BORDEAUX INP Enseirb-Matmeca

Rapport Projet de Réseau

Abdurahman EL CALIFA KANIT BENSAIDI, Mathys MONELLO, Joachim ROBERT, Joris ROUSERE, Yannis YOASSI PATIPE

Mai 2024

Eirbone : une application d'échange de fichiers via du *peer-to-peer* centralisé (P2P)

EI8RE203 Projet de Réseau
Département Informatique
S8 - Année 2023/2024

Table des matières

1 Introduction

Le partage de fichier grâce au réseau internet est possible de plusieurs manières. L'une d'elle est le partage de fichers en *peer-to-peer*. Un réseau *peer-to-peer* est un réseau d'ordinateurs connectés entre eux où chaque ordinateur peut se comporter comme un serveur mais aussi comme un client.

On distingue deux types de client : les *seeders*, ceux qui détiennent l'entièreté d'un fichier et qui le partagent, les *leechers* sont ceux qui téléchargent un fichier et qui peuvent partager la ou les parties d'un fichier qu'ils sont en train de télécharger.

Ce projet porte sur le développement d'une application d'échanges de fichiers qui utilise ce modèle. Une particularité de notre réseau est son aspect centralisé. Une entité, le *tracker*, sera là pour donner des informations dont chaque client, qu'il soit *leecher* ou *seeder*, aura besoin pour se connecter et pour partager/télécharger un fichier.

Le développement de ce projet s'est naturellemment découpé en plusieurs parties distinctes dont nous aurons l'occasion de parler plus en détails : la création d'un tracker, le développement du côté client, pouvant communiquer avec le serveur mais aussi avec les autres clients et enfin, la mise en liaison entre les différentes composantes décrites précédemment pour enfin permettre le fonctionnement de l'application. Pour ce faire, nous avons mis en place une méthode de gestion de projet, et avons aussi pu utiliser des LLM pour nous aider dans notre développement.

2 Le tracker

Le tracker est un élément essentiel de l'application. Servant de base de données pour les clients, il est mis à disposition de ceux-ci afin que tous puissent obtenir les renseignements nécessaires au téléchargement d'un fichier ainsi qu'aux conditions de connexion au reste du réseau. En effet, c'est par le tracker qu'un *leecher* connaît les clients possédant le fichier qu'il souhaite télécharger ainsi que les numéros de port associés à chacun afin d'établir une connexion et commencer à télécharger les différentes parties du fichier réparties sur le réseau.

Deux aspects différents concernant le tracker sont à comprendre avant de se lancer dans sa réalisation. D'un côté, il y a toutes les intéractions avec les clients, gérer les connexions entrantes ainsi que les différentes requêtes pouvant être émises par un client. De l'autre côté, il y a le stockage des informations, que ce soit quels clients sont connectés au réseau ou quels clients possèdent une ou plusieurs parties d'un fichier ainsi que la manière d'accéder à toutes ces informations. Nous allons voir les choix faits concernant ces caractéristiques et la manière dont l'ensemble a été implémenté.

2.1 Le tracker en tant que server

Le tracker se comporte comme un serveur, il ouvre une connexion sur un port connu par tous les clients et se met en attente de message provenant de ces derniers. Les différentes librairies *socket.h*, *in.h* et *inet.h* ont été nécessaires pour cet aspect du tracker. Une fois l'initialisation et la mise en place de l'écoute du port avec *select*, on rentre dans une boucle infinie où l'on surveille le *file descriptor* du *socket* en écoute afin d'accepter les connexions des clients. Ils sont ajoutés au fur et à mesure dans la base de données du tracker afin qu'il puisse accepter et traiter leurs messsages.

Une fois un message réceptionné, le tracker récupère tout d'abord le *file descriptor* associé à la connexion du client, son adresse IP, son port de connexion ainsi que le message qui accompagne toutes ces informations. Nous verrons par la suite comment celles-ci sont conservées.

Une fois la requête interprétée, le tracker repasse dans son mode d'attente de requête via la boucle *while(1)* et la fonction *select()*.

