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 langage de programmation

Nous avons choisi d'implémenter le **serveur en Python** car c'est un **langage** qui nous est **familier** et qui permet de **gérer** facilement le **réseau et l'asynchrone** grâce à des bibliothèques comme **asyncio**[1] et **websockets**[2]

1.b Protocole de transport

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

- Utiliser des **sockets** TCP bruts.
- Utiliser un **protocole de plus haut-niveau**.
- Utiliser un **framework** (encore plus haut-niveau).

1.b.a TCP brut

L'utilisation directe de **sockets TCP**[3] **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 (en tout cas simplement).

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 de 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 continu**, 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 octet avant le 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 **défaut**.	On peut utiliser wss:// pour **chiffrer les données en TLS**(Transport Layer Security).

1.b.b Protocole haut-niveau

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

- **WebTransport** [4]: **Intéressant** pour sa **rapidité et son multiplexage**, **mais encore peu standardisé et mal supporté en Python**.
- **WebSocket** [5] : Protocole **connecté** et **fiable**, **compatible avec tous les navigateurs et bien supporté en Python**.

1.b.c Framework

FastAPI[6] 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 les 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 **protocole WebSocket grâce à la bibliothèque python éponyme**.
C'est d'ailleurs ce **qu'utilisent** certains logiciels pour communication textuel en temps réel comme **Slack [7], Discord [8]**, etc. Ce qui nous **conforte dans notre choix**.

1.c Logging

Pour enregistrer les informations sur l'exécution de notre application, nous allons utiliser le **module intégré** en **Python** `logging`.
On le configure au début:

```
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler("server.log"),
        logging.StreamHandler()
    ]
)
logger = logging.getLogger(__name__)
```

Puis dans le code nous mettons une ligne de ce genre pour l'écrire dans le server.log.

```
logger.info("Login échoué : pseudo '%s' déjà pris.", username)
```

1.d Protocole json

Voici le **protocole** sommaire 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**. Vu que le **client** aura toujours un **salon lié** à lui, on ne trouve pas qu'une **action "quitter"** soit nécessaire, on changera juste son **salon lié**.

1.d.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.d.b Actions que le serveur peut envoyer

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

1.e 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**. On a aussi un petite liste des rooms.
(Il faudrait probablement un **deuxième dictionnaire** des **salons** en clé 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 grosse structure de données pour **simplifier** le code.)

```
clients = {} # Format: {websocket: {"user": str, "room": str}}
rooms =["general"]
```

Quand un client va se connecter, on le reçoit avec la fonction **handle_client** qui est en **asynchrone** ce qui permet de **gérer plusieurs clients en même temps sans bloquer l'exécution globale**. On **enregistre** dans le **dictionnaire** clients avec son **websocket** comme **clé** et on le met dans le **salon général par défaut**.

```
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 **précédemment.** On **envoie** également au **client** les **salons disponibles**, le mot-clé **await** permet d'attendre que le message soit envoyé avant de continuer. Tout cela se fait sans bloquer les autres clients.

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

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

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** (on pourrait opti avec un set par exemple) 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 cela, nous avons une petite **condition** qui empêche l'**utilisateur** d'**interagir** avec le serveur s'il n'est pas **connecté**.
On retrouve plus loin la gestion des autres actions. (join_room change juste la room dans le dictionnaire client)

```
        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":...
```

Les **déconnexions** sont **gérées** comme suit. Lorsqu'une erreur survient (par exemple : fermeture de la page), cela est consigné dans le **log** et le **client** est **supprimé** de notre liste.

```
  except websockets.exceptions.ConnectionClosed:
      logger.info(
          "Client déconnecté : %s",
          websocket.remote_address,
      )
  except Exception:
      logger.exception("Erreur inattendue")

  finally:
      if websocket in clients:
          del clients[websocket]
          logger.info(
              "Client supprimé : %s",
              websocket.remote_address,
          )
```

Pour ce qui est de l'**envoi d'un message** ou des **rooms** à un **client précis**, nous avons une 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, lorsqu'un **utilisateur** envoie un message, on doit l'envoyer à tous les **clients connectés** dans le même **salon**. On **appellera** la fonction **broadcast** qui parcourra le **dictionnaire clients** et enverra le message à tous les clients du salon (y compris nous), ce qui fera **apparaître** le message dans l'**historique**. **À** noter qu'on **n'a pas besoin** d'un **paramètre** (sender par exemple) 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. `asyncio.run()` démarre la **boucle événementielle** d'`asyncio`, ce qui permet ensuite d'utiliser les `await` pour gérer les opérations asynchrones comme l'envoi et la réception de messages, sans bloquer 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 langage

Nous avons choisi d'utiliser une **interface web** (**html/js/css**) car cela permet une accessibilité immédiate via un **navigateur**, sans nécessiter d'installation supplémentaire et nous permettra de mettre en application les enseignements de notre cours de PHP/HTML. Les critiques ont été entendues et partiellement comprises mais nous pensons que si nous voulons tout **gérer** en "natif", il suffirait d'ajouter une deuxième interface graphique. N'est-ce pas justement là la force des **WebSockets** ? Cela permet d'avoir plusieurs clients. On pourrait aussi imaginer un **client en CLI**. De plus, **JavaScript**[9] gère les **WebSockets** [10] nativement et ça nous montre que grâce au **protocole WebSocket** l'interface client peut être dans un langage différent de celui du serveur sans aucun problème et montre la séparation entre le **frontend** et le **backend**.
Enfin, à terme, cela permettrait également de **déployer** l'application sur un **serveur distant** (**Apache**), **accessible** depuis n'importe quel appareil connecté à **Internet**.

2.b Explication du code

Le **HTML** a été **créé** par l'**IA** puis grandement modifié et le **CSS** quant à lui est presque entièrement généré. Le reste a été fait **manuellement**.

2.b.a Connexion au serveur

Tout d'abord, nous avons un **bouton de connexion** dans le **html** qui va appeler notre **fonction `connect`** dans le **js**.
Tandis que dans notre **js** on récupère les champs **ip**, **user_name** et **port** pour ensuite envoyer l'**action "login"** au serveur avec notre nom d'utilisateur. `html`:

