Laboratoire de réseau

Application client/serveur de messagerie textuel instantanée

Auteurs
Coisne Valentin
Van der Veen Georgé

Année universitaire : 2025-2026

Table des matières

1 Serveur

1.a Language de programmation

Nous avons choisi d'implémenter le **serveur en Python** car c'est un **language** qui nous est **familier** et qui permet de **gérer** facilement le **réseau et l'asynchrone** grâce à des bibliothèques comme **asyncio** et **websockets**.

1.b Protocole

Arpsè quelques recherches, nous avons identifié **trois approches** principales pour le **protocole** de transport :

- Utiliser des **sockets** TCP bruts.
- Utiliser un **protocole existant**.
- Intégrer WebSocket via **FastAPI**.

1.b.a TCP brut

L'utilisation directe de **sockets TCP** permettrait de concevoir un **protocole de communication personnalisé**, offrant un **intérêt pédagogique** indéniable.
Cependant, cette approche **complexifierait le développement** et nous ne pourrions pas bénéficier des **optimisations** et de la **sécurité** que nous offrirait un **protocole standardisé**.
De plus, les **navigateurs** modernes **bloquent les connexions TCP brutes** pour des raisons de sécurité, ce qui **exclut** toute **interface web** côté client.

Bien que écarté de ce projet pour ces raisons, nous nous sommes quand même intéressé à l'implémentation de cette solution en python afin de mieux comprendre ce que faciliterait l'utilisation d'un protocle existant tel que WebSocket. Voici un **tableau non-exhaustif des différentes choses qui seraient bien plus complexes en tcp brut comparé à WebSocket** :

Fonctionnalités	**Socket**	**Websocket**
Connexion	Pour initialiser la connexion en tcp, il faut faire son **propre "handshake"**, où le **client** et le **serveur s'échangent** des **messages** pour **s'assurer que la connexion est bien établie** avant de commencer à envoyer des données.	Géré par la bibliothèque.
Intégrité des messages	Etant donné que **TCP** est un **flux continue**, certains **messages** pourraient être **fragmentés ou combinés**. Il faut donc implémenter un mécanisme pour **délimiter les messages**. Par exemple, utiliser des **séparateurs** ou **préciser la taille du message**	En **une ligne de code** le message est **reçu** en entier.
Déconnexion	Il faut détecter et **gérer les déconnexions** soit même et libérer le socket quand cela arrive.	Les déconnexions et le nettoyage sont gérés **automatiquement**.

Sécurité	Les données ne sont **pas chiffrées** par defaut.	On peut utiliser wss:// pour **chiffrer les données en TLS**(Transport Layer Security).

1.b.b Protocole existant

Utiliser un **protocole existant** comme vu au cours permettrait de bénéficier de **fonctionnalités avancées** et d'une **meilleure fiabilité**.

- **UDP** : **Exclu** en raison de son **absence de fiabilité et de gestion de connexion**.
- **QUIC** : **Intéressant** pour sa **rapidité et son multiplexage**, **mais encore peu standardisé et mal supporté en Python**.
- **WebSocket** : Protocole **connecté** et **fiable**, **compatible avec tous les navigateurs et bien supporté en Python**.

1.b.c WebSocket via FastAPI

FastAPI est un **framework web moderne pour Python**, idéal pour **créer des API RESTful(communication avec un serveur via requêtes) et des applications web**. Son **support natif des WebSockets** simplifie la mise en place d'une communication temps réel entre client et serveur. Il permet aussi d'ajouter facilement des **routes HTTP**, **non pas pour le chat en lui-même qui demande du temps réel mais pour des fonctionnalités annexes (authentification, historique, etc.)**.
Cependant, cette approche réduit la part de développement "manuel", ce qui **limite l'aspect pédagogique** du projet.

1.b.d Conclusion du choix de protocole

Nous optâmes donc de partir sur le **protocle WebSocket grâce à la bibliothèque python éponyme**.
C'est d'ailleurs ce **qu'utilise** les logiciles de communication en temps réel comme **Slack, Discord**, etc. Ce qui nous **conforte dans notre choix**.

1.c Protocole json

Voici le **protocole** que nous avons défini pour la **communication** entre le **client** et le **serveur**. Nous utilisons le **format JSON** pour structurer les messages échangés. Nous nous sommes limités aux **fonctionnalités de base** pour garder le serveur **simple et lisible**.

