Networks & Architectures

Projet Serveur Chat

Membres du groupe :

GUO Huaiyuan

GRARD Vincent

GRIL Florian

DEFFORGE Julien

ESILV 2020-2021

IOS 2 – A4

Sommaire

Table des matières

Introduction

Notre projet concerne le projet d'un serveur implémentant un chat avec de nombreuses fonctionnalités. Ce chat doit permettre l'interaction, l'échange de données entre plusieurs personnes grâce à l'intermédiaire du serveur. Ces personnes doivent en effet s'authentifier pour accéder au serveur et au chat via notre système de connexion basé sur SQLite. En cas de problème (enquête judiciaire), notre serveur doit aussi pouvoir nous offrir l'accès aux logs et au chat pour le résoudre.

Les fonctionnalités introduites sont donc implémentées côté client ou serveur. En lançant notre serveur, il est alors possible de se connecter via la base de données déjà créée ou alors de s'inscrire directement, pour cela un pseudo et un mot de passe sont nécessaires. Ensuite, le client a accès à de nombreuses fonctionnalités comme l'envoi de message dans le chat global, privé ou en message privé avec d'autres utilisateurs. Il peut aussi transférer des fichiers sur le serveur, en télécharger depuis le serveur et faire des échanges avec les autres utilisateurs. L'utilisateur peut avoir accès à la liste des personnes connectés et des fichiers du serveur.

Il y a aussi des fonctionnalités coté serveur permettant d'alerter les utilisateurs (via un message envoyé à tous les utilisateurs), de bannir ou plutôt kick des personnes du serveur et enfin de stopper le serveur (et donc kick tous les clients).

Dans le rapport suivant nous allons alors expliquer les différentes parties de notre code et comment nous avons implémenté les fonctionnalités présentées plus tôt. Tout d'abord avec la partie client comprenant le threading et l'envoi de certains messages. Ensuite, avec la partie serveur qui est utile pour toutes les fonctionnalités du client et du serveur. Enfin, avec la partie SQLite permettant le lien avec la base de données permettant de s'inscrire et de s'authentifier.

Nous livrons alors trois fichiers pythons comprenant chacun respectivement le code des trois parties (client, serveur, base de données), le fichier de base de données comprenant la base de données (le fichier .db) et enfin le fichier ReadMe.txt qui contient les informations pour installer et faire fonctionner notre projet server de chat.

Le client

On importe d'abord socket et threading qui permettent de créer et gérer entre autres les sockets.

D'abord, on définit l'host ici « 127.0.0.1 » pour une utilisation locale et le numéro de port libre 52020, le buffer size qui permet de définir la taille utilisée en fonction du flux qui arrive, l'END = False est lui utile lors que l'utilisation ou fermeture du Client.

```
import socket
import threading

HOST = "127.0.0.1"
PORT = 52020
BUFFER_SIZE=2048
END = False
```

Figure 1 : variables globales

On crée ensuite la classe ReceiveResponseFromServerThread() (Figure 2) prenant en entrée le Thread, cette classe gère la réception des données du serveur. Pour initialiser cette classe on utilise les différents socket client, le pseudo utilisé, l'état du client (initialisé à True). Lorsque le Thread est lancé, la méthode run est exécutée.

```
class ReceiveResponseFromServerThread(threading.Thread):
    def __init__(self, clientSocket, sender_client):
        threading.Thread.__init__(self)
        self.cSocket = clientSocket
        self.my_sender_address = str(sender_client.getsockname())
        self.Sender_client = sender_client
        self.pseudo = ''
        self.etat = True
        print("Thread pour la réception des données initialisé")

    def run(self):
        global END
        msg = ''
        try:
            while True:
                data = self.cSocket.recv(BUFFER_SIZE) # Instructic
                msg = data.decode()
                print("admin:", msg)
                if "#Response TrfD#" in msg:
                    m = msg.split('/')
                    received_f = m[1]
                    f = open(received_f, 'wb')
                    while True:
                        data = self.cSocket.recv(BUFFER_SIZE)
                        if b"#End TrfD#" in data:
                            break
                        f.write(data) # Write data to a file
                    f.close()
                if "#Kill" in msg:
                    print("Tu es kick.")
                    send_message("#Exit",self.Sender_client)
                    self.cSocket.close()
                    self.Sender_client.close()
                    self.etat = False
                    END = True
        except:
            self.cSocket.close()
            print("Fin du thread")
```

