Introduction Réseau

1 Introduction

L'objectif de cette série d'exercice est de vous proposer une introduction à l'informatique en réseau, et plus spécifiquement à l'utilisation de TCP/IP`https://fr.wikipedia.org/wiki/Transmission_Control_Protocol`. Je vous encourage vivement à passer un peu de temps sur la première partie du TP, où l'on utilisera TCP/IP *à la main*.

TCP/IP est un protocole[1] qui se caractérise par sa fiabilité : les messages sont échangés lors d'une session, et toute erreur est détectée après un certain temps d'attente (*timeout*). L'intérêt pour nous est qu'en utilisant ce protocole, nous pouvons nous reposer sur cette fiabilité, plutôt que devoir gérer les pertes de messages à la main. Pour ces raisons, il s'agit d'un des deux protocoles les plus utilisés pour les échanges sur internet (l'autre étant UDP, plus rapide mais qui n'offre aucune garantie de fiabilité).

De notre point de vue, pour utiliser un tel protocole, il nous suffit de dire à l'ordinateur « Je veux me connecter avec TCP », et l'ordinateur nous donne alors de quoi envoyer et recevoir des messages.

1.1 Organisation

Ce TP est un TP de découverte. Certes, il y a quelques questions à implémenter en deuxième partie, mais le plus important est de vous *familiariser* avec les éléments présentés. Pour cela, la Section 2 a pour but de vous faire utiliser une session TCP à la main. Il est important de bien comprendre cette partie *avant* de procéder tête baissée dans la partie Python (Section 3)

2 Telnet et la communication à la main

Telnet est à la fois un protocole qui permet de communiquer avec un ordinateur distant (un ancêtre de SSH, d'une certaine manière), et un programme qui implémente ce protocole. Aujourd'hui, telnet (le programme comme le protocole) n'est plus trop utilisé pour son usage initial en raison de failles de sécurité, mais le programme reste très utilisé de manière détournée, pour manipuler des sessions TCP *à la main*.

Dans cette première partie, l'idée est donc d'utiliser telnet (le programme) pour bien comprendre comment va fonctionner le tchat que nous allons programmer dans les parties suivantes.

2.1 Établir une connexion

Nous allons commencer par nous connecter au serveur du tchat en utilisant telnet. Pour cela, il suffit de taper ce qui suit :

```
$ telnet vassor.org 12345
Trying 141.95.162.229...
Connected to vassor.org.
Escape character is '^]'.
```

Pour l'adresse, vous pouvez utiliser ou bien `"vassor.org"`, ou bien `82.67.122.4`. Si vous donnez le nom de domaine, la bibliothèque fera automatiquement une requête DNS[2] pour obtenir l'adresse IP qui correspond.

[1] Il y a là un petit raccourci. En fait IP et TCP sont deux protocoles différents : le protocole TCP utilisant le protocole IP de manière sous-jacente. On retrouve cette distinction quand on se connecte : l'adresse IP est un paramètre du protocole IP, là où le port est un paramètre du protocole TCP.

[2] `https://fr.wikipedia.org/wiki/Domain_Name_System`

Pour le port, il faudra utiliser 12345 qui correspond au port sur lequel le programme du serveur écoute. Si vous utilisez un autre port, ou bien vous tomberez sur un port qui n'est pas ouvert, ou bien vous tomberez sur un autre service (par exemple, si vous contactez le port 80, vous trouverez un serveur Web[3]).

2.2 Communiquer avec le serveur

Tout ce qui est écrit est ensuite envoyé (ligne par ligne) au serveur du tchat. Par exemple, si on tape `Bonjour`, le serveur recevra de son côté un message de 8 octets, chacun correspondant à un caractère du message, plus le retour à la ligne.

Essayons donc :

```
$ telnet vassor.org 12345
Trying 141.95.162.229...
Connected to vassor.org.
Escape character is '^]'.
Bonjour
"Failure"
Connection closed by foreign host.
$
```

On voit alors que le serveur nous renvoie un message (`"Failure"`); que la connexion est ensuite coupée par le serveur (`Connection closed by foreign host.`); et que telnet termine donc.

