

Programmation Socket en Python

Introduction

Dans la partie précédente, on a vu comment fonctionnent TCP et UDP au niveau théorique : le three-way handshake, la fiabilité de TCP, la rapidité de UDP, les ports, etc. Maintenant, on passe à la pratique. On va utiliser le module `socket` de Python pour créer nos propres serveurs et clients, envoyer des données sur le réseau, et construire un chat en temps réel.

Pourquoi les sockets ?

Un socket, c'est un point de communication entre deux machines (ou deux programmes sur la même machine). C'est la brique de base de toute communication réseau. Quand vous ouvrez un site web, votre navigateur crée un socket vers le serveur. Quand vous envoyez un message sur une app de chat, un socket est utilisé en coulisse.

En Python, le module `socket` de la bibliothèque standard donne accès directement à l'interface socket du système d'exploitation. Pas besoin d'installer quoi que ce soit.

HTTP et WebSocket : construits sur les sockets

Tous les protocoles applicatifs que vous connaissez, HTTP, WebSocket, FTP, SMTP, reposent sur des sockets TCP (ou UDP) en dessous.

HTTP fonctionne en requête-réponse : le client ouvre une connexion TCP, envoie sa requête, reçoit la réponse, puis la connexion se ferme. C'est toujours le client qui initie la communication, et la latence est élevée puisqu'il faut établir une nouvelle connexion à chaque fois. On l'utilise pour les API REST et les pages web classiques.

WebSocket résout ces limites. La connexion démarre par un handshake HTTP classique, puis le protocole "upgrade" la connexion vers WebSocket. À partir de là, la connexion TCP reste ouverte et les deux côtés peuvent s'envoyer des messages à tout moment, sans attendre de requête. C'est ce qu'on appelle une communication **full-duplex**. La latence est faible, et on l'utilise pour le chat en temps réel, les jeux en ligne ou les dashboards live.

Cas d'usage courants des WebSockets :

- Applications de messagerie instantanée (Slack, WhatsApp Web)
- Streaming de données en temps réel (cours de bourse, scores sportifs)
- Outils collaboratifs (Google Docs, Figma)
- Jeux multijoueurs en ligne
- Données IoT et capteurs

Mais avant de construire des WebSockets, il faut comprendre ce qui se passe en dessous. Et ce qui se passe en dessous, ce sont des sockets TCP. C'est exactement ce qu'on va apprendre ici.

Socket TCP vs Socket UDP

Un **socket TCP** (type `socket.SOCK_STREAM`) nécessite une connexion explicite via `connect` / `accept`. Les données sont garanties et arrivent dans l'ordre. On l'utilise pour le chat, le transfert de fichiers ou HTTP.

Un **socket UDP** (type `socket.SOCK_DGRAM`) ne nécessite aucune connexion. Il n'offre aucune garantie de livraison ni d'ordre. On l'utilise pour le DNS, le streaming vidéo ou les jeux rapides.

Créer un serveur TCP

Un serveur TCP suit toujours le même schéma : créer un socket, le lier à une adresse, écouter les connexions, puis accepter les clients un par un.

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind(("localhost", 9999))
server_socket.listen()

print("Serveur en écoute sur le port 9999...")

client_socket, client_address = server_socket.accept()
print(f"Connexion reçue de {client_address}")

client_socket.close()
server_socket.close()
```

Décomposons chaque étape :

1. Création du socket

```
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

- `socket.AF_INET` : on utilise IPv4
- `socket.SOCK_STREAM` : on veut du TCP (un flux continu et fiable de données)

2. Bind, lier le socket à une adresse

```
server_socket.bind(("localhost", 9999))
```

On associe le socket à une adresse IP et un port. Ici, `localhost` signifie que le serveur n'écoute que les connexions locales. Pour accepter des connexions depuis d'autres machines, utilisez `"0.0.0.0"` à la place.

