Application de messagerie instantanée avec salon de discussion et partage de fichiers

EGLOFF Nicolas, LUNET Louis

RE213 - Projets de Programmation Réseaux
Réseaux et Informatique

16 octobre 2024

Supervisé par :
BRUNEAU-QUEYREIX Joachim
jbruneauqueyreix@bordeaux-inp.fr

BORDEAUX INP Enseirb-Matmeca

Table des matières

Table des figures

1 Introduction

1.1 Présentation du projet

L'objectif de ce projet est de développer une application de messagerie instantanée avec un système de salon de discussion et un partage de fichiers via un serveur central. Ce type de projet combine plusieurs aspects essentiels de la programmation réseau et de la gestion des utilisateurs, avec une application pratique : un service de communication en temps réel, similaire à des plateformes comme *Slack*, *Discord* ou *IRC*.

L'application doit fournir des fonctionnalités de base telles que la possibilité de rejoindre un salon de discussion (par exemple, un canal thématique), d'envoyer des messages en temps réel, et de partager des fichiers avec les autres utilisateurs du salon.

1.2 Objectifs du projet

1. Permettre à plusieurs utilisateurs de se connecter à un serveur, de participer à des salons de discussion, et d'échanger des fichiers en temps réel.
2. Gérer l'authentification des utilisateurs, les droits d'accès et la sécurité des échanges.
3. Offrir une expérience interactive en temps réel avec des fonctionnalités de partage et de gestion de fichiers.

1.3 Exigences techniques

Les exigences techniques de ce projet sont les suivantes :
— Gestion et authentification des utilisateurs.
— Salons de discussion.
— Messagerie en temps réel.
— Partage de fichiers.
— Gestion des droits d'accès sur les fichiers partagés.

1.4 Extensions possibles

Les extensions envisagées pour ce projet sont :
— Sauvegarde des discussions.
— **Synchronisation multi-serveurs**.
— Gestion des collisions de versions de fichiers.
— Implémentation de transferts sécurisés avec SSL/TLS.
— Utilisation de protocoles de contrôle de flux pour améliorer les performances des transferts.

Nous avons choisi d'ajouter la **synchronisation multi-serveurs** comme extension à ce projet.

2 Spécifications Fonctionnelles

2.1 Gestion des utilisateurs

2.1.1 Connexion au serveur

Les utilisateurs se connectent principalement au serveur correspondant à leur région géographique. Cela permet d'améliorer la qualité et la rapidité des connexions, tout en réduisant la latence. En accédant au serveur de leur région, ils bénéficient d'une expérience utilisateur plus fluide, car les données n'ont pas besoin de traverser de grandes distances. Cette approche favorise également une répartition équilibrée de la charge sur les serveurs, assurant ainsi une performance optimale et une meilleure gestion des ressources pour l'ensemble du réseau.

2.1.2 Authentification et gestion des sessions

Un système d'authentification par identifiant unique permet aux utilisateurs de s'authentifier. Les sessions sont gérées côté serveur pour maintenir l'intégrité des connexions.

2.1.3 Formulaire de première connexion

Lors de la première connexion, les utilisateurs remplissent un formulaire qui demande des informations personnelles (âge, sexe)

2.2 Messagerie instantanée

2.2.1 Communication en temps réel

Les utilisateurs ont la possibilité de se connecter à un groupe dédié afin de communiquer et d'échanger entre eux. Ce groupe sert de plateforme collaborative, permettant aux membres d'interagir en temps réel, de partager des idées, des informations ou des documents, et de poser des questions.

2.2.2 Fonctionnalités avancées

Une fonctionnalité avancée offerte aux utilisateurs est la possibilité de partager des fichiers entre clients via un espace de stockage centralisé, type "drive", hébergé par deux serveurs synchronisés. Cette configuration assure une redondance des données, garantissant une haute disponibilité et une sécurité accrue. Chaque fichier déposé sur le drive est automatiquement répliqué sur les deux serveurs, de sorte

qu'en cas de défaillance de l'un d'entre eux, les données restent accessibles sans interruption. Cette synchronisation en temps réel permet aux utilisateurs de collaborer de manière fluide tout en bénéficiant d'une protection contre la perte de données et d'un accès continu aux fichiers partagés.

