Documentation

Messagerie multi-clients en C

Par Jorge Rémi et Deloire Alexandre

PARTIE I : TECHNIQUE

Introduction:

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.

Description générale de l’application:

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.

De plus, afin de favoriser les interactions entre les clients, notre plateforme offre la possibilité aux utilisateurs de rejoindre des salons de discussion. Une interface agréable à l'utilisation leur permet d'accéder à ces salons et d'engager des conversations avec d'autres clients. Les utilisateurs sont libres de créer et de supprimer ces salons selon leurs besoins et préférences. De plus, il n'y a aucune limite quant au nombre de salons auxquels les clients peuvent participer, leur offrant ainsi une grande flexibilité dans leur expérience de discussion.

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

Lorsque le programme est lancé, le client est invité à entrer un nom d'utilisateur unique. Si le nom d'utilisateur fourni n'est pas unique, le client doit saisir un nouveau nom d'utilisateur jusqu'à ce qu'il en trouve un qui soit unique. Cette vérification garantit que chaque utilisateur a un nom distinct, permettant ainsi une identification claire et évitant les doublons.

Lorsqu'un client se connecte avec succès, les autres clients sont informés de sa connexion. De même, lorsque le client se déconnecte en utilisant la commande "/fin", le serveur détecte cette déconnexion et en informe les autres clients présents. Cela permet une communication transparente et une prise de conscience de l'état de connexion des utilisateurs.

En cas de saturation du serveur, c'est-à-dire lorsque le nombre maximal de clients est atteint, un client qui tente de se connecter se verra refuser l'accès jusqu'à ce qu'un autre client se déconnecte. Cela garantit que le serveur ne dépasse pas sa capacité maximale et maintient la stabilité du système.

Grâce à ces fonctionnalités et commandes, nous offrons aux clients une expérience de communication sécurisée, interactive et gérée de manière optimale. Que ce soit pour discuter avec d'autres utilisateurs, partager des fichiers ou gérer sa propre connexion, notre programme met à disposition les outils nécessaires pour une expérience fluide et conviviale.

Le client bénéficie d'une fonctionnalité pratique qui lui permet de télécharger divers types de fichiers vers le serveur. Pour ce faire, il lui suffit de placer le fichier dans son répertoire dédié aux fichiers. Ensuite, il dispose de deux options pour procéder au téléchargement : soit en tapant directement la commande "/upload nom_du_fichier" pour transférer immédiatement le fichier vers le serveur, soit en utilisant la commande "/upload" qui ouvre un menu affichant les fichiers présents dans son répertoire dédié.

Dans ce menu, le client peut naviguer en utilisant les flèches de son clavier, lui permettant ainsi de sélectionner le fichier qu'il souhaite transférer vers le serveur de manière aisée.

Par ailleurs, le client a également la possibilité de demander la liste des fichiers disponibles en téléchargement à partir du serveur, et de choisir de télécharger l'un de ces fichiers s'il le souhaite. Pour cela, il lui suffit de saisir la commande "/download", ce

qui affiche un menu présentant les fichiers disponibles. À l'aide des flèches de son clavier, le client peut parcourir ce menu et sélectionner le fichier qu'il souhaite télécharger.

Cette fonctionnalité d'upload et de download facilite le transfert de fichiers entre le client et le serveur, offrant ainsi une expérience pratique et fluide. Que ce soit pour partager des documents importants, échanger des fichiers multimédias ou tout autre besoin de transfert de données, nos commandes intuitives permettent aux clients d'effectuer ces opérations de manière simple et efficace.

L'application permet aux clients de rejoindre des salons de discussion pour une expérience encore plus interactive. Grâce à la commande "/salon", les utilisateurs peuvent accéder facilement aux salons disponibles et aux salons auxquels ils sont déjà connectés.

Lorsqu'un client utilise la commande "/salon", un menu s'affiche, répertoriant les différents salons disponibles. Ce menu offre une vue claire et organisée, permettant aux utilisateurs de choisir le salon qui les intéresse. Une fois qu'ils ont sélectionné un salon, une nouvelle fenêtre s'ouvre, offrant un espace dédié où ils peuvent échanger avec les autres personnes présentes dans le salon.

L'avantage de cette nouvelle fenêtre est qu'elle permet aux utilisateurs de discuter en temps réel avec les autres participants du salon, sans interférer avec leurs autres interactions dans l'application. Ainsi, ils peuvent continuer à envoyer des messages et à partager des fichiers avec les autres clients tout en participant activement aux conversations dans le salon.

Une flexibilité supplémentaire est offerte aux utilisateurs, car ils peuvent quitter un salon à tout moment. Lorsqu'ils décident de quitter un salon, la fenêtre se ferme automatiquement, leur permettant de passer facilement d'un salon à un autre ou de retourner à d'autres fonctionnalités de l'application.