Le port `9999` est arbitraire, vous pouvez choisir n'importe quel port entre 1024 et 65535 (les ports en dessous de 1024 sont réservés et nécessitent des droits administrateur).

3. Listen, se mettre en écoute

```
server_socket.listen()
```

Le serveur est prêt à recevoir des connexions. Vous pouvez passer un argument optionnel pour limiter le nombre de connexions en attente : `server_socket.listen(5)`.

4. Accept, accepter une connexion

```
client_socket, client_address = server_socket.accept()
```

`accept()` est **bloquant** : le programme s'arrête ici et attend qu'un client se connecte. Quand un client arrive, `accept()` retourne deux choses :

- `client_socket` : un nouveau socket dédié à la communication avec ce client
- `client_address` : un tuple (`ip`, `port`) qui identifie le client

Créer un client TCP

Le client est plus simple : il crée un socket et se connecte au serveur.

```
import socket

client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect(("localhost", 9999))

print("Connecté au serveur !")

client_socket.close()
```

`connect()` initie le three-way handshake TCP vers le serveur. Si le serveur n'est pas en écoute sur cette adresse/port, vous obtiendrez une erreur `ConnectionRefusedError`.

Tester la connexion

Pour tester, ouvrez **deux terminaux** :

Terminal 1 (serveur) :

```
python server.py
```

Terminal 2 (client) :

```
python client.py
```

Côté serveur, vous devriez voir :

```
Serveur en écoute sur le port 9999...  
Connexion reçue de ('127.0.0.1', 52314)
```

Le port du client (ici **52314**) est attribué automatiquement par le système d'exploitation. C'est ce qu'on appelle un **port éphémère**.

Envoyer des données via une connexion TCP

Se connecter c'est bien, mais l'intérêt c'est d'échanger des données. En TCP, on utilise `send()` et `recv()`.

Point important : les sockets envoient et reçoivent des **bytes**, pas des strings. Il faut encoder les chaînes avant de les envoyer, et décoder les bytes après réception.

Serveur qui reçoit un message

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind(("localhost", 9999))
server_socket.listen()

print("Serveur en écoute sur le port 9999...")

client_socket, client_address = server_socket.accept()
print(f"Connexion de {client_address}")

message = client_socket.recv(1024)
print(f"Message reçu : {message.decode()}")

client_socket.close()
server_socket.close()
```

Client qui envoie un message

```
import socket

client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect(("localhost", 9999))

client_socket.send("Salut le serveur !".encode())
```

```
client_socket.close()
```

`send()` prend des bytes en argument. `"Salut le serveur !".encode()` convertit la string en bytes UTF-8. Côté serveur, `recv(1024)` lit jusqu'à 1024 bytes depuis le socket, et `.decode()` reconvertit les bytes en string.

On y reviendra plus bas.

Serveur qui répond au client

On peut aussi faire l'inverse : le serveur envoie une réponse au client.

```
import socket

# Serveur
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind(("localhost", 9999))
server_socket.listen()

print("En attente d'un client...")
client_socket, client_address = server_socket.accept()

message = client_socket.recv(1024)
print(f"Reçu : {message.decode()}")

reponse = "Bien reçu, merci !"
client_socket.send(reponse.encode())

client_socket.close()
server_socket.close()
```

```
import socket

# Client
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect(("localhost", 9999))

client_socket.send("Salut le serveur !".encode())

reponse = client_socket.recv(1024)
print(f"Réponse du serveur : {reponse.decode()}")

client_socket.close()
```

Résultat côté client :

Réponse du serveur : Bien reçu, merci !

Créer et envoyer des données via UDP

Avec UDP, il n'y a pas de connexion à établir. On envoie directement un paquet (datagram) à une adresse. Le serveur n'a pas besoin d'appeler `listen()` ni `accept()`.

Serveur UDP

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server_socket.bind(("localhost", 9999))

print("Serveur UDP en écoute...")

data, client_address = server_socket.recvfrom(1024)
print(f"Message de {client_address} : {data.decode()}")

server_socket.sendto("Message bien reçu !".encode(), client_address)

server_socket.close()
```

Avec `recvfrom()`, le serveur reçoit à la fois les données et l'adresse de l'expéditeur. C'est indispensable en UDP puisqu'il n'y a pas de connexion : le serveur doit savoir à qui répondre.

