ENSEIRB - MATMECA

Eirbone : une application d'échange de gros Fichiers en Pair à Pair (P2P)

ROSA Mathias
ELOUAZZANI Amira
PERRIN Lucie
BOUHAJA Mohammed
Hajji-Lammouri Marwa

14 Mai 2024

Table des matières

1 Introduction

Ce projet consiste à développer une application pour le partage de fichiers en pair à pair (P2P). Dans un réseau P2P, les ordinateurs sont connectés entre eux, chaque nœud agissant comme à la fois serveur et client. Les fournisseurs, également appelés "seeders", possèdent le fichier complet tandis que les consommateurs, ou "leechers", téléchargent des parties du fichier

L'objectif principal est de mettre en place un protocole de communication entre les pairs du réseau P2P. Les communications se feront via TCP avec des messages en texte, sauf pour les données binaires. Les données seront échangées sous forme de "pièces" de taille égale. Le développement implique donc la mise en place de mécanismes pour découper et échanger efficacement ces pièces entre les pairs.

2 Architecture fonctionnelle du projet

Le répertoire du projet contient des éléments divisé en plusieurs composants, y compris un dossier peer pour les fonctionnalités liées aux pairs avec ses binaires Java et sources, un dossier tracker pour le suivi avec des fichiers de configuration, des tests et des sources en C, ainsi que divers fichiers de documentation et de configuration comme Readme.md,

3 Peer

Dans un réseau pair à pair (P2P), un pair représente un élément central capable à la fois de fournir et d'accéder à des ressources, comme des fichiers partagés. Le code du pair facilite l'établissement de connexions avec d'autres pairs et un tracker central, ce qui permet un échange de données bidirectionnel. Parmi les fonctionnalités clés du pair, on retrouve la communication via des sockets TCP, l'échange de messages structurés avec les autres pairs et le tracker, ainsi que la gestion des connexions entrantes et sortantes. Cette représentation du pair dans un réseau P2P souligne son rôle crucial dans la distribution efficace des ressources entre les membres du réseau.

3.1 Arborescence

La figure (1) ci-dessous explique la structure de l'arborescence du répertoire `Peer`.

— `config.ini.example` : contient les configurations initiales de connection telles que *PieceSize* (taille des pièces de transfert de fichier).
— `Makefile` : compile et éxecute
— `Seed` : Les sous-répertoires de chaque pair sont organisés selon leur *<Id>*, et ils contiennent les fichiers de ce pair dans un sous-répertoire nommé *Peer<Id>*.
— `src` : contient les fichiers sources dans le package *lammm*.

