

# SAE 3.02 Développer des applications communicantes

DOCUMENTATION SERVEUR

SAMUEL ROLLI RT222

## SOMMAIRE

I.	Introduction .....	3
II.	Utilisation .....	3
III.	Explication du code .....	4
IV.	Limitation & Réflexion .....	14
1)	Sécurité .....	14
2)	Limitation .....	14

Figure 1: exemple de tag "#MODIFY4PROD" .....	3
Figure 2: authentification sur le serveur .....	3
Figure 3: administrateur connecté.....	4
Figure 4: liste des commandes disponibles .....	4
Figure 5: démarrage du serveur.....	4
Figure 6: fonction Connect.....	5
Figure 7: Fonction d'identification (partie Log-in).....	6
Figure 8: Fonction d'identification (partie sign-in).....	7
Figure 9: fonction réception de messages .....	8
Figure 10: fonction de récupération des channels.....	9
Figure 11: fonction de récupération de l'historique .....	10
Figure 12: fonction de vérification d'accès à un channel.....	11
Figure 13: fonction d'entrée de commande administrateurs .....	12
Figure 14: fonction d'entrée de commande, partie BAN .....	12
Figure 16: fonction d'entrée de commande, partie liste user.....	13
Figure 17: fonction de validation d'arrêt.....	13
Figure 18: développement en cours du traitement de la décision utilisateur .....	14

## I. Introduction

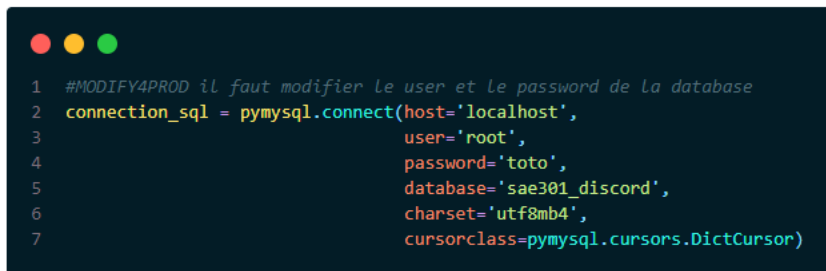
Ce document vise à vous présenter toutes les fonctions du serveur que j'ai dû créer pour la SAE 3.02 : Développer des applications communicantes.

Tout d'abord, nous allons voir comment utiliser le serveur et comment le lancer correctement. Puis, nous expliquerons le code en profondeur.

Enfin, je vous présenterai une courte réflexion sur les limitations de ce serveur et sur les options qui n'ont pas pu être mises en place.

## II. Utilisation

Avant même de démarrer, il est important de vérifier les paramètres à modifier, c'est-à-dire les informations de la base de données (présentes à trois endroits dans le code) et identifiées par le tag "#MODIFY4PROD" (= "modifier pour la version production"), ainsi que le port d'accès pour le socket.



```
1 #MODIFY4PROD il faut modifier le user et le password de la database
2 connection_sql = pymysql.connect(host='localhost',
3                                 user='root',
4                                 password='toto',
5                                 database='sae301_discord',
6                                 charset='utf8mb4',
7                                 cursorclass=pymysql.cursors.DictCursor)
```

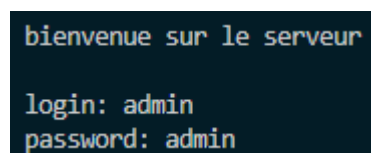
Figure 1: exemple de tag "#MODIFY4PROD"

Il est aussi nécessaire d'installer les modules pip utilisés par le serveur, ici pymysql. Pour ce faire, et pour être sûr que la bonne version des paquets est installée, il est possible d'utiliser la commande "*pip install -r requirement.txt*". "*requirement.txt*" est le fichier texte présent dans le dossier contenant les fichiers client et serveur.

Enfin, il faut installer la base de données : pour ce faire, nous avons deux possibilités, soit il faut utiliser le dump de la base SQL fournis, soit créer la base de données à la main et le code se charge de créer les tables correspondantes à la première exécution de celui-ci (cf. figure 5).

Une fois que toutes ces actions ont été effectuées, le serveur peut être lancé.