2.2 Le traitement des messages

Comme cela a été présenté avant, le client envoie des requêtes au tracker afin d'avoir toutes les informations nécessaires au bon téléchargement du fichier qu'il souhaite. Pour interpréter les requêtes reçues, il a fallu développer un *parser*, c'est à dire une fonctionnalité capable d'interpréter les requêtes et agir en conséquence. Il est implémenté via la fonction `handle_message` du fichier *message_handler*. Les différentes requêtes imposées au minimum dans le sujet du projet sont les suivantes :

- *announce* : 1ère requête que chaque client du réseau doit envoyer. De cette manière il est reconnu par le tracker et introduit les fichiers qu'il possède.
- *look* : Permet à un client de connaître l'état du réseau en récupérant la liste des fichiers vérifiant un certain nombre de paramètre que le client aura précisé en complément de sa requête.
- *getfile* : Lorsqu'un client veut télécharger un fichier, il récupère via ce message, l'ensemble des *seeders* possédant une partie ou la totalité du fichier demandé ainsi que les informations nécessaires pour se connecter à ces différentes *seeders*.
- *update* : Périodiquement, chaque client doit mettre à jour ses fichiers qu'il possède ou ceux en cours de téléchargement.

Lorsqu'un client annonce un fichier à partager, les détails du fichier sont extraits de l'annonce, tels que le nom, la taille, la taille de la pièce et la clé d'identification. En parallèle, sont enregistrées les informations du client, comme son adresse IP et son port. Tout cela doit être conservé de manière à être repartagé selon les requêtes reçues. Une architecture en **liste simplement chaînée** s'est vue être le meilleur choix pour le stockage des différentes informations. Dans un projet aussi complet, il fallait trouver un compromis entre une réalisation rapide pour ce côté du tracker tout en ayant une complexité algorithmique raisonnable quant à l'accès à certaines données, d'où ce choix de listes chaînées.

Ces listes chaînées ont été alors utilisées pour stocker les fichiers et les clients. Ces deux entités avaient une structure qui leur était propre : *client_t* et *file_t*. La question ressortant alors était quelle structure devait contenir l'autre. En effet peu importe le point de vue, ces deux entités sont dépendantes l'une de l'autre. Et donc soit chaque client était relié à la liste des fichiers qu'il possédait ou bien chaque fichier était lié aux différents clients possédant une partie ou l'entièreté de ce dernier. L'équipe du projet a penché pour la deuxième option. Cela paraissait plus pratique d'opérer de cette manière, étant donné que la plupart des requêtes annoncées par les clients concernent des données liées aux fichiers et non directement aux autres clients du réseau. C'est pourquoi on trouve dans la structure *file_t* un pointeur vers une liste de *client_t*.

Afin de ne pas passer un temps considérable dessus et d'implémenter quelque chose de non-abouti, il a été décidé pour les listes chaînées d'utiliser des structures déjà construites. Cette structure

provient des `Queue BSD`. Bien que l'implémentation puisse être difficile à apréhender aux premiers abords, la construction des listes chaînées est complète et il y a le nécessaire pour implémenter correctement l'architecture de stockage du tracker.

2.3 Architecture globale du tracker

D'après ce qui vient d'être présenté, l'ensemble des missions à accomplir par le tracker est riche et complexe, il a donc fallu décider d'une certaine architecture pour la programmation de ce service afin de rendre sa maintenance et sa modifition aisées à gérer. Son architecture se présente donc comme suit :

