

# Rapport Final

En 7 parties

Akif Aslankilic  
Frédéric Drouhin  
2024/2025



## – INTRODUCTION –

Maintenant que vous savez comment manipuler le client et le serveur, je vous propose d'essayer de comprendre les enjeux de ce projet.

Dans ce rapport, je vais vous expliquer comment j'ai élaboré l'architecture de mon projet, le choix des technologies que j'ai du utilisé pour arriver à mes fins, bien sûr, je vous montrerai comment utilisé et à quoi elles servent.

Pour arriver à cela, certains bâtons se sont hissés dans mes roues, donc j'ai du mettre en place des solutions afin de respecter tout de même le cahier des charges imposé. Je vous ferai un diagnostique de tout ce que je dois améliorer ou pouvais dans le système que je vais vous présenter.



## 1. L'architecture globale

### 1.1 Client

Le client est basée sur PyQt5, c'est lui qui va permettre de se connecter au serveur et d'effectuer toutes les tâches qu'on lui demande de faire comme se connecter au serveur, l'envoi des fichiers et la réception de message.

#### 1.1.1 Connexion au serveur

L'un des piliers de ce projet est la connexion au serveur.

Pour **établir une connexion entre un client et un serveur**, le client doit connaître le nom d'écoute réseau du serveur et, dans certains cas, également l'emplacement du serveur sur le réseau, dans notre cas, on effectue le test dans une interface virtuelle (127.0.0.1)

#### 1.1.2 L'envoi des fichiers

L'envoi des fichiers est l'essence même de ce projet, une fois la connexion établie, c'est ce paramètre qui va faire en sorte qu'on puisse **communiquer** entre le serveur et le client et le voir.

Pour ma part, j'ai utilisé l'envoi de fichier qui ont déjà été fait dans le dossier (*RapportEtDocInstallation*), mais j'aurai pu juste faire une interface ou l'utilisateur peut rentrer son texte et que sa demande soit traitée par la suite.

#### 1.1.3 La réception de message

« Bien reçu. » Eh oui, une fois un message envoyé, il faut le **réceptionner** (ce qui est tout le but d'un message..). La réception de message se fait tout autant côté serveur que côté client. (tout comme l'envoi d'ailleurs)

## 1.2 Serveur

Le serveur, ici, sera un **programme multi-thread** (envoi de plusieurs Thread en même temps) qui permettra de **gérer plusieurs connexions client en simultanées**, donc il ne doit pas refuser les autres connexions même si il a déjà un client connecté, autrement appelé..

Un serveur infidèle..

Le serveur réceptionnera les fichiers du client, les traitera, renverra les messages et **l'avantage avec ce type de serveur, c'est qu'il accepte tout les fichiers envoyées par le client et les répondra en temps réel.**

## 1.3 Communication entre le client et le serveur.

Une fois le serveur mis en place et le client connecté, nous pouvons procéder à la partie la plus concrète, la communication.

### 1.3.1 Transmission des données via des sockets TCP/IP

Le protocole **TCP/IP** va être le protocole dont nous aurons besoin pour faire communiquer nos machines entre elles.

TCP garantit que les données sont envoyées et reçues dans l'ordre correct, sans perte, ni duplication. Cela est particulièrement important dans des applications où l'intégrité des données est critique, comme le transfert de fichiers. Le serveur utilise un socket pour écouter les connexions entrantes sur une adresse IP et un port définis. Lorsqu'un client tente de se connecter, un nouveau socket dédié est créé pour gérer la session spécifique avec ce client.

### 1.3.2 Isolation des connexions pour garantir la stabilité et la sécurité

L'isolation des connexions offre une meilleure sécurité, car chaque session est indépendante, donc ça réduit les risques d'accès non autorisé aux données d'autres utilisateurs.

En cas de déconnexion ou d'erreur sur une connexion spécifique à un client, le reste des connexions continue de fonctionner normalement. Le serveur est configuré pour détecter ces interruptions et libérer les ressources associées à la connexion concernée car il est programmé pour cela.