Figure 2 : Classe ReceiveResponseFromServerThread côté Client

Cette méthode attend d'abord un message du serveur et tant qu'il n'y en a pas, on ne passe pas à la méthode suivante et donc la première instruction attendant le message est bloquante. Ce message est décodé et s'il reçoit #Response TrfD#, il écrira le document demandé à être téléchargé coté Client. S'il reçoit #Kill, le thread envoie un message au client pour qu'il se ferme automatiquement et fermera le Thread par la suite en passant son état à False ensuite. Si le try ne fonctionne pas le thread se fermera.

Nous avons ensuite créé plusieurs fonctions utiles pour connecter les sockets, envoyer des messages ou des fichiers (en public ou privé) (Figure 3).

La première fonction permet de se connecter au serveur via socket.socket et .connect() qui connecte la socket au bon host et port défini. On peut alors aussi envoyer un message ou un fichier (en privé ou en public), ces méthodes utilisent le .send() pour permettre l'envoi. De plus, pour envoyer un message, il a besoin d'être encodé (via UTF-8) et pour utiliser l'envoi de fichier nous utilisons des requêtes de messages pour permettre l'échange entre serveur et client.

```python
# Fonction pour la connexion au serveur.
def connect_to_server(HOST, PORT):
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client.connect((HOST, PORT))
    print ("Connexion vers " + HOST + " : " + str(PORT) + " réussie.")
    return client


# Fonction pour envoyer un message au serveur.
def send_message(message, client):
    message = message.encode('UTF-8')
    n = client.send(message)
    if n!=len(message):
        print("Erreur envoi")
```

```python
# Fonction pour envoyer un fichier.
def send_file(filename, socket):
    send_message("#TrfU Request# - " + filename, socket)
    f = open(filename, 'rb')
    while True:
        l = f.read(BUFFER_SIZE)
        while (l):
            socket.send(l)
            l = f.read(BUFFER_SIZE)
        if not l:
            f.close()
            send_message("#End TrfU#", socket)
            break
```

Figure 3 : fonctions de connexion et d'envoi de messages et de fichiers

Dans notre main (Figure4), on crée tout d'abord les sockets de création pour l'envoi et la réception de données, on crée ensuite le thread avec les deux sockets créés et on le start.

```python
def main():
    receiver_client = connect_to_server(HOST, PORT) # création du socket pour la réception des données.
    sender_client = connect_to_server(HOST, PORT) # création du socket pour l'envoie des données.

    #my_sender_address = str(sender_client.getsockname())
    print("Mon adresse d'envoie : " + str(sender_client.getsockname()))

    receiveResponseFromServerThread = ReceiveResponseFromServerThread(receiver_client, sender_client) #
    receiveResponseFromServerThread.start() # Démarrage du thread.
```

Figure 4 : main côté Client (part 1)

Tant qu'END n'est pas vrai, (Figure 5) alors on peut envoyer des messages depuis le thread et recevoir des réponses, avec #Help on obtient les infos sur les fonctions utilisables (implémentés dans la partie serveur) et il y a aussi les instructions permettant de transférer un fichier vers le serveur ou en privé en utilisant les fonctions send_file ou send_private_file (fait en splittant les infos sur message).

Pour un message d'exit, celui-ci ferme les sockets de création pour l'envoi et la réception de données ce qui permet de quitter le serveur (de déconnecter l'utilisateur).

```
# L'émission de message se fait dans le main.
while not END:
    message = input("Moi : ")
    if not END:
        send_message(message, sender_client)
    else:
        print ("Tu as été Kick durant ton absence.")
        break
    if message == "#Help":
        print("Liste de toutes les commandes clients : \n")
    if message == "#Exit":
        print ("Déconnexion")
        sender_client.close()
        receiver_client.close()
        break
    if "#TrfU private" in message:
        msg = message.split(' ')
        recipient = msg[2]
        filename = msg[3]
        send_private_file(recipient, filename, sender_client)
    elif "#TrfU" in message:
        msg = message.split(' ')
        filename = msg[1]
        send_file(filename, sender_client)
        print("File is uploaded on server")
```