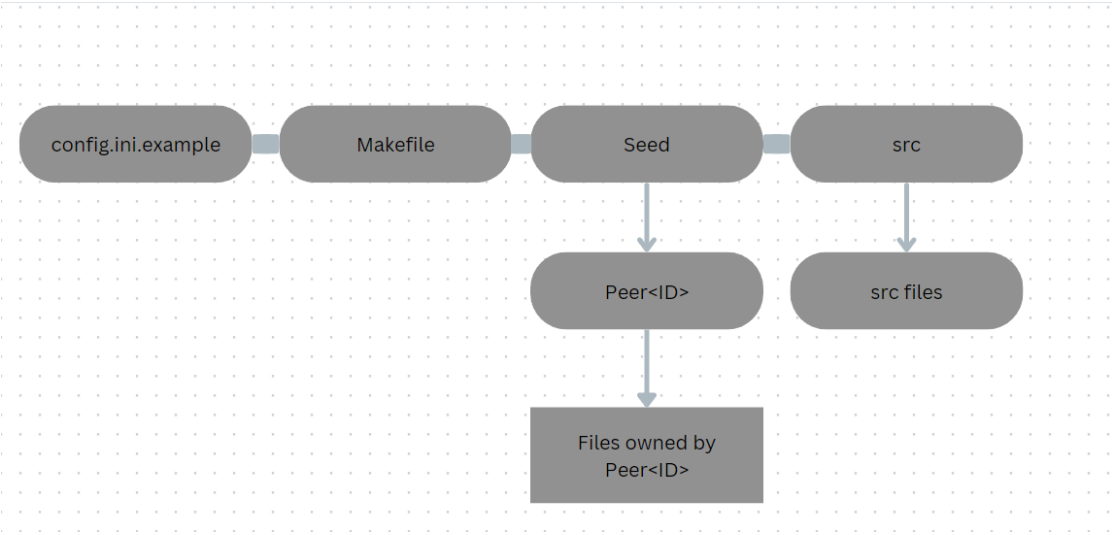


FIGURE 1 – Structure de l'arborescence du dossier Peer

3.2 Diagramme de classe

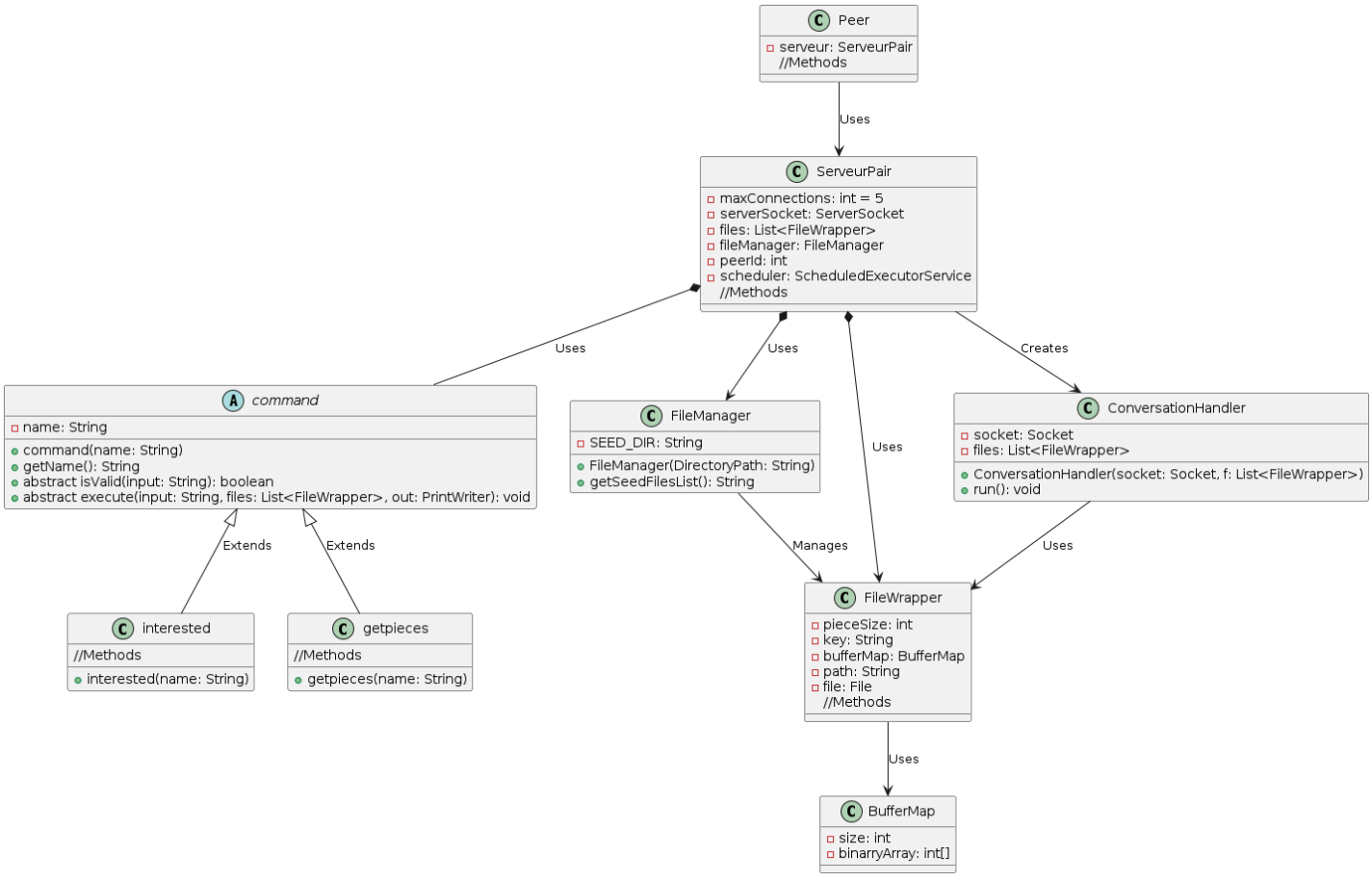


FIGURE 2 – Diagramme de classes des classes en relation avec le Pair

3.3 Connection Pair-Pair

Chaque pair connecté au Tracker agit simultanément en tant que serveur et client. Ainsi, la classe `Pair` possède une instance de la classe `ServeurPair` pour gérer ses fonctionnalités de serveur. Cette conception permet de séparer clairement l'implémentation des fonctionnalités liées au serveur de celles liées au client, favorisant ainsi la modularité du système. Lors du lancement du programme, chaque instance de `Pair` initialise son propre serveur dans un thread distinct, assurant ainsi que le programme ne soit pas bloqué et permettant au pair de continuer à communiquer avec le Tracker tout en écoutant de nouvelles connexions pair à pair. La commande *exit* permet de quitter la connection avec le pair et d'être redirigé vers le tracker en premier lui, puis de quitter le programme si la conversation est déjà avec le tracker.