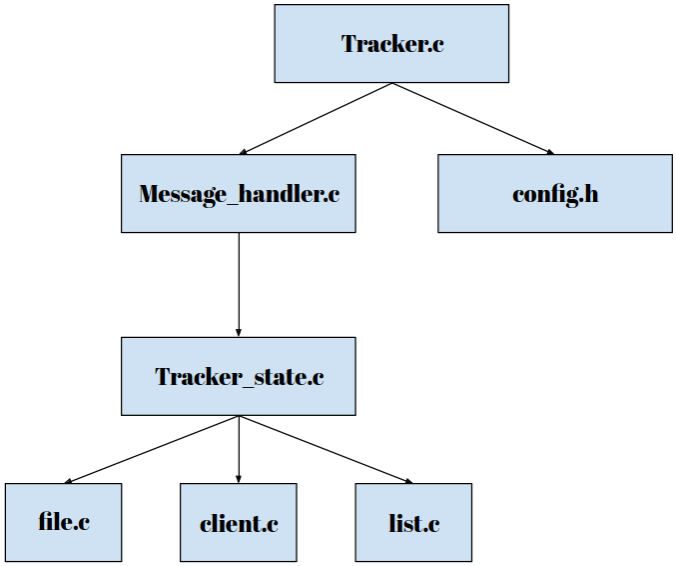


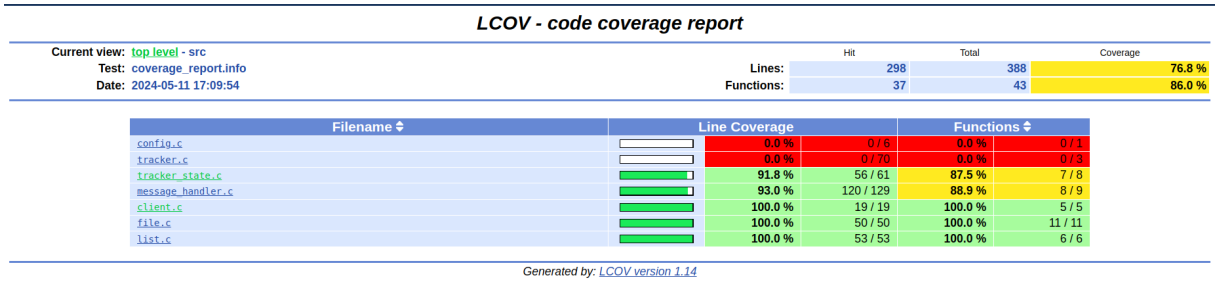
FIGURE 1 – Diagramme de dépendance du tracker.

— C'est dans le fichier `tracker.c` que les listes contenant les fichiers et les clients sont initialisées. Le serveur communiquant avec les différents clients est également créé et géré dans ce script.
— Dès qu'un message est reçu, il est géré par les fonctions dans `message_handler.[c-h]`. A chaque requête correspond une fonction dans le fichier *message_ handler* permettant son *parsing* ainsi que le traitement qui lui est associé. Cette interface permet entre autres la construction des messages de réponse et l'appel au fonctions de mise à jour de l'état du *tracker*.
— Pour mettre à jour les listes de fichiers et de clients, ce sont les scripts `tracker_state.[c-h]` qui sont utilisés pour cela. Ici une certaine "sécurité" a été mise en place afin que l'état du *tracker* reste fiable malgré des potentiels appels de fonction incomplets ou incohérents. Par exemple, il fallait penser à ne pas dupliquer un fichier annoncé par deux *peers* ou encore être capable d'identifier un client malgré une adresse et un port identique. Ainsi nous avons une interface indépendante offrant un accès sécurisé à l'état du tracker. De plus, le fait d'avoir centralisé les modifications du *tracker* dans ce fichier nous aurait permis dans le futur du multithreading assez simplifié.

Il a paru important que ça ne soit pas au tracker de gérer l'interprétation directe des messages reçus ainsi que les modifications des informations stockées. Le tracker était vu comme un simple serveur utilisant des traitements extérieurs pour mettre à jour le réseau symbolisé par les deux listes chaînées présentées précédemment.

2.4 Les tests

Pour les petits fichiers, c'est à dire, *file.c client.c* et *list.c* des tests unitaires ont été mis en place très tôt afin d'avoir des bases solides. Ensuite, un seul test a été utilisé pour le reste du code, il consiste en une mise en situation d'utilisation du tracker, on obtient sur l'ensemble des tests le *coverage* suivant :



LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level - src	Lines:	298	388	76.8 %
Test:	coverage_report.info	Functions:	37	43	86.0 %
Date:	2024-05-11 17:09:54				

Filename	Line Coverage			Functions	
config.c		0.0 %	0 / 6	0.0 %	0 / 1
tracker.c		0.0 %	0 / 70	0.0 %	0 / 3
tracker_state.c		91.8 %	56 / 61	87.5 %	7 / 8
message_handler.c		93.0 %	120 / 129	88.9 %	8 / 9
client.c		100.0 %	19 / 19	100.0 %	5 / 5
file.c		100.0 %	50 / 50	100.0 %	11 / 11
list.c		100.0 %	53 / 53	100.0 %	6 / 6

Generated by: LCOV version 1.14

FIGURE 2 – Couverture des tests du tracker et de sa structure.

Pour *tracker.c* les tests on été réalisés avec les clients. De plus, il n'a pas paru pertinent d'en faire pour *config.h*. D'où les pourçentages de couverture nuls pour ces deux fichiers.