## 2. Choix des technologies

### 2.1 Python

Comme vous avez pu le voir dans la doc. d'installation, ce projet est programmé en Python. L'un des avantages de ce langage est que c'est l'un des plus simple à comprendre. De plus, pour la compréhension, ça reste l'un des langages les plus faciles à interpréter même pour une personne qui débute.

En plus de cela, Python a la particularité de posséder plusieurs librairies qui vont nous être grandement utile.

Et le dernier point que souhaiterai aborder, et le fait que ce langage est accessible facilement sur toutes les plateformes, les OS, ou autres.

### 2.2 PyQt5

PyQt5 est un ensemble de bibliothèques C++ multiplateformes qui mettent en œuvre des API de haut niveau pour accéder à de nombreux aspects des systèmes de bureau et mobiles modernes. Il s'agit notamment des services de localisation et de positionnement, du multimédia, de la connectivité NFC et Bluetooth, d'un navigateur Web basé sur Chromium, ainsi que du développement traditionnel de l'interface utilisateur (D'après [pypi.org](https://pypi.org))

Dans ce projet, PyQt5 fournit entre autres des outils pour créer une interface graphique moderne et en plus interactive.



PyQt5 intègre des éléments comme des **boutons**, des **boîtes de dialogue**, ou encore des **barres de progression** (chose que je n'ai pas intégré dans mon projet), enfaite PyQt donne une bonne plage d'option a utiliser pour mettre l'utilisateur dans de bonnes circonstances, ce qui nous est très favorable pour ce projet.

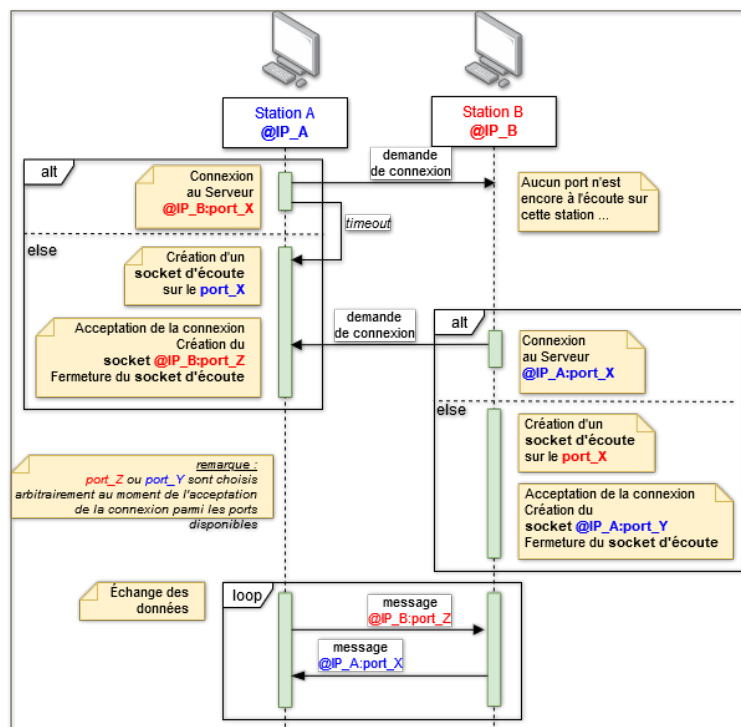
## 2.3 Les Sockets

Venons en a l'élément de base de la communication entre 2 entités, les sockets..

Les sockets permettent d'établir la connexion via le protocole TCP/IP.

Ils sont la pour assurer une **communication bidirectionnelle** fiable entre le client et le serveur. Nous pouvons le voir sur l'image ci-dessous :

(Source : <https://info.blaisepascal.fr/nsi-sockets-python/> )



## 2.4 Les Threads

Le threading permet de faire fonctionner différentes parties du programme simultanément et peut la simplifier comme, la gestion simultanée de plusieurs connexions clients, par exemple.

L'avantage principal du Thread est qu'il garantit une exécution fluide des tâches qu'on lui donne sans pour autant compromettre les performances du programme ou ralentir ce dernier.