3.4 Coté serveur

3.4.1 Boucle de connection et La classe ConversationHandler

Aprés avoir instancié le *ServeurPair*, celui-ci se met en écoute sur un port choisit dynamiquement par le système grâce à la méthode *listen*. Le paramètre 0 utilisé lors de l'instanciation d'un *ServerSocket* permet de choisir le port dynamiquement. Dans la boucle, le serveur commence par accepter la connexion grâce à la méthode `serverSocket.accept()` fournie par la classe `ServerSocket`. Cette méthode renvoie un objet `Socket` qui caractérise la connexion avec le client distant. Ce socket nous permet d'interagir spécifiquement avec ce client et d'échanger des messages. Ensuite, un objet `ConversationHandler` est créé en passant le `Socket` et la liste des fichiers connus par le serveur en tant que paramètres. La liste des fichiers connus est stockée dans une *ArrayList<>*. En offrant une structure de données facilement modifiable, l'*ArrayList<>* permet une gestion aisée des fichiers partagés dans un réseau dynamique où les ajouts et suppressions sont fréquents. Sa simplicité facilite également le développement en rendant la logique de stockage des fichiers plus intuitive et moins sujette aux erreurs.

Listing 1 – Attribut du ServeurPair : la liste de fichiers

```
private List<FileWrapper> files = new ArrayList<>();
```

Listing 2 – Création d'un ServeurSocket

```
public void listen() {
    try {
        serverSocket = new ServerSocket(0);
        int port = serverSocket.getLocalPort();
        System.out.println("Le pair est en  coute  sur le port :" + port + "\n");
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Listing 3 – Boucle d'acceptation des connexions avec les autres pairs

```
public void communicateWithPairs() {
    try {
        executor.submit(() -> {
            while (true) {
                try {
                    Socket socket = serverSocket.accept();
```

```
                    System.out.println("Le serveur a accept la connexion\n");

                    ConversationHandler clientsock = new ConversationHandler(socket,
                    new Thread(clientsock).start();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        });
    } finally {
        executor.shutdown();
    }
}
```

La classe `ConversationHandler` implémente l'interface `Runnable`, permettant ainsi de lancer chaque conversation avec une connexion entrante dans un thread séparé. Ainsi, le serveur peut gérer plusieurs connexions simultanément, adoptant ainsi une approche multithreadée.

```
public void run() {
    PrintWriter outPeer = null;
    BufferedReader inPeer = null;
    try {
        outPeer = new PrintWriter(socket.getOutputStream(), true);
        inPeer = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        String commandStr = "";
        while (true) {
            int maxAttempts = 3;
            int attempts = 0;
            boolean isValid = false;

            System.out.println("\n>");
            Command command = null;
            while (attempts < maxAttempts  !isValid) {
                if (inPeer.ready()) {
                    commandStr = inPeer.readLine();
                    System.out.println(commandStr);

                    if (commandStr.startsWith("interested")) {
                        command = new InterestedCommand(commandStr);
                        isValid = command.isValid(commandStr);
                    } else if (commandStr.startsWith("getpieces")) {
                        command = new GetPiecesCommand(commandStr);
                        isValid = command.isValid(commandStr);
                    } else if (commandStr.equals("exit")) {
                        break;
                    }
                    if (!isValid) {
                        attempts++;
                        outPeer.println("Commande invalide, essayez une nouvelle fois");
                    }
                }
            }
            if (isValid) {
                command.execute(commandStr, files, outPeer);
            } else {
                break;
            }
```

```
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                if (outPeer != null) {
                    outPeer.close();
                }
                if (inPeer != null) {
                    inPeer.close();
                    outPeer.println("Vous allez tre d connect ");
                    socket.close();
                    System.out.println("Peer disconnected successfully\n");
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
}
```

3.4.2 La classe abstraite *Command*

Afin de permettre plus de modularité, nous avons créer la classe abstraite `Command`. Ainsi, on peut modifier les réponses su serveur, sans trop toucher à son code propre :

```
public abstract class Command {
    private String name;

    public Command(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }

    public abstract boolean isValid(String input); // Checks if the input follows the right
    public abstract void execute(String input, List<FileWrapper> files, PrintWriter out); /
}
```

L'attribut `name` représente le nom de la commande. Par conséquent, la méthode `getName` est utilisée pour comparer la commande reçue par le serveur avec celle des classes concrètes qui héritent de `Command`. Grâce au polymorphisme, il est possible de convertir un objet `Command` en son type correspondant après la comparaison des chaînes, ce qui permet d'exécuter les bonnes réponses.

La méthode `isValid` permet de vérifier que le message reçue est bien formaté. Cela est nécessaire afin de pouvoir parser le message et en extraire les bons arguments. Afin de se faire, nous avons utiliser la bibliothèque `import java.util.regex.*;` qui permet de faciliter les méthode de reconnaissance des patterns. Comme par exemple dans le code de `getpieces` afin d'extraire les indices des pièces à télécharger :