```
<input type="text" id="serverIp" value="127.0.0.2" required>
<input type="number" id="serverPort" value="8001" required>
<input type="text" id="username" required maxlength="20" placeholder="Votre
pseudo...">
<button type="button" class="btn btn-primary" onclick="connect()">Se connecter</
button>
```

`js`:

```
function connect() {
    const ip = document.getElementById("serverIp").value
    const user_name = document.getElementById("username").value
    const port = document.getElementById("serverPort").value
    socket = new WebSocket(`ws://${ip.trim()}:${port.trim()}`)
    socket.onopen = () => {
        socket.send(JSON.stringify({ action: 'login', user: user_name }))
        connectionPage.style.display = "none"
        chatPage.style.display = "flex"
    };
}
```

2.b.b Gestion de la reception des messages

On rajoute toujours dans la fonction connect : `socket.onmessage` qui dira à notre code comment réagir en fonction des différentes actions. Pour le moment, nous mettons juste comment le client doit gérer les messages qu'il reçoit.

```
    socket.onmessage = (event) => {
        const data = JSON.parse(event.data);
        if (data.action === 'message') {
            add_message(data.message);
        }
    }
```

Nous nous occupons donc de la réception des messages, on crée une fonction **add_message** qui édite notre **`div`** pour ajouter du **texte** dedans. On appellera cette fonction à chaque fois qu'on souhaitera ajouter des choses dans le **chat**. Sans trop rentrer dans les détails, ça ajoute le **message**, scroll tout en bas et vide le champ d'entrée

```
function add_message(message) {
    const message_input = document.getElementById("messageInput")
    const new_message = document.createElement("div")
    const message_container = document.getElementById("messagesContainer")
    new_message.textContent = message
    message_container.appendChild(new_message)
    message_container.scrollTop = message_container.scrollHeight
    message_input.value = ""
}
```

On recevra aussi les **salons** du serveur (condition rajoutée dans le **socket.onmessage**). Globalement la **gestion des salons** est gérée de la même manière que les messages, pour éviter la redondance on ne l'expliquera pas.

2.b.c Gestion de l'envoi des messages

Nous avons un **champ d'entrée** et un **bouton** dans notre **html**. Le script **js** récupère le contenu du champ une fois le bouton pressé. Le contenu est donc envoyé avec l'action **`send_message`** au **serveur** qui va le recevoir et le renvoyer ensuite à tous les **clients** comme nous l'avons vu précédemment. Ensuite tous les clients recevront le message qui sera ajouté dans le **chat** grâce au code vu juste au-dessus. `html`:

```
<input type="text" id="messageInput" placeholder="Tapez votre message..."
required>
<button type="button" onclick="send_message()">Envoyer</button>
```

`js`:

```
function send_message() {
    const message = document.getElementById("messageInput").value
    socket.send(JSON.stringify({ action: 'send_message', message: message }))
}
```

2.b.d Asynchrone

Comme on peut le voir nous n'utilisons pas vraiment l'**asynchrone** ici mais c'est parce que nous avons des **événements** à la place (`socket.onmessage`, `onclick`) qui ne bloquent pas le navigateur.

3 Répartition des tâches

Nous avons utilisé **GitHub** afin de travailler simultanément sur les parties **client** et **serveur**, puis de les **regrouper** facilement par la suite.

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

Références

[1] P. S. Foundation, *Python Asyncio Documentation*. 2025. [Online]. Available: https://docs.python.org/3/library/asyncio.html

[2] A. Augustin, “websockets: A library for building WebSocket servers and clients in Python.” [Online]. Available: https://websockets.readthedocs.io/

[3] W. R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994.

[4] IETF, “WebTransport: A new protocol for real-time communication,” *Internet-Draft*, 2023, [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-webtrans-webtransport/

[5] IETF, *The WebSocket Protocol*. 2011. [Online]. Available: https://tools.ietf.org/html/rfc6455

[6] S. Ramírez, “FastAPI: Modern, fast (high-performance) web framework for building APIs with Python.” [Online]. Available: https://fastapi.tiangolo.com/

[7] S. Engineering, “How Slack Works: Technical Overview.” [Online]. Available: https://slack.engineering/

[8] “How Discord Handles Millions of Messages in Real Time (and Doesn’t Crash)-2,” 2025, [Online]. Available: https://medium.com/@pikachuzombie2/how-discord-handles-millions-of-messages-in-real-time-and-doesnt-crash-2-2579820959e0

[9] D. Flanagan, *JavaScript: The Definitive Guide*. O'Reilly Media, 2020.

[10] M. D. Network, “WebSocket API - Web APIs | MDN.” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebSocket