Documentation Sprint 2
Messagerie multi-clients en C

Par Jorge Rémi et Deloire Alexandre

Description:

Ce document décrit la messagerie multi-clients, qui permet de gérer plusieurs clients en même temps grâce à un serveur et un protocole de communication en temps réel. Le document comprend une description du protocole de communication, de l'architecture de l'application, des difficultés rencontrées, de la répartition du travail entre les membres de l'équipe, et des instructions pour la compilation et l'exécution du code.

Protocole de Communication:

Le protocole de communication entre les clients et le serveur est basé sur TCP/IP. Chaque client se connecte au serveur via un socket et envoie des messages sous forme de chaînes de caractères. Le serveur reçoit les messages des clients et les transmet aux clients concernés.

Le client peut également exécuter plusieurs commandes, qui sont mentionnées à la fin de ce paragraphe.

Lorsqu'un client lance le programme, il doit entrer un nom d'utilisateur unique. Tant que le nom d'utilisateur fourni n'est pas unique, le client ne pourra pas discuter avec les autres clients. Le programme demandera au client de saisir un nouveau nom d'utilisateur jusqu'à ce qu'il soit unique.

Lorsqu'un client se connecte, les autres clients sont informés de sa connexion. De même, lorsqu'un client se déconnecte, le serveur le détecte et informe les autres clients de sa déconnexion. Pour se déconnecter, l'utilisateur doit entrer la commande "/fin".
Lorsque le serveur est plein, un client qui tente de se connecter ne pourra pas se connecter tant qu'un autre client ne se déconnecte pas.

Voici les commandes que le client peut exécuter :

@<pseudo> <message>
   Mentionne une personne spécifique sur le serveur, affiche le message en évidence

@<everyone> <message>
   Mentionne toutes les personnes actuellement sur le server

/fin
   Permet de mettre fin au protocole de communication et fermer le programme

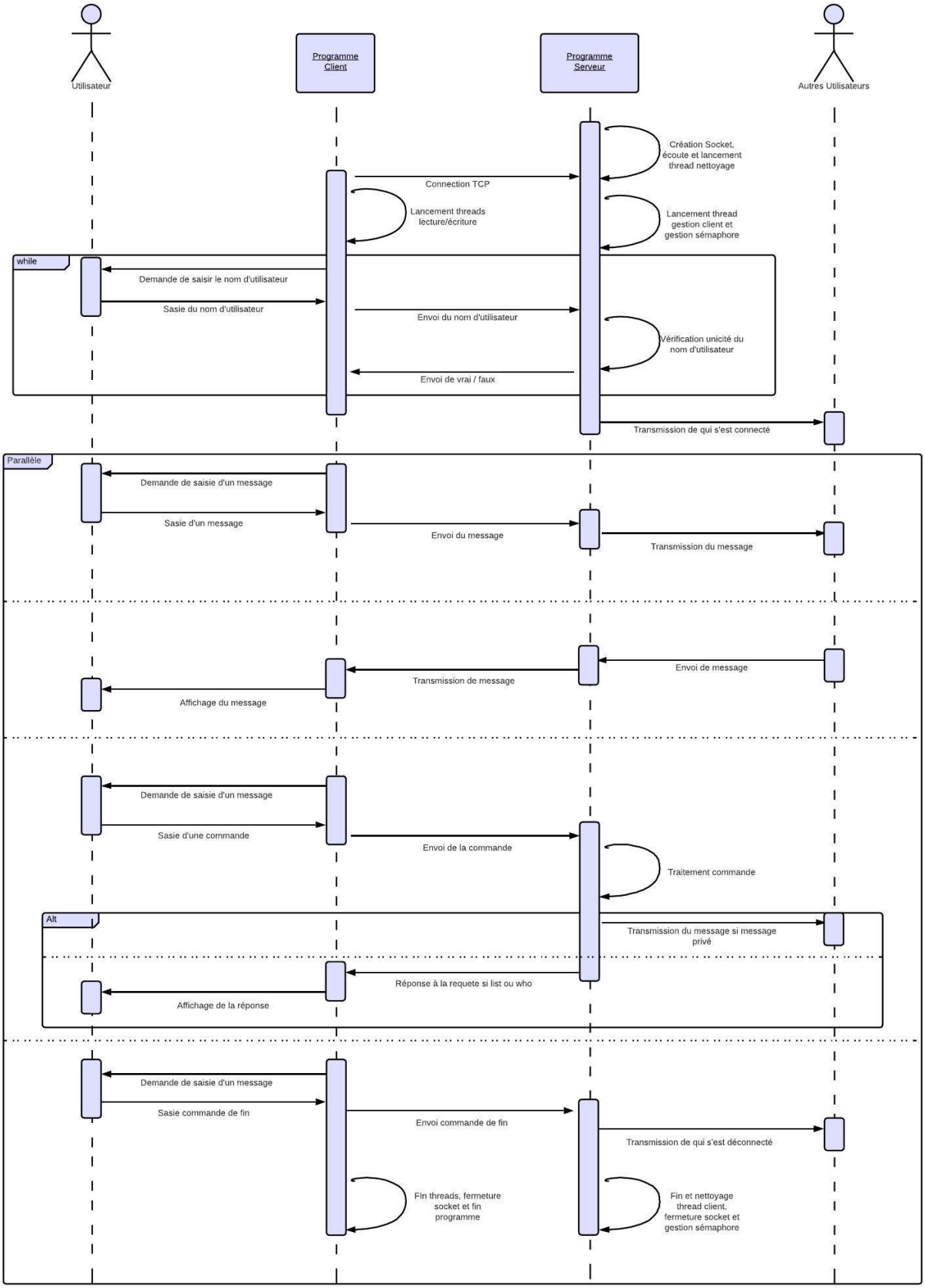
/mp <pseudo> <message>
   Envoie un message privé à la personne mentionnée par le pseudo