1.c.a Actions que le serveur peut recevoir

```
{"action": "login", "user": "AlphaXZero"}
{"action": "send_message", "message": "<message_content>"}
{"action": "create_room", "room": "<room_name>"}
{"action": "join_room", "room": "<room_name>"}
```

1.c.b Actions que le serveur peut envoyer

```
{"action": "message", "message": "<message_content>"}
{"action": "rooms", "rooms": [<room_name>, <room_name>, ...]}
```

1.d Explication du code

Nous nous sommes efforcés de garder le code du serveur **simple et lisible** tout en **implémentant les fonctionnalités principales** demandées afin de bien comprendre comment tout cela fonctionnait.
Le serveur pourrait évidemment être amélioré de bien des façons.

Tout d'abord on importe les bibliothèques nécessaires : `asyncio` pour la gestion asynchrone, `websockets` pour la communication WebSocket, et `json` pour le formatage des messages.

Nous avons ensuite un dictionnaire **clients** qui stockera le **websocket en clé** et un **dictionnaire** avec le **nom** d'utilisateur et la **salon** de chat en **valeur**.
Il faudrait probablement un deuxième dictionnaire des salons avec chaque utilisateur connecté dessus en valeur, cela permettrait de gérer plus facilement l'envoi de messages à tous les utilisateurs. Mais étant donné, le peu de clients que nous avions à gérer, nous avons préféré garder une seule structure de données pour simplifier le code.

```
clients = {} # Format: {websocket: {"user": str, "room": str}}
```

Quand un client va se connecter, on le reçoit avec la fonction handle_client.On enregistre dans le dictionnaire clients avec son websocket comme clé et on .

```
async def handle_client(websocket):
  clients[websocket] = {"user": None, "room": "general"}
```

On reçoit ensuite les messages du client chaque message est attendu au format json décrit précedemment.(on envoie également au client les salons disponibles)

```
  try:
    await send_rooms(websocket)
    async for message in websocket:
        data = json.loads(message)
        action = data.get("action")
```

On peut ensuite traiter les différentes actions que le client peut envoyer au serveur en fonction de la clé "action" du message reçu.
Par exemple, pour l'action "login", on parcours tout le dictionnaire clients pour voir si le pseudo est déjà pris puis on enregistre le nom d'utilisateur du client dans le dictionnaire clients s'il est bien disponible.

```
        if action == "login":
            username = data.get("user")
            already_taken = False
            for i in clients.values():
                if i["user"] == username:
                    already_taken = True
            if already_taken:
                await send_message(websocket, f"pseudo invalide")
            else:
                clients[websocket]["user"] = username
                await send_message(websocket,f"Bienvenue")
```

Après ça on a une petite condition qui empêche d'interragir avant de s'être connecté.
On retrouve plus loin la gestion des autres actions.

```
        elif clients[websocket]["user"] is None:
            await send_message(
                websocket, "veuillez choisir un pseudo avant de pouvoir chatter"
            )
        elif action == "join_room":...
        elif action == "send_message":...
        elif action == "create_room":...
```

Pour ce qui est de l'envoi d'un message ou des rooms à un client précis, nous avons une petite fonction qui formate le message en json avant de l'envoyer.

```
async def send_message(websocket, message):
    await websocket.send(json.dumps({"action": "message", "message": message}))
```

Par contre lorsque un utilisateur envoie un message, on doit l'envoyer à tous les clients connectés dans le même salon. On appelera la fonction broadcast qui parcourera le dictionnaire clients et enverra le message à tous les clients du salon même nous comme ça on appraitra aussi dans l'historique. A noter qu'on a pas besoin d'un parametre sender car on mettra ce dernier dans le message envoyé. `await broadcast(room, clients, f"{user}: {message}")`

```
async def broadcast(room, clients, message):
    for websocket, client in clients.items():
        if client["room"] == room:
            await send_message(websocket, message)
```

Nous pouvons enfin lancer le serveur.

```
async def main(ip, port):
    server = await websockets.serve(handle_client, ip, port)
    print(f"Serveur démarré sur ws://{ip}:{port}")
    await server.wait_closed()
asyncio.run(main("127.0.0.2", 8001))
```

2 Client

2.a Language

Nous avons choisi d'utiliser une interface web car cela permet une accessibilité facile via un navigateur, sans nécessiter d'installation supplémentaire.
De plus, JavaScript gère les WebSockets nativement et ça nous montre que grâce au protocole WebSocket, l'interface client peut être dans un language différent du serveur sans aucun problèmes et montre la séparaison frontend/backend.
Enfin, à terme, cela permettrait également de déployer l'application sur un serveur distant, accessible depuis n'importe quel appareil connecté à Internet.

2.b Explication du code

WORK IN PROGRESS

3 Répartition des tâches

Partie	Responsable
Serveur	Van der Veen Georgé
Client	Coisne Valentin
Rapport	Van der Veen Georgé et Coisne Valentin