Il y a deux choses à distinguer ici :

- du point de vue de TCP, tout fonctionne à merveille : les messages ont été transmis (que ce soit le `Bonjour` qu'on a envoyé au serveur, comme le `"Failure"` que le serveur nous a renvoyé), puis l'une des deux parties (le serveur en l'occurrence) a mis fin à la session;
- du point de vue du serveur, il a reçu un message qu'il n'a pas compris, et a donc préféré clôturer la communication plutôt que de s'entêter.

Le pseudo-code du serveur ressemble à l'Algorithme 1.

```
1 Supposons qu'un client est connecté et a une session S
2 recevoir msg
3 si msg compris alors
4 |   traiter le message...
5 sinon
6 |   envoyer "Failure" sur S
7 |   clôturer S
8 fin
```

Algorithme 1 : Pseudocode du serveur

2.3 Protocole du tchat

Avant de pouvoir continuer, il nous faut donc expliciter précisément les messages qui sont attendus par le serveur. C'est-à-dire le protocole de communication du tchat.

> J'insiste sur la distinction entre le protocole de l'application que l'on écrit (à savoir le tchat), et le protocole de communication sous-jacent (à savoir TCP/IP). Utiliser TCP, c'est ce qui nous permet d'utiliser des commandes comme **recevoir** et **envoyer**; là où protocole du serveur est la manière dont on utilise ces commandes.

