Université de Pau et des Pays de l'Adour

Collège STEE

Systèmes Distribués & Systèmes d'exploitation II

Projet d'implémentation d'un outil de chat multi-utilisateurs

Rapport de projet

par

Iban LEGINYORA

Année universitaire 2023 - 2024

Table des matières

Introduction

Ce rapport vise à présenter et détailler l'implémentation d'un système de chat multi-utilisateurs du point de vue des systèmes distribués et d'exploitation.

Dans ce contexte, nous examinerons les différentes étapes de conception, de développement et de déploiement de ce système, en mettant en évidence les aspects techniques et les choix d'architecture.

Nous commencerons par présenter le fonctionnement général du système, en décrivant son utilisation en mode local et en réseau, ainsi que son arborescence de fichiers et les commandes de compilation associées.

Ensuite, nous aborderons l'exécution de l'application, en détaillant les étapes nécessaires au lancement du serveur et des clients, ainsi que l'utilisation de l'interface utilisateur et l'affichage de l'interprétation des requêtes côté serveur.

Enfin, nous examinerons l'implémentation du système du point de vue des systèmes distribués et d'exploitation, en mettant en évidence les différents aspects techniques et les choix de conception.

Nous conclurons ce rapport en présentant quelques pistes d'amélioration et de développement futur pour le système de chat multi-utilisateurs.

1

Fonctionnement général

Le système de chat peut être utilisé en mode local sur un seul ordinateur, mais également en mode réseau. Dans ce cas, il est nécessaire d'avoir au moins trois ordinateurs : l'un servant de serveur de la partie Gestion Comptes, l'un de serveur de la partie centrale et les autres en tant que client.

Voici un rappel de l'architecture générale de l'application fournie dans le sujet :

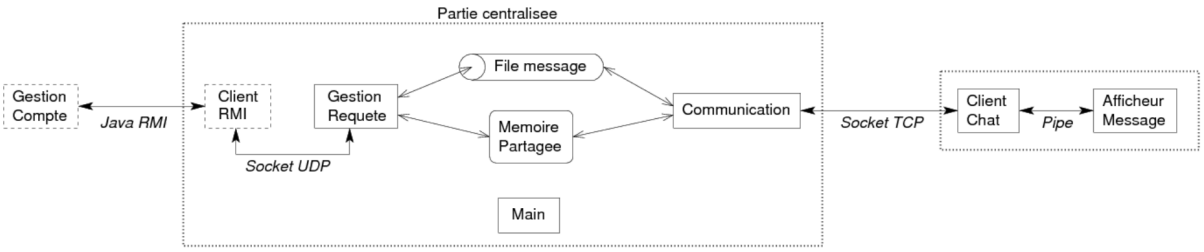

Figure 1.1 – Architecture générale du système

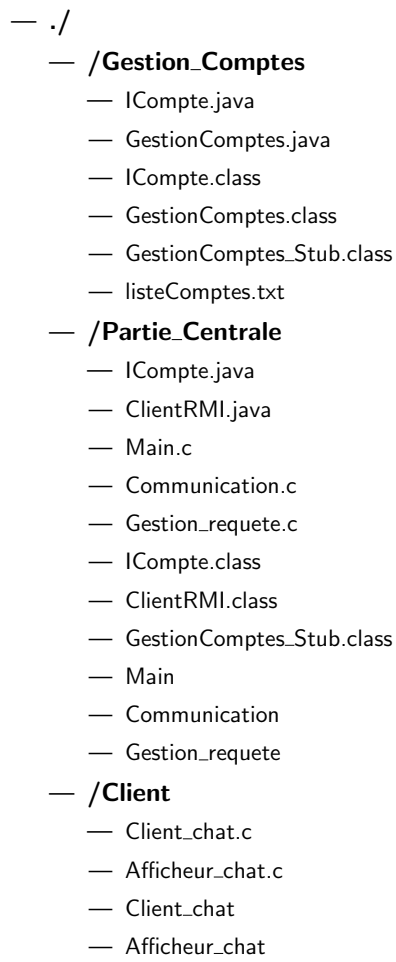

Figure 1.2 – Arborescence des fichiers du projet

1.1 Compilation du projet

Bien que les exécutables du projet soient fournis, voici les différentes commandes de compilation des fichiers nécessaires :