Nous serons accueillis par un message nous demandant de nous connecter en tant qu'administrateur sur le serveur (utilisateur par défaut : admin, mot de passe par défaut : admin)



```
bienvenue sur le serveur
login: admin
password: admin
```

Figure 2: authentification sur le serveur

Une fois que nous sommes bien authentifiés, un message de bienvenue ainsi qu'un message d'information sur les différentes commandes disponibles apparaît.



Figure 3: administrateur connecté

Il est possible à tout moment de lister les différentes commandes, en utilisant "?" ou "help"

```
Les commandes disponibles sont :
- /arret qui arrettera le serveur
- /liste user qui listera les pseudo utilisateur
- /ban nom_d'utilisateur temps(en minute) qui ban l'utiliasteur sur la periode de temps définie
- /liste demande qui listera les demandes pour les channels
```

Figure 4: liste des commandes disponibles

Nous pouvons voir que les commandes disponibles sont : "/arret", "/liste user", "/ban" et "/liste demande" et leurs descriptions sont données.

### III. Explication du code

Pour cette partie, j'ai décidé d'expliquer le code dans la chronologie de son exécution. C'est pourquoi, je commence par expliquer la fin.

```
1 if __name__ == '__main__':
2     admin_connected = False
3     while not admin_connected:
4         print("\nbienvvenue sur le serveur\n")
5         admin = input('login: ')
6         pwd = input('password: ')
7         if admin == "admin" and pwd == "admin":
8             admin_connected = True
9             print(f"\n\nhello_admin\n\n")
10            print("press "?" or "help" to list all the command available")
11
12    with connection_sql:
13        with connection_sql.cursor() as cursor:
14            sql = "CREATE TABLE IF NOT EXISTS users(id INT UNSIGNED PRIMARY KEY AUTO_INCREMENT, user varchar(40), password varchar(40))"
15            cursor.execute(sql)
16            sql = "CREATE TABLE IF NOT EXISTS messages(id INT UNSIGNED PRIMARY KEY AUTO_INCREMENT, sender varchar(40), recever varchar(40), message varchar(100) NOT NULL)"
17            cursor.execute(sql)
18            sql = "CREATE TABLE IF NOT EXISTS access (id INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,user VARCHAR(40), channel VARCHAR(40), etat TINYINT(1))"
19            cursor.execute(sql)
20            sql = "CREATE TABLE IF NOT EXISTS ban (id INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,user VARCHAR(40) NOT NULL, start_ban DATETIME NOT NULL, duration INT NOT NULL)"
21            cursor.execute(sql)
22
23    connection_sql.commit()
24    server_socket.listen(5)
25    command_entry_tread = threading.Thread(target=command_entry)
26
27    command_entry_tread.start()
28    while not server_stop:
29        client_socket, address = server_socket.accept()
30        if not server_stop:
31            connection = threading.Thread(target=connect, args=[client_socket])
32            connection.start()
33            at_least_one_client = True
34
35    if at_least_one_client:
36        connection.join()
37    server_socket.close()
```

Figure 5: démarrage du serveur

Commençons par le démarrage du serveur. Nous pouvons observer à la ligne 4 le début de la séquence d'identification de l'administrateur.

Ensuite, à partir de la ligne 12, c'est l'initialisation de la base SQL qui est faite avec les commandes "CREATE TABLE IF NOT EXISTS". Ce qui va créer les tables si elles n'existent pas mais ne va rien faire dans le cas contraire. J'ai préféré inclure ces lignes dans le cas où le serveur SQL aurait eu un problème ou une corruption de données. Cela nous permettrait de quand même utiliser le serveur, de plus dans le cas où le Dump de la base aurait un problème, toutes les tables seraient créées correctement.

Puis les lignes 22 à 24, servent à la création et à l'exécution du Thread qui est chargé d'interpréter et d'exécuter les commandes données par l'administrateur sur le serveur (cf. figure 13)

Par la suite, à la ligne 25 commence la boucle d'acceptation des clients. Tant que le flag "server\_stop" n'est pas activé, elle ne s'arrêtera pas. Elle se charge de créer un thread d'identification par client et de les lancer.

Enfin, les trois dernières lignes servent à arrêter le serveur correctement. Elles vérifient qu'il y a au moins un client connecté, et donc au moins un thread de connexion ouverte, pour attendre la fin de l'exécution de celle-ci. Puis elles ferment le socket ouvert.