3 Le client

Dans le modèle de partage de fichier en *peer-to-peer*, le client, pair ou *peer* se décompose en deux composantes. La première est celle qui communique avec le tracker, et la seconde est celle qui se comporte comme un serveur avec les autres pairs qui font partie du réseau.

3.1 Architecture du pair

Dans cette section, nous allons présenter le diagramme de classe du client simplifié, c'est-à-dire ne comportant ni les méthodes, ni les attributs des classes.

La création du pair nécessite l'implémentation de plusieurs fonctionnalités, ainsi nous avons décidé de découper le pair en plusieurs composantes qui sont visibles sur la figure 3. Le développement du pair est réalisé en `JAVA`, ce qui permet d'utiliser des propriétés du langage menant vers une *Programmation Orientée Objet* du pair comme le montreront certains liens qui seront explicités par la suite.

3.2 Un pair qui communique avec le tracker

Cette partie concerne la manière dont le pair envoie des commandes au tracker et comment il gère les fichiers pour envoyer les informations que le tracker désire.

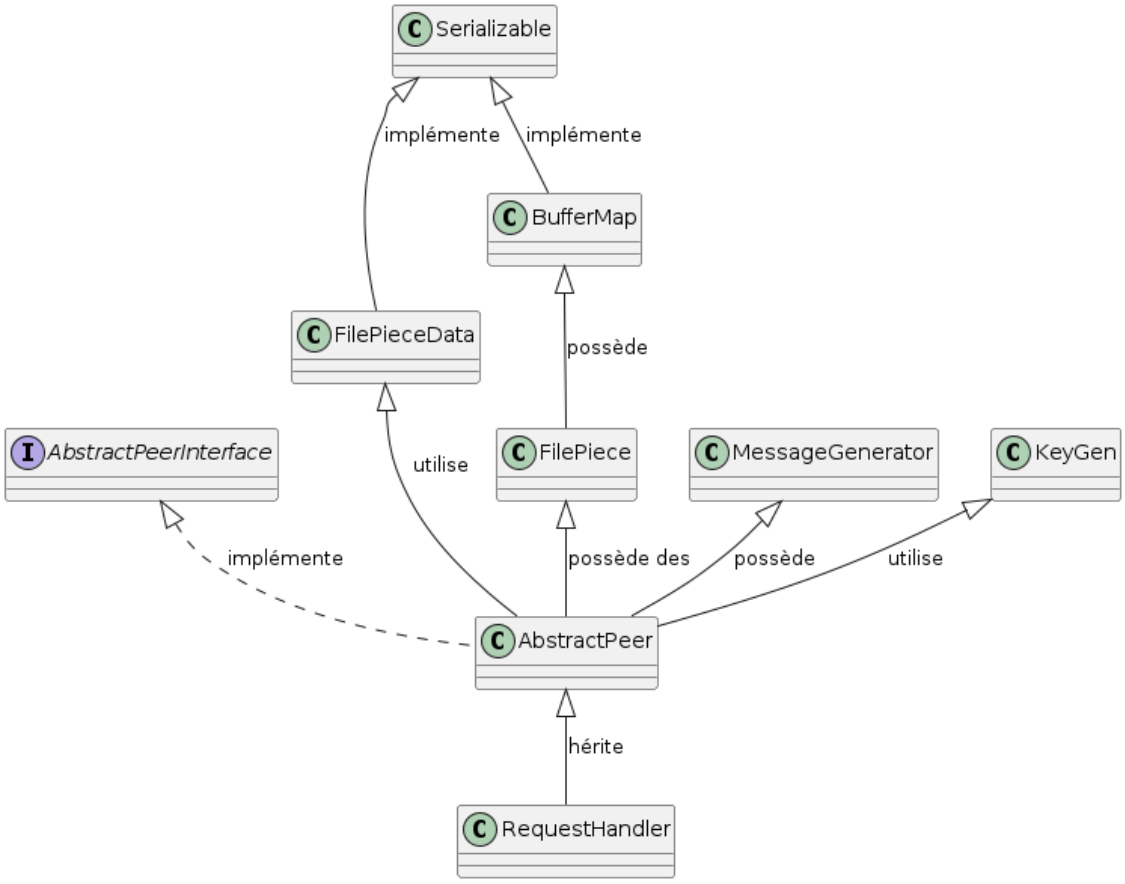


FIGURE 3 – Diagramme de classe du peer

3.2.1 La communication

La première étape pour permettre au pair de communiquer avec le tracker est de transmettre les informations liées au tracker à tout pair afin qu'il soit capable de s'y connecter. Pour ce faire, nous avons utilisé la classe **Scanner** qui permet de récupérer les informations de connexion du tracker. Nous avons choisi d'utiliser cette classe puisqu'elle permet de faire du *parsing* dans un fichier ou même dans le terminal. Cette classe est d'ailleurs utilisée également pour récupérer les commandes écrites par l'utilisateur dans le terminal. Elle implémente déjà les différentes méthodes nécessaires à l'interprétation des entrées et à leurs traitements.

Ensuite, pour communiquer avec le tracker, nous avons dû nous familiariser avec la classe **Socket** qui permet la communication entre deux machines. Pour envoyer et recevoir les messages venant du tracker, nous avons également dû utiliser deux autres classes qui sont **BufferedReader** et **PrintWriter**. Ces deux classes servent respectivement à lire et écrire dans la **Socket** reliée au tracker. Cette instance de la classe **Socket** permet d'initier une connexion `TCP` entre le pair et le tracker. Une fois

tout cela réalisé, nous avons commencé à nous intéresser à la partie qui gère les fichiers.

3.2.2 La génération des informations relatives aux fichiers