3 Architecture générale de l'application

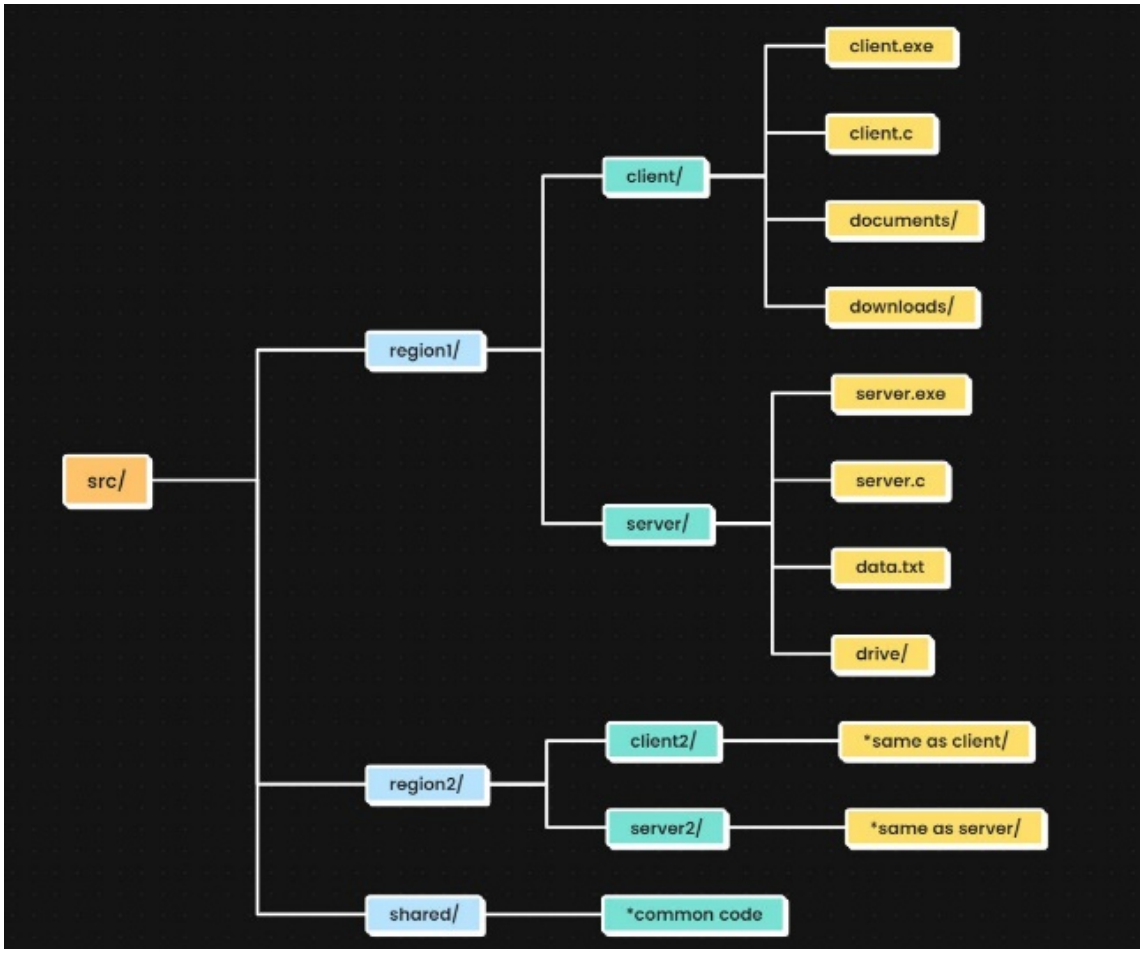

FIGURE 1 – Architecture de l'application

Dans le cadre de la synchronisation entre deux serveurs, nous avons simulé deux environnements distincts pour représenter des "machines" différentes. L'architecture du code est divisée en deux répertoires, `region1` et `region2`, qui séparent les groupes client/serveur respectifs. Chaque région contient deux sous-dossiers : `client` et `server`. Ces répertoires hébergent le code client et serveur spécifiques à chaque région (client2 et server2 pour region2).