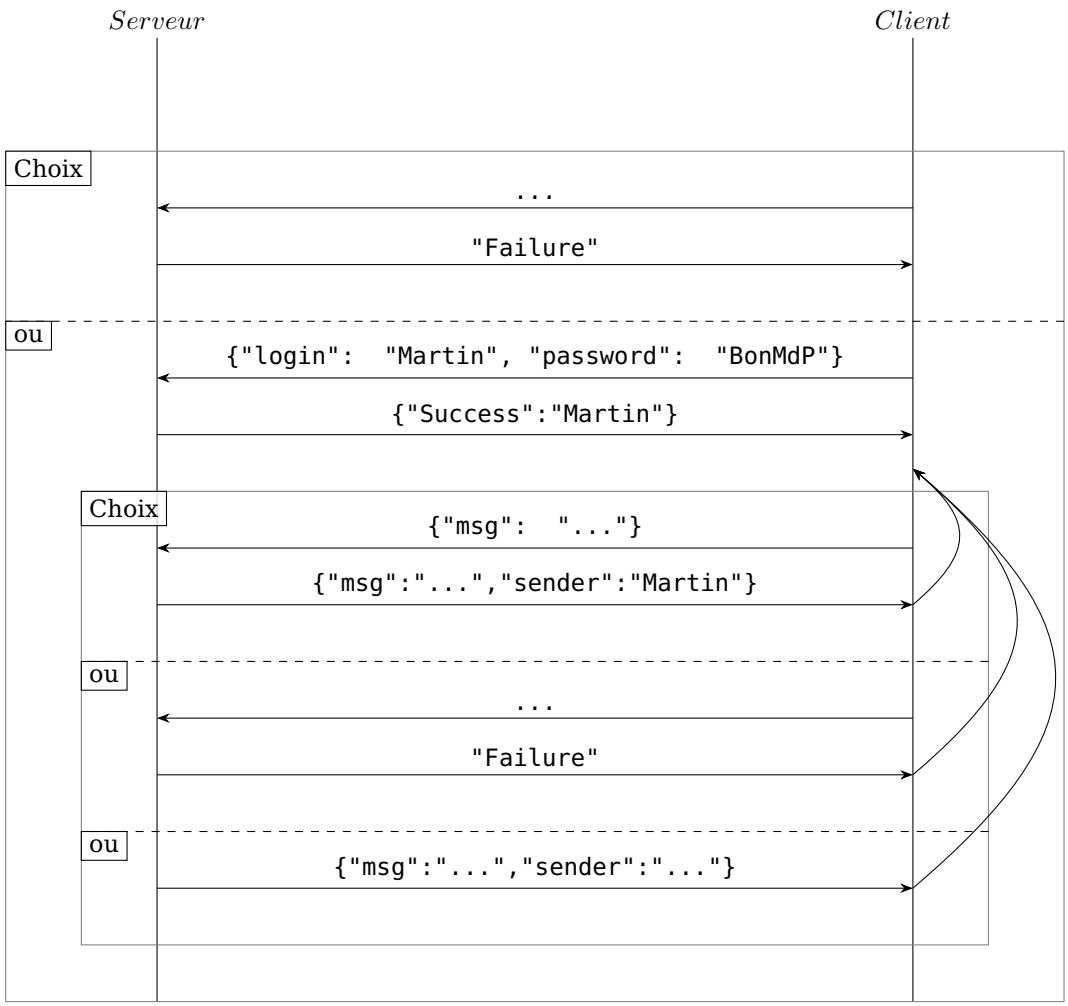

Figure 1: Protocole pour le tchat

La Figure 1 représente le protocole.
On voit que tout d'abord, deux choses peuvent se produire :

- ou bien le client tente de se connecter et échoue (que ce soit parce que le mot de passe n'est pas le bon, ou parce que le message n'est pas bien formé); auquel cas le serveur renvoie `"Failure"` et la session s'arrête (c'est ce qui s'est passé quand on a envoyé `Bonjour` au début);
- ou bien le client envoie correctement ses identifiants de connexion, auquel cas le serveur répond avec `{"Success": "LOGIN"}`, où `LOGIN` est l'identifiant de l'utilisateur.

Essayons de voir ce qu'il se passe si, dans telnet, on se connecte correctement[4] :

```
$ telnet vassor.org 12345
Trying 141.95.162.229...
Connected to vassor.org.
Escape character is '^]'.
{"login": "Martin", "password": "..."}
{"Success": "Martin"}
```

On observe bien que cette fois, le serveur nous a répondu avec un message de succès; et surtout, la session n'est pas finie, on est toujours en ligne avec le serveur ! Regardons donc la suite du protocole, dans la Figure 1.

On observe qu'en cas de succès d'authentification, 3 choix sont offerts :

- la première possibilité est d'envoyer un message. Pour cela, il faut envoyer une chaîne de caractères de la forme `{"msg": "MSG"}`, où `MSG` est le message à publier. Dans ce cas là, naturellement, le serveur publie le message, que nous recevons donc en réponse (ainsi que toutes les autres personnes connectées). La réponse est de la forme `{"msg": "MSG", "sender": "LOGIN"}`, où `MSG` est le message publié (en l'occurrence le notre), et `LOGIN` le nom de l'utilisateur·ice qui a envoyé le message (en l'occurrence notre identifiant).
- la seconde possibilité est d'envoyer quelque chose qui ne correspond pas à un message bien formé, auquel cas le serveur nous répond avec `"Failure"`, qui indique une erreur qu'une erreur s'est produite.
- la troisième possibilité est qu'un·e autre utilisateur·ice publie un message, que le serveur nous envoie alors sans que nous n'ayons rien à faire. Dans ce cas, nous recevons un message `{"msg": "MSG", "sender": "LOGIN"}` (avec `MSG` et `LOGIN` comme indiqué précédemment).

Dans tous les cas, ces trois possibilités se produisent en boucle.

D'un point de vue formel, le protocole ne correspond pas exactement à ce qui est présenté là, car plusieurs participant·e·s peuvent interagir et les messages peuvent donc apparaître dans des ordres différents. Par exemple, on peut tenter d'envoyer un message, puis recevoir un message publié par un tiers, puis seulement ensuite le message qu'on a envoyé. La présentation faite ici est toutefois suffisamment proche de la réalité pour que ça ne soit pas gênant *dans notre cas*.

Essayons donc d'envoyer un message à la main :

```
$ telnet vassor.org 12345
Trying 141.95.162.229...
Connected to vassor.org.
Escape character is '^]'.
{"login": "Martin", "password": "..."}
{"Success": "Martin"}
{"msg": "Bonjour"}
{"msg":"Bonjour","sender":"Martin"}
```

[3] Vous pouvez vous inspirer de l'exemple donné sur cette page `https://fr.wikipedia.org/wiki/Hypertext_Transfer_Protocol#HTTP_1.0` pour essayer d'obtenir à la main la page `/index.html` du serveur.

[4] Bien évidemment, utilisez les identifiants que je vous ai envoyé.