Pour la gestion des fichiers par l'utilisateur, nous avons décidé de créer un répertoire où les fichiers qui seront contenus dedans seront partagés sur le réseau. Ainsi, lors de l'instanciation du pair, nous récupérons automatiquement les informations des fichiers qui s'y trouvent et dont nous avons besoin pour remplir les différents champs de la classe **FilePiece** qui est présentée sur la figure 4. Cela est possible grâce aux classes **System** et **File**, la première permet de chercher dans le répertoire de l'exécutable, le répertoire contenant ces fichiers, et la seconde permet de parcourir le répertoire pour récupérer les informations et générer celles qui doivent l'être.

FIGURE 4 – Classe FilePiece

Dans les attributs de la classe **FilePiece**, nous pouvons voir que nous utilisons un lien ***A-un*** avec la classe **BufferMap**. La classe **BufferMap** est illustrée par la figure 5. Cette classe permet de conserver les informations concernant les parties d'un fichier que possède un pair ainsi que sa clé. Cette clé qui est la même entre les deux classes est générée grâce à la classe **KeyGen**. La classe **KeyGen** ne possède qu'une seule méthode qui est définie comme *static*. Nous pouvons donc l'utiliser sans créer

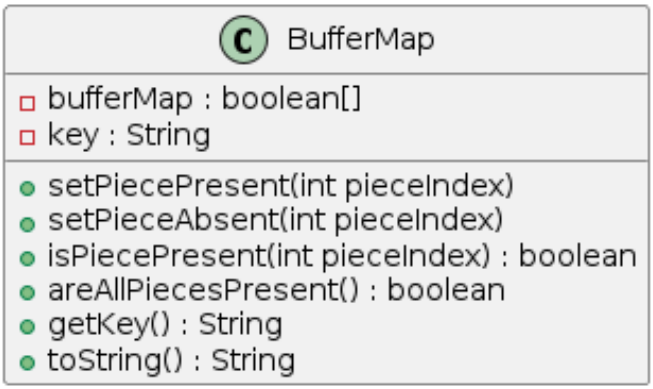


FIGURE 5 – Classe BufferMap

d'instance, car un objet n'est pas nécessaire pour y accéder. Il est alors possible de calculer la clé *MD5* générée à partir du contenu du fichier et non pas de son nom, permettant ainsi de différencier deux fichiers portant le même nom.

Toutes ces fonctionnalités permettent ainsi de créer la classe **AbstractPeer**.

3.2.3 AbstractPeer

On obtient enfin la classe **AbstractPeer** qui est présentée plus explicitement sur la figure 6. Comme montré par la figure 3, la classe **AbstractPeer** hérite de l'interface **AbstractPeerInterface**. Cette interface, qui est illustrée sur la figure 7, nous permet de spécifier les méthodes que nous jugeons obligatoires pour implémenter la communication entre le pair et le tracker. C'est également dans cette classe que nous avons choisi de mettre en attribut les informations du client ainsi que celle du serveur du pair. Par conséquent, toutes les classes héritant d'**AbstractPeer** ont toutes les informations qu'un pair est supposé avoir pour être jugé comme tel. Le multithreading est également implémenté à ce niveau là afin de permettre à cette classe de déjà pouvoir utiliser un thread de même que pour les classes qui en héritent pourront utiliser les threads libres restant.