Figure 5 : main côté Client (part 2)

Le serveur

La partie serveur de notre projet est la pièce centrale. En effet notre structure est de type « centralisée » et fonctionne sur le même principe que la topologie en étoile. Chaque client va se connecter au serveur et lui envoyer des messages. Le serveur s'occupe ensuite de redistribuer les messages aux autres clients. Le programme python qui « reçoit » et gèrent les connexions entrantes venant des clients s'appelle « server_multi_thread » (Figure 6). Cette gestion des connexions entrantes se fait dans le « main » à la fin du fichier python dans le « While SERVER_SWITCH » où SERVER_SWITCH est défini à True (cf. Figure 7) au début du programme et tourne à False que lorsque nous écrierons : #Exit (cf. commandes Admin).

```
## Main ##
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # Création du serveur.
server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server.bind((HOST, PORT)) # Bind avec l'addresse et le numéro de port.
print("Server started")
print("Waiting for client request...")

commandThread = CommandThread() # Creation du Thread qui va gérer les commandes pour le serveur.
commandThread.start() # Démarrage du Thread.

lst_files = load_files_secure()

while SERVER_SWITCH:
    server.listen(1)

    # Gestion du socket receiver_client. Uniquement besoin de l'ajouter dans la liste des clients connectés.
    clientSocketReceive, clientAddress = server.accept() # Instruction bloquante. Passage à la ligne suivante quand un client se connecte.

    # Gestion du socket sender_client. Uniquement besoin de créer un Thread.
    clientSocketSender, clientAddress = server.accept() # Instruction bloquante. Passage à la ligne suivante quand un client se connecte.

    # Instanciation du client.
    new_client = Client(clientAddress, clientSocketSender, clientSocketReceive, CONNECTED_CLIENTS)
    #connected_clients.append(new_client) # Ajout à la liste des clients connectés.

    # Instanciation du Thread.
    newThread = ClientThread(new_client) # Création du Thread pour ce nouveau client.
    newThread.start() # Démarrage du Thread.

    CONNECTED_CLIENTS = CONNECTED_CLIENTS + 1
```

Figure 6 : Le main du serer_multi_thread.py

```
HOST = "127.0.0.1"
PORT = 52020
BUFFER_SIZE = 2048
SERVERADDRESS = ">>>"
CONNECTED_CLIENTS = 0
connected_clients = [] # liste qui va contenir tous les clients connectés.
lst_files = []
SERVER_SWITCH = True
```

Figure 7 : Les variables globales

Au préalable le serveur a été créé et lié à l'adresse localhost et au numéro de PORT 52020 (cf. Figure 2). Le serveur est ensuite mis à l'écoute et attend la connexion d'un client (cf. Figure 8 ligne 64). Lorsqu'un client se connecte, le processus s'enclenche.

```
55      def run(self):
56          global connected_clients
57          print("Connection from : ", self.Client.address)
58          authentication_result = ask_authentication(self.Client)
59          if authentication_result and authentication_result[0]:
60              self.Client.pseudo = authentication_result[1]
61              write_in_log_file("Connexion de " + self.Client.pseudo + " / Adresse IP : " + str(self.Client.address)  + " à " + str(datetime.now()))
62              while True:
63                  # Instruction "bloquante". Tant que le Thread ne reçoit pas de message, il ne passe pas à la ligne suivante.
64                  data = self.Client.senderSocket.recv(BUFFER_SIZE)
65                  # Décodage de ce message
66                  data = data.decode()
67                  # Affichage du message
68                  print(self.Client.pseudo + " : " + data)
69                  write_in_chat_file(self.Client.pseudo + " : " + data)
70                  #data: "input du client"
```