Super, le message qu'on a envoyé a bien été reçu ! D'ailleurs, si un·e autre utilisateur·ice envoie un message par la suite, il est bien affiché (ici, envoyé par `Martin2`), c'est la troisième option de la boucle :

```
$ telnet vassor.org 12345
Trying 141.95.162.229...
Connected to vassor.org.
Escape character is '^]'.
{"login": "Martin", "password": "..."}
{"Success": "Martin"}
{"msg": "Bonjour"}
{"msg":"Bonjour","sender":"Martin"}
{"msg":"Comment vas—tu ?","sender":"Martin2"}
```

Vous pouvez bien entendu démarrer plusieurs fois telnet en parallèle et vous connecter plusieurs fois simultanément pour vous envoyer à vous-même des messages depuis plusieurs sessions (vous pouvez aussi vous coordonner avec votre voisin·e de table).

3 Client en Python

Dans la section précédente, nous avons vu comment interagir *à la main* avec le serveur du tchat. Bien entendu, tout faire à la main est difficile. Dans notre cas, il faut écrire les messages bien-formés manuellement, et d'autres protocoles de la vie réelle sont encore plus compliqués. En pratique, on va donc vouloir écrire un *client*, c'est-à-dire un programme (ici en Python) qui rend facile l'utilisation d'un protocole donné (ici, le protocole du tchat).

3.1 Fichiers fournis

Le fichier fourni contient déjà une grande partie du code du client :

- le code qui gère l'interface utilisateur
- puisqu'on peut envoyer et recevoir des messages simultanément, une partie du code gère la *simultanéité* de ces évènements (on parle de *concurrence*).

Tout ce qu'il nous reste à faire, c'est de gérer l'interface du programme avec le serveur, via une connexion TCP. Le code que nous allons écrire doit interagir avec le code fourni; en particulier, même si le rôle du TP n'est pas d'étudier l'interface utilisateur, ni la concurrence, nous allons devoir adopter quelques conventions.

Afficher un message : L'interface utilisateur fournie gère déjà pas mal de choses de manières transparente. Le seul point sur lequel nous allons devoir explicitement interagir avec l'interface sera pour *demander l'affichage de texte dans la zone de texte*. Pour cela, il faut disposer d'une instance de `MessageWidget` (objet qui gère la zone de texte). Pour ajouter du texte, il faudra *envoyer un message* à cet objet, en appelant : `widget.post_message(MessageWidget.NewMessage(Texte à afficher))`, où `widget` est notre instance de `MessageWidget`.

Utiliser des coroutines : Puisque certaines parties de notre programmes doivent pouvoir s'exécuter simultanément, nous allons utiliser des *coroutines*, qui sont une manière d'exécuter des morceaux de code *en arrière-plan*, pendant qu'une autre partie s'exécute *au premier-plan*. Nous n'aurons malheureusement pas le temps d'entrer dans le détail de la programmation concurrente, ce qui nécessiterait au moins un semestre à des programmeur·euse·s chevronné·e·s.

Dans le cadre de ce TP, on rencontrera les mots-clefs `async` et `await` à différents endroits. Il faudra en particulier penser à les indiquer à certains moments (voir dans les questions plus bas). Sans rentrer dans les détails, ces mots-clefs servent à gérer la mise en arrière-plan de certaines parties du code.

3.1.1 Organisation générale du client

Lorsqu'il est lancé, le client commence par demander des informations de connexion à l'utilisateur (Figure 2), et appelle ensuite la fonction `connection` (que vous aurez à compléter pour établir la connexion). Ensuite, le programme passe à l'interface principale qui comprend un champ de texte où les messages reçus seront affichés, et un champ libre, qui permet d'envoyer des messages (Figure 3).

Dans l'état fourni, au lieu de pouvoir envoyer et recevoir des messages, le client affiche un message par seconde dans la zone de texte, ainsi que les messages que l'on demande d'envoyer.

Pour la gestion des réceptions de messages, vous aurez à compléter la fonction `get_new_message`. Pour la gestion des envois, vous aurez à compléter la fonction `send_message`.