The image shows a code editor with a dark background and light-colored text. At the top left, there are three colored circles (red, yellow, green). The code is a Python function named 'connect' that takes a 'client' argument. It uses global variables 'server\_stop', 'clients', 'connection\_sql', and 'connected\_pseudo'. The function starts with a 'try' block. Inside, it connects to a MySQL database using 'pymysql.connect'. Then, it creates a thread 'identification\_thread' with 'target=id\_process' and 'args=[client, connection\_sql]', starts it, and joins it. Next, it uses a 'with lock:' block to append the client to the 'clients' list. Then, it creates a thread 'reception' with 'target=get' and 'args=[client, connection\_sql]', starts it. A 'while not server\_stop:' loop follows. Inside the loop, it checks if 'server\_stop' is True. If so, it iterates over 'clients' and sends ':server\_stop'.encode()' to each client, then joins the 'reception' thread. After the loop, there's an 'except pymysql.Error as e:' block that prints the error. Finally, a 'finally:' block uses a 'with lock:' block to iterate over 'clients' and close each client, then calls 'arret\_server()'. The line numbers 1 through 29 are visible on the left side of the code block.

```
1 def connect(client):
2     global server_stop
3     global clients
4     global connection_sql
5     global connected_pseudo
6     try:
7         with pymysql.connect(host='localhost', user='root', password='toto', database='sae301_discord'...) as connection_sql:
8             identification_thread = threading.Thread(target=id_process, args=[client, connection_sql])
9             identification_thread.start()
10            identification_thread.join()
11
12            with lock:
13                clients.append(client)
14
15            reception = threading.Thread(target=get, args=[client, connection_sql])
16            reception.start()
17
18            while not server_stop:
19                if server_stop:
20                    for client in clients:
21                        client.send(":server_stop".encode())
22            reception.join()
23        except pymysql.Error as e:
24            print(f"MySQL Error: {e}")
25        finally:
26            with lock:
27                for client in clients:
28                    client.close()
29            arret_server()
```

Figure 6: fonction Connect

La fonction "connect" est appelée directement à l'acceptation d'un client, c'est elle qui va se charger d'identifier les clients, de créer et d'exécuter les différents threads utilisés au cours de l'exécution du programme. Elle ne gère pas directement les messages, c'est plutôt un thread de gestion.

Elle va notamment démarrer le thread d'identification, pour permettre au client de se connecter avec un compte et donc d'avoir la possibilité d'avoir un suivi d'historique.

Puis, une fois que le client est passé par la phase d'identification, elle va l'ajouter à une liste de clients, qui n'est pas la liste de tous les clients mais plutôt la liste des clients connectés et identifiés à un instant T. Elle se charge aussi de démarrer le thread de réception de ce client.

Toute la suite de cette fonction a pour but de gérer l'arrêt volontaire ou involontaire du serveur.

```

1  def id_process(client, connection_sql):
2      global server_stop
3      global pseudo
4      global clients
5      global connected_pseudo
6      logged_in = False
7      while not logged_in:
8          if not server_stop:
9              try:
10                 message = client.recv(1024).decode()
11                 print(f"message reçu {message}")
12             except ConnectionResetError:
13                 print("Client forcibly disconnected")
14                 break
15             except ConnectionAbortedError:
16                 print("Client forcibly disconnected")
17                 break
18             except OSError:
19                 print("Client has probably disconnected")
20         else:
21             # LOGGING IN
22             if message[:1] == ":":
23                 message = message[1:]
24                 if message[:5] == "login":
25                     message = message[5:].split(':')
26                     try:
27                         with connection_sql.cursor() as cursor:
28                             sql = "SELECT `password` FROM `users` WHERE `user`=%s"
29                             cursor.execute(sql, (message[0],))
30                             result = cursor.fetchone()
31                     except pymysql.Error as e:
32                         print(f"MySQL Error: {e}")
33                     else:
34                         if message[0] not in connected_pseudo:
35                             if result and result['password'] == message[1]:
36                                 client.send('logged-in'.encode())
37                                 pseudo[client] = message[0]
38                                 print(f'bienvenue {message[0]}')
39                                 with lock:
40                                     connected_pseudo[pseudo[client]] = client
41                                 logged_in = True
42                             else:
43                                 print('Wrong credentials')
44                                 client.send("error".encode())
45                         else:
46                             print(f"{message[0]} is already connected. Please logout from your other devices.")
47                             client.send('already connected'.encode())
48

```

Figure 7: Fonction d'identification (partie Log-in)

La fonction d'identification gère, comme son nom l'indique, l'identification des différents clients. C'est-à-dire, qu'elle gère la connexion ou l'inscription des différents clients.