Client UDP

```
import socket

client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

client_socket.sendto("Hello UDP !".encode(), ("localhost", 9999))

data, server_address = client_socket.recvfrom(1024)
print(f"Réponse : {data.decode()}")

client_socket.close()
```

Différences clés avec TCP

	TCP	UDP
Type de socket	<code>SOCK_STREAM</code>	<code>SOCK_DGRAM</code>
Connexion	<code>connect()</code> / <code>accept()</code> obligatoire	Pas de connexion

	TCP	UDP
Envoi	<code>send()</code>	<code>sendto(data, address)</code>
Réception	<code>recv()</code>	<code>recvfrom()</code> retourne data + adresse
Serveur	<code>listen() + accept()</code>	Juste <code>bind()</code>

Le buffer size : comprendre `recv(1024)`

Vous avez remarqué le **1024** dans `recv(1024)` ? C'est la **taille du buffer** : le nombre maximum de bytes que `recv()` va lire en une seule fois.

Pourquoi un buffer ?

Les données arrivent du réseau de façon imprévisible : par petits bouts, avec des délais variables, parfois plus vite que votre programme ne peut les traiter. Sans buffer, chaque octet reçu devrait être traité immédiatement, ce qui ralentirait énormément le programme (un appel système à chaque octet, c'est très coûteux).

Le buffer agit comme une **zone de stockage temporaire**. Le système d'exploitation y accumule les données qui arrivent du réseau. Quand votre programme appelle `recv()`, il vient piocher dans ce buffer d'un coup, au lieu de lire octet par octet. C'est le même principe qu'une boîte aux lettres : le facteur y dépose les courriers au fil de la journée, et vous les récupérez tous en une seule fois quand vous êtes prêt.

Ce mécanisme existe dans tous les langages de programmation qui font du réseau ou de la lecture de fichiers, pas seulement en Python. L'idée est toujours la même : **réduire le nombre d'allers-retours entre le programme et le système d'exploitation** pour gagner en performance.

Que se passe-t-il si le message est plus grand que le buffer ?

`recv()` ne lit que ce qui tient dans le buffer. Le reste attend dans le buffer du socket pour le prochain appel à `recv()`.

Démonstration :

```
import socket

# Serveur avec un petit buffer
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind(("localhost", 9999))
server_socket.listen()

client_socket, _ = server_socket.accept()

# On lit en plusieurs fois avec un petit buffer
chunk1 = client_socket.recv(10)
print(f"Chunk 1 : {chunk1.decode()}")

chunk2 = client_socket.recv(10)
```

```
print(f"Chunk 2 : {chunk2.decode()}")

chunk3 = client_socket.recv(10)
print(f"Chunk 3 : {chunk3.decode()}")

client_socket.close()
server_socket.close()
```

```
import socket

# Client qui envoie un long message
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect(("localhost", 9999))

client_socket.send("ABCDEFGHIIJKLMNOPQRSTUVWXYZ".encode())

client_socket.close()
```

Résultat côté serveur :

```
Chunk 1 : ABCDEFGHIIJ
Chunk 2 : KLMNOPQRST
Chunk 3 : UVWXYZ
```

Le message de 26 bytes a été lu en trois morceaux de 10, 10 et 6 bytes.