Figure 8 : Début ClientThread

Dans ce script il y a deux classes héritant de « threading.Thread » : ClientThread et CommandThread. Le ClientThread est la classe permettant de gérer tous les clients en parallèle. Lorsqu'un client va se connecter au serveur, un nouveau « ClientThread » sera créé et lui sera associé. La classe « Client » (Figure 9) permet de faire cette association entre le client et le thread.

```
25 # Classe représentant un client connecté au serveur.
26 class Client():
27     def __init__(self, clientAddress, senderSocket, receiverSocket, clientNumber):
28         self.address = clientAddress
29         self.senderSocket = senderSocket # Socket utilisé par le client pour envoyer des données.
30         self.receiverSocket = receiverSocket # Socket utilisé par le client pour recevoir des données.
31         self.number = clientNumber
32         self.pseudo = ""
33         self.privateMode = False # Variables pour gérer la communication privée.
```

Figure 9 : class Client côté serveur

Afin de pouvoir envoyer et recevoir des messages de façon simultanée le client ouvre deux canaux de communications (voir la partie sur le programme « Client »), donc deux sockets. On retrouve ces deux sockets dans la classe présentée ci-dessus :

- senderSocket correspond au socket utilisé par le client pour envoyer des données. Donc lorsque le serveur recevra un message du client il devra « attendre » (recv) sur la « senderSocket ».
- receiverSocket correspond au socket utilisé par le client pour recevoir des données. Donc lorsque le serveur voudra envoyer un message au client il devra se servir de ce socket.

Revenons au ClientThread. La méthode run (héritée de threading.Thread) est la méthode exécutée lorsque le Thread est démarré (newThread.start()). On utilise un « while True » pour laisser le Thread

s'exécuter. La toute première action entre le client et le serveur est la gestion de l'authentification. Si le client à un compte dans la BDD il peut accéder au serveur, sinon il peut en créer un. Si l'utilisateur saisi un mauvais mot de passe il ne pourra pas accéder au serveur et devra se déconnecter puis se reconnecter.

Une fois le client authentifié, lorsque celui-ci enverra un message au serveur, le message sera analysé. La ligne surlignée (dans la capture suivante) correspond à une instruction « bloquante », c'est ici que le thread est en attente de réception d'un message venant du client.

Le message est ensuite décodé (converti en string) et analysé. S'il s'agit d'une commande celle-ci sera analysée et l'opération correspondante sera exécutée. S'il s'agit d'un message celui-ci sera redistribué à tout le monde ou seulement au « correspondent » du client si ce dernier est en mode « private ».

Dans ce programme il y a aussi de nombreuses fonctions permettant par exemple de lister les fichiers (cf. figure11), d'écrire dans le fichier de log (cf. figure 10), d'écrire dans le fichier de sauvegarde des chats, d'envoyer des messages, des fichiers, de recevoir des fichiers, tout cela de façon publique ou privée.

```python
# Fonction pour écrire dans le fichier de log.
def write_in_log_file(message):
    message += "\n"
    filename = "LogFile.txt"
    f = open(filename, "a")
    f.write(message)
    f.close()
```

Figure 10 : Fonction d'écriture dans le fichier de log

```python
def list_files_v2():
    L='Liste des fichiers du serveur : \n'
    files = os.listdir()
    files.remove('database_sqlite_access_v3.py')
    files.remove('LogFile.txt')
    files.remove('ChatFile.txt')
    files.remove('__pycache__')
    for file in files:
        L=L+file+' \n'
    return L
```