/man
   Affiche le guide d'utilisation

/list
   Affiche tous les utilisateurs connectés

/who
   Renvoie le pseudo

Le diagramme de séquence UML suivant illustre le protocole de communication :


Utilisateur
Programme Client
Programme Serveur
Autres Utilisateurs
Création Socket, écoute et lancement thread nettoyage
Connection TCP
Lancement threads lecture/écriture
Lancement thread gestion client et gestion sémaphore
while
Demande de saisir le nom d'utilisateur
Sasie du nom d'utilisateur
Envoi du nom d'utilisateur
Vérification unicité du nom d'utilisateur
Envoi de vrai / faux
Transmission de qui s'est connecté
Parallèle
Demande de saisie d'un message
Sasie d'un message
Envoi du message
Transmission du message
Envoi de message
Transmission de message
Affichage du message
Demande de saisie d'un message
Sasie d'une commande
Envoi de la commande
Traitement commande
Alt
Transmission du message si message privé
Réponse à la requete si list ou who
Affichage de la réponse
Demande de saisie d'un message
Sasie commande de fin
Envoi commande de fin
Transmission de qui s'est déconnecté
Fin threads, fermeture socket et fin programme
Fin et nettoyage thread client, fermeture socket et gestion sémaphore

Architecture:

L'application est constituée d'un fichier serveur et d'un fichier client. Le fichier client est voué à être lancé plusieurs fois, chaque client va lancer une instance du programme. Le serveur n'est lancé qu'une seule fois.

Threads:
Chaque client a deux threads : un thread qui va recevoir et afficher les messages provenant du serveur, et un autre qui va récupérer les messages de l'utilisateur, les formater suivant le protocole et les envoyer au serveur.

Le serveur a un thread principal qui va accepter les connexions. Le serveur a un thread pour chaque client qui se connecte. Le thread qui gère le client va recevoir les messages du client, traiter les commandes demandées par le client, et relayer les messages aux bons clients. Le serveur a aussi un thread qui est chargé de nettoyer les threads clients quand les clients se déconnectent.

Pseudo:
Pour pouvoir discuter avec d'autres clients, il faut donner un pseudo. Ce pseudo est unique. Le thread dans le serveur qui gère le client va assurer l'unicité de ce pseudo avant de laisser le client communiquer avec les autres clients. Tant que le client n'a pas donné un pseudo unique, il ne pourra pas discuter avec les autres et le programme client lui demandera d'en rentrer un autre.

