Documentation Sprint 3
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. Les clients peuvent également télécharger des fichiers depuis et sur le serveur tout en continuant d'échanger des messages avec les autres clients.

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.

Le client peut télécharger n'importe quel type de fichier vers le serveur. Pour ce faire, il doit placer le fichier dans son répertoire dédié aux fichiers, puis il a deux options : soit il tape directement "/upload nom_du_fichier" pour télécharger directement le fichier vers le serveur, soit il tape simplement "/upload" pour ouvrir un menu affichant les fichiers de ce dossier. Le client peut parcourir ce menu à l'aide des flèches de son clavier pour choisir le fichier qu'il souhaite transférer.

De plus, le client peut demander la liste des fichiers disponibles en téléchargement depuis le serveur, puis télécharger l'un des fichiers s'il le souhaite. Il lui suffit de taper la commande "/download" pour afficher un menu des fichiers disponibles, qu'il peut naviguer à l'aide des flèches de son clavier.

Voici le guide d'utilisation de la messagerie:

@<pseudo> <message>
	Mentionne une personne specifique sur le server, affiche le message en evidence

@<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

/upload <fichier>
	Telecharge le <fichier> qui se trouve dans le repertoire de client_files vers le server

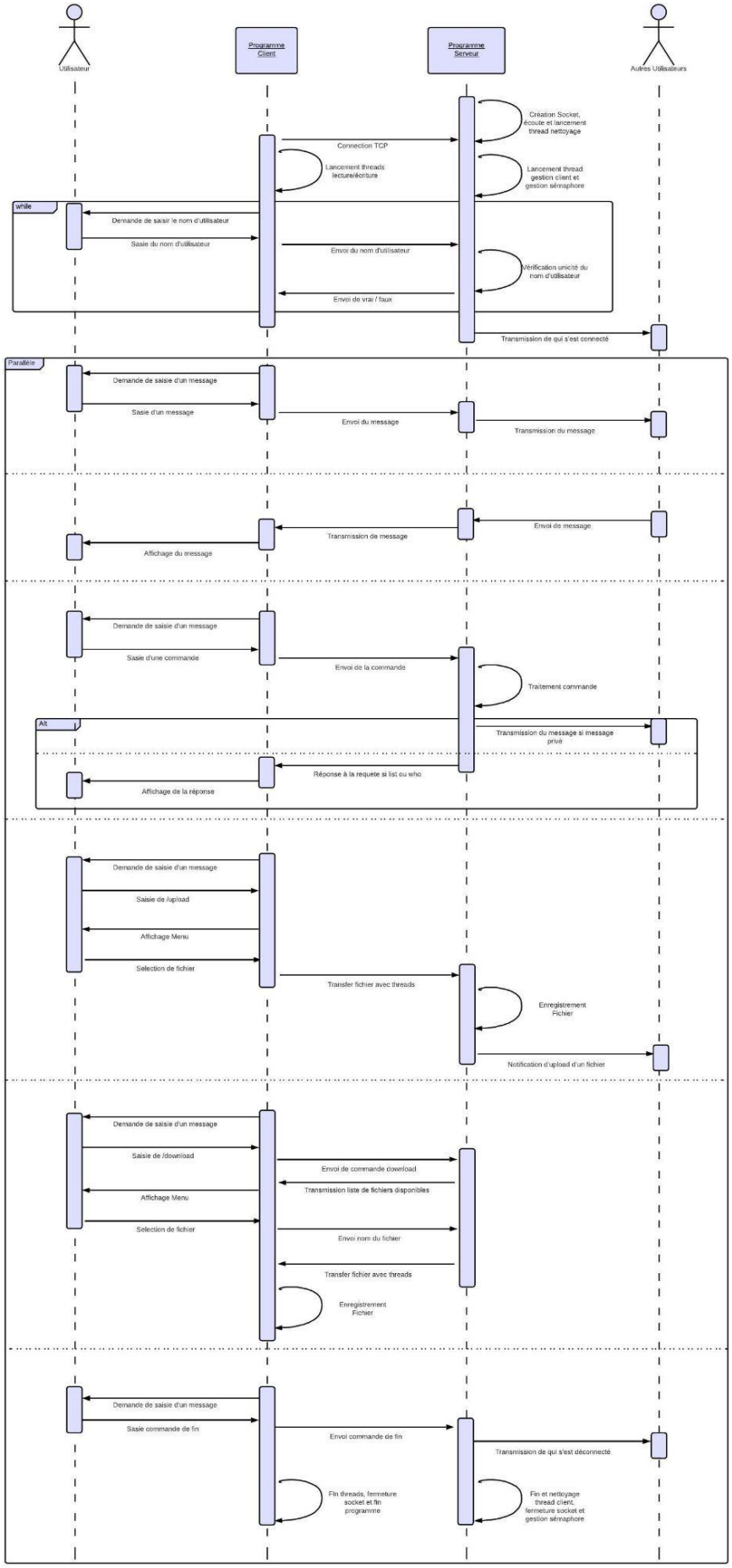
/upload

Ouvre le menu de selection de fichier afin d'envoyer un fichier de client_files vers le server

