre classe en cascade.

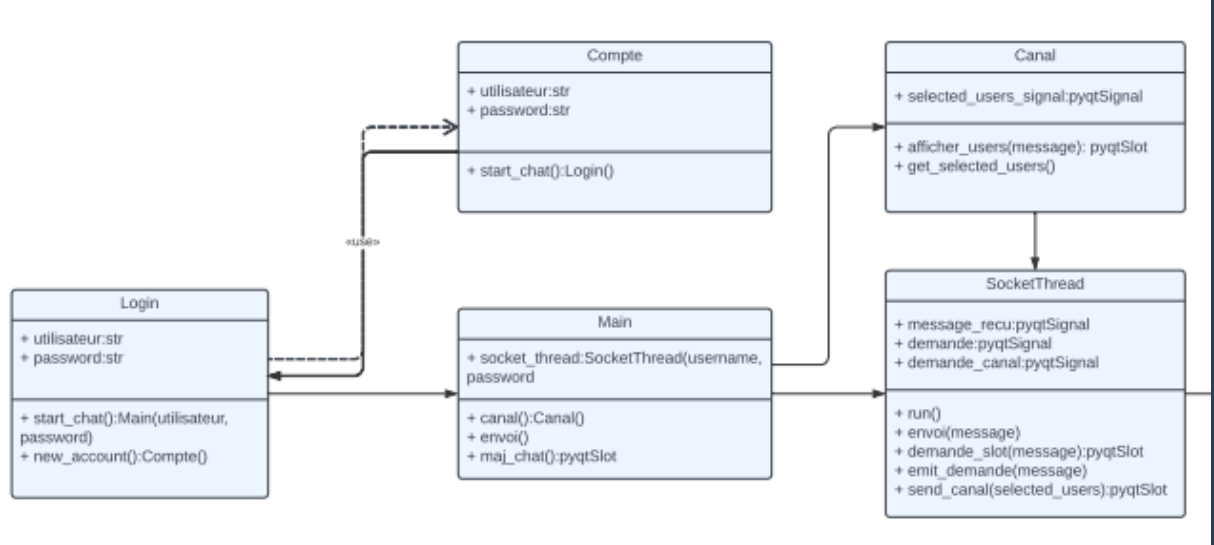

Schéma de la structure du code client.

1/ Classe Login

La classe Login représente la fenêtre d'authentification de l'application. Elle est la première classe à être sollicitée lors du lancement du programme.

Elle demande sous forme graphique un nom d'utilisateur et un mot de passe. Ces données peuvent être envoyé au serveur sans message codé, auquel cas le serveur tentera d'authentifier le client avec ces données. Il y a un bouton de création de compte, qui chargera la page de création de compte, de la classe Compte.

Lorsque les identifiants ont été rentré par l'utilisateur, la classe Login ouvrira la page Main avec ces identifiants. Alors, la classe Main pourra se charger d'envoyer la demande d'authentification au serveur.

2/ Classe Compte

La classe compte est appelé par la classe Login et ouvre une fenêtre permettant de créer un nouveau Compte. De la même façon que Login, la classe Compte va demander à l'utilisateur sous forme graphique un identifiant et un mot de passe.

La différence est que ces données seront envoyées au serveur avec un code spécifique au début du message afin que le serveur puisse le reconnaître et créer le compte avec ces identifiants plutôt que d'essayer de les authentifier.

```
code = (f"FWwCXNb9u3l0E1Ej3OqsRBGlR9zkyHIv {username} {password}")
```

3/ Classe Main

La classe Main est la classe principale de l'application. Elle ouvre la fenêtre de chat, l'essence même du produit.

Elle est appelée par la classe Login avec un nom d'utilisateur et un mot de passe. Avec ces données, elle va appeler la classe SocketThread (Classe responsable de la communication directe vers le serveur) avec les données comme attribut.

```
self.socket_thread = SocketThread(username, password)
```

La classe main a deux fonctions principales.