```
public boolean isValid(String input) {
    String[] parsedInput = input.split("␣");
    int startIndex = input.indexOf("[");
    int endIndex = input.indexOf("]");
    if (parsedInput.length > 1) {
        if (parsedInput[0].equals("getpieces")  parsedInput[1].length() == 32  startIndex >
                endIndex > -1) {
            String pattern = "\\[\\d+(␣\\d+)*\\]";
            String list = input.substring(startIndex, endIndex + 1);
            Pattern regex = Pattern.compile(pattern);
            Matcher matcher = regex.matcher(list);
            return matcher.matches();
        }
    }
    return false;
}
```

L'expression régulière utilisée recherche des motifs dans une chaîne de caractères qui commencent par [, suivis d'une séquence d'un ou plusieurs chiffres, suivis par zéro ou plusieurs occurrences d'un espace suivi d'une séquence de chiffres, et se terminent par].

La méthode `execute` permet d'éxecuter la réponse du serveur.

3.5 Coté client

Après avoir établi la connexion avec le Tracker et obtenu la liste des ports auxquels se connecter pour télécharger le fichier désiré, le pair se connecte automatiquement à ces ports. Il lance ensuite la communication avec les autres pairs. Son message est envoyé simultanément à tous les pairs, et leurs réponses sont ensuite reçues et traitées.

3.6 Fonctionalités implémentées

Voici un exemple de déroulement de connection :

3.7 Détails de communication

3.7.1 Utilisation des sockets

Dans ce code, des objets Socket sont utilisés pour établir des connexions avec le tracker et d'autres pairs, tandis que des objets ServerSocket sont utilisés pour écouter les connexions entrantes sur le pair local.

Le pair demande la connection au pair ou tracker grâce au code suivant :

```
public Socket connect(int port) {
    Socket socket = null;
    try {
        socket = new Socket("localhost", port);
        System.out.println("Pair␣connect ␣au␣port␣:" + port + "\n");
    } catch (Exception e) {
        System.out.println("Le␣port␣n'est␣pas␣correct,␣essayez␣  ␣nouveau");
        // e.printStackTrace();
    }
```

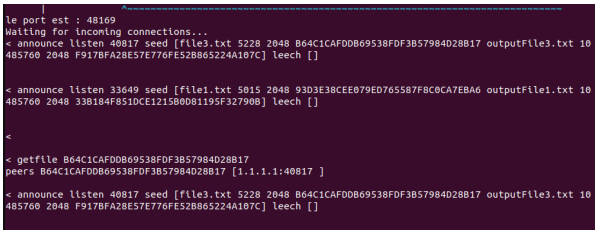


FIGURE 3 – Vision du Tracker

FIGURE 4 – Pair client

FIGURE 5 – Pair Serveur

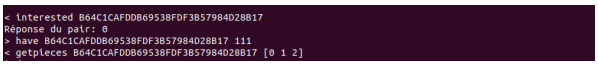


FIGURE 6 – Commandes interested et getpieces

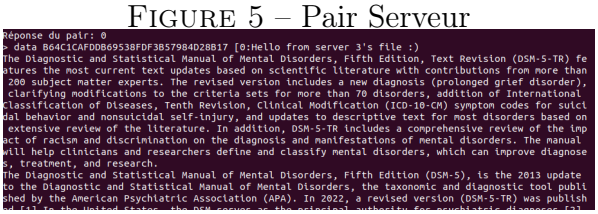


FIGURE 7 – Parties du fichier téléchargé

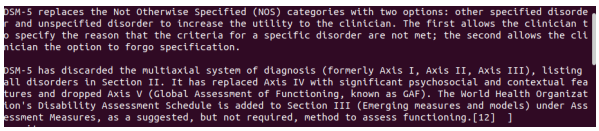


FIGURE 8 – Parties du fichier téléchargé

FIGURE 9 – Your overall caption for all the images

```
    return socket;
}
```

Puisque les pairs et le tracker se trouvent sur la même machine, on utilise l'option `"localhost"` pour l'adresse IP. Cela facilite les tests dans un premier temps. Ensuite, il est envisageable de le paramétrer dans le fichier de configuration initiale.

3.7.2 Entrée/Sortie Stream

Le code utilise les flux d'entrée et de sortie (InputStream et OutputStream) pour échanger des données avec le tracker et d'autres pairs via les sockets. Les messages envoyés sont généralement des chaînes de caractères qui suivent un protocole spécifique défini pour le système P2P. Exemple :

```
outPeer = new PrintWriter(socket.getOutputStream(), true);
inPeer = new BufferedReader(new InputStreamReader(socket.getInputStream()));
```

PrintWriter est une classe utilisée pour écrire des données formatées dans un flux de sortie. Elle propose diverses méthodes pour écrire différents types de données, tels que des chaînes de caractères, des entiers, des doubles, et bien plus encore. Dans ce contexte, elle est utilisée via `socket.getOutputStream()` pour transmettre les données vers le socket distant. L'option `true`

activée lors de la création du **PrintWriter** permet le flush automatique, assurant ainsi que les données sont immédiatement vidées vers le flux de sortie associé.

BufferedReader, quant à lui, est utilisé pour lire des données à partir d'un flux d'entrée, caractère par caractère ou ligne par ligne. Fréquemment employé pour la lecture de données depuis un fichier ou une connexion réseau, il propose des méthodes pour lire des caractères, des lignes de texte, voire des données formatées. Une fois de plus, il est instancié à partir du socket de communication.

3.8 Transfert de fichiers

Chaque pair est instancié ayant un certain nombre de fichiers en totalité. Il peut distribué ceux-ci aux autres pairs lorsque la connection est établie.

3.8.1 BufferMap

La classe `BufferMap` représente un tableau binaire de taille `file.length/file.pieceSize`, utilisé pour indiquer la disponibilité des pièces. Lorsqu'une pièce est disponible, elle est représentée par une valeur de 1 dans la `BufferMap`.

3.8.2 Gestion des fichiers

La classe `FileWrapper` conserve les métadonnées de chaque fichier ainsi que sa *BufferMap* correspondante et la taille de ses pièces. L'attribut `PieceSize` est paramétrable dans le fichier de configuration initiale et est directement lu à partir de celui-ci, puis assigné lors de l'instantiation d'un objet `FileWrapper`. Cette classe permet notamment de lire une pièce spécifique d'un fichier ou d'y écrire via un socket, correspondant ainsi partiellement au processus de téléchargement d'un fichier.