De plus, notre fonctionnalité de salon ne limite pas le nombre de salons auxquels les clients peuvent participer simultanément. Les utilisateurs sont libres de rejoindre autant de salons qu'ils le souhaitent, leur donnant ainsi la possibilité d'explorer divers sujets et d'interagir avec une multitude de personnes en même temps.

Pour compléter cette expérience interactive, les clients ont également la possibilité de créer leurs propres salons. Ils peuvent ainsi donner vie à des espaces de discussion dédiés à des sujets spécifiques ou à des communautés particulières. De plus, ils

peuvent supprimer les salons qu'ils ont créés s'ils ne sont plus nécessaires ou s'ils souhaitent les retirer de la liste des salons disponibles.

Cette fonctionnalité de salon renforce notre engagement à fournir une plateforme de communication conviviale, offrant aux clients une expérience enrichissante et adaptable à leurs besoins. Profitez de ces salons pour partager vos idées, rencontrer de nouvelles personnes et élargir vos horizons dans notre communauté en ligne.

Voici la liste des commandes:

@<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

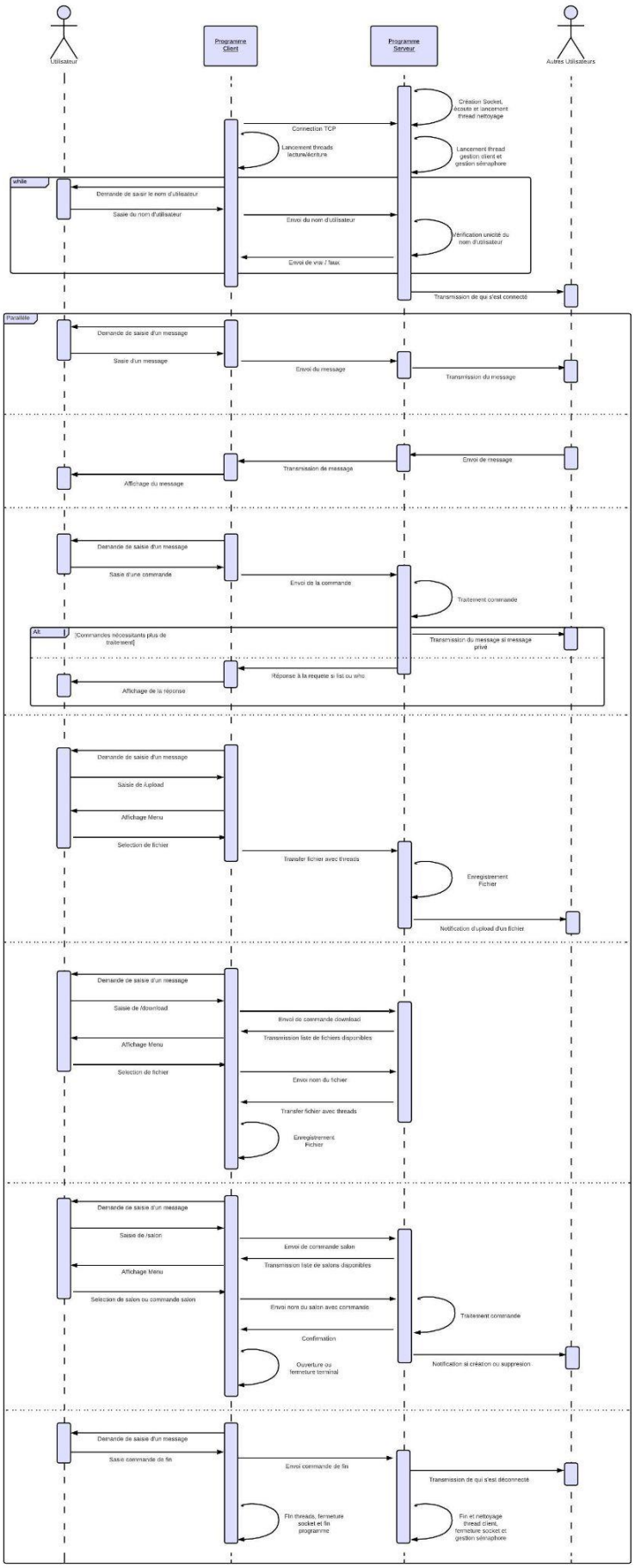
/salon

Ouvre le menu des salons pour pouvoir créer, rejoindre, quitter et supprimer des salons

/exit

Commande à taper dans un salon. Permet de quitter le salon. La fenêtre du salon se fermera automatiquement.

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
[Commandes nécessitants plus de traitement]
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 de /salon
Envoi de commande salon
Transmission liste de salons disponibles
Affichage Menu
Selection de salon ou commande salon
Envoi nom du salon avec commande
Traitement commande
Confirmation
Notification si création ou suppresion
Ouverture ou fermeture terminal
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, d'un fichier client et d'un fichier client_salon. 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.