Elle va commencer par recevoir un message protocolaire envoyé par le client, reconnaissable par le fait qu'il commence par ":" (voir ligne 22 figure 7). Elle vérifie s'il s'agit d'une demande de connexion ou d'inscription. Ensuite, elle va traiter le message, c'est-à-dire récupérer le pseudo et le mot de passe pour les comparer à sa base de données. S'ils correspondent, elle va envoyer une confirmation au client, ajouter le client et le pseudo dans un dictionnaire et finir son exécution. Dans le cas contraire, ou si le l'utilisateur est déjà connecté, elle va lui envoyer les réponses appropriées, ici "error" si le mot de passe n'est pas bon et "already connected" si l'utilisateur est déjà connecté.

```
1  # SIGNING IN
2  elif message[:6] == "signin":
3      message = message[6:].split(':')
4      try:
5          with connection_sql.cursor() as cursor:
6              sql = "SELECT `password` FROM `users` WHERE `user`=%s"
7              cursor.execute(sql, (message[0],))
8              result = cursor.fetchone()
9      except pymysql.Error as e:
10         print(f"MySQL Error: {e}")
11     else:
12         if result:
13             client.send("already".encode())
14         else:
15             try:
16                 with connection_sql.cursor() as cursor:
17                     sql = "insert into users (user,password) values (%s,%s);"
18                     cursor.execute(sql, (message[0], message[1]))
19                     sql = "insert into access (user,channel, etat) values (%s,%s,%s)"
20                     cursor.execute(sql, (message[0], 'General', 1))
21                     connection_sql.commit()
22             except pymysql.Error as e:
23                 print(f"MySQL Error: {e}")
24             else:
25                 client.send('signed-in'.encode())
26                 print(f'bienvenue {message[0]}')
27                 pseudo[client] = message[0]
28                 logged_in = True
```

Figure 8: Fonction d'identification (partie sign-in)

La deuxième partie du thread d'identification, est l'inscription. Dans cette partie le serveur vérifie que le message protocolaire venant du client est bien "signin" puis découpe et traite le message reçu pour en extraire le pseudo et le mot de passe de l'utilisateur. Ensuite, elle vérifie qu'aucun utilisateur déjà inscrit n'a le pseudo, si c'est le cas elle va envoyer "already" et inviter l'utilisateur à changer de nom d'utilisateur. Dans le cas où le nom d'utilisateur est disponible, elle va ajouter l'utilisateur dans la base de données et lui donner l'accès au channel général. Enfin elle envoie au client une validation et ajoute le client et le pseudo dans un dictionnaire et finit son exécution.



```

1  def get(client, connection_sql):
2      global server_stop
3      global pseudo
4      global clients
5      global connected_pseudo
6      global requesting_channel
7      while not server_stop:
8          if client in clients:
9              if not requesting_channel:
10                 try:
11                     message = client.recv(1024).decode()
12                     print(message)
13                 except ConnectionResetError:
14                     print("Client forcibly disconnected")
15                     print(clients)
16                     clients.remove(client)
17                 except ConnectionAbortedError:
18                     print("Client forcibly disconnected")
19                     clients.remove(client)
20                 except OSError:
21                     print("Client has probably disconnected")
22                 else:
23                     if message[:1] == ":":
24                         message = message[1:]
25                         if message[:11] == "chanel_list":
26                             get_chanel(client, connection_sql, pseudo[client])
27                         elif message[:11] == "get_history":
28                             get_history(client, connection_sql, message[11:], pseudo[client])
29                         elif message[:12] == 'check_access':
30                             verify_access(client, connection_sql, message[12:])
31                         elif message == 'bye':
32                             print(f"{pseudo[client]} has disconnected")
33                             with lock:
34                                 clients.remove(client)
35                                 connected_pseudo.pop(pseudo[client])
36                                 del pseudo[client]
37                             client.send(":okbye".encode())
38                             client.close()
39                     else:
40                         message = message.split(":to:")
41                         with connection_sql.cursor() as cursor:
42                             sql = "insert into messages (sender,message,receiver) values (%s,%s,%s);"
43                             cursor.execute(sql, (pseudo[client], message[0], message[1]))
44                             connection_sql.commit()
45                         message_send = f"{pseudo[client]}:to:{message[1]}:to:{message[0]}"
46                         for client_receve in clients:
47                             if client_receve != client:
48                                 client_receve.send(message_send.encode())