```
public void readPiece(PrintWriter out, int pieceIndex) {
    if (isAvailable(pieceIndex)) {
        try {
            // Nouvel objet fichier
            File file = new File(this.path);

            FileInputStream fileInputStream = new FileInputStream(file);
            BufferedInputStream bufferedInputStream = new BufferedInputStream(fileInputStrea

            byte[] buffer = new byte[this.pieceSize];
            int bytesRead;
            long bytesToSkip = (long) pieceIndex * this.pieceSize;

            // Passer les octets correspondant aux pi ces pr c dentes
            long skippedBytes = bufferedInputStream.skip(bytesToSkip);
            if (skippedBytes != bytesToSkip) {
                bufferedInputStream.close();
                throw new IOException("Impossible de passer  la pi ce " + pieceIndex);
            }

            // Lire les octets de la pi ce actuelle
            bytesRead = bufferedInputStream.read(buffer, 0, this.pieceSize);
            if (bytesRead != -1) {
                // Convertir les octets lus en cha ne et l'envoyer au client
```

```
                String data = new String(buffer, 0, bytesRead);
                // Encoder en base64
                String encodedString = Base64.getEncoder().encodeToString(data.getBytes());

                System.out.println(data);
                out.print(encodedString);
            }

            System.out.println("T l chargement termin !\n");

            // Fermer le flux
            bufferedInputStream.close();

        } catch (IOException e) {
            e.printStackTrace();
        }
    } else {
        out.println("Pi ce non disponible");
    }
}
```

Les classes `import java.nio.file.Files;` et `import java.nio.file.Paths;` de Java sont particulièrement utilisées pour interagir avec le système de fichiers Linux. Elles permettent de trouver la taille d'un fichier, son nom, ainsi que de lire ou modifier son contenu.

Parmi les attributs de la classe `FileWrapper`, se trouve *key*, qui stocke le hachage du fichier. Ce dernier est automatiquement généré lors de l'instanciation grâce à la méthode `generateKey()`, exploitant l'algorithme de hachage `MD5`. Il est à noter qu'aucune méthode n'est prévue pour modifier directement le hachage du fichier en le passant en paramètre, afin de préserver la sécurité du contenu du fichier.

```
private String generateKey() {
    String key = "";
    try {
        MessageDigest md = MessageDigest.getInstance("MD5");
        md.update(Files.readAllBytes(Paths.get(this.path)));
        byte[] digest = md.digest();
        // Convertir les octets en cha ne hexad cimale
        StringBuilder sb = new StringBuilder();
        for (byte b : digest) {
            sb.append(String.format("%02X", b));
        }
        key = sb.toString();
    } catch (NoSuchAlgorithmException | IOException e) {
        e.printStackTrace();
    }
    return key;
}
```

La classe `FileManager` permet de lire le contenu d'un répertoire et de sauvegarder les informations sur les fichiers qu'il contient. Chose qui va servir pour créer le message d'annonce au Tracker. De plus, le message d'annonce au Tracker est mis à jour périodiquement en utilisant l'objet *ScheduledExecutorService*.

```
import java.io.*;
```

```java
import java.nio.file.*;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class FileManager {
    // private static final String SEED_DIR = "./seed";
    private String SEED_DIR; // d pend de l'identifiant du pair

    public FileManager(String directoryPath) {
        this.SEED_DIR = directoryPath;
    }

    public String getSeedFilesList() throws IOException {
        try (Stream<Path> paths = Files.walk(Paths.get(this.SEED_DIR))) {
            return paths
                    .filter(Files::isRegularFile)
                    .map(path -> new FileWrapper(path.getFileName().toString(), path.toStri
                    .collect(Collectors.joining("␣", "[", "]"));
        }
    }
}
```

3.8.3 Transfert des données

Une fois que l'utilisateur a spécifié les fichiers à télécharger ainsi que la clé de fichier (*filekey*) à l'aide de la commande `getpieces`, le processus de téléchargement commence. Tout d'abord, le fichier est lu à l'aide de l'objet `FileInputStream`, puis il est transféré à travers les sockets de communication entre les pairs. Ensuite, le fichier est écrit (et créé s'il n'existe pas déjà) à l'aide de la méthode `writePiece`, qui utilise *file.write*.

Cependant, comme l'objet `BufferedReader` lit ligne par ligne et ne supporte que les données textuelles, la lecture du fichier s'arrête après chaque retour à la ligne. Pour garantir l'envoi de la totalité du contenu du fichier, il a été décidé d'encoder les données en Base64 avant l'envoi. Cela permet de représenter les données binaires sous forme de texte ASCII, assurant ainsi une transmission complète et sans altération.

Le retour du serveur est dans le format suivant : > *data $Key [$Index1 :$Piece1 $Index2 :$Piece2 $Index3 :$Piece3 ...]* Pour faciliter l'analyse du message, chaque pièce du fichier envoyé a été isolée entre %Piece0%. Ensuite, la reconnaissance de motifs a été utilisée pour lire chaque pièce du fichier et reconstituer le produit final.