Figure 11 : lister les fichiers

Il y a aussi des fonctions permettant de gérer les commandes du serveur (Kill, lister les fichiers, lister les utilisateurs connectés, afficher l'aide et alerter les clients). Ces commandes sont d'ailleurs traitées via le CommandThread(cf. figure 12). Grâce à ce thread, le serveur peut envoyer des messages, recevoir des messages, etc. tout en laissant à un administrateur la possibilité d'agir sur le serveur :

```
def run(self):
    global SERVER_SWITCH
    start = True
    while start:
        print("__Welcome in our chatbox!__")
        print("If you need help or learn more about it: #Help")
        working = True
        while(working):
            action_on_server = input("Your action on the server : ")
            action_on_server = action_on_server.rstrip() #Void Blank

            if action_on_server in ["#Help", "#help"]:
                helpCommand()

            if action_on_server in ["close","#Exit","#exit"]:
                working, start = False, False
                close_all_client_Thread()
                SERVER_SWITCH = False


            if action_on_server == "hello":
                print("hello")

            if action_on_server.startswith(("#Kill","#kill")):
                killCommand(action_on_server)

            if action_on_server in ["#ListU","#listU","#Listu","#listu"]:
                print("la liste des utilisateurs:")
                for i in connected_clients:
                    ourClient = ["Wonderful", "Strongest", "Richest","Smart"]
                    print(ourClient[random.randint(0,len(ourClient)-1)]," ",i)

            if action_on_server.startswith(("#Alert","#alert")):
                alertCommand(action_on_server)

            if action_on_server in ["#ListF","#Listf"]:
                if len(lst_files) != 0:
                    for i in len(lst_files):
                        print(i, ". ",lst_files[i])
                else:
                    print("Il n'y a pas de fichier disponible")

            if action_on_server.startswith(("#Private", "#private")):
                privateCommand(action_on_server)
```

Figure 12 : Les commandes du Thread serveur

Avec les commandes en question :

```
408  def killCommand(action_on_server, raison = ""):
409      userPseudo = ""
410
411      if (action_on_server == "#Kill" or
412                              action_on_server == "#kill"):
413                          userPseudo = input("User pseudo: ")
414      else:
415                          userPseudo = action_on_server.split(" ")[1]
416      reason = input("reason:") if raison == "" else raison
417      for client in connected_clients:
418          if client.pseudo == userPseudo:
419              #connected_clients.remove(client)
420              print("###KILL### :",userPseudo, " kicked for ",reason)
421              send_msg_to_client(client.receiverSocket,"#Kill".encode('UTF-8'))
422              #client.working = False
423              return 0
424      print("Je ne trouve pas le client comportant le pseudo: ",userPseudo )
425
426
427      #Manque la facon de l'éjecter
428
429
430  def alertCommand(message):
431      if message in ["#Alert","#alert"]:
432          message = input("Saisie de ton message: ")
433      else:
434
435          message = message[7:]
436      finalMessage = ("\033[31m"+
437          "\n ####################ALERT######################## \n"+
438                          message + " \n " +
439                          "####################ALERT########################"
440                          + "\033[0m")
441      send_all(finalMessage, "Admin")
442
443  def privateCommand(message):
444      pseudo = ""
445      if message in ["#Private","#private"]:
446          pseudo = input("Saisie le pseudo de la personne:")
447      else:
448          pseudo = message.split(" ")[1]
449      for c in connected_clients:
450          if c.pseudo == pseudo:
451              message = input("Saisie de ton message: ")
452              finalMessage = ("[Private from admin]: " + message).encode('UTF-8')
453              send_msg_to_client(c.receiverSocket,finalMessage)
454              return 0
455      print("pseudo invalide")
456
457  def close_all_client_Thread():
458      alertCommand("#Alert              ____fermeture du serveur____")
459      for c in connected_clients:
460          killCommand("#Kill " + c.pseudo,"MAINTENANCE")
461
462      return 0
```

Figure 13 : les commandes liant le Thread Serveur

La fonction pour gérer la réception d'un fichier est la suivante :

```
211 def receive_file(client, filename):
212     lst_files.append(filename)
213     f = open(filename, 'wb')
214     while True:
215         data = client.senderSocket.recv(BUFFER_SIZE)
216         if b"#End TrfU#" in data:
217             break
218         f.write(data) # Write data to a file
219     f.close()
```

Figure 14 : fonction de réception de fichier

On ajoute dans la liste des fichiers le nom du fichier reçu, on l'ouvre, et tant qu'il y a réception des données on écrit dans le fichier. La fonction se stoppe lorsque le client envoie un « tag » particulier (« #End TrfU# »).

