UGA - IUT2 - Département informatique
BUT Semestre 6 - Programmation Concurrente - 2024/2025

TP05 : Serveur multi-thread et mini-chat en C++

L'objectif du TP est de maîtriser la programmation d'un client et d'un serveur TCP, tous les 2 multi-threads, ainsi que la spécification d'un protocole simple de niveau application.

Nous allons adopter une approche incrémentale en commençant par un programme minimal (version 1), puis en l'enrichissant progressivement (version 2 et bonus).

1. Présentation

On fournit les squelettes de 2 logiciels :

- Dans le répertoire client : un logiciel `Client`, composé de la classe du même nom et de 2 classes annexes permettant à terme (dans une version V2) un fonctionnement multi-threads.
- Dans le répertoire serveur : un logiciel `Serveur`, composé de 3 classes :
  - la classe `Serveur` qui s'occupe d'attendre les clients
  - la classe `ClientConnecte` qui stocke les informations liées à un client connecté
  - la classe `ThreadClient` qui s'occupe du dialogue avec un client connecté grâce à un thread dédié

2. Serveur multi-threads et client mono-thread (V1)

Dans cette 1ère version (V1), le serveur envoie au client un numéro, mais ensuite, seul le client envoie des chaînes de caractères au serveur.

2.1. Serveur

Observation du code Serveur (partie réseau)

Cette partie d'observation est destinée à vous faire comprendre la structure du code.

1. Cette question est uniquement pour vous aider à comprendre le code (aucune modification de code n'est nécessaire). Comme indiqué dans le cours, le serveur a besoin de stocker un objet de type `socket` par client connecté. Dans la classe `Serveur`, un attribut de type `std::map` a été défini et est paramétré par les types suivants :
   - clés de type `unsigned long`
   - valeurs de type `ClientConnecte`, classe qui inclut les informations liées à un client (dont le `socket` qui sera utilisé pour communiquer avec lui)

   Regardez cette classe, et retrouvez le nom de cet attribut.
2. Cette question est uniquement pour vous aider à comprendre le code (aucune modification de code n'est nécessaire). Dans la classe `Serveur`, trouvez l'attribut nommé `numero_nouveau_client`. Cet attribut est initialisé à 1 (l'indice 0 est réservé au serveur lui-même), incrémenté au bon moment, et le serveur devra l'envoyer au client.
3. Cette question est uniquement pour vous aider à comprendre le code (aucune modification de code n'est nécessaire). Dans la méthode `Serveur::attend_clients()`, lors de la connexion d'un nouveau client, regardez où se fait la création d'un objet `ClientConnecte` qui, une fois la connexion établie, est stocké dans la map. Notez l'utilisation de la méthode préconisée en cours, qui permet de contourner la difficulté qu'un objet de type `socket` n'est pas copiable !

Modification du code Serveur

1. Une fois que le client est connecté, il affiche ses paramètres IP. Créez ensuite un thread grâce à la classe `ThreadClient` pour s'occuper du dialogue avec ce client. Attention : ce thread ne sera pas attendu par le thread principal, il faut donc le *détacher* dès sa création (méthode `detach()`).
2. Dans la méthode `ThreadClient::dialogue_avec_client()` :
   - Recevoir les chaînes de caractères envoyées par le client, en utilisant les méthodes de communications fournies (`EnvoiReception::recevoir_chaine`).
   - Quand une exception réseau fait sortir de la boucle infinie, cela signifie que le client n'est plus connecté. Retirer alors son entrée dans la map.
3. Les affichages (accès aux variables globales `cout` et `cerr`) étant faits simultanément par plusieurs threads, ils doivent être faits en exclusion mutuelle. Pour cela :
   - Rajouter un mutex dans la classe `Serveur`.
   - Faire en sorte que ce mutex soit verrouillé lors de chaque accès à `cout/cerr` (y compris dans `ThreadClient.cpp`).
4. La map étant maintenant accédée simultanément par plusieurs threads, il faut que tous les accès soient faits en exclusion mutuelle. Pour cela :
   - Rajouter un mutex dans la classe `Serveur`.
   - Faire en sorte que ce mutex soit verrouillé lors de chaque accès à la map.
5. Compléter la méthode `Serveur::nb_clients()`.