Les fonctions communes utilisées par les deux serveurs sont regroupées dans le fichier `shared/utils`, qui centralise la logique partagée.

3.1 Synchronisation des serveurs

Le système repose sur une architecture à deux serveurs synchronisés en permanence, qui partagent les données et les opérations effectuées par les utilisateurs. Cette synchronisation garantit la redondance des informations et assure la haute disponibilité du service. Toutefois, certaines actions spécifiques ne sont pas synchronisées entre les serveurs :

— Connexion des utilisateurs (`login`).
— Opérations de listage des fichiers (`list_files`).
— Listage des groupes (`list_groups`).

Ces actions sont gérées indépendamment par chaque serveur afin d'optimiser la répartition de la charge et d'éviter toute duplication inutile de données temporaires, propres à chaque session utilisateur. Cependant, toutes les autres données, telles que les fichiers partagés, les mises à jour ou les modifications, sont entièrement répliquées entre les deux serveurs. Cela garantit la continuité du service en cas de panne de l'un des serveurs.

Concernant la gestion des fichiers, lors de l'initialisation du serveur, des dossiers sont créés dans le sous-répertoire `drive`. Un dossier est dédié à chaque groupe d'utilisateurs respectifs, assurant un stockage organisé et sécurisé.

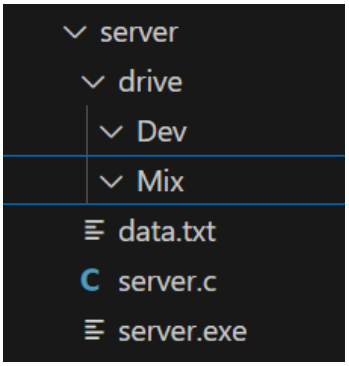

FIGURE 2 – Emplacement du stockage

3.2 Gestion de la persistance des données utilisateur

La persistance des données utilisateur repose sur une stratégie de stockage sélectif. Seuls les documents partagés via le répertoire `drive`, ainsi que les informations relatives aux comptes utilisateurs et aux groupes, sont stockés de manière permanente. Ces données sont synchronisées en temps réel entre les deux serveurs afin de garantir une haute disponibilité et de prévenir la perte de données.

En revanche, les messages échangés entre utilisateurs ne sont pas stockés de manière persistante. Ils sont traités de manière éphémère et ne sont pas conservés sur le long terme. Cette approche permet de préserver la confidentialité des discussions tout en réduisant la charge de stockage sur les serveurs.

4 Choix de Conception

4.1 Protocole de communication utilisé

Le protocole de communication entre les clients et les serveurs, ainsi qu'entre les serveurs eux-mêmes, est implémenté dans `shared/socket_utils`. Les principales méthodes utilisées sont `send_message` et `receive_message`.

```
void print_error(int result, char *s);
int read_message_from_socket(int fd, char *buffer, int size);
void write_on_socket(int fd, char *s);
int read_int_from_socket(int fd);
void write_int_as_message(int fd, int val);
void send_message(int fd, char *message, int size, int flag);
void receive_message(int fd, char *buffer, int size, int flag);
```

FIGURE 3 – Functions

Le protocole repose sur l'envoi de la taille du message avant l'envoi du contenu du message lui-même. Cela permet de s'assurer que le destinataire reçoit le message complet sans coupures, même en cas de latence réseau.

```c
void send_message(int fd, char *message, int size, int flag)
{
    write_int_as_message(fd, size);
    printf("Message size: %d\n", size);
    write_on_socket(fd, message);
}

void receive_message(int fd, char *buffer, int size, int flag)
{
    int message_size = read_int_from_socket(fd);
    printf("Message size: %d\n", message_size);

    int read_status = read_message_from_socket(fd, buffer, message_size);
    if (read_status <= 0)
    {
        printf("Server disconnected or error occurred.\n");
        // exit(EXIT_FAILURE);
    }
    buffer[message_size] = '\0';
}
```