Pour envoyer un fichier nous avons développé cette fonction :

```
221 def send_file(filename, client):
222     send_msg_to_client(client.receiverSocket, (str(SERVERADDRESS) + " #Response TrfD# /" + filename).encode('UTF-8'))
223     f = open(filename, 'rb')
224     while True:
225         l = f.read(BUFFER_SIZE)
226         while (l):
227             client.receiverSocket.send(l)
228             l = f.read(BUFFER_SIZE)
229         if not l:
230             f.close()
231             send_msg_to_client(client.receiverSocket, (str(SERVERADDRESS) + " #End TrfD#").encode('UTF-8'))
232             break
```

Figure 15 : fonction d'envoi de fichier

Pour commencer, le serveur indique au client qu'il va envoyer un fichier avec le tag « #Response TfD# » avec le nom du fichier séparé par un slash. Ainsi le client pourra se préparer et ouvrir un fichier avec ce nom. Ensuite le serveur lit le fichier et l'envoie au client. Lorsqu'il n'y a plus de données à lire, le fichier se ferme et le serveur indique au client que l'envoi de fichier est terminé.

Afin de lister les fichiers déjà présents dans le dossier du serveur lors d'un démarrage du programme, nous chargeons les fichiers préexistants. C'est la fonction « load_files » qui s'en charge.

```
def load_files_secure():
    files = os.listdir()
    removelist = ['database_sqlite_access_v3.py','LogFile.txt','ChatFile.txt','__pycache__']
    for elmt in removelist:
        if elmt in files:
            files.remove(elmt)
    return files
```

Figure 16 : fonction de chargement des fichiers

La base de données

La base de données SQLite permet au client de s'inscrire et de s'authentifier.

Le code python sqlite nommé « database_sqlite_access.py » a pour but d'exploiter la database « server_chat_database.db » pour vérifier l'existence ou le mot de passe d'un utilisateur.

Pour s'y connecter, la fonction initialisation() utilise la commande : conn = sqlite3.connect(db_file) avec db_file le chemin d'accès à la database.

Ensuite, lors de la connexion d'un client, il lui est demandé s'il possède ou non un compte. S'il n'en a pas, il peut répondre « non » et s'inscrire en écrivant « username/password ». La fonction add_user intervient et envoie une requête sql : conn.execute("INSERT INTO users (username, password) VALUES(?, ?)", (username, password,)). Ainsi, le nouveau client est inscrit.

```
def add_user(conn, username, password):
    result = ""
    try:
        conn.execute("INSERT INTO users (username, password) VALUES(?, ?)", (username, password,))
        conn.commit()
        result = "Utilisateur ajouté"
    except Error as e:
        print("exception from dao", e)
        result = e
    return result
```

Figure 17 : les commandes liant le Thread Serveur

Lors d'une nouvelle connexion, il pourra s'identifier. Ceci à l'aide de la fonction authentification (conn, username, password). Celle-ci se connecte à la database et va sélectionner l'utilisateur correspondant. A l'aide des fonctions cursor() et fetchall(), on accède à une liste de tuples qui ont donc le même utilisateur (unique ici car la database renvoie une erreur lors de l'inscription si l'utilisateur existe déjà, une contrainte d'unicité sur la colonne « username » a été ajoutée lors de la création de la table). Dans ce tuple, la 3eme donnée étant celle du mot de passe correspondant, on peut donc valider ou non l'authentification.

```
def authentication(conn, username, password):
    status_message = ''
    authentication_result = False
    curseur = conn.cursor()
    curseur.execute("SELECT * FROM users WHERE username=? ;", (username,))
    resultats = curseur.fetchall()
    if resultats:
        if password == resultats[0][2]:
            status_message = "Authentification réussie"
            authentication_result = True
        else:
            status_message = "Authentification échouée"
    else:
        status_message = "Utilisateur non existant"
    return (authentication_result, status_message)
```

Figure 18 : les commandes liant le Thread Serveur

Par le même procédé, on peut également vérifier lors de la connexion si l'identifiant existe avant d'essayer de vérifier son mot de passe. La fonction check_if_user_exist(conn, username) regarde simplement si le .fetchall() a bien un résultat.

```
def check_if_user_exist(conn, username):
    result = False
    curseur = conn.cursor()
    try:
        curseur.execute("SELECT * FROM users WHERE username=? ;", (username,))
        resultats = curseur.fetchall()
        if resultats:
            result = True
    except Error as e:
        result = False
        print("exception from dao", e)
    return result
```

Figure 19 : les commandes liant le Thread Serveur