Mise en œuvre d'un système de chat interactif avec architecture client-serveur en utilisant les sockets Python

L'objectif de ce projet est de développer un système de chat interactif utilisant Python et la programmation réseau par sockets, permettant la communication en temps réel entre un serveur et plusieurs clients. Le serveur écoute les connexions des clients et leur permet d'échanger des messages simultanément. Ce projet sert à démontrer les concepts de programmation réseau, de gestion de sockets et de multi-threading en Python.

Dans ce projet, le serveur écoute les connexions entrantes des clients et leur permet d'échanger des messages en temps réel. Chaque client peut se connecter au serveur et envoyer des messages, qui sont ensuite affichés sur le terminal du serveur et sur celui du client. L'échange se termine lorsque l'une des parties envoie le message "exit".

Le système suit une architecture de type client-serveur, où le serveur est responsable de la gestion des connexions et de l'échange des messages entre les clients. Chaque client se connecte au serveur via une adresse IP et un port spécifiques.

Code du Serveur :

Le serveur écoute sur une adresse IP et un port spécifiques. Lorsqu'un client se connecte, le serveur crée un nouveau thread pour gérer la communication avec ce client. Chaque client peut envoyer des messages, et le serveur répond en conséquence.

```python
import socket
import threading

# Handle communication with each client
def handle_client(client_socket, client_address):
    print(f"Connection established with {client_address}")

    while True:
        # Receive the message from the client
        message_client = client_socket.recv(1024).decode()

        if message_client == 'exit':
            print(f"Client {client_address} has exited the conversation.")
            break

        print(f"Message from {client_address}: {message_client}")

        # Server's response
        message_server = input(f"Server (to {client_address}): ")
        client_socket.send(message_server.encode())

        if message_server == 'exit':
            print(f"Server has ended the conversation with {client_address}.")
            break

    # Close client connection
    client_socket.close()
```

```python
# Server setup to accept multiple clients
def server():
    # Create the server socket
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # Server IP and port
    server_ip = "10.100.21.60"  # Your server IP
    server_port = 12345  # Port on which the server listens

    # Bind the server to the specified IP and port
    server_socket.bind((server_ip, server_port))

    # Start listening for incoming client connections
    server_socket.listen(5)
    print(f"Server listening on {server_ip}:{server_port}...")

    while True:
        # Accept incoming client connections
        client_socket, client_address = server_socket.accept()

        # Start a new thread for each client
        client_thread = threading.Thread(target=handle_client, args=(client_socket, client_address))
        client_thread.start()

if __name__ == "__main__":
    server()
```

Le code du serveur crée un socket (server_socket) qui écoute sur l'adresse IP spécifiée et un port donné. Lorsqu'un client se connecte, une nouvelle connexion est acceptée et un thread est lancé pour gérer ce client spécifiquement. La fonction handle_client permet au serveur de recevoir des messages du client et d'y répondre. Si l'un des participants envoie le message "exit", la connexion est fermée proprement.

Code du Client :

Le client se connecte à l'adresse IP et au port du serveur, envoie des messages et reçoit les réponses du serveur. Le client interagit avec le serveur en temps réel.

```python
import socket


def client():
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # Adresse IP et port du serveur
    server_ip = "10.100.21.60"  # Adresse IP du serveur
    server_port = 12345  # Port du serveur

    # Connexion au serveur
    client_socket.connect((server_ip, server_port))

    print(f"Connecté au serveur {server_ip}:{server_port}")

    while True:
        # Envoi d'un message au serveur
        message_client = input("Votre message : ")
        client_socket.send(message_client.encode())

        if message_client == 'exit':
            print("Vous avez quitté la conversation.")
            break

        # Réception de la réponse du serveur
        message_server = client_socket.recv(1024).decode()
        print(f"Message du serveur : {message_server}")

        if message_server == 'exit':
            print("Le serveur a quitté la conversation.")
            break
```

```python
    # Fermeture de la connexion
    client_socket.close()


if __name__ == "__main__":
    client()
```

Le code du client permet de se connecter au serveur via l'adresse IP et le port spécifiés. Le client envoie un message au serveur et attend une réponse. Si l'une des parties envoie le message "exit", la connexion est fermée.

Étapes pour tester la communication :

1. **Exécution du serveur** : Sur **l'ordinateur 1** (serveur), lancez le programme server.py dans un terminal en exécutant la commande :

```
python server.py
```

Le serveur commencera à écouter sur l'adresse IP et le port définis (par exemple, 10.100.21.60:12345).

Exécution du client 1 : Sur **l'ordinateur 2**, lancez le programme client.py en exécutant la commande :

```
python client.py
```

Le client 1 se connectera au serveur et pourra envoyer des messages. Il recevra aussi les réponses du serveur.

Exécution du client 2 : Sur **l'ordinateur 3**, lancez également le programme client.py en exécutant la commande :

```
python client.py
```

1. Le client 2 se connectera également au serveur, permettant une communication simultanée avec le serveur.

Exemple de communication :

- **Client 1 (10.100.22.150)** :
    - Envoie : "Bonjour depuis Client 1"
    - Reçoit : "Bonjour depuis le Serveur"
- **Client 2 (10.100.22.174)** :
    - Envoie : "Bonjour depuis Client 2"
    - Reçoit : "Bonjour depuis le Serveur"

Chaque client peut envoyer et recevoir des messages en temps réel. L'échange se termine lorsque l'une des parties envoie le message "exit".

Gestion de plusieurs clients :

Le serveur gère plusieurs connexions en même temps grâce à la **multi-threading**. Chaque client est associé à un thread distinct, ce qui permet au serveur de gérer plusieurs clients simultanément sans conflit. Ainsi, chaque client peut envoyer et recevoir des messages de manière indépendante.

Conclusion :

Ce projet montre la mise en œuvre d'un système de chat en utilisant Python, les sockets et la gestion de la concurrence par les threads. Le serveur est capable de gérer plusieurs clients simultanément, ce qui permet une communication fluide et en temps réel. Ce type de système est la base de nombreuses applications réseau comme les chats en ligne ou les jeux multijoueurs.