Compilation de la partie Gestion_Comptes

```
cd Gestion_comptes
javac GestionComptes.java
```

Générer le talon de `GestionComptes.class` (dans le cadre de l'utilisation de Java RMI), et le copier dans le dossier `Partie_centrale`

```
cd Gestion_comptes
rmic GestionComptes
cp GestionComptes_Stub.class ../Partie_centrale
```

Compilation de la partie Partie_Centrale

```
cd Partie_Centrale
javac ClientRMI.java
gcc -o Gestion_requete Gestion_requete -pthread -lrt
gcc -o Communication Communication.c -pthread -lrt
gcc -o Main Main.c -lrt
```

Compilation de la partie Client

```
cd Client
gcc -o Afficheur_chat Afficheur_chat.c
gcc -o Client_chat Client_chat.c -pthread
```

Remarques

L'option `-pthread` lie le programme à la bibliothèque `pthread`, permettant ainsi l'utilisation des threads.

L'option `-lrt` lie le programme à la bibliothèque `librt`, facilitant l'utilisation des fonctionnalités liées au temps, notamment dans le contexte de la mémoire partagée.

1.2 Exécution de l'application

1.2.1 Lancement du serveur

Partie Gestion Comptes

```
cd Gestion_Comptes
rmiregistry &
java GestionComptes
```

Partie centrale

```
cd Partie_centrale
./Main <nom_serveur_gestion_comptes> <numero_port>
```

Pour exécuter le fichier Main, il faut spécifier le nom de la machine où est lancé le processus `GestionComptes` (`ip addr` ou `hostname`) et un port d'écoute composer de 4 chiffres.

1.2.2 Lancement d'un client

Pour lancer un client, il faut connaître le nom du serveur `Main` (`ip addr` ou `hostname`) et le numéro de port (spécifier au-dessus) :

```
cd Client
./Client_chat <nom_serveur_main> <numero_port>
```

1.3 Utilisation des interfaces

1.3.1 Côté Client

Une fois connecté au système, le client dispose, dans un premier temps de trois options :

— Se connecter
— Créer un compte
— Quitter

Les comptes sont enregistrés dans le fichier `listeComptes.txt` avec un pseudonyme et un mot de passe.

Une fois authentifié, l'utilisateur pourra accéder à plusieurs fonctionnalités :

— Afficher les utilisateurs présents
— Envoyer un message
— Se déconnecter
— Quitter

Parallèlement, un terminal s'ouvre pour afficher les messages reçus. Si l'utilisateur choisit de se déconnecter, il sera renvoyé au menu de connexion initial. Le programme s'arrête lorsque l'utilisateur quitte le système avec la fonctionnalité fournie, ou exécutant la combinaison `CTRL+C`.

1.3.2 Côté Serveur

Dans le terminal exécutant `Main`, le programme `Communication` indique l'état du serveur (état de connexion, numéro de port...) et interprète les requêtes envoyées, en affichant le type de requête (connexion, déconnexion, liste utilisateur, message...) ainsi que le retour positif ou négatif. Cette affichage est utile en cas de dysfonctionnement du système et permet de localiser précisément le problème.
Il est possible d'arrêter volontairement le serveur en utilisant la combinaison `CTRL+C` ou en saisissant la commande '`exit`' dans le terminal du `Main` de la partie centrale.

2

Implémentation du système

La conception du système repose sur l'interface `ICompte.java` fournie. Dans cette section, nous détaillerons l'ensemble des logiques d'implémentation. Pour ce faire, nous commencerons par explorer la mise en œuvre des fonctionnalités liées aux systèmes distribués, telles que la communication entre les clients et le serveur, la gestion des comptes utilisateur et la coordination des échanges de messages. Ensuite, nous aborderons l'implémentation des fonctionnalités liées aux systèmes d'exploitation, notamment la création et la gestion des processus, la communication inter-processus, ainsi que la gestion des ressources partagées, telles que la mémoire partagée et les files de messages.