3.2 Établir la connexion

Notre première étape consiste à établir la connexion. Dans telnet, lorsque nous démarrons le programme, nous voyons quelques lignes :

```
$ telnet vassor.org 12345
Trying 141.95.162.229...
Connected to vassor.org.
```

Ces lignes correspondent à l'établissement d'une session TCP entre notre ordinateur local et le serveur distant. D'un point de vue pratique, nous aimerions obtenir un ou plusieurs *objet(s)* python qui nous permettent de lire les messages en attente et d'envoyer des messages.

La manière la plus primitive serait d'utiliser la bibliothèque `socket`[5], qui nous donne un objet `socket`. La difficulté d'utiliser cette bibliothèque est... qu'elle est très primitive : il nous faudrait alors gérer la réception de messages à la main. Sachant que les messages peuvent être découpés en plusieurs parties envoyés séparément, ou au contraire, plusieurs messages peuvent être assemblés et envoyés ensemble; il nous faudrait alors regarder les caractères envoyés un par un, puis attendre d'obtenir quelque chose qui ressemble à un message complet, etc... Bref, c'est tellement compliqué que les développeurs de Python ont écrit une bibliothèque qui fait tout ce travail pour nous : `asyncio`[6], qui gère de manière asynchrone (comprendre : de manière concurrente) les entrées et sorties, y-compris les entrées et sorties sur le réseau. Nous allons donc utiliser cette bibliothèque.

Pour établir une connexion, il suffit la fonction `open_connection` fournie par la bibliothèque. Cette fonction prend plusieurs arguments, mais nous lui en donnerons que deux : l'adresse du serveur à contacter, et le port à contacter. La bibliothèque devinera toute seule qu'il s'agit d'une connexion en TCP à établir.

La fonction `open_connection` est une coroutine, il vous faudra l'appeler à l'aide de `await` : `... = await asyncio.open_connection(...)`.

Cette fonction renvoie deux objets : un `StreamReader` et un `StreamWriter`. Ces deux objets encapsulent le `socket` que nous aurions eu en utilisant la bibliothèque `socket`. Le premier objet nous permet *d'écrire* des messages et le second *d'en recevoir*. L'autre intérêt est que ces objets gèrent tout ce qui concernent le buffering[7] des communications. Avec ces objets, nous pourrons simplement dire *recevoir une ligne* et *envoyer telles données*.

Consigne

Complétez la fonction `connection` afin d'établir une connexion. *Il est important que la fonction renvoie le `StreamReader` et le `StreamWriter` obtenus.* (2 lignes de code)

[5] https://docs.python.org/3/library/socket.html
[6] https://docs.python.org/3/library/asyncio.html
[7] https://fr.wikipedia.org/wiki/Memoire_tampon