Fichiers et dossiers :
L'application est composée des fichiers serveur, client, et client_salon, ainsi que de différents dossiers dédiés au stockage des fichiers et des informations des salons.
Les fichiers du serveur sont stockés dans server_files. Les fichiers du client sont stockés dans client_files. Chaque salon est stocké sous la forme d'un fichier. Les salons sont stockés dans un dossier server_channels. Ces éléments sont essentiels au bon fonctionnement de l'application, permettant aux clients de se connecter, d'échanger des messages, des fichiers et de gérer les salons de discussion.

Gestion des threads :

Coté client:
Chaque instance du programme client lance trois threads initiaux:

- Un thread qui est responsable de la réception et de l'affichage des messages provenant du serveur.
- Un thread qui récupère les messages de l'utilisateur, les formate selon le protocole et les envoie au serveur.
- Un thread qui est chargé du nettoyage des autres threads lorsqu'ils terminent.

De plus, lorsqu'un client effectue un téléchargement de fichier depuis ou vers le serveur, un thread dédié à cette action est lancé pour chaque fichier. Cette approche permet au client de continuer à recevoir et à envoyer des messages aux autres clients pendant le transfert de fichiers.

Lorsqu'un client ouvre le menu des salons, une nouvelle socket est utilisée avec un nouveau thread pour permettre la réception simultanée de messages. Lorsqu'un client rejoint un salon, un thread est lancé pour relayer les messages entre le salon et le serveur.

Chaque salon possède deux threads pour gérer l'envoi et la réception simultanée de messages.

Coté serveur:
Le serveur possède un thread principal qui est chargé d'accepter les connexions des clients et d'initialiser leur informations dans la variables globales correspondantes.

Pour chaque client qui se connecte, le serveur crée un thread dédié à la gestion de ce client. Ce thread est responsable de la réception des messages du client, du traitement des commandes demandées et du relais des messages aux destinataires appropriés.

De plus, lorsqu'un client effectue un téléchargement de fichier depuis ou vers le serveur, un thread dédié est également lancé pour chaque fichier. Ces threads supplémentaires permettent au serveur de continuer à recevoir et à envoyer des messages aux autres clients pendant les transferts de fichiers.

Le serveur dispose également d'un thread chargé de nettoyer les threads clients lorsque les clients se déconnectent, ainsi que de nettoyer les threads associés aux téléchargements et aux téléversements de fichiers.

Un autre thread est lancé pour gérer les actions de connexion, déconnexion, création et suppression des salons en parallèle qui permet au client de recevoir des messages même quand il est dans le menu et d'envoyer des messages sur les autres salons déjà ouverts.

Gestion des sockets :
Coté client:
Le client créer deux sockets principales:

- Un socket pour se connecter au serveur.
- Un socket pour gérer les différents salons. Chaque salon ouvert va se connecter a cette socket pour que le programme client principal puisse ensuite relayer les messages vers le serveur.

Coté serveur:
Le serveur crée quatre sockets pour accepter les connexions des clients:

- Un socket est dédié à la connexion initiale des clients pour l'échange de messages.
- Un socket est dédié à la gestion des uploads des fichiers du client au serveur.
- Un socket est dédié à la gestion des downloads des fichiers du serveur vers le client.
- Un socket est utilisé pour la gestion des actions liées aux salons, telles que la connexion, la déconnexion, la création et la suppression.

Les communications entre le client et les fenêtres des salons se font également par le biais de sockets, permettant ainsi le relais des messages entre le serveur et les fenêtres des salons.

Gestion des pseudonymes :
Afin de permettre la communication entre les clients, un pseudonyme unique doit être fourni. Le thread du serveur qui gère chaque client s'assure de l'unicité du pseudonyme avant de permettre au client de communiquer avec les autres. Tant que le client n'a pas fourni un pseudonyme unique, il ne pourra pas participer aux discussions avec les autres clients et le programme client lui demandera d'en choisir un autre.

Déconnexion :
Chaque client a la possibilité de se déconnecter à tout moment en envoyant la commande "/fin" ou en quittant le programme.

Coté serveur:
Lorsqu'un client se déconnecte, son socket est fermé, son thread est nettoyé et une place se libère sur le serveur. Lors de la déconnexion, l'indice du client est ajouté à une file d'attente, ce qui permet au thread de nettoyage de prendre en charge le nettoyage de chaque client, même si plusieurs clients se déconnectent simultanément.
De plus, les autres clients sont informés de la déconnexion de ce client spécifique.
Si un client se déconnecte d'un salon, sa liste de salon est mise à jour.

Coté client:
Lorsqu'un client se déconnecte d'un salon, la fenêtre du salon correspondant se ferme et les ressources et threads associés sont nettoyés et libérés.

Lorsqu'un client se déconnecte du serveur, son socket est fermé, les threads sont nettoyés, les ressources sont libérées et le programme se ferme proprement.

Variables globales :
Plusieurs variables globales sont utilisées dans l'application.