```

Figure 9: fonction réception de messages

La fonction "get" a un fonctionnement très simple, elle attend qu'un message arrive, puis elle vérifie si c'est un message protocolaire (ligne 23 figure 9). Si c'est le cas, elle vérifie le type de message et appelle les fonctions correspondantes. Sinon, elle formate le contenu du message, l'ajoute dans la base de données, le reformate correctement et l'envoie en broadcast à tous les clients connectés.

```
1 def get_chanel(client, connection_sql, pseudo = None):
2     print("requisition channel")
3     try:
4         with connection_sql.cursor() as cursor:
5             sql = "SELECT `user` FROM `users`"
6             cursor.execute(sql)
7
8             result = cursor.fetchall()
9     except pymysql.Error as e:
10        print(f"MySQL Error: {e}")
11    else:
12        chanel_list = ""
13        for i in channels_are:
14            chanel_list = chanel_list + f",{i}"
15        for chanel in result:
16            if chanel.get('user') != pseudo:
17                chanel_list = chanel_list+f",{chanel.get('user')}"
18        print(chanel_list)
19        print("sending channels")
20        client.send(chanel_list.encode())
```

Figure 10: fonction de récupération des channels

Cette fonction a pour unique but de lister les différents utilisateurs et channels. Pour ce faire, elle fait une requête sur la base de données pour avoir tous les utilisateurs inscrits, puis elle y ajoute la liste des channels "*channels\_are*" déclarée en amont. En suite elle convertit cette liste en "*str*" car il n'est pas possible d'envoyer un objet de type "*list*" avec le module socket, et l'envoie au client.

```

1 def get_history(client, connection_sql, channel, pseudo):
2     try:
3         if channel not in channels_are:
4             with connection_sql.cursor() as cursor:
5                 sql = "SELECT `sender`,`message` FROM `messages` WHERE (`receiver`=%s and `sender`=%s) or (`receiver`=%s and `sender`=%s);"
6                 cursor.execute(sql, (channel, pseudo, pseudo, channel))
7                 result = cursor.fetchall()
8             else:
9                 with connection_sql.cursor() as cursor:
10                    sql = "SELECT `sender`,`message` FROM `messages` WHERE `receiver`=%s"
11                    cursor.execute(sql, channel)
12                    result = cursor.fetchall()
13        except pymysql.Error as e:
14            print(f"MySQL Error: {e}")
15        else:
16            client.send(":start_history".encode())
17            time.sleep(0.1)
18            if result:
19                for i in result:
20                    message = i.get('sender')+":said:"+i.get('message')
21                    client.send(message.encode())
22                    time.sleep(0.01)
23            client.send(":end_history".encode())
24        else:
25            client.send(":end_history".encode())

```

Figure 11: fonction de récupération de l'historique

Cette fonction prend en argument le nom d'utilisateur et le nom du correspondant ou channel dont on souhaite récupérer l'historique, et interroge la base de données de deux manières différentes. Tout d'abord, si l'historique demandé est une discussion privée, la requête à la base sera : tous les messages qui ont pour envoyeur l'utilisateur qui demande l'historique et comme receveur l'autre personne de la discussion privée, et inversement (voir ligne 5 figure 11). Si l'historique demandé est celui d'un channel, la requête sera : tous les messages qui ont pour receveur le nom du channel.

Ensuite, elle formate correctement les données et les envoie au client. Avant de commencer l'envoi des messages d'historique, elle envoie un message protocolaire, `":start_history"` pour informer le thread de réception du client qu'elle ne doit plus intercepter les messages qui vont arriver et laisser la bonne fonction s'en charger. Lorsque l'historique a fini d'être envoyé, un message protocolaire `":end_history"` est envoyé, pour indiquer au client que son fonctionnement peut reprendre à la normale.