2.1 Partie Systèmes distribués

2.1.1 Interopérabilité logicielle

Pour assurer une meilleure interopérabilité entre les systèmes, permettant le traitement des données entre des langages différents, entre autre, le choix a été d'opter pour la transmission des données sous forme de chaînes de caractères. Voici les types de requête :

<table>
<tr><th>Type</th><th>Données d'envoie</th><th colspan="2">Données retour</th></tr>
<tr><td>CONNEXION</td><td rowspan="4">pseudo_mdp</td><td rowspan="4">VALIDE</td><td rowspan="4">INVALIDE</td></tr>
<tr><td>DECONNEXION</td></tr>
<tr><td>CREERCOMPTE</td></tr>
<tr><td>SUPPRIMERCOMPTE</td></tr>
<tr><td>LISTEUTILISATEUR</td><td></td><td colspan="2">pseudo1_pseudo2_pseudo3</td></tr>
<tr><td>MESSAGE</td><td>@pseudo>message</td><td rowspan="2">0</td><td rowspan="2">1</td></tr>
<tr><td>QUITTER</td><td>SORTIE | pseudo_mdp</td></tr>
</table>

Exemple : `CONNEXION_Jean_aout64`

2.1.2 Communication

Pour communiquer entre eux, différentes méthodes ont été employées afin de satisfaire aux mieux les exigences du système.

Socket UDP

Dans la partie centrale, le protocole UDP (User Datagram Protocol) a été utilisé entre le processus `ClientRMI` et `Gestion_requete` avec un port spécifié en dur dans le code (7777). Etant donné que la communication s'effectue au sein de la même machine, il n'y a pas besoin de se soucier de la fiabilité du transport.

Socket TCP

En revanche pour avoir une garantie de la transmission des données entre les processus `Client_chat` et `Communication`, il a été obligé d'utiliser des sockets avec le protocole TCP (Transmission Control Protocol). La gestion des retours de requête et de leur éventuelle erreur durant leur traitement s'effectue au sein du processus `Client_chat`. Voici les différents états de retours possibles :

- `0` : succès de la requête ;
- `1` : échec de la requête ;
- `2` : le traitement de la requête a été trop long, on suppose un plantage du serveur.
- `4` : Echec de la reqûete, cette partie n'a pas été implémentée côté serveur.

Il y a un cas particulier, si la requête concerne la liste d'utilisateurs actifs, cela signifie que le processus `Communication` a retourné une liste (format : `pseudo1_pseudo2_pseudo3`).

Middleware Java RMI

Le middleware Java RMI a été employé pour appeler à distance les méthodes du processus `Gestion_comptes`, afin quelles soient utilisées dans le processus `ClientRMI`. Pour ce faire, lors du lancement de la partie centrale par le processus `Main`, `ClientRMI` est appelé avec l'adresse IP de du serveur `Gestion_requete` précisé en paramètre du `Main`. Il est alors nécessaire de générer un talon de l'interface `ICompte` côté client et serveur.

2.1.3 Transparence de l'application

Dans le cadre du projet, plusieurs aspects de la transparence peuvent être identifiés et mis en évidence. La transparence fait référence à la capacité d'un système à agir de manière claire et unie pour l'utilisateur. On peut identifier quatre de ces aspects :

— **Transparence de localisation :** L'application vise à offrir une expérience transparente aux utilisateurs, indépendamment de leur emplacement physique. Cela signifie que les utilisateurs peuvent accéder aux fonctionnalités de l'application de n'importe où, tant qu'ils ont une connexion réseau.

— **Transparence de concurrence :** Pour garantir un fonctionnement fluide de l'application, il a fallu utiliser de threads au niveau du processus `Client_chat`. Cela permet de gérer efficacement les différentes tâches en parallèle, comme le traitement des requêtes (envoi/réception) et la réception des messages des autres utilisateurs, tout en évitant les blocages ou les ralentissements.

— **Transparence d'accès :** L'application offre une transparence d'accès en permettant aux utilisateurs d'accéder à la liste des comptes enregistrés sur un autre ordinateur. Cela signifie que des utilisateurs peuvent s'inscrire et se désinscrire de la liste des comptes (`listeComptes.txt`), même s'ils sont connectés à un serveur distant.

— **Transparence de performance :** Le système peut être reconfiguré pour augmenter la taille des ressources allouées, avec les constantes qui permettent de limiter le nombre de connexions ou de requêtes simultanées. Cela permet d'adapter l'application aux besoins changeants et de garantir des performances optimales même dans des conditions variables.