Coté serveur:

- Un tableau stocke les descripteurs de fichiers des clients en cours de connexion (n'ayant pas encore fourni de pseudonyme unique), tandis qu'un autre tableau stocke les descripteurs de fichiers des clients connectés avec un pseudonyme unique (ceux qui peuvent communiquer entre eux). Le dernier tableau est inclus dans le premier.

- Un tableau est également utilisé pour stocker les identifiants des threads.
- Une file pour stocker les identifiants de threads qui se sont terminés.
- Un tableau de liste est utilisé pour représenter les salons auxquels les clients sont connectés.
- Les différents mutexs et sémaphores relatifs à la synchronisation des threads.

Coté Client:

- Variables globales relatives à l'affichage pour synchroniser les threads.
- Une file pour stocker les identifiants de threads qui se sont terminés.
- Une liste des salons du client auxquels il est connecté.
- Des descripteurs fichiers des différents sockets.
- Les différents mutexs et sémaphores relatifs à la synchronisation des threads.

Gestion des sémaphores :
Deux sémaphores sont utilisés dans l'application. Le premier sémaphore indique le nombre de places restantes sur le serveur, ce qui permet de gérer les connexions lorsque le serveur est complet. Le deuxième sémaphore indique le nombre de threads terminés qui doivent être nettoyés. Ce dernier sémaphore se trouve à la fois chez le client et le serveur.

Gestion des mutex :
Pour chaque variable globale, un mutex est associé. Lors de chaque lecture et écriture sur ces variables, le mutex correspondant est verrouillé pour éviter tout problème potentiel de concurrence. Ceci est côté client et serveur.

Gestion des signaux :
Le programme client gère le signal Ctrl-C en demandant à l'utilisateur d'envoyer la commande "/fin" pour une déconnexion propre. Le programme serveur gère le signal Ctrl-C en déconnectant tous les clients en les prévenant, en fermant les sockets, en nettoyant tous les threads et en libérant proprement toutes les ressources utilisées par le serveur.

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 à la fois côté serveur et côté client pour gérer cette action. Ce thread utilise une nouvelle socket pour établir une connexion distincte avec le serveur et transférer le fichier. Le thread propose ainsi une liste des fichiers disponibles pour l'envoi au serveur grace a

un menu que le client peut parcourir avec les flèches du clavier. Ceci étant fait en parallèle, le client peut continuer de recevoir des messages des autres clients lorsqu'il est dans le menu.

Une fois le fichier choisi et les informations du fichier envoyé au serveur, 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 côté serveur et côté client pour chaque téléchargement. Le thread propose ainsi une liste des fichiers disponibles pour téléchargement grâce à un menu que le client peut parcourir avec les flèches du clavier. Ceci étant fait en parallèle, le client peut continuer de recevoir des messages des autres clients lorsqu'il est dans le menu.

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.

MultiSalons:

Les salons constituent une fonctionnalité essentielle de l'application, offrant aux utilisateurs la possibilité de participer à des discussions thématiques distinctes.

Coté Client:

Lorsqu'un client décide de rejoindre un salon, un thread est lancé pour récupérer la liste des salons disponibles et afficher un menu de gestion de salon, que le client peut parcourir avec les flèches du clavier. Ceci étant fait en parallèle, le client peut continuer de recevoir des messages des autres clients lorsqu'il est dans le menu.

Le client peut rejoindre un salon en appuyant sur la touche “Entrer” du clavier, un étoile s'affiche à côté du nom du salon pour indiquer que le client est connecté la dessus, et

un nouveau processus est lancé dans un terminal distinct, qui représente le nouveau salon. Ce processus communique avec la fenêtre principale du client via une socket dédiée. Cette approche permet à l'utilisateur de bénéficier d'une expérience agréable, avec la possibilité de gérer plusieurs salons simultanément, tout en envoyant et en recevant des messages de manière parallèle. Grâce à cette architecture, l'utilisateur peut facilement naviguer entre les différents salons, interagir avec d'autres participants et garder une vue d'ensemble des discussions en cours. Cette fonctionnalité favorise une communication fluide et efficace, en offrant aux utilisateurs un moyen pratique d'organiser leurs conversations et d'interagir de manière ciblée dans chaque salon.

Le client peut quitter un salon soit en tapant “/exit” dans le salon soit en appuyant sur la touche “Entrer” sur le salon qui à une étoile à côté de son nom dans le menu de gestion des salons. Ceci qui termine le processus du salon en nettoyant les threads et les ressources et les informations du client sont mis à jour sur le serveur.

Le client peut également créer un salon, le thread du menu de gestion de salon lui demandera de saisir le nom et la description du salon, qui seront envoyés au serveur et le salon sera créé. Tous les utilisateurs seront prévenus de la création du nouveau salon.