La fonction **connexionTracker** illustre parfaitement cette approche : en lançant un nouveau thread pour gérer la communication réseau, elle permet à l'application de rester réactive tout en gérant plusieurs connexions simultanément. Ce thread spécifique créé un socket et gère l'échange de commandes de façon continue, permettant ainsi des interactions fluides et non bloquantes jusqu'à la déconnexion demandée par l'utilisateur.

La création d'un lien ***Est-un*** entre la classe **AbstractPeer** et **AbstractPeerInterface** permet l'utilisation de *Mock* lors de la réalisation des tests unitaires de certaines fonctions.

Tout cela, nous a permis de poser la base de la création du pair. Il a ensuite fallu passer à la partie serveur et à ce qu'elle doit implémenter pour pouvoir envoyer des fichiers.

FIGURE 6 – Classe AbstractPeer

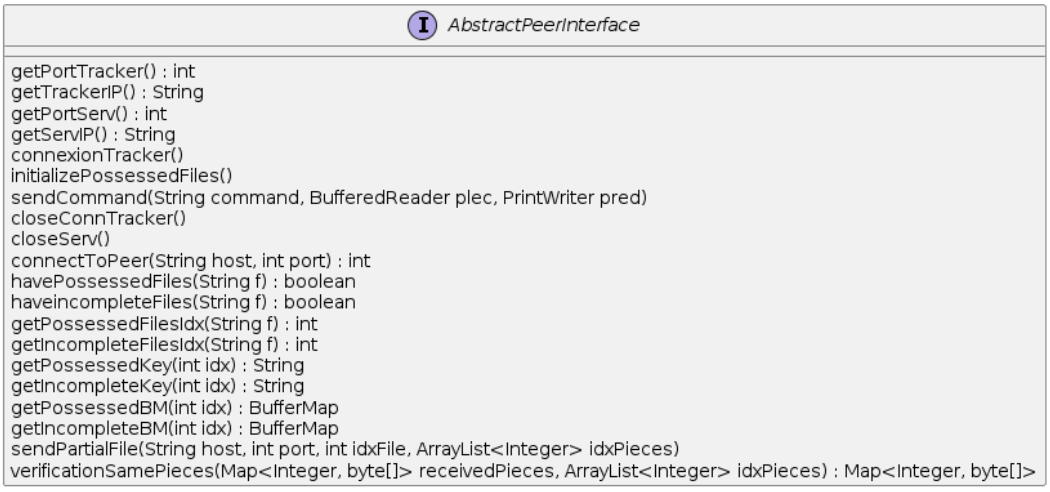


FIGURE 7 – Interface AbstractPeerInterface.

3.3 Un serveur-client qui envoie des fichiers

Dans cette partie, seront détaillées la manière dont nous avons implémenté notre serveur ainsi que notre implémentation pour envoyer des fichiers.

3.3.1 Un serveur

Pour mettre en place le serveur, nous avons eu besoin d'une fonction qui serait capable de démarrer un serveur sur l'attribut **ServerSocket** défini dans la classe **AbstractPeer**. C'est pourquoi nous avons décidé que nous allions créer une classe qui se nomme **RequestHandler**. Cette classe est illustrée par la figure 8 et elle hérite de la classe **AbstractPeer**. Ainsi, la classe **RequestHandler**

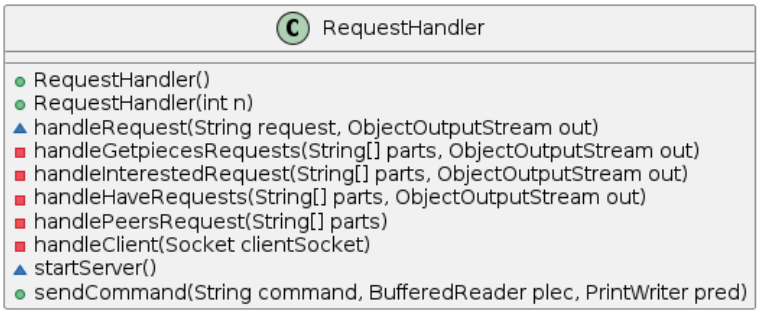


FIGURE 8 – Classe RequestHandler

implémente déjà la partie utilisateur grâce à l'héritage et elle peut désormais mettre en place la partie serveur en ne donnant pas la possibilité à l'utilisateur d'avoir un impact sur le serveur.