FIGURE 4 – Send and receive functions

4.2 Structure des messages échangés

Les messages envoyés sont sous la forme de chaînes de caractères (de type `char *`) avec la structure suivante :

```
Command arg1 arg2 arg3 arg4
```

Selon la commande envoyée, des gestionnaires spécifiques (handlers) sont déclenchés en prenant en compte les arguments fournis. Cela permet d'exécuter différentes actions selon la commande reçue.

5 Requêtes et Réponses

5.1 Connexion et Authentification

La connexion d'un client au serveur s'effectue automatiquement lors de l'exécution du programme client. En cas d'échec de connexion, le client s'arrête immédiatement. Des comptes utilisateurs prédéfinis sont initialisés sur le serveur pour permettre la connexion des utilisateurs. Il est également possible de créer de nouveaux comptes utilisateurs.

L'authentification est obligatoire pour certaines commandes sensibles, telles que la consultation ou l'adhésion à un groupe. Le processus de connexion est géré par une fonction dédiée qui associe chaque utilisateur connecté à un descripteur de fichier (`fd`). Cette association permet de faciliter la communication entre le serveur et l'utilisateur, notamment lors de l'envoi de messages ou des actions liées à un groupe (par exemple, quitter un groupe).

Le login est assuré par la fonction suivante :

```
*/
void handle_login(int client_fd, char *username, char *password)
{
    for (int i = 0; i < user_count; i++)
    {
        if (strcmp(users[i].username, username) == 0 && strcmp(users[i].password, password) == 0)
        {

            send_message(client_fd, "Login successful\n", 17, 0);
            add_client(username, client_fd);
            return;
        }
    }

    send_message(client_fd, "Login failed\n", 13, 0);
}
```

FIGURE 5 – Handle_login

Par la suite, le serveur associe un fd à un username en se servant de la struct suivante :

```
struct client_info
{
    char username[50];
    int fd;
};
```

FIGURE 6 – Struct client_info

5.2 Envoi et réception des commandes client-serveur

Le client fonctionne avec deux boucles `while(1)`. La première est dédiée à la gestion des commandes de type "menu" telles que :

- `login`
- `create_user`

— `list_groups`
— `join_group`

La seconde boucle est utilisée lorsque le client a rejoint un groupe. Elle utilise la bibliothèque `pollfd` pour écouter simultanément les messages entrants provenant du serveur et traiter les commandes de l'utilisateur. Lorsqu'une commande est reçue par le serveur, elle est analysée (`parsed`) pour en extraire la commande et ses arguments. Le serveur appelle ensuite le gestionnaire (handler) correspondant à la commande spécifique.

```
void add_client(const char *username, int fd);
void remove_client(int fd);
void handle_login(int client_fd, char *username, char *password);
void handle_create_user(int client_fd, char *username, char *gender, int age, char *password);
void handle_upload_file(int client_fd, const char *group_name, const char *file_name);
void handle_download_file(int client_fd, const char *group_name, const char *file_name);
void handle_list_files(int client_fd, const char *group_name);
void handle_list_groups(int client_fd);
void handle_join_group(int client_fd, char *username, char *group_name);
int get_client_fd_by_username(const char *username);
void handle_message(int client_fd, char *group, char *user, char *message, int type);
void handle_client(int client_fd);
void print_data();
```

FIGURE 7 – Handles functions

5.3 Envoi et réception des commandes serveur-serveur

Les serveurs établissent d'abord une connexion entre eux avant d'accepter toute autre communication. Par convention, le descripteur de fichier pour la communication inter-serveurs est défini par le numéro 4. Un `#define` dans le code permet d'identifier cette connexion particulière.

```
#define OTHER_SERVER_FD 4
```

FIGURE 8 – define other_server_fd

La communication serveur/serveur est limitée à certaines commandes spécifiques, notamment :