Le client peut supprimer un salon, ce qui prévient et déconnecte tous les utilisateurs connectés à ce salon, et tous les utilisateurs seront prévenus de la suppression du salon. Le salon sera effectivement supprimé côté serveur.

Cote Serveur:

Le serveur a un tableau de liste, ou chaque liste représente la liste des salons auxquels un client est connecté. A chaque fois qu’un client lance le menu de gestion des salons, un thread est lancé, ce qui permet au client de recevoir les messages des autres clients et de discuter avec les autres clients sur les autres salons. Les informations du client vis-à-vis des salons et les salons eux-mêmes sont mis à jour lors des différentes demandes.

Le serveur garantit une communication sécurisée car il envoie les messages seulement aux salons pour lesquels les messages sont destinés.

Protocoles:

Le protocole utilisé dans cette application est TCP (Transmission Control Protocol). TCP est un protocole fiable et orienté connexion, offrant plusieurs avantages :

1. Fiabilité : TCP garantit la fiabilité de la communication en fournissant un mécanisme de retransmission des données perdues ou corrompues. Cela assure que les messages et les fichiers sont correctement reçus par les destinataires.
2. Contrôle de flux : TCP utilise un mécanisme de contrôle de flux pour réguler la quantité de données échangées entre les parties. Cela évite les problèmes de congestion du réseau et assure des transmissions fluides et efficaces.
3. Ordre de livraison : TCP maintient l'ordre de livraison des segments de données, ce qui signifie que les messages et les fichiers sont reçus dans le même ordre dans lequel ils ont été envoyés. Cela est particulièrement important pour les applications qui dépendent de l'ordre des données, comme les conversations en temps réel.
4. Gestion des erreurs : TCP dispose de mécanismes de détection et de correction d'erreurs intégrés, tels que les numéros de séquence et les accusés de réception. Cela permet de détecter et de corriger les erreurs de transmission, garantissant ainsi l'intégrité des données échangées.

Envoi de Messages:

Dans notre application, les messages sont envoyés et reçus en utilisant une structure appelée "Message". Cette structure admets ces champs:

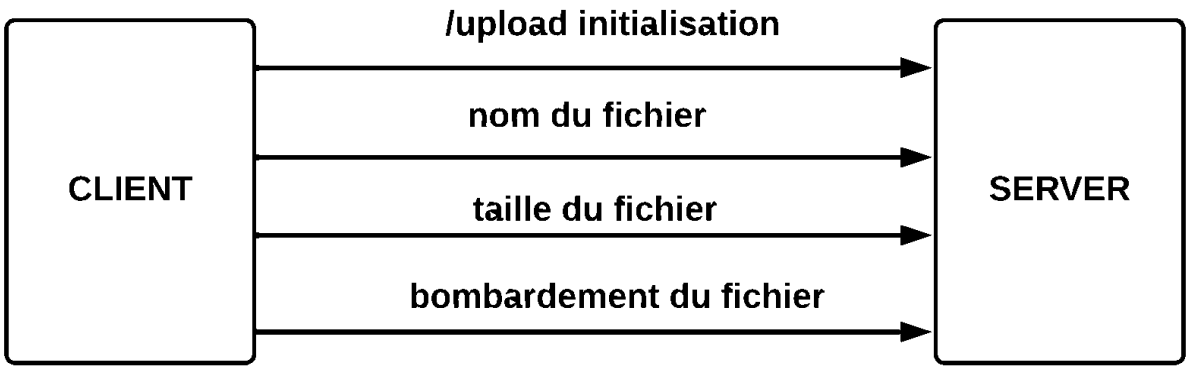
1. **cmd** : Ce champ représente la commande associée au message. Il peut prendre différentes valeurs en fonction de l'action souhaitée, telles que la création d'un salon, l'envoi d'un message, la demande de la liste des utilisateurs, etc. Les différentes commandes possibles sont décrites en détail dans la documentation.
2. **from** : Ce champ indique le nom d'utilisateur du client qui envoie le message. Il peut être le nom d'un utilisateur connecté ou "Server" si le message est envoyé par le serveur lui-même.
3. **`to`** : Ce champ représente le nom d'utilisateur du client destinataire du message. Il peut également être "Server" si le message est destiné au serveur.
4. **`channel`** : Ce champ contient le nom du salon dans lequel le message est envoyé. Un salon est un espace de discussion spécifique où les utilisateurs peuvent échanger des messages.

5. **`message`**: Ce champ contient le contenu du message lui-même. Il peut s'agir d'un texte ou d'autres informations selon le contexte.
6. **`color`** : Ce champ spécifie la couleur du message, permettant de personnaliser son apparence visuelle. Cela peut être utilisé pour différencier les types de messages ou pour ajouter une touche esthétique.

Envoi de Fichiers:

Voici le protocole utilisé pour l'upload et le download de fichiers. Les données des fichiers sont envoyées par morceaux dans des buffers de taille fixe. La taille est vérifiée à la fin de l'envoi pour veiller au bon transfert du fichier.