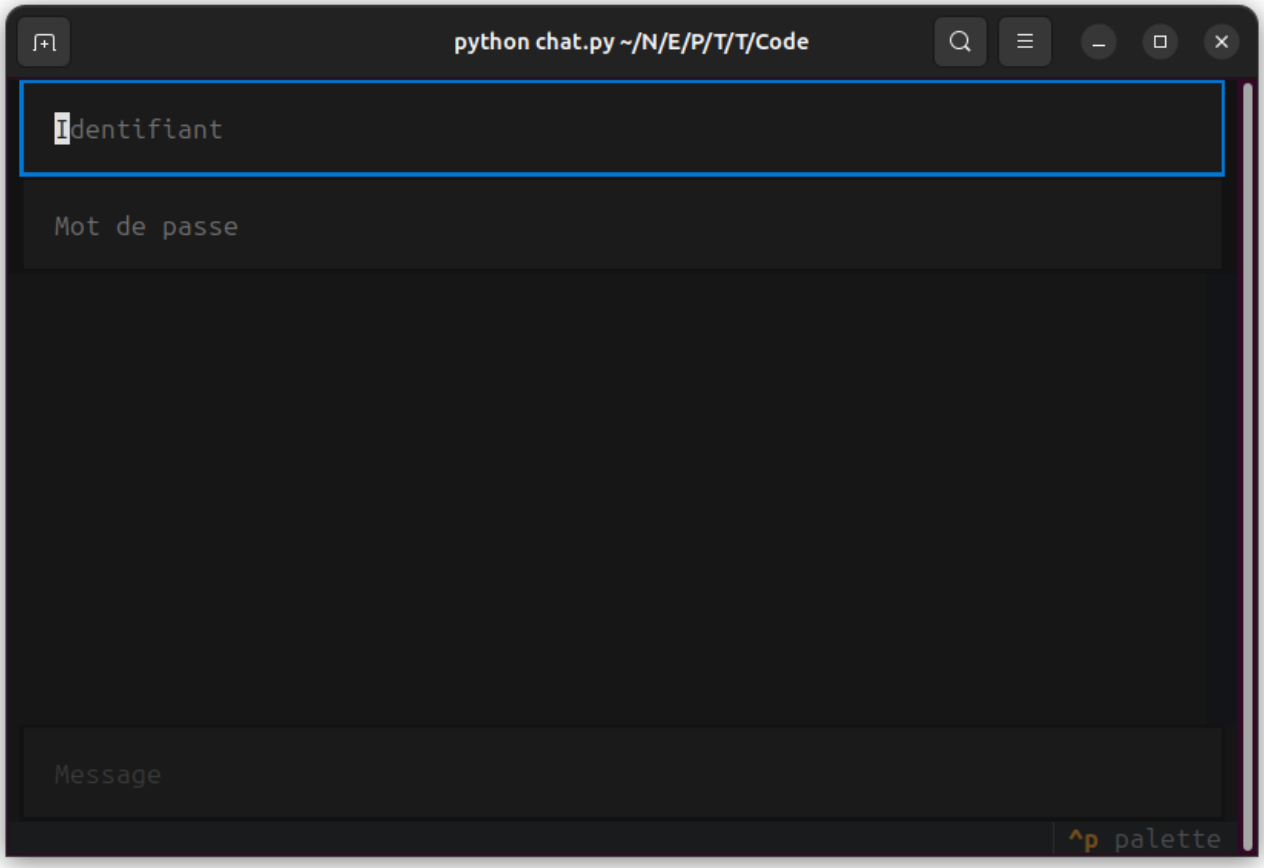

Figure 2: Écran d’authentification

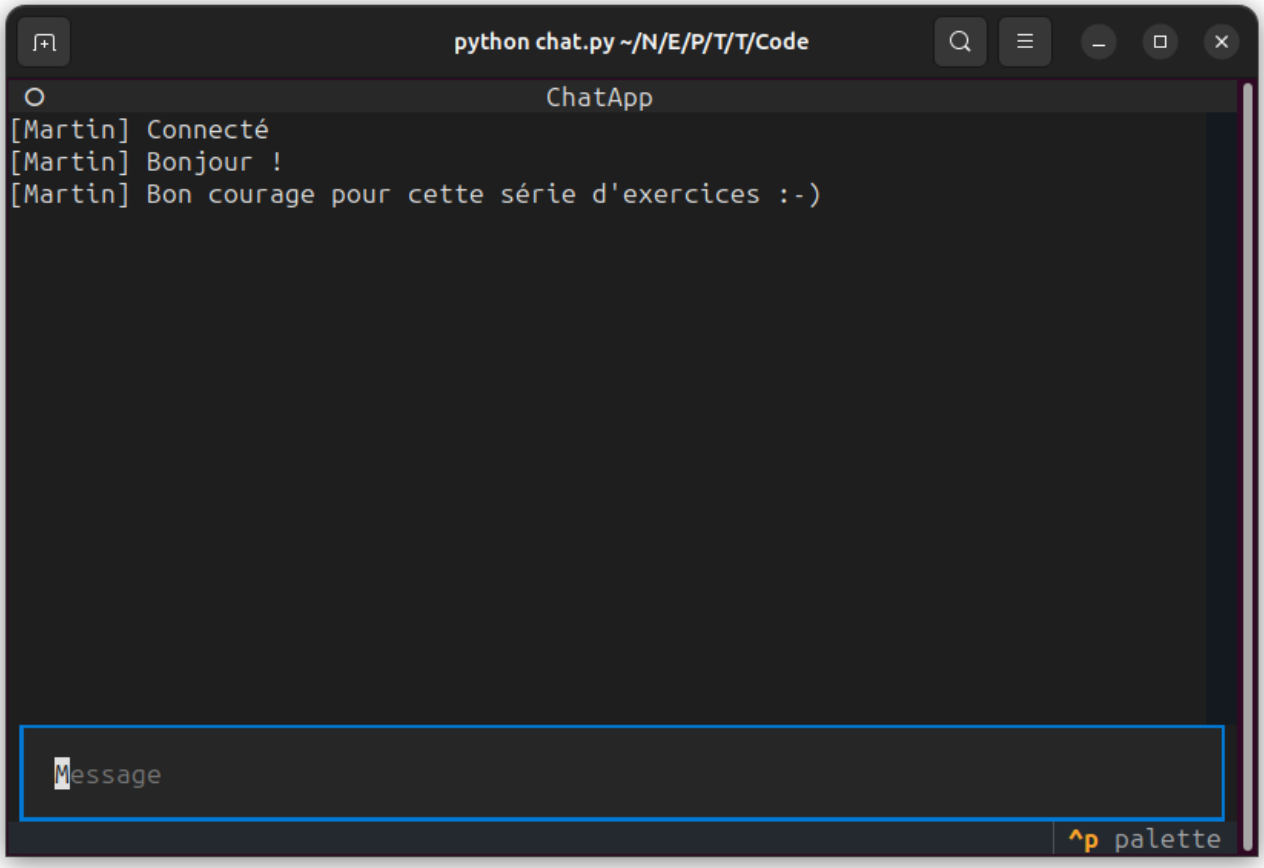

Figure 3: Écran principal

3.3 Gérer l'authentification

Avoir une connexion, c'est bien; s'en servir, c'est mieux. Nous allons maintenant nous authentifier auprès du serveur. La fonction `connection` prend 3 paramètres : le nom d'utilisateur·ice, le mot de passe (les deux sont ceux qui sont fournis par l'utilisateur·ice à l'ouverture du programme), ainsi que l'objet `log`, qui est le `MessageWidget` qui se charge de l'affichage des messages.

Le but est donc maintenant d'utiliser les informations de connexion pour envoyer un message, attendre la réponse du serveur, et en fonction, ou bien quitter le programme (en cas d'erreur), ou bien afficher que l'utilisateur·ice est bien connecté·e sur la fenêtre.

Envoyer le message d'authentification : Pour s'authentifier, il nous faut d'abord préparer le message (le mettre sous une forme qui sera comprise par le serveur), puis l'envoyer.

Le message que nous envoyons est un objet JSON[8]. Nous pourrions réutiliser le parser que nous avons écrit, mais nous allons plutôt utiliser la bibliothèque `json`[9] qui nous permet de faire facilement la conversion entre chaînes de caractères JSON et objets python.

Le serveur attend des messages *ligne-par-ligne*. Il faudra donc bien ajouter un symbole `"\n"` en fin de message.

Consigne