/download

Ouvre le menu de selection de fichier afin de télecharger le fichier choisi depuis le server

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
Saisie 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
Saisie d'un message
Envoi du message
Transmission du message
Envoi de message
Transmission de message
Affichage du message
Demande de saisie d'un message
Saisie 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
Saisie de /upload
Affichage Menu
Selection de fichier
Transfer fichier avec threads
Enregistrement Fichier
Notification d'upload d'un fichier
Demande de saisie d'un message
Saisie de /download
Envoi de commande download
Transmission liste de fichiers disponibles
Affichage Menu
Selection de fichier
Envoi nom du fichier
Transfer fichier avec threads
Enregistrement Fichier
Demande de saisie d'un message
Saisie 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 principaux : 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. De plus, à chaque fois que le client télécharge un fichier depuis ou vers le serveur, un thread dédié à cette action est lancé pour chaque fichier. Cela, avec l'aide d'une nouvelle socket, permet au client de continuer à recevoir et envoyer des messages aux autres clients.

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. A chaque fois que le client télécharge un fichier depuis ou vers le serveur, un thread dédié à cette action est lancé pour chaque fichier. Cela, avec l'aide d'une nouvelle socket, permet au client de continuer à recevoir et envoyer des messages aux autres clients. Le serveur a aussi un thread qui est chargé de nettoyer les threads clients quand les clients se déconnectent, et de nettoyer les threads qui gèrent les uploads et les downloads.

Sockets:
Le serveur crée trois sockets qui serviront à accepter les connexions des clients. Une socket est dédiée à la connexion initiale des clients pour l'échange de messages. Les deux autres sockets sont utilisés pour le téléchargement et le transfert de fichiers, permettant ainsi aux clients de communiquer et de transférer des fichiers simultanément.

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 threads qui se sont terminés et qui 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.

Envoi et réception de fichiers:
L'application permet également aux clients d'envoyer et de recevoir des fichiers de n'importe quel type. Les fichiers des clients sont stockés dans le répertoire "client_files", tandis que les fichiers du serveur sont stockés dans le répertoire "server_files".

Lorsqu'un client souhaite envoyer un fichier au serveur, un thread dédié est créé pour gérer cette action. Ce thread utilise une nouvelle socket pour établir une connexion distincte avec le serveur et transférer le fichier. Pendant le transfert du fichier, le client peut continuer à recevoir et à envoyer des messages aux autres clients.

De même, lorsqu'un client souhaite télécharger un fichier depuis le serveur, un thread dédié est créé pour chaque téléchargement. Ce thread utilise également une nouvelle

socket pour établir une connexion distincte avec le serveur et transférer le fichier demandé vers le répertoire "client_files" du client.

Les transferts de fichiers utilisent un protocole spécifique pour assurer une transmission fiable des données. Les fichiers sont découpés en paquets de taille appropriée et envoyés un par un. Le serveur et le client s'accordent sur un mécanisme de validation des paquets pour s'assurer de l'intégrité des données transférées.

Il est important de noter que l'application de messagerie permet l'envoi et la réception de fichiers de n'importe quel type. Cela signifie que les clients peuvent partager des images, des documents, des vidéos, etc. Le protocole de communication prend en charge ces différents types de fichiers et garantit leur transmission correcte entre les clients.

Commandes:
Chaque client peut à tout moment consulter la liste des commandes.
Voici le guide d'utilisation de la messagerie:

@<pseudo> <message>
    Mentionne une personne specifique sur le server, affiche le message en evidence

@<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

/upload <fichier>

Telecharge le <fichier> qui se trouve dans le repertoire de client_files vers le server

/upload

Ouvre le menu de selection de fichier afin d'envoyer un fichier de client_files vers le server

/download

Ouvre le menu de selection de fichier afin de télecharger le fichier choisi depuis le server

Difficultés rencontrées:

Une difficulté que nous avons dû surmonter était un problème lié à l'envoi et à la réception de paquets lors du transfert de fichiers entre le serveur et un client. En effet, nous avons constaté un décalage entre les envois et les réceptions, sans pouvoir en expliquer la raison. Cependant, nous avons résolu ce problème en vérifiant attentivement le nombre d'octets reçus à la fin du transfert, car le protocole TCP garantit la transmission et l'ordre des paquets.

La terminaison simultanée de plusieurs threads de natures différentes engendrait des problèmes avec leur nettoyage, mais nous avons réussi à surmonter ce problème en modifiant la file existante.

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