Upload:

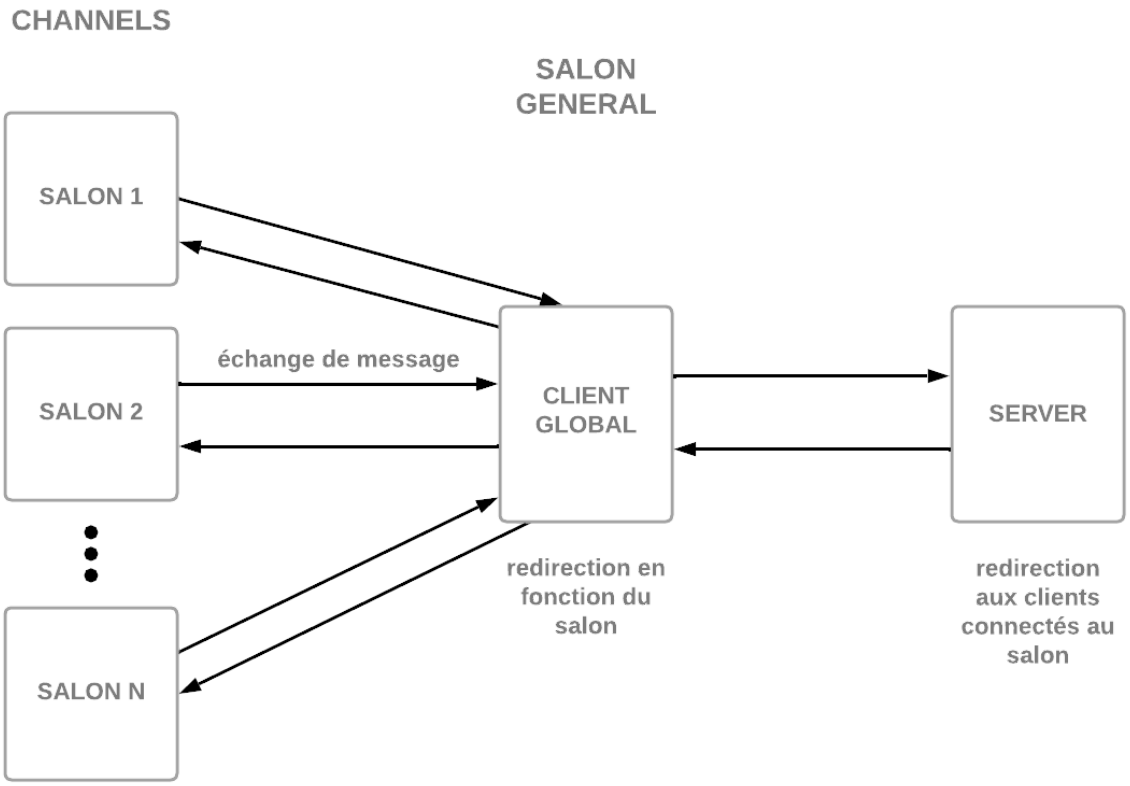


Download:

Multi Salon:

En ce qui concerne le protocole utilisé pour la fonctionnalité des multi salons, chaque salon ouvert va communiquer avec le programme client main ou global (qui est aussi le salon global) via un socket, et le client main va relayer ces messages au serveur et également rediriger les messages reçu du serveur au bon salon.

Voici un schéma général qui illustre le protocole utilisé:



Commandes:
Chaque client peut à tout moment consulter la liste des commandes:

@<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 répertoire 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 télécharger le fichier choisi depuis le server

/salon

Ouvre le menu des salons pour pouvoir créer, rejoindre, quitter et supprimer des salons

/exit

Commande à taper dans un salon. Permet de quitter le salon. La fenêtre du salon se fermera automatiquement.

Compilation et Execution:

Pour compiler et exécuter le code de la séance souhaitée, veuillez suivre les étapes suivantes :

1. Tout d'abord, ouvrez un terminal et placez-vous dans le répertoire contenant le fichier bash "compil.sh". Assurez-vous d'être dans le bon répertoire.
2. Exécutez le script bash en utilisant la commande "./compil.sh". Ce script créera un répertoire appelé "bin" dans le répertoire courant. Ce répertoire sera utilisé pour stocker les fichiers binaires générés.
3. **Déplacez-vous dans le répertoire "bin"** nouvellement créé à l'aide de la commande "cd bin". Assurez-vous d'être maintenant dans ce répertoire.
4. À l'intérieur du répertoire "bin", vous trouverez deux fichiers binaires compilés : "client" et "server". Ces fichiers seront utilisés respectivement pour lancer le client et le serveur du programme.
5. Tout d'abord, exécutez le serveur en spécifiant le port sur lequel vous souhaitez qu'il écoute les connexions. Utilisez la commande "./server <port>" (par exemple : "./server 3000"). Cela lancera le serveur et il sera prêt à accepter les connexions des clients.
6. Ensuite, pour chaque client que vous souhaitez exécuter, ouvrez un nouveau terminal et déplacez-vous dans le répertoire "bin" comme précédemment. Utilisez la commande "./client <server_ip> <server_port>" pour lancer un client et vous connecter au serveur. Remplacez "<server_ip>" par l'adresse IP du serveur et "<server_port>" par le port sur lequel le serveur écoute (par exemple : "./client 162.111.186.34 3000").
7. Répétez l'étape 6 pour chaque client que vous souhaitez exécuter, en ouvrant un nouveau terminal pour chaque client.
8. Assurez-vous de lancer le serveur et chaque client dans des terminaux séparés.
9. Une fois que vous avez démarré le serveur et les clients, vous êtes prêt à commencer à discuter sur la messagerie. Chaque client aura sa propre interface pour envoyer et recevoir des messages.