Comme vous l'avez probablement remarqué, à la ligne 22, une pause de 0.01 seconde a été implémentée, car en son absence, avec le temps de traitement du côté du client, certains messages été concaténés, ou pire ignorés.

```

1  def verify_access(client,connection_sql,chanel):
2      if chanel in channels_are:
3          try:
4              with connection_sql.cursor() as cursor:
5                  sql = "SELECT `user`,`channel`,`etat` FROM `access` WHERE `user`=%s and `channel`=%s;"
6                  cursor.execute(sql,(pseudo[client], chanel))
7                  result = cursor.fetchall()
8          except pymysql.Error as e:
9              print(f"MySQL Error: {e}")
10         else:
11             if result:
12                 if result[0].get('etat') == 1 and result[0].get('channel') == chanel and result[0].get('user') == pseudo[client]:
13                     client.send(':start_check'.encode())
14                     time.sleep(0.001)
15                     client.send(":subed".encode())
16                 else:
17                     client.send(':start_check'.encode())
18                     time.sleep(0.001)
19                     client.send(":notsubed".encode())
20                     time.sleep(0.001)
21                     # wait_conf_sub(client,connection_sql)
22                     client.send(":end_check".encode())
23             else:
24                 client.send(':start_check'.encode())
25                 time.sleep(0.001)
26                 client.send(":notsubed".encode())
27                 time.sleep(0.001)
28                 client.send(":end_check".encode())
29
30     else:
31         client.send(':start_check'.encode())
32         time.sleep(0.001)
33         client.send(":subed".encode())
34         time.sleep(0.001)
35         client.send(":end_check".encode())
36         time.sleep(0.001)
37
38
39

```

Figure 12: fonction de vérification d'accès à un channel

Cette fonction a pour but de savoir si un utilisateur a accès à un channel restreint, si sa demande est en cours ou s'il n'a jamais fait la demande.

Cette fonction est appelée automatiquement par le client peu importe si la cible est bien un channel ou si c'est un fil de message privé entre deux utilisateurs, donc il est important de vérifier que ce soit bien un channel. Puis on vérifie que ce soit bien autorisé, puis, on envoie un message protocolaire `":start_check"`, comme vu précédemment pour l'historique. On envoie ensuite la réponse en elle-même : `":subed"` si la personne a accès au channel ou `":notsubed"` si ce n'est pas le cas. Enfin, on envoie `":end_check"` pour informer du retour au comportement de base du client.

```

1 def command_entry():
2     global server_stop
3     connection_sql = pymysql.connect(host='localhost', user='root', password='toto', database='sae301_discord')
4     while not server_stop:
5         command = input()
6         if any(substring in command for substring in ['?', 'help', 'commande']):
7             print("Les commandes disponibles sont :\n - /arret qui arrete le serveur\n - /liste demande qui listera les demandes pour les channels")
8         if command == "/arret":
9             for client in clients:
10                 client.send("server_stop".encode())
11             server_stop = True
12             arret_server()
13         if command == "/liste demande":
14             try:
15                 with connection_sql:
16                     with connection_sql.cursor() as cursor:
17                         sql = "SELECT `user`,`channel` FROM `access` WHERE `etat`=0;"
18                         cursor.execute(sql)
19                         result = cursor.fetchall()
20
21             except pymysql.Error as e:
22                 print(f"MySQL Error: {e}")
23             else:
24                 if not result:
25                     print("No pending requests.")
26                 else:
27                     for row in result:
28                         print(f"{row['user']} demande pour le channel {row['channel']}")

```

Figure 13: fonction d'entrée de commande administrateurs

Cette fonction est celle qui permet à l'administrateur de rentrer des commandes au serveur, comme `/arret` par exemple.

Elle a accès à la base de données pour pouvoir lister les demandes d'accès aux channels en attente par exemple.

```

1 if command[:4] == "/ban":
2     command = command.split(" ")
3     try:
4         with connection_sql:
5             with connection_sql.cursor() as cursor:
6                 sql = "SELECT `user` FROM `users`;"
7                 cursor.execute(sql)
8                 result = cursor.fetchall()
9
10    except pymysql.Error as e:
11        print(f"MySQL Error: {e}")
12    else:
13        try:
14            with connection_sql:
15                with connection_sql.cursor() as cursor:
16                    sql_insert = "INSERT INTO ban (user, start_ban, duration) VALUES (%s, NOW(), %s);"
17                    cursor.execute(sql_insert,[command[1],command[2]])
18                    connection_sql.commit()
19                    connection_sql.close()
20
21        except pymysql.Error as e:
22            print(f"MySQL Error: {e}")
23        else:
24            print(f"{command[1]} is ban for {command[2]}")
25