## 3. Instructions de Déploiement

### 3.1 Préparation

#### 3.1.1 Installation des outils nécessaires

Tout d'abord, il vous faut télécharger et installez la dernière version de Python comme on l'avait fait dans le document d'installation et d'utilisation.

Puis, il nous faut installer les bibliothèques dont nous avons besoin :

Nous devons taper dans un terminal, les lignes suivantes :

- pip install PyQt5
- pip install socket

#### 3.1.2 Organisation des fichiers

Une fois fait, nous devons placer nos fichiers '`client.py`' et '`serveur.py`' dans un répertoire commun pour simplifier la gestion du système dans son ensemble.



## 3.2 Démarrage

### 3.2.1 Côté Serveur

Pour démarrer le programme :

Exécutez '**serveur.py**' en spécifiant l'adresse IP et le port comme arguments, comme ceci:

```
python3 serveur.py 127.0.0.1 4200
```

Une fois le serveur démarré, un message nous affiche bien qu'il s'est lancé et que le serveur est à la recherche de client :

```
akif@MacBook-Pro-de-Akif SAE3.02 % python3 serveur.py 127.0.0.1 4200
Serveur en attente de connexion...
```


'Serveur en attente de connexion...' Cela démontre bien ce que nous avons expliqué tout à l'heure, et je vous expliquerai comment le code fonctionne plus tard dans le rapport.

### 3.2.2 Côté Client

Pour démarrer le programme :

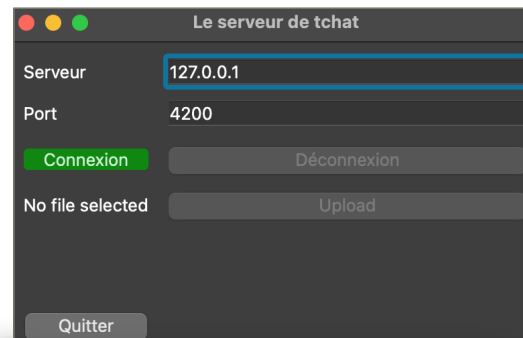
Exécutez seulement '**client.py**' (pensez à être bien dans le dossier où se trouve les fichiers) comme ceci :

```
akif@MacBook-Pro-de-Akif SAE3.02 % /usr/bin/python3 /Users/akif/Documents/SAE3.02/client.py
2024-12-24 23:17:12.457 Python[28747:1939511] +[IMKClient subclass]: chose IMKClient_Modern
```

À partir d'ici, le fichier s'est bien exécuté correctement comme on peut le voir sur la ligne d'après. J'ai lancé le programme grâce au bouton  sur Visual Studio Code, c'est pourquoi il me lance le programme avec l'arborescence des fichiers -> /usr/bin/python3 – Alors qu'on pourrait juste écrire -> python3 – pour référencé Python.



Noël est passé mais la magie opère quand même car  
notre programme se lance parfaitement :



Grâce aux multi-threading, plusieurs clients peuvent se connecter en même temps, et peuvent être gérés par le serveur en simultané. Exemple de deux clients envoyant un message au serveur en même temps :

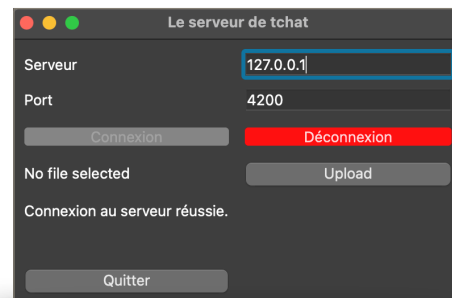
```
Affiche de la sortie dans : 3
Affiche de la sortie dans : 2
Affiche de la sortie dans : 1
Affiche de la sortie dans : 3
Hello
Le message 3.01523 seconde(s) à s'établir avec le client.
Affiche de la sortie dans : 2
Affiche de la sortie dans : 1
Hello
Le message 3.00925 seconde(s) à s'établir avec le client.
```

Le problème ici, c'est qu'on arrive pas à reconnaître qui est qui.. Cependant on voit bien que le programme du client n°2 s'est lancé pendant que le serveur traitait la demande du client n°1. C'est ça le multi-threading.