La première est la fonction « envoi », qui récupère le message écrit par l'utilisateur via l'interface graphique de la classe, et l'envoi à la classe SocketThread pour qu'il puisse l'envoyer au serveur.

```
def envoi(self):
    '''
    **py:func::envoi
    '
    Cette méthode est appelée lorsque le
    '''
    message = self.message_envoi.text()
    self.socket_thread.envoi(message)
    self.message_envoi.clear()
```

Ensuite, la seconde fonction est « maj_chat », pour mise à jour du chat. Cette fonction reçoit des messages venant de SocketThread, qui les as reçus du serveur. Ces messages sont affichés les uns après les autres.

```python
@pyqtSlot(str)
def maj_chat(self, message):
    '''
    **py:func::maj_chat
    '
    La méthode maj_chat récupère les messages reçu par SocketThrea
    '''
    current_text = self.chat_history.toPlainText()
    self.chat_history.setPlainText(current_text + '\n' + message)

    cursor = self.chat_history.textCursor()
    cursor.movePosition(QTextCursor.MoveOperation.End)
    self.chat_history.setTextCursor(cursor)
```

Une possibilité de création de canal personnalisé a été implémenté. La classe Main peut ouvrir la fenêtre de création de canal en appelant la classe Canal.

4/ Classe Canal

La classe canal est appelé par la classe Main lorsque l'utilisateur demande à créer un nouveau canal.

Cette classe va d'abord envoyer un message à SocketThread pour qu'il puisse demander au serveur une liste des utilisateurs connectés.

Une fois la liste récupérée, il va les afficher sous forme de checkboxe, permettant à l'utilisateur de choisir quels autres clients il souhaitera intégrer dans son canal.

Lorsque l'utilisateur aura validé son choix, la classe Canal enverra une liste d'utilisateur à la classe SocketThread pour qu'il l'envoie au serveur avec un code spécifique. Le serveur saura créé le canal personnalisé avec la liste d'utilisateur.

```python
#Ce slot attend une donnée de SocketThread, cette donnée est la liste d
@pyqtSlot(str)
def afficher_users(self, message):
    '''
    **py:func::afficher_users
    '
    La méthode affiche les utilisateurs connectés sous forme de checkbo
    '''
    #On divise la lise pour individualiser chaque utilisateur.
    self.users = message.split(',')
    #On vide les checkboxes (S'il y en a).
    for i in reversed(range(self.users_container.layout().count())):
        item = self.users_container.layout().itemAt(i)
        widget = item.widget()
        if widget:
            widget.setParent(None)
    #Pour chaque utilisateur trouvé dans la liste, on lui affecte une c
    for row, user in enumerate(self.users):
        checkbox = QCheckBox(user)
        self.users_container.layout().addWidget(checkbox, row, 0)

    self.adjustSize()
```

```
def get_selected_users(self):
    '''
    py:func::get_selected_users
    '
    Cette méthode est appelé après que le bouton "Ok" soit sollicité. Elle récupère les utilisateurs séléctionés et les envoie a
    '''
    selected_users = list(checkbox.text() for checkbox in self.users_container.findChildren(QCheckBox) if checkbox.isChecked())
    self.selected_users_signal.emit(','.join(selected_users))
```

5/ Classe SocketThread

La classe SocketThread est la seule classe du code qui ne dispose pas d'interface graphique.

Assurément, cette classe n'a pas besoin de l'interaction de l'utilisateur, elle gère les demander des autres classes et assure la communication entre le client et le serveur.

Il enverra tous les messages que les autres classes essaient d'envoyer, et recevra tous les messages venant du serveur.

```
def __init__(self, username, password):
    super().__init__()
    #Connextion au serveur, syntaxe vu dans la classe "Compte".
    self.server_address = address
    self.server_port = port

    self.client_socket = socket.socket()
    self.client_socket.connect((self.server_address, self.server_port))
    #Ayant récupérer les identifiants et mot de passe de la classe Main,
    self.client_socket.send(username.encode())
    time.sleep(3)
    self.client_socket.send(password.encode())

    self.demande.connect(self.demande_slot)
```

Code du SocketThread pour se connecter au serveur et envoyer les identifiants.

```
while True:
    try:
        #On reçoit le message du serveur.
        message = self.client_socket.recv(1024).decode()
        #Si ce message est ce code précisément, on ferme le client. I
        if message == "q5rN0rt81mwgr87FzuCv7QSdZTyb1mLt":
            self.client_socket.close()
            self.close()
        #Si ce message commence avec ce code, alors on envoie la suit
        elif message.startswith("9AlaoKX1XBgF4PpOouj5M7ULgShcN0HF"):
            self.demande.emit(message[32:])
        self.message_recu.emit(message)
    except (ConnectionResetError, ConnectionAbortedError, OSError):
        print("Connection error or client exit.")
        self.client_socket.close()
        break
```

Cette boucle reçoit les messages du serveur et les envois à la fonction de Main qui affiche les messages.