Recevoir un message complet

En pratique, quand on ne connaît pas la taille du message à l'avance, on lit en boucle jusqu'à ce qu'il n'y ait plus rien :

```
def receive_all(sock, buffer_size=1024):
    data = b""
    while True:
        chunk = sock.recv(buffer_size)
        if not chunk:
            break
        data += chunk
    return data
```

`recv()` retourne une chaîne de bytes vide (`b""`) quand le client ferme la connexion. C'est ce qui permet de sortir de la boucle.

Quelle taille choisir ?

Taille	Usage
1024 (1 Ko)	Petits messages texte, commandes
4096 (4 Ko)	Usage général, bon compromis
65535 (64 Ko)	Transfert de fichiers, gros volumes

Un buffer trop petit oblige à faire plus d'appels à `recv()`. Un buffer trop grand gaspille de la mémoire. En pratique, **4096** est un bon choix par défaut.

Chat bidirectionnel, Partie 1 : mise en place serveur/client

On va construire un petit programme de chat où le serveur et le client peuvent s'envoyer des messages à tour de rôle.

Le serveur

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server_socket.bind(("localhost", 9999))
server_socket.listen(1)

print("Serveur de chat démarré. En attente d'un client...")

client_socket, client_address = server_socket.accept()
print(f"Client connecté : {client_address}")
print("Chat prêt ! Tapez vos messages (tapez 'quit' pour quitter).")
print()
```

`setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)`. Cette ligne permet de réutiliser le port immédiatement après avoir fermé le programme. Sans ça, si vous relancez le serveur trop vite, vous obtiendrez une erreur **Address already in use**, le système d'exploitation garde le port réservé pendant quelques secondes après la fermeture.

Le client

```
import socket

client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect(("localhost", 9999))

print("Connecté au serveur de chat !")
print("Chat prêt ! Tapez vos messages (tapez 'quit' pour quitter).")
print()
```

Pour l'instant, la connexion est établie mais personne ne peut encore parler. C'est ce qu'on va ajouter dans la partie suivante.

Chat bidirectionnel, Partie 2 : activer le chat

Le principe : le serveur et le client alternent entre envoyer et recevoir. Le serveur attend d'abord un message du client, puis répond, et ainsi de suite.

Le serveur complet

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server_socket.bind(("localhost", 9999))
server_socket.listen(1)

print("Serveur de chat démarré. En attente d'un client...")

client_socket, client_address = server_socket.accept()
print(f"Client connecté : {client_address}")
print("Chat prêt ! (tapez 'quit' pour quitter)")
print()

while True:
    # Recevoir un message du client
    message = client_socket.recv(4096).decode()

    if not message or message.lower() == "quit":
        print("Le client a quitté le chat.")
        break

    print(f"Client : {message}")

    # Envoyer une réponse
    reponse = input("Vous : ")

    if reponse.lower() == "quit":
        client_socket.send("quit".encode())
        break

    client_socket.send(reponse.encode())

client_socket.close()
server_socket.close()
print("Chat terminé.")
```

Le client complet

```
import socket

client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect(("localhost", 9999))

print("Connecté au serveur de chat !")
print("Chat prêt ! (tapez 'quit' pour quitter)")
print()

while True:
    # Envoyer un message au serveur
    message = input("Vous : ")

    if message.lower() == "quit":
        client_socket.send("quit".encode())
        break

    client_socket.send(message.encode())

    # Recevoir la réponse du serveur
    reponse = client_socket.recv(4096).decode()

    if not reponse or reponse.lower() == "quit":
        print("Le serveur a quitté le chat.")
        break

    print(f"Serveur : {reponse}")

client_socket.close()
print("Chat terminé.")
```

Comment ça fonctionne

Le flux de communication est le suivant :

1. Le client tape un message et l'envoie avec `send()`
2. Le serveur le reçoit avec `recv()`, l'affiche, puis attend une réponse via `input()`
3. Le serveur envoie sa réponse avec `send()`
4. Le client la reçoit avec `recv()` et l'affiche
5. Retour à l'étape 1

Si l'un des deux tape `quit`, le message est envoyé, l'autre côté le détecte et ferme la connexion. Si la connexion est coupée brutalement, `recv()` retourne une chaîne vide, ce qui déclenche aussi la sortie de boucle.