## 4. Tests et Vérifications

### 4.1 Connexion

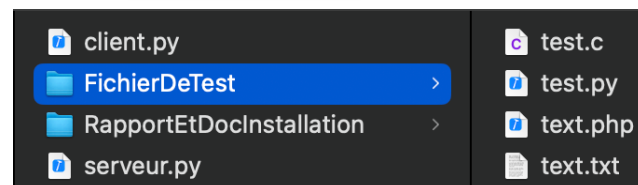
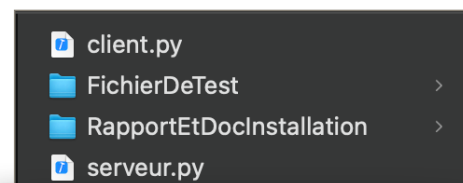
Nos programmes sont lancés, passons aux choses concrètes comme par exemple tester et vérifier que le client peut établir une connexion avec le serveur. Perdons pas de temps, voilà ce que cela donne lorsque je me connecte au serveur avec mon interface graphique client :



Comme vous pouvez l'apercevoir, la connexion au serveur a été un succès. À présent, les boutons '**Déconnexion**' et '**Upload**' sont cliquables.

### 4.2 Envoi des fichiers

Maintenant on va pouvoir envoyer des fichiers, depuis le client vers le serveur. Pour cela, il faut se rendre dans le dossier « **FichierDeTest** » que j'ai créé dans lequel vous trouverez les fichiers à utiliser. Une fois appuyer sur « **Upload** », voici le dossier :





Comme vous pouvez le constater, les fichiers tests sont bien présents, on y trouve 4 différents types de fichiers.


En partant du **C**, passant par du **Python** et du **PHP**, jusqu'à arriver à du fichier **texte**.

#### 4.2.1 Tests positifs

Je vais procéder à l'envoi de fichiers qui auront pour extension :

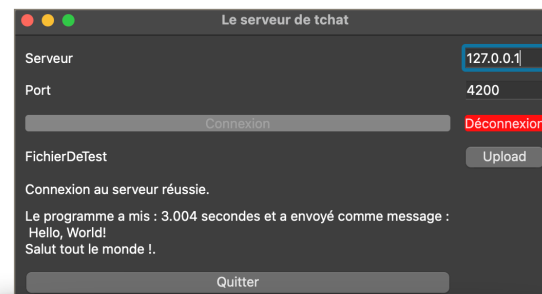
 **.py** (Pour les fichiers Python)

 **.c** (Pour les fichiers en C)

 **.txt** (Pour les fichiers texte)

Ces extensions ont pour caractéristiques d'être **White-listed** du programme, c'est à dire que si l'on envoie une autre extension au serveur, le serveur va nous rejeter le programme qu'on lui demande.

Commençons par **C** :



L'envoi du programme s'est bien effectué, le fichier contenait le code ci-dessous :

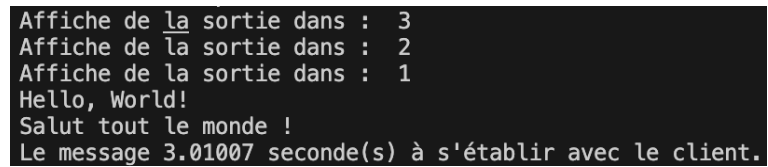


```
serveur.py  test.c  X
FichierDeTest > C test.c > main()
1  #include <stdio.h>
2
3  int main()
4  {
5      printf("\nHello, World!");
6      printf("\nSalut tout le monde !");
7
8      return 0;
9  }
```

Le fichier contient un programme simple où l'on va juste envoyer sur la console :

« Hello, World » suivi d'un saut de ligne et « Salut tout le monde »

Côté serveur, on reçoit bien le message indiqué dans le code.



```
Affiche de la sortie dans : 3
Affiche de la sortie dans : 2
Affiche de la sortie dans : 1
Hello, World!
Salut tout le monde !
Le message 3.01007 seconde(s) à s'établir avec le client.
```

Passons à **Python** :



L'envoi du programme s'est bien effectué, le fichier contenait le code ci-dessous :



Suis-je vraiment obligé d'expliquer ??