— `join_group`
— `message`
— `upload_file`

Cette communication est gérée par un simple filtre conditionnel (`if`), afin de s'assurer que seules les commandes concernées transitent entre les serveurs.

```c
if (client_fd != OTHER_SERVER_FD &&
    (strncmp(buffer, "join_group", 10) == 0 ||
     strncmp(buffer, "message", 7) == 0 ||
     strncmp(buffer, "create_user", 11) == 0))
{
    // Forward the command to the second server
    printf("sending command to server\n");
    send_message(OTHER_SERVER_FD, buffer, strlen(buffer), 0);

    // Receive response from the second server
    char response[BUFFER_SIZE];
    printf("\n\nAwaiting for recv ...\n\n");

    receive_message(OTHER_SERVER_FD, response, sizeof(response), 0);
}
```

FIGURE 9 – Filtre conditionnel

6 Gestion des Fichiers

L'application implémente des fonctionnalités robustes pour l'envoi et la réception de fichiers entre les clients et les serveurs, utilisant des protocoles de communication fiables pour garantir que les fichiers sont transmis de manière sécurisée et complète. Cette section décrit les processus de téléchargement (upload) et de téléchargement inverse (download) des fichiers entre un client et un serveur.

6.1 Upload de fichiers (Client vers Serveur)

Lorsqu'un client souhaite uploader un fichier vers le serveur, la fonction `handle_upload_file` gère ce processus côté serveur. Les étapes principales de ce processus sont les suivantes :

1. **Recherche du groupe** : Le serveur commence par identifier le groupe auquel le fichier doit être envoyé en parcourant la liste des groupes. Si le groupe n'est pas trouvé, le serveur envoie un message d'erreur au client : `"Group not found"`.
2. **Création du chemin de fichier** : Une fois le groupe trouvé, le serveur construit le chemin complet du fichier en utilisant le nom du groupe et le nom du fichier, garantissant une organisation structurée dans des répertoires spécifiques à chaque groupe.
3. **Ouverture du fichier** : Le serveur tente ensuite d'ouvrir le fichier en mode écriture binaire (`wb`). Si l'ouverture échoue (par exemple, à cause de droits d'accès ou de répertoires manquants), le serveur informe le client de l'erreur avec un message d'erreur : `"Error opening file"`.
4. **Réception des données** : Le serveur notifie le client qu'il est prêt à recevoir le fichier en envoyant le message `"SERVER_READY"`. Le client envoie ensuite la taille du fichier, et le serveur confirme avec un message `"SIZE_OK"`. Le serveur commence alors à recevoir les blocs de données et les écrit directement dans le fichier. Ce processus continue jusqu'à ce que le fichier complet soit reçu.

5. **Vérification de l'intégrité** : Le serveur compare la quantité totale de données reçues avec la taille du fichier attendue. Si ces valeurs correspondent, le serveur signale que le fichier a été reçu avec succès ; sinon, il informe l'utilisateur d'une réception incomplète.

6.2 Téléchargement de fichiers (Serveur vers Client)

Le téléchargement d'un fichier du serveur vers un client est géré par la fonction `handle_download_file`. Les étapes clés de ce processus sont :

1. **Création du chemin de fichier** : Comme pour l'upload, le serveur commence par construire le chemin d'accès complet au fichier demandé par le client.
2. **Ouverture du fichier** : Le serveur tente d'ouvrir le fichier en mode lecture binaire (`rb`). Si le fichier n'existe pas ou s'il ne peut être ouvert, un message d'erreur est envoyé au client : `"Error opening file"`.
3. **Préparation de la réception par le client** : Le serveur attend une confirmation de la part du client indiquant qu'il est prêt à recevoir le fichier. Si le client envoie le message `"SERVER_READY"`, le serveur continue le processus.
4. **Envoi de la taille du fichier** : Avant de commencer le transfert de données, le serveur envoie la taille du fichier au client. Cela permet au client de connaître à l'avance la quantité de données à recevoir et de gérer correctement la réception.
5. **Transfert des données** : Le serveur lit le fichier par blocs et envoie ces blocs au client. Le transfert continue jusqu'à ce que l'intégralité du fichier soit transmise.
6. **Vérification et clôture** : Une fois le transfert terminé, le fichier est fermé et le serveur affiche un message confirmant que le fichier a été envoyé avec succès.

