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 Explication du code

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.

WORK IN PROGRESS

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.
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