```java
Pattern pattern = Pattern.compile("%(.*?)%");
Matcher matcher = pattern.matcher(responseP);

while(matcher.find()) {
    // code
}
```

matcher.find() recherche toutes les occurrences des chaînes correspondant au motif spécifié, dans la réponse du serveur, puis écrit chaque fois la pièce correspondante dans le fichier.

Notons que les méthodes de lecture et d'écriture sur les fichiers ont été conçues pour permettre la lecture et l'écriture à partir d'une partie spécifique du fichier en octets.

3.9 Eléments d'organisation

3.9.1 Gestion des erreurs

La gestion des exceptions est cruciale dans tout code réseau. Le code utilise des blocs try-catch pour intercepter et gérer les exceptions qui pourraient survenir lors de l'établissement de connexions, de l'envoi ou de la réception de messages. De plus, pour chaque exception interceptée, on print la trace de la pile.

```
try {
    //Some code
} catch (exception e) {
    e.printStackStrace();
}
```

Cela a permis de remonter à la source de chaque bogue et ainsi de faciliter le processus de débogage.

3.9.2 Interactions utilisateur

Le code permet à l'utilisateur d'entrer des commandes via la console, telles que le port du tracker, ou des commandes spécifiques comme "connect" pour établir une connexion avec un autre pair. Les saisies utilisateur sont lues à l'aide de l'objet BufferedReader. De plus, l'utilisateur peut utiliser le fichier `config.ini.example` pour modifier la valeur de certaines configurations telles que *PieceSize*.

3.9.3 Makefile

Pour compiler le code Java, nous avons mis en place une infrastructure maven afin de transformer notre code en jar et aussi d'installer des dépendances. Malheureusement nous n'avons pas réussi a installer de dépendances mais nous arrivons bien à utiliser maven pour compiler automatiquement les Peers. Le code source est donc maintenant placé dans le dossier 'src/main/-java/lammm' et est compilé avec la commande 'mvn package'

Nous avons quand même un Makefile présent à la racine du fichier qui possèdes des règles afin de compiler et lancer les pairs sans avoir à se rappeler de ces commandes.

Pour compiler le peer, la règle 'make package' peut être utilisée. Pour l'exécuter il suffit d'exécuter la commande 'make run'. Par défaut, la commande 'make' compile et exécute le programme.

On nous demandera alors de spécifier le port du tracker.

4 Tracker

Le tracker est un serveur qui permet de lier plusieurs pair entre eux afin de partager des fichiers. Mais avant d'établir la connexion avec le tracker, un port libre est automatiquement recherché, bien que les pairs puissent également spécifier un port manuellement pour la connexion. Une fois cette étape franchie, la connexion est établie à l'aide de sockets. Pour chaque nouvelle connexion, le serveur initie un thread spécifique afin de gérer cette connexion de manière isolée, en exécutant la fonction connectionHandler() au sein de ce thread dédié.

4.0.1 Communication Pair-Tracker

Pour établir la communication entre les pairs et le tracker, l'utilisation de sockets était indispensable. Cette communication repose sur la structure struct sockaddr_in. De plus, nous avons introduit la structure de données peer_info qui contient deux attributs, sock et ip, pour stocker respectivement les adresses IP et les sockets des pairs.

Listing 4 – Définition de la structure peer_info

```
typedef struct {
    int sock;
    char ip[INET_ADDRSTRLEN];
} peer_info;
```

Afin de gérer la communication entre le tracker et les pairs, nous utilisons les fonctions implémentées dans le fichier connection_handler.c. Ces fonctions permettent de traiter les messages du tracker et de construire des réponses adaptées aux requêtes, telles que les demandes de téléchargement de fichiers.

Pour gérer les informations sur les fichiers partagés, nous avons créé une structure de données AnnouncedFile. Cette structure stocke les détails des fichiers dans une liste globale, ce qui facilite l'accès et la gestion des fichiers annoncés par les pairs.

Listing 5 – Définition de la structure AnnouncedFile

```
typedef struct {
    char filename[256];
    unsigned int length;
    unsigned int pieceSize;
    char key[256];
    unsigned int port;
    char ip[256];
} AnnouncedFile;