Tester le chat

Terminal 1 :

```
python chat_server.py
```

Terminal 2 :

```
python chat_client.py
```

Exemple d'échange :

```
# Terminal 2 (client)
Vous : Salut !

# Terminal 1 (serveur)
Client : Salut !
Vous : Hey, comment ça va ?

# Terminal 2 (client)
Serveur : Hey, comment ça va ?
Vous : Bien et toi ?

# Terminal 1 (serveur)
Client : Bien et toi ?
Vous : quit
```

Limites de cette approche

Ce chat fonctionne, mais il a une limitation importante : la communication est **alternée**, pas simultanée. Le serveur ne peut pas envoyer un message tant qu'il n'a pas reçu celui du client, et inversement. C'est un modèle **half-duplex**.

Pour un vrai chat où les deux peuvent écrire en même temps (**full-duplex**), il faudrait utiliser du **threading** (un thread pour envoyer, un thread pour recevoir), un concept qu'on a déjà vu dans les cours précédents.

L'idée : on sépare l'envoi et la réception dans deux threads distincts. Un thread écoute en permanence les messages entrants, l'autre attend les inputs de l'utilisateur. Les deux tournent en parallèle, ce qui permet d'envoyer et de recevoir en même temps.

Le serveur full-duplex (`chat_server_v2.py`) :

```
import socket
import threading

def receive_messages(sock):
    while True:
        try:
```

```

        message = sock.recv(4096).decode()
        if not message:
            print("\nLe client s'est déconnecté.")
            break
        print(f"\nClient : {message}")
    except:
        break

def send_messages(sock):
    while True:
        message = input()
        if message.lower() == "quit":
            sock.send("quit".encode())
            sock.close()
            break
        sock.send(message.encode())

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server_socket.bind(("localhost", 9999))
server_socket.listen(1)

print("Serveur de chat v2 démarré. En attente d'un client...")

client_socket, client_address = server_socket.accept()
print(f"Client connecté : {client_address}")
print("Chat prêt ! (tapez 'quit' pour quitter)\n")

thread_recv = threading.Thread(target=receive_messages, args=(client_socket,))
thread_send = threading.Thread(target=send_messages, args=(client_socket,))

thread_recv.start()
thread_send.start()

thread_recv.join()
thread_send.join()

server_socket.close()
print("Chat terminé.")

```

Le client full-duplex (**chat_client_v2.py**) :

```

import socket
import threading

def receive_messages(sock):
    while True:
        try:
            message = sock.recv(4096).decode()
            if not message or message.lower() == "quit":
                print("\nLe serveur a quitté le chat.")

```

```

        break
    print(f"\nServeur : {message}")
except:
    break

def send_messages(sock):
    while True:
        message = input()
        if message.lower() == "quit":
            sock.send("quit".encode())
            sock.close()
            break
        sock.send(message.encode())

client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect(("localhost", 9999))

print("Connecté au serveur de chat v2 !")
print("Chat prêt ! (tapez 'quit' pour quitter)\n")

thread_recv = threading.Thread(target=receive_messages, args=
(client_socket,))
thread_send = threading.Thread(target=send_messages, args=(client_socket,))

thread_recv.start()
thread_send.start()

thread_recv.join()
thread_send.join()

print("Chat terminé.")

```

Avec cette version, plus besoin d'attendre son tour : les deux côtés peuvent écrire et lire en même temps. C'est un vrai chat full-duplex.

WebSockets

Le problème de HTTP pour le temps réel

HTTP fonctionne en requête-réponse. Le client demande, le serveur répond, la connexion se ferme. C'est parfait pour charger une page ou interroger une API REST. Mais pour du temps réel, c'est une contrainte majeure : **seul le client peut initier une communication**.