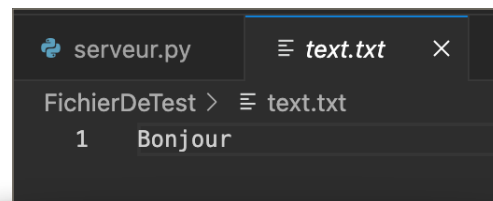
**Côté serveur :**

```
Affiche de la sortie dans : 3
Affiche de la sortie dans : 2
Affiche de la sortie dans : 1
Hello
Le message 3.0127 seconde(s) à s'établir avec le client.
```

Pour finir, voyons voir avec un fichier **texte** :



L'envoi du programme s'est bien effectué, le fichier contenait ceci :



Voilà, voilà...

#### Côté serveur :

```
Affiche de la sortie dans : 3
Affiche de la sortie dans : 2
Affiche de la sortie dans : 1
Bonjour
Le message 3.00761 seconde(s) à s'établir avec le client.
```

Bien sûr, libre à vous de choisir ce que vous désirez effectué dans le programme, ceci ne sont que des exemples.



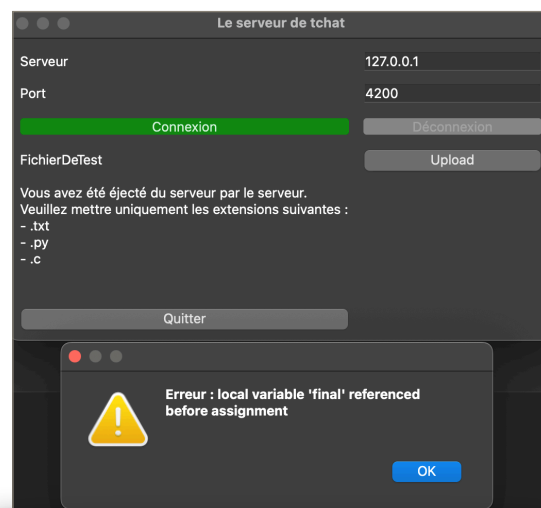
#### 4.2.2 Tests négatifs

À présent, passons aux tests négatifs.

Comme je l'avais introduit tout à l'heure, j'ai White-lister 3 extensions, donc les autres extensions sont sensés ne pas être accepté par le programme.

Sur le screen qui introduit la partie 4, vous pouvez voir qu'il y a un fichier PHP qu'on va utiliser pour tester notre programme, voir si il prend en compte ou pas le fichier.

Envoie du fichier [PHP](#)...



Comme on peut le voir, le programme nous ressort bien une **erreur**. L'interface graphique affiche aussi un message **d'erreur** qui indique que le fichier ne peut pas être réceptionner. De plus, nous sommes **déconnectés**. Tout est **OK**.

– Voilà pour les démonstrations d'envoi de fichiers, je tiens à rajouter que le serveur peut-être tenu 24H/24 sans être déconnecté, et l'envoi de fichiers peut se faire quand on le souhaite. –

## 5. Défis et Solutions

### 5.1 Compatibilité des Fichiers

Le **défi** : Tous les formats ne sont pas pris en charge.

La **solution** : Ajoutez un filtre qui accepte uniquement certains types de fichiers, comme `.txt`, `.py`, ou `.c`

Comme vous avez pu le constater juste avant, tout les fichiers ne sont pas pris en charge. Le principe était de mettre 3 extensions qui sont les plus utilisés et de les faire fonctionner correctement dans le programme.

Au lieu de black-lister mille et une extension, j'ai préféré choisir ces 3 extensions et rejeté le reste, le serveur m'a remercié pour cela.

### 5.2 Gestion des Déconnexions

Le **défi** : Une déconnexion imprévue peut interrompre le processus.

La **solution** : Implémentez des mécanismes qui détectent et gèrent les déconnexions automatiquement.