Déconnexion:
Chaque client peut se déconnecter quand il veut en envoyant /fin (ou en quittant son programme). Son thread est donc nettoyé et une place est libérée sur le serveur. Lors de la déconnexion, l'indice du client est mis dans une file pour que le thread qui nettoie puisse nettoyer chaque client, même si plusieurs clients se déconnectent au même instant. Les autres clients sont informés de qui se déconnecte.

Variables globales:
Nous avons un tableau pour stocker les descripteurs fichiers des clients qui sont en connexion (qui n'ont pas donné de pseudo unique) et un tableau pour stocker les descripteurs fichiers des clients qui sont bien connectés avec un pseudo unique (ils peuvent discuter entre eux). Le dernier tableau est inclus dans le premier.
Il y a un tableau pour stocker les identifiants des threads.
Il y a une file pour stocker les indices des clients qui se sont déconnectés.

Sémaphores:
Il y a un sémaphore qui indique le nombre de places restantes sur le serveur.
Il y a un sémaphore qui indique le nombre de clients qui se sont déconnectés dont les threads doivent être nettoyés.

Mutex:
Pour chaque variable globale, il y a un mutex associé. À chaque lecture et écriture, le mutex est verrouillé pour pallier tout problème possible.

Signaux:
Il y a une gestion du signal Ctrl-C dans le programme client qui demande à l'utilisateur de saisir la commande "/fin" pour permettre une déconnexion propre.

Commandes:
Chaque client peut à tout moment consulter la liste des commandes.
Voici les commandes que nous avons fournies et leurs explications :

@<pseudo> <message>
Mentionne une personne spécifique sur le serveur, affiche le message en évidence

@<everyone> <message>
Mentionne toutes les personnes actuellement sur le server

/fin
Permet de mettre fin au protocole de communication et fermer le programme

/mp <pseudo> <message>
Envoie un message privé à la personne mentionnée par le pseudo

/man
Affiche le guide d'utilisation

/list
Affiche tous les utilisateurs connectés

/who
Renvoie le pseudo

Difficultés rencontrées:

Nous avons rencontré une difficulté particulière lors de la mise en place de l'affichage sur le terminal. En effet, le terminal est un outil assez primitif qui offre peu de fonctionnalités pour réaliser un affichage agréable pour l'utilisateur. Cependant, nous avons réussi à surmonter cette difficulté en combinant plusieurs séquences d'échappement pour pouvoir garantir un affichage optimal pour l'utilisateur.

La déconnexion simultanée de plusieurs clients engendrait des problèmes avec la gestion des variables globales, mais nous avons réussi à surmonter ce problème en utilisant une file.

Répartition du travail:

Nous avons travaillé en binôme pour le développement de ce projet. Nous avons réparti le travail en deux parties distinctes, l'une pour le programme client et l'autre pour le programme serveur. Pour assurer une coordination harmonieuse et s'assurer que les deux programmes puissent fonctionner ensemble sans problèmes, nous avons commencé par définir ensemble les spécifications exactes de ce que nous voulions accomplir. Ensuite, nous avons chacun défini une API que l'autre devait respecter. Cette approche a très bien fonctionné et nous a permis de minimiser le nombre de problèmes rencontrés tout au long du développement.

Compilation et Execution:

Vous pouvez compiler le code de la séance souhaitée en exécutant le fichier bash compil.sh dans un terminal avec la commande : “./compil.sh” (assurez-vous d’être dans le bon répertoire si le script n’est pas trouvé). Ce script crée un répertoire *bin* dans le répertoire courant. Déplacez-vous dans ce répertoire *bin.* Deux codes compilés y sont présents : client et server. Commencez par exécuter le server sur le port de votre choix avec la commande : “./server <port>” (ex: “./server 3000”). Ensuite vous pouvez exécuter les clients avec la commande “./client <server_ip> <server_port>” (ex: “./client 162.111.186.34 3000”). Exécutez le serveur et chaque client dans un terminal différent. Vous êtes à présent prêt à discuter sur la messagerie.

Ce projet est open source ! Voici le lien code: https://github.com/RemiJorge/Projet-Far