Imaginez un dashboard de prix en bourse. Avec HTTP classique, vous devez interroger le serveur toutes les X secondes pour savoir si les prix ont changé. C'est du **polling** : inefficace, lent, et gourmand en ressources.

Une première tentative de solution : le **long-polling**. Le client envoie une requête, le serveur la garde ouverte jusqu'à ce qu'une donnée soit disponible, puis répond. C'est mieux, mais ça reste du bricolage : chaque réponse nécessite une nouvelle requête, et la connexion TCP est recréée à chaque fois.

WebSocket résout le problème à la racine : une seule connexion TCP ouverte, et les deux côtés peuvent envoyer des messages à tout moment.

Ce que WebSocket change

	HTTP	WebSocket
Qui initie	Toujours le client	Les deux côtés
Connexion	Ouverte par requête, puis fermée	Persistante
Latence	Élevée (reconnexion à chaque fois)	Faible (connexion déjà ouverte)
Cas d'usage	API REST, pages web	Chat, live, jeux, IoT

Comment ça fonctionne

Une connexion WebSocket démarre toujours par un handshake HTTP. Le client envoie une requête HTTP ordinaire avec un en-tête spécial :

```
GET /chat HTTP/1.1
Host: localhost:8765
Upgrade: websocket
Connection: Upgrade
```

Si le serveur accepte, il répond avec un code **101 Switching Protocols**. À partir de là, HTTP est abandonné et la connexion devient WebSocket : bidirectionnelle, persistante, à faible latence.

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
```

La connexion TCP reste ouverte. Les deux côtés peuvent maintenant s'envoyer des **frames** (messages) à tout moment, sans attendre de requête.

Lien avec les sockets TCP

WebSocket repose sur TCP, exactement comme les sockets vus dans ce chapitre. La différence : WebSocket ajoute un protocole standardisé par-dessus TCP, avec un handshake, un format de messages, et une gestion des connexions. C'est pour ça qu'on peut l'utiliser depuis un navigateur web, ce qu'un socket TCP brut ne permet pas.

```
Navigation ←—— WebSocket (protocole) ——→ Serveur Python
                |
                TCP en dessous
```

Installation

La bibliothèque **websockets** est la référence en Python. Elle est construite sur asyncio.

```
pip install websockets
```

Serveur WebSocket

```
# server.py
import asyncio
import websockets

async def handle_client(websocket):
    print(f"Nouvelle connexion : {websocket.remote_address}")

    async for message in websocket:
        print(f"Reçu : {message}")
        await websocket.send(f"Echo : {message}")

    print(f"Connexion fermée : {websocket.remote_address}")

async def main():
    async with websockets.serve(handle_client, "localhost", 8765):
        print("Serveur WebSocket en écoute sur ws://localhost:8765")
        await asyncio.Future() # garde le serveur actif indéfiniment

asyncio.run(main())
```

async for message in websocket : itère sur les messages entrants. La boucle se termine automatiquement quand le client ferme la connexion. C'est l'équivalent async du **recv()** en boucle qu'on faisait avec les sockets TCP.

await websocket.send() : envoie un message au client. La connexion est déjà ouverte, pas besoin de se reconnecter.

await asyncio.Future() : crée une coroutine qui ne se termine jamais. C'est le moyen idiomatique de garder un serveur asyncio actif indéfiniment sans occuper le CPU.

Client WebSocket

```
# client.py
import asyncio
import websockets

async def main():
    uri = "ws://localhost:8765"
```



```
async with websockets.connect(uri) as websocket:
    await websocket.send("Bonjour !")
    reponse = await websocket.recv()
    print(f"Réponse : {reponse}")

    await websocket.send("Comment ça va ?")
    reponse = await websocket.recv()
    print(f"Réponse : {reponse}")

asyncio.run(main())
```

async with websockets.connect(uri) : ouvre la connexion WebSocket et la ferme proprement à la sortie du bloc, même en cas d'erreur.

await websocket.recv() : attend le prochain message du serveur. Bloquant de façon coopérative : l'event loop peut tourner pendant l'attente.

Terminal 1 :