À l'aide de la fonction `dumps` de la bibliothèque `json`[a], préparer un message à envoyer. (2 lignes de code)

[a] `https://docs.python.org/3/library/json.html#json.dumps`

Le message étant prêt, il nous suffit maintenant de l'envoyer, c'est-à-dire l'écrire dans le `StreamWriter` obtenu lors de la question précédente. Toutefois, le `StreamWriter` s'attend à recevoir une séquence d'octets et non une chaîne de caractères. Il nous faut donc l'encoder à l'aide de `str.encode('utf-8')`, où `str` est la chaîne de caractères à envoyer.

L'envoi en lui-même se fait avec la méthode `write` du `StreamWriter`. Le seul point auquel il faut faire attention est que le `StreamWriter` et le socket sous-jacent n'ont aucune obligation d'envoyer le message tout de suite, ils peuvent (par exemple) attendre qu'il y ait plus de données à envoyer[10]. Pour forcer l'envoi, il suffit d'appeler la méthode `drain` du `StreamWriter`.

La méthode `drain` est une coroutine, il vous faudra l'appeler à l'aide de `await` : `await writer.drain()`.

Consigne

Encodez et envoyez le message d'authentification au serveur. (2 lignes de code)

Recevoir l'accusé de réception de l'authentification Si l'on suit le protocole (Figure 1), le serveur peut nous renvoyer ou bien `"Failure"`, ou bien `{"Success": "..."}`. Dans un premier temps, il va nous falloir obtenir cette chaîne de caractères, puis l'analyser dans un second temps.