Je ne l'ai peut-être jamais cité dans ce rapport, mais j'avais des difficultés à trouver un moyen de faire comprendre au serveur que le client s'était déconnecté, la première connexion fonctionnait, je me déconnecte, je me reconnecte au serveur, tout fonctionne, mais lorsque je voulais effectué une troisième fois la connexion au serveur, celui-ci ne m'indiquait pas sur le terminal que le client s'était connecté, et l'interface graphique plantait.

J'ai peut-être passé 2-3 heures à comprendre comment cela fonctionnait..

```
akif@MacBook-Pro-de-Akif SAE3.02 % python3 serveur.py 127.0.0.1 4200
Serveur en attente de connexion...
Une connexion est en cours...
Le client d'IP ('127.0.0.1', 62357) s'est connecté
Test 1
Test 2
La connexion a prit 0.0005 seconde(s) à s'établir avec le client.
```



Voici le résultat maintenant :

Ici, le Test 1 et 2 font parti du débogage, j'ai mis ces lignes pour voir ou est-ce que le programme plantait au bout de la troisième connexion.

Je vais vous les laisser pour que vous voyez par vous même, de base j'en avais mis 5, oui, 5.

Cependant, j'ai enfin trouvé, et la réponse est humiliante..

J'ai remplacé un 'break' par un 'return' comme ci-dessous :

```
def Reception(client):  
    # Fonction pour recevoir des messages du client  
    while True:  
        try:  
            requete_client = client.recv(500) # Réception de données  
            if not requete_client: # Si la connexion est perdue  
                print("Aucun client connecté. En attente d'une connexion...")  
                return
```

Je ne sais pas pourquoi les deux premiers fonctionnait, mais apparemment le fait de mettre un 'return' marche. Et maintenant, je peux me déconnecter et me reconnecter à l'infini.

## 6. Améliorations Futures

### 6.1 Authentification

J'avais une idée qui m'était venu lors de la création du programme, lorsque je me connecté avec plusieurs clients, le serveur me disait juste l'adresse IP et le port de la personne qui s'était connecté.

J'aurai bien voulu faire un système d'authentification, où, au moins, le client peut mettre son nom et peut être reconnu avec un nom et pas juste avec un port qui change, comme ci-dessous :

#### Client 1 :

```
Le client d'IP ('127.0.0.1', 62357) s'est connecté
```

#### Client 2 :

```
Le client d'IP ('127.0.0.1', 62371) s'est connecté
```

C'est donc de cela dont je parlais tout à l'heure. Ici, seulement le port change, rien d'autre. La prochaine fois, mettons un nom à la connexion.

### 6.2 Interface de gestion serveur

Une fois l'authentification faite, j'aurai aimé pouvoir développer un système pour surveiller et gérer les connexions qui sont actives. Vu qu'on leur a refiler des noms, on peut mieux choisir qui éjecté si l'un d'eux nous a envoyé un fichier en PHP..

Il faudrait donc ajouter une nouvelle interface graphique pour le serveur.

## 7. Analyse des Parties Importantes des Programmes

### – Affichage et Boutons –

```
# Paramètres de base
self.setWindowTitle("Le serveur de tchat") # Nom de la fenetre
self.resize(400,200) # redimensionnement de la taille
self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # Connexion avec le serveur
self.setCentralWidget(widget) # ligne qui sert à centralisé la variable widget sur le self

self.label_serveur = QLabel("Serveur") # Label
self.text_serveur = QLineEdit(serveur) # Button Text

self.label_port = QLabel("Port") # Label
self.text_port = QLineEdit(port) # Button Text

self.demarrage = QPushButton("Connexion") # Button Démarrage
self.demarrage.setStyleSheet("background-color:green; border-radius: 3px")
self.arret = QPushButton("Déconnexion")

self.label = QLabel('No file selected') # Mettre le fichier
self.upload_button = QPushButton('Upload')
self.upload_button.setDisabled(True)

self.arret.setDisabled(True)
self.affichage = QLabel("") # Button affichage
self.rebours = QLabel() # Compte à rebours

self.quit = QPushButton("Quitter") # Button Quitter
```

Ici, vous pouvez voir tout les boutons, ou label utilisé pour venir à nos fins. On peut aussi choisir le titre de notre fenêtre ou sa taille.