En suivant ces étapes, vous pourrez compiler, exécuter et interagir avec le programme de messagerie en utilisant le serveur et les clients fournis.

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

PARTIE II : SYNTHESE

Démarche:

En entamant ce projet, notre objectif était de créer une messagerie instantanée permettant aux clients de communiquer entre eux. Cependant, nous avons accordé une grande importance à rendre cette expérience aussi agréable que possible grâce à une interface conviviale, une connexion sécurisée et des fonctionnalités de haute qualité. Pour ce faire, nous avons organisé notre travail en plusieurs étapes, appelées "sprints", chacune étant dédiée à l'implémentation d'une fonctionnalité spécifique.

Le premier sprint consistait à développer un serveur capable de gérer un nombre N de clients qui s'échangent des messages très simples. Dans le deuxième sprint, nous avons amélioré le code du sprint précédent en le rendant plus propre, tout en ajoutant les premières commandes, telles que les messages privés, les pseudonymes et les mentions. Le troisième sprint était dédié à une fonctionnalité majeure : l'upload et le téléchargement de fichiers de toutes tailles à partir du serveur. Enfin, nous avons créé une fonctionnalité permettant la gestion des salons, incluant l'ajout, la suppression, la connexion et la déconnexion.

Étant donné que nous avons terminé le sprint plus tôt que prévu, nous avons décidé d'implémenter une fonctionnalité de multi-salon sur laquelle nous avons travaillé jusqu'à sa version finale.

Pour réaliser ce projet, nous avons utilisé plusieurs outils, notamment Git et Github, qui nous ont permis de diviser le travail en deux parties distinctes : le client d'un côté et le serveur de l'autre. Cela nous a permis de travailler de manière indépendante et d'avancer beaucoup plus rapidement. Cependant, afin d'éviter tout désordre et d'assurer une coordination efficace, nous avons pris soin d'établir un protocole de communication préalable entre les clients et le serveur. À cette fin, nous avons utilisé des applications telles que LucidChart, qui nous ont permis de créer des diagrammes de séquence de qualité pour visualiser et planifier notre protocole de communication.

En résumé, notre projet de messagerie instantanée s'est déroulé en plusieurs étapes, avec une attention particulière portée à l'interface utilisateur, à la sécurité et à la qualité des fonctionnalités. Grâce à une organisation en sprints, à l'utilisation d'outils tels que Git et Github, ainsi qu'à l'établissement d'un protocole de communication clair, nous avons pu progresser efficacement et réaliser un produit final satisfaisant.

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.

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 engendre des problèmes avec leur nettoyage, mais nous avons réussi à surmonter ce problème en modifiant la file existante.

L'ajout des multi-salons a également posé des difficultés supplémentaires. Dans un premier temps, nous avons envisagé de créer plusieurs terminaux en utilisant uniquement un fichier client et en gérant plusieurs threads. Malheureusement, la création de terminaux à partir d'un seul fichier client s'est avérée difficile, car cela nécessite des processus lourds. Nous avons alors exploré une architecture utilisant des pipes, mais encore une fois, cela s'est révélé impossible, car la création de nouveaux fichiers dédiés aux salons était nécessaire. Après avoir hésité entre l'utilisation de files de messages et de sockets, nous avons finalement opté pour une communication par

socket entre le client principal et ses salons. Cependant, si nous voulions créer plusieurs clients sur la même machine (ne serait-ce que pour les tests), le port de ce socket ne pouvait pas être fixe pour ne pas interférer entre les différents clients. Ainsi, au démarrage du programme, le client principal cherche une socket disponible pour communiquer avec ses salons. Enfin, la dernière difficulté liée à ces salons était le nombre de cas d'erreur possibles avec plusieurs salons en communication simultanée. En effet, même sur un seul terminal, nous devions déjà faire attention à l'entrée utilisateur pour assurer le bon fonctionnement du programme. Maintenant, imaginez les problématiques supplémentaires qui peuvent survenir lorsque nous avons un nombre potentiellement illimité de salons qui communiquent avec ce même terminal, en plus du serveur. Ces difficultés ont nécessité des ajustements dans notre approche initiale, mais nous avons réussi à surmonter les obstacles en choisissant une architecture adaptée et en prenant en compte les contraintes liées à la communication entre les différents éléments du système.