```

Figure 14: fonction d'entrée de commande, partie BAN

Même si elle n'est pas finie et lève systématiquement une erreur, cette fonction est sensée ajouter le ban d'un utilisateur dans la base de données. Elle récupère l'entrée de l'admin sous la forme `/ban user temps` puis elle découpe la commande selon les espaces et fait une requête SQL pour ajouter le ban dans la table ban.

```
1  if command == "/liste user":
2      try:
3          with connection_sql:
4              with connection_sql.cursor() as cursor:
5                  sql = "SELECT `user` FROM `users`;"
6                  cursor.execute(sql)
7                  result = cursor.fetchall()
8
9      except pymysql.Error as e:
10         print(f"MySQL Error: {e}")
11     else:
12         print("\nliste des users : \n")
13         for row in result:
14             print(f"- {row['user']}")
15         print()
```

Figure 15: fonction d'entrée de commande, partie liste user

La commande `"/liste user"` a pour unique but de lister les pseudos inscrits sur l'application, pour permettre à l'administrateur de bien orthographier un nom pour le ban.

```
1  def arret_server():
2      server = socket.socket()
3      server.connect(("localhost", port))
4      server.close()
5
```

Figure 16: fonction de validation d'arrêt

Cette fonction a pour seul but de faire arrêter le serveur correctement. C'est-à-dire que sans celle-ci, toutes les fonctions du serveur sont arrêtées, hormis, la fonction d'acceptation de connexion des clients (cf. figure 5 lignes 26). Elle fait une tentative de connexion sur lui-même ce qui sort le serveur de son état d'attente et lui permet de finir son exécution proprement.

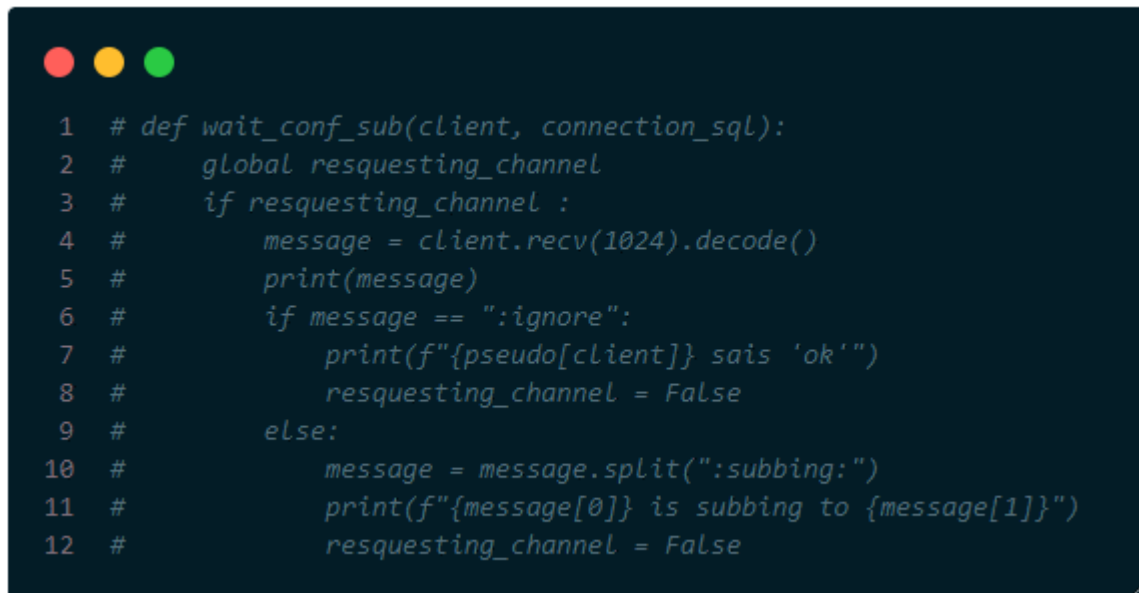
## IV. Limites & Réflexion

### 1) Sécurité

L'un des plus gros problèmes de sécurité est que le mot de passe et le nom d'utilisateur administrateur sont "*hard coded*" dans le code du serveur, ce qui laisserait l'occasion à une personne mal intentionnée de le lire ou de le modifier librement. Cela est aussi vrai pour le mot de passe de la base de données, il est inscrit en clair ainsi que le nom d'utilisateur et l'adresse IP du serveur SQL, ce qui est relativement dangereux.