```
python server.py
```

Terminal 2 :

```
python client.py
```

Résultat côté client :

```
Réponse : Echo : Bonjour !
Réponse : Echo : Comment ça va ?
```

Application de chat en temps réel

L'exemple echo est utile pour comprendre, mais WebSocket prend tout son sens quand le serveur diffuse des messages à plusieurs clients simultanément.

Le serveur maintient une liste de tous les clients connectés. Quand un client envoie un message, le serveur le retransmet à tous les autres.

server.py

```
import asyncio
import websockets

clients_connectes = set()
```

```

async def handle_client(websocket):
    clients_connectes.add(websocket)
    print(f"Connexion : {websocket.remote_address} -
    {len(clients_connectes)} client(s) connecté(s)")

    try:
        async for message in websocket:
            print(f"Message de {websocket.remote_address} : {message}")
            destinataires = clients_connectes - {websocket}
            if destinataires:
                await asyncio.gather(
                    *[client.send(message) for client in destinataires]
                )
    finally:
        clients_connectes.discard(websocket)
        print(f"Déconnexion : {websocket.remote_address} -
        {len(clients_connectes)} client(s) restant(s)")

async def main():
    async with websockets.serve(handle_client, "localhost", 8765):
        print("Serveur de chat sur ws://localhost:8765")
        await asyncio.Future()

asyncio.run(main())

```

client.py

```

import asyncio
import websockets

async def recevoir(websocket):
    async for message in websocket:
        print(f"\n[Message reçu] {message}")

async def envoyer(websocket):
    while True:
        message = await asyncio.get_event_loop().run_in_executor(None,
input, "Vous : ")
        await websocket.send(message)

async def main():
    uri = "ws://localhost:8765"
    async with websockets.connect(uri) as websocket:
        print("Connecté au chat. Tapez vos messages.")
        await asyncio.gather(
            recevoir(websocket),
            envoyer(websocket),
        )

asyncio.run(main())

```

`run_in_executor(None, input, ...)` : `input()` est une fonction bloquante. L'appeler directement dans une coroutine gèlerait l'event loop et empêcherait la réception de messages pendant la saisie.

`run_in_executor` l'exécute dans un thread séparé, de façon transparente.

`asyncio.gather(recevoir, envoyer)` : les deux coroutines tournent en même temps. Le client peut envoyer et recevoir des messages simultanément.

Lancez le serveur, puis plusieurs instances du client dans des terminaux différents :

```
# Terminal 1
python server.py

# Terminal 2
python client.py

# Terminal 3
python client.py
```

Tout ce que vous tapez dans le terminal 2 apparaît dans le terminal 3, et vice versa. En temps réel.

Les pièges classiques

Confondre `ws://` et `http://` : WebSocket utilise ses propres schémas d'URL : `ws://` pour une connexion non chiffrée, `wss://` pour une connexion chiffrée (WebSocket over TLS, l'équivalent de HTTPS).

Modifier un set pendant l'itération : dans le serveur de chat, on utilise `clients_connectes` – `{websocket}` pour créer une copie sans le client expéditeur. Itérer directement sur `clients_connectes` et le modifier en même temps provoque une `RuntimeError`.

Oublier le bloc `finally` : si un client se déconnecte brutalement, `websocket` lève une exception et sort de la boucle `async for`. Sans `finally`, le client reste dans `clients_connectes` et le serveur essaiera de lui envoyer des messages indéfiniment.

Pour tester : lancez le serveur dans un terminal, puis ouvrez deux ou trois terminaux clients avec des pseudos différents. Chaque message envoyé par un client apparaît chez tous les autres.