TP : Implémentation d’un Chat Multiclient en TCP

Objectif

Ce TP a pour but de développer un système de chat en ligne avec un serveur en C/C++ et un client en Python. Le système doit permettre à plusieurs utilisateurs de se connecter simultanément au serveur, d’envoyer et de recevoir des messages. L’architecture du serveur sera multithreadée pour gérer plusieurs clients en parallèle.

Le projet se décompose en 3 parties principales :

1. Définition du protocole de communication et implémentation d’un client en Python et d’un serveur en C/C++ simple.
2. Développement d’un serveur multithreadé en C++ pour gérer plusieurs clients simultanément.
3. Implémentation d’un client Python avec la gestion de la connexion et l’envoi de messages, ainsi que la validation du fonctionnement en mode multiclient.

Partie 1 : Définition du protocole et développement client/serveur simple

Objectifs :

- Définir un protocole de communication en JSON pour l’échange de messages entre le client et le serveur.
- Implémenter un client en Python capable de se connecter à un serveur C/C++.
- Implémenter un serveur en C/C++ capable d’accepter plusieurs connexions et de gérer les messages simples.

Spécifications du protocole JSON :

1. **Login Serveur** :
    - Le client envoie un message de connexion au serveur contenant un nom d’utilisateur unique.
    - Le serveur doit valider si le nom est disponible et retourner un message de confirmation.

    Exemple de requête de login (client -> serveur) :

```
{
  "type": "login",
  "username": "nom_utilisateur"
}
```

Réponse serveur (serveur -> client) :

```
{
  "status": "success",
  "message": "Bienvenue, nom_utilisateur"
}
```

2. **Envoi de message** :
   - Le client envoie un message texte à tous les autres clients connectés (broadcast).

   Exemple d'envoi de message (client -> serveur) :

```
{
  "type": "message",
  "username": "nom_utilisateur",
  "content": "Hello tout le monde!"
}
```

   Réponse serveur (serveur -> client) :

```
{
  "status": "success",
  "message": "Message envoyé à tous les clients"
}
```

3. **Commande de liste des utilisateurs** :
   - Le client peut demander la liste des utilisateurs connectés au serveur.

   Exemple de requête de liste des utilisateurs (client -> serveur) :

```
{
  "type": "list"
}
```

   Réponse serveur (serveur -> client) :

```
{
  "status": "success",
  "users": ["user1", "user2", "user3"]
}
```

Partie 2 : Serveur Multithreadé en C++

Objectifs :

- Implémenter un serveur en C++ capable de gérer plusieurs connexions simultanées à l'aide de threads.
- Implémenter la gestion des clients connectés (ajout, suppression et liste des clients).

Spécifications :

1. **Serveur en écoute** :
    - Le serveur doit accepter les connexions entrantes via TCP.
    - Il doit pouvoir gérer plusieurs clients en parallèle en créant un thread par client.
2. **Gestion des threads** :
    - Un thread distinct doit être créé pour chaque client.
    - Chaque thread doit écouter les messages du client et répondre en fonction du protocole défini (connexion, envoi de message, liste des utilisateurs).
3. **Liste des clients** :
    - Le serveur doit maintenir une liste des clients connectés (par exemple sous forme de tableau ou de liste chaînée).
    - Lorsque qu'un client se déconnecte, le serveur doit supprimer ce client de la liste.
4. **Gestion des clients invalides** :
    - Le serveur doit vérifier la validité des connexions. Si un client se déconnecte de manière inopinée, il doit être supprimé de la liste des clients.

Exemple de structure du serveur en C++ :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <netdb.h>      // Pour getaddrinfo
#include <sys/socket.h>
#include <arpa/inet.h>

#define PORT "12345"    // Le port d'écoute du serveur

void handle_client(int client_socket) {
    char buffer[1024];
    int bytes_received;

    // Boucle pour recevoir des messages du client
    while ((bytes_received = recv(client_socket, buffer, sizeof(buffer) - 1, 0)
        buffer[bytes_received] = '\0';  // Ajouter un terminateur de chaîne
        printf("Message reçu: %s\n", buffer);

        // Répondre au client
        send(client_socket, buffer, bytes_received, 0);
    }

    if (bytes_received == 0) {
        printf("Le client s'est déconnecté\n");
    } else {
        perror("recv");
    }