2.1.4 Mode d'intéraction

Dans l'application, plusieurs modes d'interaction sont mis en place pour assurer le bon fonctionnement du système :

— **Modèle client/serveur :** Le modèle client/serveur est au cœur de l'architecture. Il permet une communication efficace entre les différents processus de l'application. Principalement, il établit une communication entre le client et la partie centrale du système, mais aussi entre les différents composants de la partie centrale, tels que `ClientRMI` et `Gestion_requete`.

— **Diffusion de message :** La diffusion de message est utilisée pour transmettre à tous les utilisateurs les messages envoyés par l'un des clients. Cette fonctionnalité garantit que tous les utilisateurs connectés reçoivent les messages en temps réel, favorisant ainsi une communication fluide et instantanée.

— **Mémoire partagée :** Elle permet de stocker les utilisateurs actifs. Cette mémoire est initialisée par le processus `Main`, puis modifiée (ajout/suppression d'utilisateurs) par le processus `Gestion_requete`. Enfin, elle est lue par le processus `Communication` pour transmettre la liste des utilisateurs actifs au client ayant fait la demande. Cette approche garantit une synchronisation efficace des données et une mise à jour en temps réel de la liste des utilisateurs.
La mémoire partagée est initialisée avec comme adresse le numéro de port renseigné lors du lancement du processus `Main`.

2.1.5 Reprise et gestion des erreurs

Lorsqu'un des clients se termine de manière inattendue, un mécanisme de gestion des erreurs est activé. De même, si le serveur rencontre un dysfonctionnement, cela est détecté grâce à une limite de temps pour le retour des requêtes. Si cette limite est dépassée, le client est informé d'une erreur système. Toutefois, la partie ayant planté peut être relancée sans causer de problème externe.

2.2 Partie Systèmes d'exploitation

2.2.1 Manipulation des processus

Certains de ces processus sont lancés via d'autres processus. C'est notamment le cas pour la partie centrale de l'application. Le processus `Main` va manipuler deux processus enfants :

- `ClientRMI` (en Java) avec les fonctions :
  - `LancerClientRMI`
  - `TerminerClientRMI`
- `Communication` (en C) avec les fonctions :
  - `LancerCommunication`
  - `TerminerCommunication`

Les fonctions de lancement vont créer les processus fils respectivement à l'aide de `fork`. Dans les processus fils, la fonction `execlp` et utiliser pour exécuter un programme (`ClientRMI` et `Communication`).
Dans la partie cliente, il y a une manipulation du processus au niveau du lancement de l'afficheur de chat. On peut retrouver cette manipulation dans la fonction `LancerAfficheur(Client *client)`, où il y a un nouveau processus qui est créé pour exécuter un terminal `GNOME` où l'afficheur de message sera lancé.

2.2.2 Communication inter-processus

Cette communication est multiple, la mémoire partagée, les tubes de communication et les signaux ont été employés dans ce système.

La mémoire partagée a été détaillé dans la partie des Systèmes distribuées (cf. 2.1.4 Mémoire partagée).

Un tube de communication a été utilisé dans `Client_chat` pour transmettre les messages reçus des autres clients vers `Afficheur_message`. Le tube est créé dans la fonction `CreerTube(Client *client)` par la méthode open avec un nom unique générer à partir du `PID` du processus client. L'écriture dans le tube est gérée par la fonction `EcrireTube(Client *client, const char *message)`. Cette fonction utilise la fonction `write` pour écrire le message dans le tube.

Un gestionnaire de signal est mis en place pour intercepter le signal `SIGINT` (généralement déclenché par `CTRL+C`). Il est implémenté dans tous les processus ayant une interaction directe avec les utilisateurs. Nous allons nous concentrer sur celui de la partie centrale, se situant donc

dans le processus `Main`. Lorsque le signal est détecté, le gestionnaire de signal appelle les fonctions `TerminerCommunication` et `TerminerMemoirePartagee` pour terminer proprement la communication et supprimer la mémoire partagée.