Il est également possible de créer une classe intermédiaire qui aurait un lien ***A-un*** avec la partie serveur et la partie client mais il aurait fallu écrire un nombre conséquent de méthodes pour mettre en commun certaines informations qui doivent être partagées par les deux parties. La classe **RequestHandler** implémente donc les méthodes qui permettent au pair de communiquer avec le serveur d'un autre pair. Également, ce sont elles qui permettent de gérer les requêtes envoyées par un autre pair qui désire télécharger un fichier.

Grâce aux threads qui sont initialisés lors de l'instanciation de la classe mère **AbstractPeer**, le serveur peut ainsi lancer un thread qui s'occupe d'un client à chaque fois qu'un se connecte. On a donc un thread associé à un client durant l'échange d'informations. Une fois le serveur prêt à accepter la connexion de clients et à repondre à leurs requêtes, nous avons mis en place l'envoi des fichiers.

3.3.2 L'envoi des fichiers

Pour l'envoi de fichiers, trois classes ont été centrales dans son développement : **FilePiece**, **BufferMap** et surtout **FilePieceData**, qui est véritablement au cœur du processus. Voici son diagramme de classe ci-dessous :

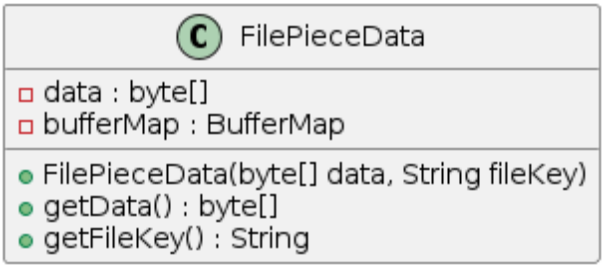


FIGURE 9 – Classe FilePieceData.

En effet, la classe **FilePieceData** (figure 9) joue un rôle clé dans le mappage qui associe un entier (l'index d'une pièce) à une instance de cette classe (un mappage que renvoie la fonction **getPiecesTosend** dans la classe **FilePiece** (figure 4)). Elle permet de lier chaque pièce à la **BufferMap** de son fichier. Ainsi, pour chaque pièce envoyée par le serveur, le client sait quelle taille doit avoir le tableau de pièces à créer et où chaque pièce doit être placée dans ce tableau, ce qui facilite grandement la tâche.

Dans notre code, deux fonctions sont responsables de l'envoi, de la réception et du placement des pièces de données : du côté serveur, dans la classe **RequestHandler**, nous avons **handleGetPiecesRequests** ; et du côté client, dans la classe **AbstractPeer**, la fonction **connectToPeer**. Tout commence avec la réception de la requête **getpieces** côté serveur. Après la réception de cette requête par **handleGetPieces**, la requête est transformée en un mappage entre le numéro de la pièce et une instance de la classe **FilePieceData**. Cette instance est ensuite transmise au client grâce aux classes **ObjectOutputStream** et **ObjectInputStream**, qui sont très utiles car elles permettent de transférer des objets sous forme de bits à condition qu'ils soient sérialisables, ce qui est le cas des objets issue des classes **BufferMap** et **FilePieceData** 3.

Une fois reçues par le client, trois structures nous permettent de gérer les pièces : **possessedFile**, **incompleteFile** et **MapReceivedOriginal**. **MapReceivedOriginal** est un mappage entre la clé

du fichier côté serveur et la clé du fichier côté client afin de se souvenir que lorsqu'on reçoit une pièce d'un fichier de clé $k1$ du serveur, elle doit être placée dans le fichier de clé $k2$ côté client.