    // Fermer la connexion avec le client
    close(client_socket);
}

int main() {
    struct addrinfo hints, *res;
    int server_socket, client_socket;
    struct sockaddr_storage client_addr;
    socklen_t addr_size;
    int status;

    // Initialiser la structure hints
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_INET;        // Utiliser IPv4
    hints.ai_socktype = SOCK_STREAM;  // Utiliser le type de socket TCP
    hints.ai_flags = AI_PASSIVE;      // Indique que l'on veut une adresse IP l

    // Résoudre l'adresse et le port
    if ((status = getaddrinfo(NULL, PORT, &hints, &res)) != 0) {
        fprintf(stderr, "getaddrinfo failed: %s\n", gai_strerror(status));
        return 1;
```

```
    }

    // Créer la socket
    server_socket = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
    if (server_socket == -1) {
        perror("socket");
        freeaddrinfo(res);
        return 1;
    }

    // Lier la socket à l'adresse et au port
    if (bind(server_socket, res->ai_addr, res->ai_addrlen) == -1) {
        perror("bind");
        close(server_socket);
        freeaddrinfo(res);
        return 1;
    }

    // Écouter les connexions entrantes
    if (listen(server_socket, 5) == -1) {
        perror("listen");
        close(server_socket);
        freeaddrinfo(res);
        return 1;
    }

    // Libérer la mémoire allouée par getaddrinfo
    freeaddrinfo(res);

    printf("Serveur en écoute sur le port %s...\n", PORT);

    // Accepter les connexions des clients
    while (1) {
        addr_size = sizeof(client_addr);
        client_socket = accept(server_socket, (struct sockaddr *)&client_addr, 
        if (client_socket == -1) {
            perror("accept");
            continue;
        }

        printf("Client connecté\n");

        // Gérer le client dans un thread ou un processus séparé
        handle_client(client_socket);
    }

    // Fermer la socket du serveur
    close(server_socket);
    return 0;
}
```

Partie 3 : Client Python avec Gestion Multiclients

Objectifs :

- Développer un client Python capable de se connecter au serveur, envoyer des messages, et recevoir des réponses.
- Implémenter la gestion de la connexion, la saisie des messages et la commande pour afficher la liste des utilisateurs.
- Vérifier le bon fonctionnement avec plusieurs clients connectés simultanément.

Spécifications :

1. **Connexion au serveur** :
    - Le client doit se connecter au serveur en utilisant le protocole TCP.
    - Il doit envoyer un message de login avec son nom d'utilisateur.
2. **Envoi de messages** :
    - Le client doit permettre à l'utilisateur d'envoyer des messages à tous les autres clients connectés via un prompt.
3. **Commande Liste** :
    - Le client doit pouvoir demander la liste des utilisateurs connectés en envoyant une commande spécifique.
4. **Multiclient** :
    - Vérifier que plusieurs clients peuvent se connecter au serveur, envoyer et recevoir des messages en simultané.

Exemple de client en Python :

```
import socket
import json
import threading

def receive_messages(client_socket):
    while True:
        try:
            message = client_socket.recv(1024).decode('utf-8')
            if message:
                print(f"Message reçu: {message}")
            else:
                break
        except:
            break

def send_message(client_socket, username):
    while True:
        msg = input(f"{username} > ")
        if msg:
            message = json.dumps({"type": "message", "username": username, "con
            client_socket.send(message.encode('utf-8'))

def main():
    server_address = ('localhost', 12345)
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_socket.connect(server_address)

    username = input("Entrez votre nom d'utilisateur: ")
    login_message = json.dumps({"type": "login", "username": username})
    client_socket.send(login_message.encode('utf-8'))

    # Thread pour recevoir les messages
    receive_thread = threading.Thread(target=receive_messages, args=(client_soc
    receive_thread.start()

    # Thread pour envoyer des messages
    send_thread = threading.Thread(target=send_message, args=(client_socket, us
    send_thread.start()

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

Conclusion

Ce TP vous permettra d’apprendre les bases des communications réseau avec des sockets et en Python, ainsi que la gestion de plusieurs connexions simultanées à l’aide de threads. Vous serez amené à implémenter des fonctionnalités essentielles pour un système de chat multiclient, telles que la gestion des connexions, l’envoi de messages, et la gestion de la liste des utilisateurs connectés.