### 2) Limites

Comme expliqué dans la documentation client, la possibilité de demander l'accès à un channel n'a pas pu être mise totalement au point, vous pourrez trouver dans le code un reliquat d'un essai de mise au point de cette option. En effet, il m'a été impossible de finir cette mise au point. Je n'arrivais pas à récupérer la réponse de l'utilisateur par rapport à sa décision, demander l'accès ou ignorer le refus d'accès.

A screenshot of a code editor with a dark background and light-colored text. The code is a Python function named `wait_conf_sub` that takes `client` and `connection_sql` as arguments. It uses a `global` variable `resquesting_channel`. The function checks if `resquesting_channel` is true. If so, it receives a message from the client, prints it, and checks if it's `":ignore:"`. If yes, it prints a message and sets `resquesting_channel` to `False`. Otherwise, it splits the message by `":subbing:"`, prints the first part as the user and the second as the channel, and sets `resquesting_channel` to `False`.

```
1 # def wait_conf_sub(client, connection_sql):
2 #     global resquesting_channel
3 #     if resquesting_channel :
4 #         message = client.recv(1024).decode()
5 #         print(message)
6 #         if message == ":ignore":
7 #             print(f"{pseudo[client]} sais 'ok'")
8 #             resquesting_channel = False
9 #         else:
10 #             message = message.split(":subbing:")
11 #             print(f"{message[0]} is subbing to {message[1]}")
12 #             resquesting_channel = False
```

Figure 17: développement en cours du traitement de la décision utilisateur

D'un autre côté, l'option "ban" n'a pas pu être terminée à cause d'une erreur PyMySQL de fermeture de connexion, dont je n'ai pas réussi à trouver la solution. Hormis cette erreur, le ban est fonctionnel. C'est-à-dire que, pour tester cette fonction, il suffit de rentrer manuellement le ban dans la base de données avec la commande `"INSERT INTO ban (user, start ban, duration) VALUES ('toto', NOW(), 5);"` par exemple, pour ban le user "toto" pour 5 minutes.

## V. Annexe

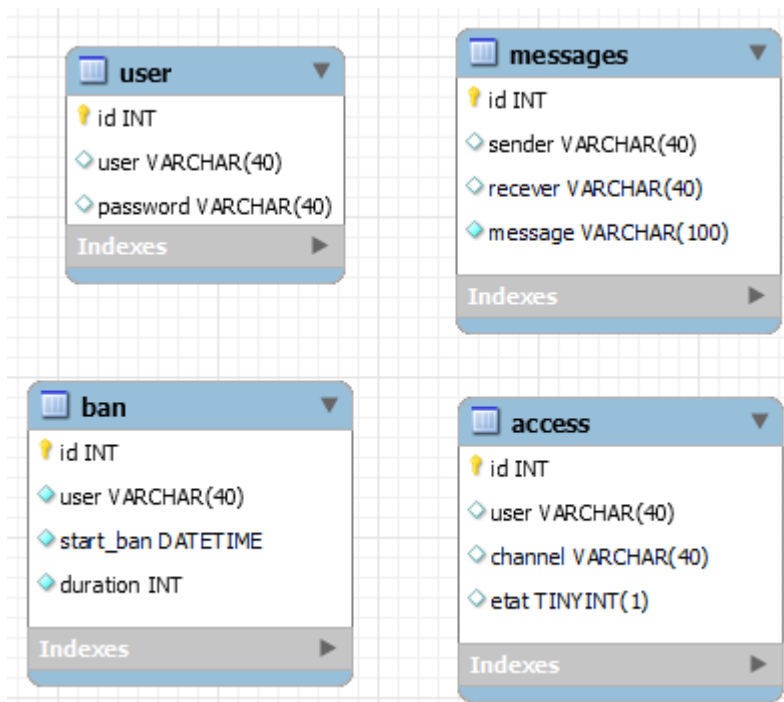


Figure 18: Schéma relationnel de la base de donnée