Pour l'obtenir, on va simplement utiliser le `StreamReader` qui permet de lire les données reçues. Ici, on va utiliser le fait que les messages du serveur sont envoyé *ligne-par-ligne* pour demander au `StreamReader` d'accumuler des données jusqu'à ce qu'il ait une ligne complète à nous donner. Pour cela, nous allons utiliser la méthode `readline()`[11]. À l'inverse de l'envoi, les données que nous recevons ne sont pas directement une chaîne de caractères, mais simplement une suite d'octets. Nous pouvons demander à Python de les convertir à l'aide de `octets.decode('utf-8')`.

[8] c.f. Semaine 2, vous êtes maintenant des expert·e·s de JSON.
[9] `https://docs.python.org/3/library/json.html`
[10] En faisant ainsi, on peut par exemple diminuer l'utilisation du réseau en mutualisant le coût fixe qu'il y a à envoyer un message.
[11] `https://docs.python.org/3/library/asyncio-stream.html#asyncio.StreamReader.readline`

La méthode `readline` est une coroutine, il vous faudra l'appeler à l'aide de `await` :
`msg = await reader.readline(...)`.

Consigne

Recevoir et décoder le message d'accusé de réception de l'authentification. (1 ligne de code)

La chaîne de caractères obtenue est une valeur JSON (soit une chaîne JSON, en cas d'échec, soit un objet JSON en cas de succès). Nous pouvons donc demander à la bibliothèque `json` de convertir cette chaîne en objet Python (donc soit la chaîne de caractère `Failure`, soit un dictionnaire), puis d'utiliser ce qui est obtenu pour continuer ou quitter le programme.

Consigne

Utilisez `json.loads`[a] pour convertir la réponse, et utilisez le résultat pour quitter le programme (à l'aide de `log.app.exit("message")`) ou afficher un message de succès (à l'aide de `log.post_message(MessageWidget.NewMessage("message"))`). (5 lignes de code)

[a] `https://docs.python.org/3/library/json.html#json.loads`

Une fois cela fait, nous sommes prêt·e·s à préparer l'envoi et la réception de messages.

3.4 Recevoir des messages

Pour ajouter la réception de messages, il faudra compléter la fonction `get_new_message`. Cette fonction prend en argument `log`, le `MessageWidget` en charge de l'affichage des messages, et `reader`, le `StreamReader` permettant de lire les messages venant du serveur.

Cette fonction est un peu magique : elle n'est pas appelée à chaque fois qu'il y a un nouveau message; au contraire, elle est appelée une fois après la connexion. C'est la raison pour laquelle elle contient une boucle infinie. Lorsque vous lirez les données du `reader`, cela bloquera l'exécution de la boucle jusqu'à ce qu'un message soit effectivement disponible, empêchant ainsi le programme de partir en cacahuètes.

Consigne

En vous inspirant de ce que nous avons fait pour l'accusé de réception de l'authentification, complétez la boucle infinie de `get_new_message` pour recevoir et afficher les messages depuis le `reader`.

⚠ Pensez à bien utiliser `await` lorsque vous appelez `readline()`.

(3 lignes de code)

3.5 Envoyer des messages

Pour ajouter l'envoi de message, il vous faudra compléter la fonction `send_message`. Cette fonction est appelée à chaque fois que l'utilisateur·ice appuie sur la touche `Entrée` dans la ligne de messages. Cette fonction prend comme argument `text` (le message à envoyer), et `writer`, qui permet d'envoyer les messages.

Consigne

En vous inspirant de ce que nous avons fait pour l'authentification, mettez en forme le message et envoyez-le. Vous pouvez supposer que vous n'aurez pas de `"Failure"` en réponse. Toutefois, si ça

devait arriver, votre programme crashera. Pensez à bien utiliser `drain`, et à l'appeler avec `await`.
(3 lignes de code)

4 Pour aller plus loin...

Consigne

Le protocole que nous utilisons pour ce tchat est très, voire trop simple. Pouvez-identifier quelques soucis de sécurité ?

Solution:

Consigne

Vous devriez sans trop de problèmes pouvoir écrire un petit robot qui envoie périodiquement des messages (e.g. l'heure toutes les minutes).