2.2.3 Synchronisation des processus

Pour synchroniser les processus entre eux, l'utilisation des threads et des sémaphores a été essentielle, notamment dans le processus `Client_chat` afin d'avoir en parallèle la gestion des commandes basiques (gestion du compte, affichage des utilisateurs actifs, envoie de message) et la réception en continu des messages reçus par l'utilisateur. Voici les threads correspondant :

- `Client.threadEnvoi` : permet d'envoyer et de recevoir les requêtes.
- `Client.threadEcoute` : permet d'écouter les messages transmis par le serveur de la partie centrale.

Ces threads sont créés avec la fonction `pthread_create()` et exécute des fonctions spécifiques :

- `Client.threadEnvoi` : `*ThreadPrincipal(void *arg)`
- `Client.threadEcoute` : `*ThreadEcoute(void *arg)`

Les sémaphores sont des mécanismes utilisés pour contrôler l'accès concurrentiel aux ressources partagées. Dans le client, ils sont utilisés pour garantir que seule la tâche `threadEnvoi` peut accéder aux variables de réponse dès qu'une requête est saisie par l'utilisateur. On va alors verrouiller le mutex (`pthread_mutex_lock()`) avant l'envoi de la requête au serveur et débloquer après la réception du serveur et le traitement de sa réponse (`pthread_mutex_lock()`).

2.2.4 Ordonnancement des processus

Étant donné que le but de cette application et l'échange de message entre plusieurs utilisateurs, le processus `Communication` peut être amené a recevoir des requêtes client en simultané. Pour cela il a été nécessaire d'implémenter une file de message. Cette file de message est initialisée dans le processus `Communication` avec la structure `FileMessage` :

```
typedef struct {
    ClientRequete tab_messages[MAX_CLIENTS];
    int debut;
    int fin;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
} FileMessage;
```

Cette structure est appeler et utiliser par la fonction `*ProcessusRequetes(void *arg)`. Cette file respecte la politique `FIFO`. Étant donné qu'il n'y a pas de notion de priorité d'une requête par rapport à une autre, ça sera la requête qui arrive en première qui sera traitée en première.

3

Remarques

3.1 Améliorations envisageables

De nombreuses pistes d'amélioration sont envisageables permettant de rendre l'expérience utilisateur plus agréable. Une option pourrait consister à intégrer la possibilité d'interrompre une fonctionnalité en cours d'utilisation. Par exemple, lorsqu'un utilisateur souhaite créer un compte, il est actuellement contraint de saisir un pseudonyme et un mot de passe sans possibilité de revenir en arrière, étant obligé de terminer toute la procédure. Il serait intéressant d'inclure une fonctionnalité permettant d'interrompre cette opération à tout moment et de l'appliquer à l'ensemble des fonctionnalités proposées. De plus, l'ajout d'une fonctionnalité permettant, non pas d'envoyer à tous les utilisateurs, mais de cibler un utilisateurs et donc d'ouvrir des conversations privées serait un plus. Au lieu d'envoyer des messages à tous les utilisateurs, cette fonctionnalité offrirait la possibilité d'ouvrir une conversation individuelle et privée, améliorant ainsi la personnalisation et la confidentialité des communications.

Conclusion

En conclusion, ce projet d'implémentation d'un outil de chat multi-utilisateurs a permis d'utiliser de nombreux concepts liés aux systèmes distribués et aux systèmes d'exploitation.

Le rapport a permis d'observer et analyser en détail le fonctionnement général de l'application, en mettant en évidence son architecture, sa conception et son déploiement, ainsi que les différentes étapes de sa mise en œuvre.

Du point de vue des systèmes distribués, nous avons étudié les mécanismes de communication inter-processus, en utilisant des technologies telles que les sockets TCP et UDP, le middleware Java RMI et la mémoire partagée. Nous avons également vu les concepts de transparence, de concurrence et de synchronisation des processus, en mettant en œuvre des threads et des sémaphores pour garantir un bon fonctionnement en parallèle de l'application.

Du côté des systèmes d'exploitation, nous avons manipulé les processus, géré la communication inter-processus à l'aide de tubes et de signaux, et assuré la synchronisation des processus à l'aide d'une file de message et de mécanismes de verrouillage.

Enfin, nous avons identifié plusieurs pistes d'amélioration pour rendre l'expérience utilisateur plus agréable, telles que l'ajout de fonctionnalités pour interrompre les opérations en cours et ouvrir des conversations privées.