Tout est déjà commenté, passons à autre chose.

### – Bouton Démarrage –

```
def fonction_demarrage(self):
    try:
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # Recréer le socket
        serveur = self.text_serveur.text()
        port = int(self.text_port.text())

        self.sock.connect((serveur, port))
        time.sleep(0.5)
        self.affichage.setText("Connexion au serveur réussie.")

        self.demarrage.setDisabled(True) # Trouver grâce à une vidéo Youtube --- Sers à rendre non accessible une fois connecté de reappuyer sur 'Connexion'
        self.demarrage.setStyleSheet("background-color:grey; border-radius: 3px")

        self.arret.setDisabled(False)
        self.arret.setStyleSheet("background-color:red; border-radius: 3px")

        self.upload_button.setDisabled(False)

        self.running = True # Juste une variable booléenne pour dire quand s'arrêter aux threads

        self.envoi = Thread(target=Envoie, args=(self.sock, lambda: self.running)) # lambda --> permet aux threads d'évaluer dynamiquement tout changement de valeur.
        self.recep = Thread(target=Reception, args=(self.sock, lambda: self.running))

        self.envoi.start()
        self.recep.start()
```

Ici, c'est la fonction qui se lance une fois avoir appuyé sur « Connexion »

J'ai donc du mettre le port en valeur numérique car sinon ça me ressortait une erreur.

J'ai aussi trouvé une option en regardant une vidéo sur [Youtube](#), l'option sert à **désactiver** les boutons au moment où ils nous servent à rien. Comme le bouton « **Connexion** » après une connexion, ça évite aussi les mauvaises manipulations côté client. On change aussi les couleurs des boutons pour un rendu plus esthétique.

### – Bouton Arrêt –

```
def fonction_arret(self):
    self.affichage.setText("Déconnexion en cours...")
    self.running = False # Arrêt de la boucle dans le thread.
    self.sock.close() # Déconnexion du client
    # time.sleep(2) -- Mauvaise idée :(
    self.affichage.setText("Vous êtes déconnecté du serveur.")
    self.arret.setDisabled(True)
    self.arret.setStyleSheet("background-color:grey; border-radius: 3px")
    self.demarrage.setDisabled(False)
    self.demarrage.setStyleSheet("background-color:green; border-radius: 3px")
    self.upload_button.setDisabled(True)
```

### – Bouton Upload –

Je vais vous montrer seulement une partie car la fonction est trop longue..

```
# Enregistrer le fichier txt (le fichier changera de nom aussi lors de l'envoi (fichier.txt)).
try:
    with open(filename, 'r') as f:
        contenu = f.read()
        self.rebours.setText(contenu)
        # Vérifier l'extension du fichier
        extensionDuFichier = os.path.splitext(contenu)[1]

        # Exécute le fichier mis par l'utilisateur
        if extensionDuFichier == '.py':
            # Fonction subprocess() trouvée grâce à Chat GPT
            resultat = subprocess.run(
                ['python3', contenu], # Commande pour exécuter le fichier
                capture_output=True, # Capture la sortie
                text=True # Garde la sortie en format texte
            )
            final = resultat.stdout.strip() # Prends le résultat de la sortie
        elif extensionDuFichier == '.txt':
            final = f.read()
        elif extensionDuFichier == '.c':
            # Un peu fait à l'arrache mais en gros on compile le fichier avec 'gcc' et on l'exécute
            resultat = subprocess.run(
                ['gcc', contenu], # Commande pour exécuter le fichier
                capture_output=True, # Capture la sortie
                text=True # Garde la sortie en format texte
            )
            # celui-ci va nous créer un fichier qui s'appelle a.out et puis on va l'exécuter
            resultat = subprocess.run(
                ['./a.out'], # Commande pour exécuter le fichier
                capture_output=True, # Capture la sortie
                text=True # Garde la sortie en format texte
            )
            final = resultat.stdout.strip() # Prends le résultat de la sortie
        else:
            # Le serveur n'accépte pas les autres extensions
            self.affichage.setText("Vous avez été éjecté du serveur par le serveur. \nVeuillez mettre uniquement les extensions suivantes : \n- .txt \n- .py \n- .c")
            self.running = False
```