AnnouncedFile announced_files[MAX_FILES];
unsigned int num_announced_files = 0;
```

Lors de la réception de chaînes de caractères du tracker, il est nécessaire de les convertir en informations exploitables pour le stockage des fichiers. Pour cela, la structure WordList a été introduite. Elle permet de stocker les informations nécessaires extraites des messages. Grâce aux fonctions implémentées dans le fichier file.c, nous pouvons extraire ces informations et peupler la liste des fichiers annoncés.

Listing 6 – Définition de la structure WordList

```
typedef struct {
    char words[MAX_WORDS][MAX_WORD_LENGTH];
    int count;
} WordList;
```

4.0.2 Fonctionnalités implémentées

Nnous avons pu implémenter et tester les fonctionnalités suivantes :

1. La commande **"announce"** qui permet de communiquer le numéro du pair qui se connecte, sa clé ainsi que les fichiers qu'il contient.

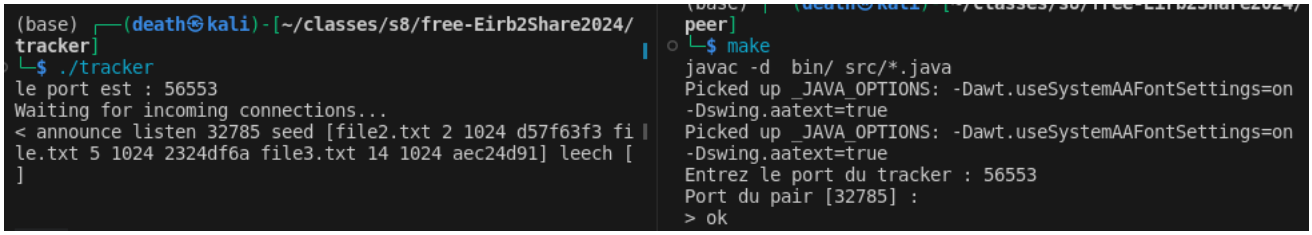


FIGURE 10 – Illustration de la commande **"announce"**.

2. La commande **"look"** qui permet à un pair de connaître les fichiers présents sur le réseau.

FIGURE 11 – Illustration de la commande **"look"**.

3. La commande **"getfile"** permet de renvoyer à un pair la liste des pairs détenteurs d'un certain fichier, pour lequel il est interessé.

FIGURE 12 – Illustration de la commande **"getfile"**.

Cette dernière commande nous permet également de faire la communication pair à pair. Ayant ces informations, un pair pourra lancer des téléchargements de fichiers.

5 Faussaires

5.0.1 Faux Tracker

On a mit en place un "faux tracker" dans un réseau pair à pair (P2P), permettant d'accepter les connexions entrantes des pairs. Il crée un socket TCP IPv4, l'associe à un port spécifique et écoute les connexions entrantes. Lorsqu'une connexion est détectée, il crée un nouveau thread

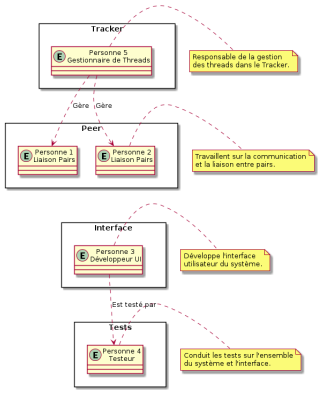
dédié à cette connexion, permettant ainsi de gérer plusieurs connexions simultanément sans bloquer le processus principal. Le code gère également les erreurs potentielles lors de la création du socket ou de l'acceptation de connexion, et démarre le faux tracker en appelant la fonction startFakeTracker() dans la fonction principale main().

5.0.2 Faux Peer et File Manager

Le faux peer (FauxPeer1) et le faux gestionnaire de fichiers (FauxFileManager) sont des composants simulés dans un système de partage de fichiers pair à pair (P2P). Le faux peer interagit avec l'utilisateur en lisant les commandes entrées via la console et en simulant le traitement de ces commandes à l'aide de la méthode processCommand(). Il peut ainsi afficher des messages pour simuler des actions telles que le téléchargement de fichiers. D'autre part, le faux gestionnaire de fichiers fournit une liste de fichiers partagés simulés à travers la méthode getSeedFilesList(). Cette liste est générée à partir d'une collection de noms de fichiers préétablis et est utilisée pour simuler la disponibilité des fichiers à télécharger dans le réseau P2P. Ces composants factices sont utilisés à des fins de démonstration et de test, permettant de simuler le comportement du système sans avoir à implémenter toutes ses fonctionnalités réelles.

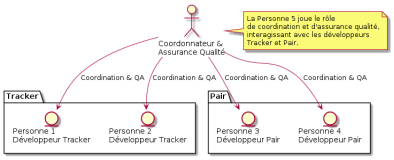
6 Gestion de projet

Pour réussir ce projet, notre équipe a adopté une organisation structurée en deux phases principales. Initialement, comme le montre la figure suivante, nous nous sommes divisés en groupes spécialisés sur le tracker et les pairs, avec une cinquième personne assurant la coordination et l'assurance qualité. Cette étape a été cruciale pour poser les bases du projet et assurer une progression harmonieuse.



Les réunions régulières nous ont permis d'échanger sur les avancées, de résoudre ensemble les difficultés, et d'éviter les conflits sur Git, en veillant notamment à ne pas intégrer de fonctionnalités inachevées sur la branche principale. La communication continue a été essentielle pour clarifier les doutes et solliciter de l'aide, garantissant ainsi une collaboration efficace et adaptée aux défis rencontrés.

Face à l'évolution du projet, nous avons réajusté notre stratégie pour mieux répondre aux besoins émergeant, comme illustré dans la figure suivante. Cette nouvelle organisation a redistribué les rôles au sein de l'équipe, allouant des ressources au renforcement des liaisons entre pairs, au développement de l'interface utilisateur, aux tests et à la gestion optimisée des threads dans le tracker.