6.3 Gestion des erreurs

Le système de gestion des fichiers comprend plusieurs mécanismes de traitement des erreurs pour s'assurer que les problèmes sont gérés efficacement. Si une erreur se produit lors de l'ouverture, la lecture, ou l'écriture d'un fichier, ou si une déconnexion du client survient de manière inattendue, des messages d'erreur sont immédiatement envoyés au client. Le serveur prend également des mesures pour fermer proprement la connexion et éviter la corruption de données.

6.4 Résumé

La gestion des fichiers dans cette application permet une communication fluide entre les clients et les serveurs, tout en garantissant que les fichiers sont transmis

de manière sécurisée et fiable. L'organisation structurée des fichiers dans des dossiers spécifiques aux groupes, combinée à des mécanismes robustes de gestion des erreurs et de contrôle d'intégrité des fichiers, assure une utilisation efficace et sans interruption du système de partage de fichiers.

7 Conclusion et axes d'améliorations

7.1 Conclusion

Le projet que nous avons réalisé, une application de messagerie instantanée avec partage de fichiers, a permis de répondre à l'ensemble des objectifs fixés au départ. L'application offre une communication en temps réel au sein de salons de discussion, tout en permettant aux utilisateurs de partager des fichiers via une infrastructure de serveurs synchronisés. La synchronisation entre serveurs assure une haute disponibilité des fichiers, et la gestion des utilisateurs et de leurs droits d'accès garantit une utilisation sécurisée du service.

Cette expérience nous a permis de mettre en pratique des concepts essentiels de la programmation réseau, notamment l'authentification, la gestion de sessions, la communication en temps réel, ainsi que la gestion des transferts de fichiers entre clients et serveurs. Le développement de ce projet a également souligné l'importance de la persistance des données et de la gestion efficace des ressources réseau.

7.2 Axes d'amélioration

Bien que le projet soit fonctionnel, plusieurs aspects pourraient être améliorés pour optimiser l'application et ajouter de nouvelles fonctionnalités. Voici quelques pistes d'amélioration envisageables :

- **Amélioration de la sécurité des transferts** : Actuellement, les échanges de données se font en clair. L'implémentation de protocoles de chiffrement tels que SSL/TLS pour sécuriser les communications entre clients et serveurs pourrait être une amélioration majeure pour assurer la confidentialité des échanges.
- **Gestion des versions de fichiers** : Une gestion plus fine des versions des fichiers partagés serait utile pour éviter les conflits lors des modifications simultanées. Un système de contrôle de version pourrait être ajouté pour suivre les différentes versions des fichiers et prévenir les pertes de données en cas de modifications concurrentes.
- **Sauvegarde et récupération des messages** : À l'heure actuelle, les messages échangés ne sont pas persistants et sont perdus après déconnexion. La mise en place d'un système de sauvegarde des conversations permettrait aux utilisateurs de retrouver leurs discussions après s'être déconnectés.

— **Évolutivité de l'application** : L'ajout de la synchronisation multi-serveurs est une première étape vers une architecture distribuée plus robuste. Cependant, une répartition plus fine des ressources et l'ajout de fonctionnalités de répartition de charge entre serveurs, ainsi que l'intégration de serveurs supplémentaires dans différentes régions géographiques, pourraient améliorer la scalabilité de l'application.
— **Optimisation des performances** : Enfin, pour améliorer les performances des transferts de fichiers, des protocoles de contrôle de flux pourraient être introduits afin d'optimiser l'utilisation de la bande passante et de réduire les temps de transfert, notamment lors des envois de fichiers volumineux.

En résumé, ce projet constitue une bonne base pour une application de messagerie instantanée robuste. Les améliorations suggérées permettraient d'accroître la sécurité, l'efficacité et l'expérience utilisateur de cette application.