Commençons par l'ouverture du fichier :

```
with open(filename, 'r') as file: # le 'with' servira à fermer le fichier sans qu'on le demande
```

Comme noté en commentaire, le « with » va nous faciliter la tâche, une fois le fichier lancé, pas besoin de penser à le refermer une fois l'avoir ouvert.

Extension du fichier :

```
# Vérifier l'extension du fichier  
extensionDuFichier = os.path.splitext(contenu)[1]
```

Ici, on ne va pas prendre le nom du fichier mais plutôt son extension avec « splitext » le [1] signifie qu'on prend la deuxième partie du moment car on commence de 0.

Exemple : `client.py.hoe.rez`

Ici, si on tape [3], par exemple, « extensionDuFichier » aura pour valeur « rez ».

Passons au traitement des différentes extensions..

Abordons tout d'abord, Python.

```
# Exécute le fichier mis par l'utilisateur  
if extensionDuFichier == '.py':  
    # Fonction subprocess() trouvé grâce à Chat GPT  
    resultat = subprocess.run(  
        ['python3', contenu], # Commande pour exécuter le fichier  
        capture_output=True, # Capture la sortie  
        text=True # Garde la sortie en format texte  
    )  
    final = resultat.stdout.strip() # Prends le résultat de la sortie
```

Si l'extension du fichier est égale à .py, on va utiliser **Subprocess**.

SubProcess sert à exécuter dans le terminal ce que lui demande avec l'option « subprocess.run », mais pas forcément obligé de l'affichage directement **DANS** le terminal, on peut stocker cette valeur dans une variable

Ici, on souhaite qu'il exécute « python3 » suivi du 'contenu' qui est autre que notre **fichier test**.

Donc, en gros c'est comme si j'exécuté le programme moi même et le rediriger quelque part d'autre, mais sauf que la c'est le programme qui le fait pour moi. Le reste est expliqué dans les commentaires.

#### – Le fichier texte –

```
elif extensionDuFichier == '.txt':  
    final = file.read()
```

Lire le **fichier**, et le mettre dans la variable 'final'.

#### – Fichier C –

C'est là où cela devient intéressant car la méthode pour lire un fichier C est beaucoup plus complexe que je ne le pensais.

```
elif extensionDuFichier == '.c':  
    # Un peu fait à l'arrache mais en gros on compile le fichier avec 'gcc' et on l'exécute  
    resultat = subprocess.run(  
        ['gcc', contenu], # Commande pour exécuter le fichier  
        capture_output=True, # Capture la sortie  
        text=True # Garde la sortie en format texte  
    )  
    # celui-ci va nous créer un fichier qui s'appelle a.out et puis on va l'exécuter  
    resultat = subprocess.run(  
        ['./a.out'], # Commande pour exécuter le fichier  
        capture_output=True, # Capture la sortie  
        text=True # Garde la sortie en format texte  
    )  
    final = resultat.stdout.strip() # Prends le résultat de la sortie
```

Lorsqu'on souhaite afficher la sortie d'un fichier C, il faut le compiler à l'aide de la commande '**gcc**' suivi du **nom du fichier**.

Une fois compilé, il crée un fichier appelé « **a.out** », grâce à cela, on peut directement exécuter le fichier ( **./a.out** ) et récupérer la sortie !

Voilà les options et les explications de mon programme, j'ai mis les sources des options un peu spécial en commentaire dans mon code, j'ai aussi directement mis leurs utilités, j'ai essayé de commenter un maximum pour que vous comprenez au mieux le programme !

# FIN

Cette bande jaune signifie que c'est la fin de mon rapport.

J'espère avoir été clair sur tout les aspects, j'ai essayé de détaillé au maximum et de pas trop en faire non plus.

Merci d'avoir lu mon rapport !