Documentation Sprint 1

Messagerie multi-clients en C

Par Jorge Rémi et Deloire Alexandre

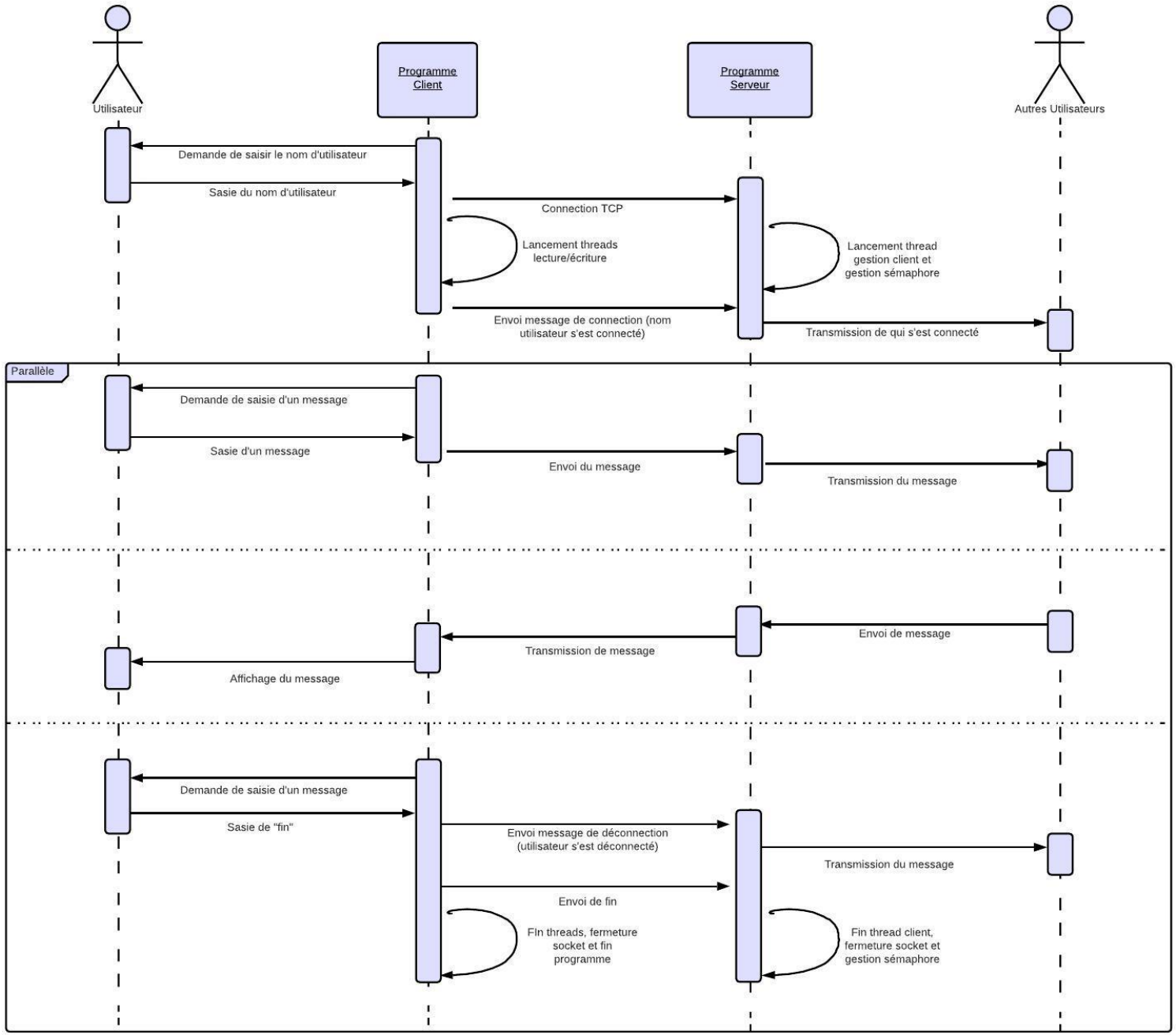
Description:

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 le protocole 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 à tous les autres clients connectés.
Lorsqu'un client lance le programme, il doit entrer un nom d'utilisateur.
Lorsqu'un client se connecte, les autres clients sont informés de qui s'est connecté . Lorsqu'un client se déconnecte, le serveur le detecte et les autres clients sont informés de qui s'est déconnecté. Pour se déconnecter, il faut que l'utilisateur entre le mot "fin".
Lorsque le serveur est plein, un client qui tente de se connecter ne pourra pas tant que quelqu'un d'autre ne se déconnecte pas.

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


Utilisateur
Programme Client
Programme Serveur
Autres Utilisateurs
Demande de saisir le nom d'utilisateur
Sasie du nom d'utilisateur
Connection TCP
Lancement threads lecture/écriture
Lancement thread gestion client et gestion sémaphore
Envoi message de connection (nom utilisateur s'est connecté)
Transmission de qui s'est connecté
Parallèle
Demande de saisie d'un message
Sasie d'un message
Envoi du message
Transmission du message
Transmission de message
Envoi de message
Affichage du message
Demande de saisie d'un message
Sasie de "fin"
Envoi message de déconnection (utilisateur s'est déconnecté)
Transmission du message
Envoi de fin
FIn threads, fermeture socket et fin programme
Fin thread client, fermeture socket et gestion sémaphore

Architecture:

L'application est composée d'un serveur et de plusieurs clients. Le serveur utilise un thread principal pour accepter les connexions des clients et un thread par client pour gérer la communication avec chaque client. Chaque thread client écoute les messages du client via un socket et les diffuse aux autres clients en utilisant un tableau partagé pour stocker les identifiants des sockets des clients connectés.

Pour gérer le nombre de clients qui peuvent être connectés, nous utilisons un sémaphore qui représente le nombre de places disponibles sur le serveur.

Le programme client demande un nom d'utilisateur pour se connecter, puis lance deux threads, un qui gère la saisie des messages par l'utilisateur et les envoie et un qui gère la réception et l'affichage des messages provenant du serveur.

Toutes les séances ont un README.txt qui explique les fonctionnalités spécifiques et incluent notamment des instructions spécifiques pour compiler et exécuter les programmes.

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.

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:

Pour chacune des séances 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. Pour la séance 3 un pseudo vous sera demandé avant la procédure de connexion. 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