2.2. Client

1. Dans la méthode `Client::dialogue_avec_serveur()`, recevoir le numéro envoyé par le serveur, en utilisant les méthodes de communications fournies (`EnvoiReception::recevoir_entier`).
2. Puis appeler la méthode `ThreadLecture::lire_clavier_et_envoyer_messages()` (sans créer de thread pour l'instant).
3. Compléter la méthode `ThreadLecture::lire_clavier_et_envoyer_messages()` pour envoyer au serveur chaque chaîne lue depuis l'entrée standard.

2.3. Tests

1. Vérifier que maintenant le serveur est capable d'accepter plusieurs connexions en même temps, et de recevoir des données provenant de plusieurs clients.
2. Déconnecter tous les clients. Exécuter cette commande sur votre machine :

   ```
   ps -LU NOM_DE_LOGIN
   ```

   L'option `-L` permet de lister les threads secondaires des processus (en plus du thread principal). Vérifier que vous voyez le processus de votre serveur. Connecter un client et réexécuter la même commande. Vérifier qu'un thread a bien été lancé. Vérifier que le numéro de thread (LWP dans l'affichage de la commande) est le même que le TID affiché par le serveur.
3. Tester avec au moins 3 clients, et vérifier que le serveur est capable de gérer plusieurs clients dont les numéros ne sont pas consécutifs.
4. Vérifier que le serveur se comporte bien dans le cas suivant :
   - déconnexion du client par Ctrl-c sur le client
5. Vérifier que le client se comporte bien dans le cas suivant :
   - fin de transmission du client par Ctrl-d

3. Mini-chat et client multi-threads (V2)

On veut maintenant réaliser un mini système de chat où une phrase saisie sur un client sera renvoyée par le serveur vers tous les clients connectés.

3.1. Serveur

1. Quand le serveur reçoit un message d'un client (une chaîne), il doit parcourir la map pour réémettre ce message (le numéro du client émetteur et la même chaîne) vers tous les autres clients, et écrire un message d'information dans son terminal pour chaque envoi. Programmer ce parcours et ces envois dans la méthode `Serveur::envoyer_message_vers_tous_clients()`. Ne pas oublier les sections critiques, mais attention toutefois à ne pas créer d'interblocage.
2. Le serveur peut maintenant avertir les clients déjà connectés qu'un nouveau client vient d'arriver. Pour cela, appeler la méthode `Serveur::envoyer_message_vers_tous_clients()` dans `Serveur::attend_clients()`. NB : l'appel est déjà présent dans `ThreadClient::dialogue_avec_client()`.

3.2. Client

Pour que chaque client puisse recevoir les messages du serveur, il faut maintenant rendre le client multi-threads.

1. Dans la méthode `Client::dialogue_avec_serveur()` remplacer l'appel de `ThreadLecture::lire_clavier_et_envoyer_messages()` par le lancement de 2 threads :
   1. le thread qui va attendre les entrées au clavier, grâce à la classe `ThreadLecture`
   2. le thread qui va attendre les données provenant du réseau, grâce à la classe `ThreadReception`
2. À la fin de la méthode `Client::dialogue_avec_serveur()`, le thread principal (qui ne fera donc rien) doit attendre la fin de l'un des 2 threads secondaires.
3. Compléter la méthode `ThreadReception::recevoir_et_afficher_messages()` pour recevoir chaque message transmis par le serveur.

3.3. Tests

1. Connecter au moins 3 clients à votre serveur.
2. Vérifier que chaque client reçoit bien les phrases tapées dans les autres clients.
3. Vérifier que le client se comporte bien dans le cas suivant :
   - déconnexion du serveur par Ctrl-c sur le serveur

4. Limitation du nombre de clients connectés simultanément (Bonus)

On veut maintenant que le serveur n'accepte qu'un nombre donné maximal N de clients en même temps. Les autres clients devront être mis en attente.

1. Pour cela, ajouter un attribut dans la classe `Serveur`.
2. À quelle valeur initiale cet attribut doit-il être initialisé ?
3. Utiliser l'attribut au bon endroit dans le serveur, pour bloquer le thread principal en cas d'un nombre de clients excessif, et le débloquer quand un client se déconnecte.