Résultats Intermédiaires :

Chaque sprint a été accompli dans les temps impartis, avec la réalisation des fonctionnalités demandées. Durant le sprint 1, nous avons réussi à mettre en place un serveur capable de relayer des messages textuels entre deux clients. Nous avons utilisé le protocole TCP et géré les échanges de messages jusqu'à ce qu'un client envoie le message "fin". Dans les séances suivantes, nous avons amélioré le serveur et le client en les rendant multi-threadés, ce qui nous a permis de gérer l'envoi de messages dans n'importe quel ordre. Dans le sprint 2, nous avons mis en place un serveur capable de gérer plusieurs clients en utilisant un tableau partagé pour stocker leurs identifiants de sockets. Nous avons également ajouté des fonctionnalités telles que les messages privés, la déconnexion, la gestion des erreurs, la synchronisation des threads et la liste des fonctionnalités disponibles pour le client. Enfin, dans le sprint 3, nous avons ajouté la possibilité d'envoyer et de recevoir des fichiers, ainsi que la gestion des salons de discussion. Chaque sprint a été un succès et nous avons respecté les délais fixés pour chaque livraison.

Résultat Final :

Lors de la réalisation du projet, nous avons réussi à atteindre et même dépasser les objectifs fixés lors de l'introduction. Nous avons implémenté toutes les fonctionnalités demandées, en commençant par un serveur capable de relayer des messages entre deux clients, puis en évoluant vers un système multi-threadé permettant la gestion de plusieurs clients simultanément. De plus, nous avons ajouté des fonctionnalités supplémentaires telles que les messages privés, la déconnexion sécurisée, la gestion des fichiers et la liste des commandes disponibles. Nous avons même innové en introduisant l'utilisation de menus déroulants pour faciliter l'interaction avec l'application, et nous avons réussi à mettre en place un système de multisalon, permettant aux utilisateurs de créer et de rejoindre différentes chaînes de discussion. En fin de compte, nous sommes fiers d'annoncer que nous avons pleinement atteint nos objectifs initiaux et que notre application de messagerie offre une expérience riche et conviviale pour les utilisateurs.

Prospectives d'évolution :

En ce qui concerne les perspectives d'évolution, il existe plusieurs fonctionnalités intéressantes que nous aurions pu implémenter si nous avions disposé de plus de temps. Tout d'abord, nous aurions aimé ajouter un historique des messages spécifiques à chaque salon, permettant aux utilisateurs de consulter les discussions passées. De plus, l'ajout de comptes utilisateurs avec des rôles d'administrateur aurait permis une gestion plus avancée de l'application, offrant des fonctionnalités telles que la modération des discussions et la gestion des utilisateurs.

Une autre fonctionnalité que nous avions envisagée était la gestion de fichiers spécifiques à chaque salon. Cela aurait permis aux utilisateurs de partager des documents et des ressources au sein de leurs discussions et de les trier selon le thème.

Malheureusement, en raison des contraintes de temps, nous avons dû nous concentrer sur la mise en place d'un multi-salon robuste et fonctionnel. Cependant, il est important de noter que notre code a été conçu de manière modulaire et extensible, ce qui facilite l'ajout de nouvelles fonctionnalités à l'avenir. Ainsi, avec un peu plus de temps et de ressources, il serait possible d'étendre notre application pour inclure ces fonctionnalités supplémentaires, offrant ainsi une expérience encore plus enrichissante pour les utilisateurs.

Conclusion/retour sur le projet :

En conclusion, ce projet a été mené à bien et nous sommes extrêmement satisfaits du résultat obtenu. La réalisation de cette application de messagerie nous a permis de développer nos compétences en matière de collaboration et de communication au sein d'une équipe. Nous avons appris à diviser le travail de manière efficace, à fixer des objectifs clairs et à coordonner nos efforts pour atteindre ces objectifs dans les délais impartis. Cette expérience nous a également appris l'importance de la flexibilité et de l'adaptabilité face aux défis qui se sont présentés tout au long du projet. Les aspects techniques auxquels nous avons fait face nous ont permis d'approfondir notre compréhension du langage C et d'élargir notre boîte à outils de programmation.

En fin de compte, nous sommes fiers du produit final que nous avons livré. Notre application de messagerie répond aux objectifs fixés et offre une expérience solide et fonctionnelle pour les utilisateurs. Ce projet a été une occasion précieuse pour nous de grandir en tant que développeur. Nous sommes confiants dans notre capacité à appliquer les connaissances et les compétences acquises dans des projets futurs, et nous sommes impatients de continuer à élargir nos horizons dans le domaine de la programmation.