Nous avons conservé cette dernière stratégie jusqu'à la fin du projet.

7 Utilisations des LLM

Nous avons utilisé des LLM tels que ChatGPT pour nous aider à développer le projet. En effet, il peut être utile pour comprendre certaines erreurs que l'on peut avoir en cours de développement en complément à des sites comme "stack overflow". En particulier, ChatGPT nous a aidé à trouver comment parser les messages parmis d'autres. Les propositions faites par ces modèles de langages n'étant pas totalement fiable, nous prenons à chaque fois du recul quant aux solutions apportées. Un usage de GitHub Copilot a également pu être fait pendant ce projet, mais plus dans une démarche d'effectuer des complétions de lignes en cours d'écriture que d'écrire une fonction complète.

8 Pistes de développements

Introduire de la modularité dans le traitement des réponses côté client serait une excellente idée pour améliorer la gestion des communications pair-à-pair. Tout comme nous avons utilisé une classe abstraite `Command` pour le serveurPair, nous pourrions envisager d'implémenter une structure similaire du côté client. Cela permettrait de mieux organiser le code, de le rendre plus lisible et plus facile à maintenir. Par exemple, chaque type de réponse du serveur pourrait être associé à une classe spécifique qui encapsule la logique de traitement associée.

En ce qui concerne les threadPools, bien qu'ils n'aient pas été complètement implémentés faute de temps, les intégrer dans le projet pourrait considérablement améliorer ses performances. Les threadPools permettent de gérer efficacement l'exécution simultanée de plusieurs tâches, ce qui est particulièrement utile dans un environnement pair-à-pair où de nombreuses opérations peuvent être exécutées en parallèle. Ils contribueraient à optimiser l'utilisation des ressources système et à réduire les temps d'attente, améliorant ainsi globalement la réactivité du système.

Ayant bien utilisé des threads dans certaines parties du projet, l'introduction de mécanismes de synchronisation aurait été bénéfique, en particulier pour le transfert de fichiers, surtout lorsqu'il s'agit de transferts simultanés impliquant plusieurs serveurs. Cela aurait permis de garantir la cohérence des opérations et d'éviter les conflits dans les situations où plusieurs threads tentent d'accéder ou de modifier les mêmes données de manière concurrente.

En outre, pour renforcer la robustesse du système, il pourrait être intéressant d'implémenter des mécanismes de reprise sur erreur, tels que la retransmission des paquets perdus ou la gestion des pannes de connexion. Cela garantirait une meilleure fiabilité des communications et une expérience utilisateur plus fluide, même dans des conditions réseau moins que parfaites.

Nous aurions aussi aimé créer une interface en utilisant un serveur websocket dans les peers qui aurait envoyé les données à un frontend en JavaScript permettant de mieux visualiser les transferts mais nous avons eu des soucis lors de l'installation des dépendances Java et n'avons pas pu l'implémenter.

De plus, une meilleure gestion des fichiers auraient été appréciée.

En résumé, en introduisant davantage de modularité dans le traitement des réponses, en intégrant les threadPools pour une gestion efficace des tâches concurrentes, et en renforçant la robustesse du système avec des mécanismes de reprise sur erreur, le projet de communication pair-à-pair pourrait être considérablement amélioré en termes de performances, de fiabilité et de maintenabilité.

9 Conclusion

Au cours de ce projet, nous avons eu l'occasion d'appliquer et de consolider les concepts appris au fil des semestres en réseaux. Cela nous a permis de mieux comprendre les mécanismes sous-jacents des protocoles de communication et des architectures réseau. De plus, en utilisant des langages de programmation tels que Java et C dans le cadre d'un projet d'envergure, nous avons pu approfondir nos compétences en développement logiciel et acquérir une expérience pratique précieuse dans la conception et la mise en œuvre de systèmes réseau complexes. Ce projet nous a également donné l'opportunité de collaborer efficacement en équipe, de résoudre des problèmes techniques et de relever les défis rencontrés lors du développement et du déploiement d'une application réseau fonctionnelle. En résumé, cette expérience a été enrichissante à la fois sur le plan académique et professionnel, nous préparant ainsi à relever de nouveaux défis dans le domaine des réseaux et de la programmation.