3.4 Réalisation de test

Pour les tests, nous en avons réalisé un maximum lorsque cela était possible. Cela est caractérisé par l'ensemble des tests unitaires pour les classes **KeyGen** et **MessageGenerator** par exemple. Nous n'avons pas réalisé de test pour les fonctions *getter* et *setter*. En ce qui concerne les autres classes, nous avons essayé d'utiliser des **Mock** pour réaliser un faux tracker qui répond *yes* lorsqu'on lui envoie une requête qu'il doit recevoir ou encore réaliser un faux serveur qui répond aux requêtes d'un client pour voir comment il interprète certaines requêtes. Pour certaines fonctions qui servent à l'envoi de fichier, nous avons réalisé des tests pour voir si nous arrivions à envoyer certaines informations entre deux programmes.

Pour certaines fonctionnalités, nous avons priviliégié des tests qui se réalisent directement en connectant le pair au tracker car il est plus simple de tester directement de cette manière plutôt que d'écrire certains programmes de tests qui nécessitent une surchage de la méthode, ce qui ne reviendrait pas au final à tester la méthode initiale.

4 Utilisation de LLM

L'utilisation de différents LLM a été autorisée dans le cadre de ce projet, mais il est important de les utiliser avec prudence. Il est facile de tomber dans le piège de simplement copier-coller un morceau de code généré par un LLM sans le comprendre, ce qui pourrait nuire à la durabilité du projet. Pour éviter cela, nous avons décidé de les utiliser pour des tâches spécifiques seulement.

Nous avons utilisé au cours du projet majoritairement `ChatGPT` et `co-pilot`, le premier surtout pour de la documentation, trouvant cette approche plus simple. Ensuite, nous l'avons utilisé pour les tests unitaires, ce qui nous a permis d'obtenir un ensemble de tests après avoir fourni le code. Pour le débogage, nous voulions utiliser `co-pilot`, seulement nous avons assez rapidement abandonné l'idée pusique les réponses obtenues n'étaient que très rarement pertinentes.

5 Gestion du projet

Pour réaliser notre projet, nous avons décidé de scinder l'équipe en deux afin que chacun puisse travailler sur des parties indépendantes du projet. La répartition s'est faite de la manière suivante :

— Côté **Tracker** (programmation C) : **Abdurahman**, **Joachim** et **Joris** s'en sont occupés. Abdurahman au niveau de la connexion avec les clients ainsi que du gestionnaire des requêtes client-serveur, Joris de l'interprétation des différents messages pouvant être reçus et des différents diagrammes de classe de ce rapport. Quant à Joachim, il s'est concentré sur le stockage et la mise à jour des informations reçues provenant des requêtes des clients.

— Côté **client** (programmation Java) : **Mathys** et **Yannis** y sont associés. Mathys s'est occupé de réaliser la fonction permettant de créer la clé, de traiter les requêtes des autres pairs lors

d'intéraction pair à pair, des fontions permettant de récupérer les informations pour se connecter au tracker et pour remplir les informations des fichiers puis réaliser la connexion, Yannis s'est occupé de la création de la classe `FilePiece`, de la classe `BufferMap`, de l'implémentation des threads mais aussi du téléchargement des fichiers

A chaque séance de projet, nous faisions un point sur la situation de chacun, pour savoir si une ou plusieurs personnes avaient des difficultés particulières puis nous nous accordions sur les choses à faire durant la séance. Avant la fin de cette dernière nous essayions de se donner du travail à réaliser avant la nouvelle. Pour avoir une idée plus clair de la réalisation du projet, son découpage la sous forme de User Story est illustré par la figure 10

FIGURE 10 – Découpage du projet sous forme de User Story.

6 Conclusion

Les fonctionnalités principales du réseau comme le partage de fichier ont été implémentées et sont fonctionnelles dans notre application. Nous avons développé un tracker centralisé qui fournit aux clients les informations nécessaires pour leurs échanges. La gestion des interactions clients, le traitement des messages et le stockage des données ont été mis en place de manière efficace grâce à l'utilisation de listes chaînées. De plus, le client a été développé avec une structure bien définie pour sa communication avec le tracker et son rôle de serveur vis-à-vis des autres pairs du réseau.

Les LLM ont été intégrés dans notre processus de développement pour enrichir notre code, et une répartition judicieuse des tâches au sein de l'équipe a assuré une réalisation continue du réseau

au cours des séances de projet.

Cependant notre projet a ses limites, en effet plusieurs fonctionnalités n'ont pas pu être abordées. Par exemple le pair qui envoie périodiquement **Have** automatiquement n'a pas été implémentée, seule la réponse côté serveur l'est. Nous n'avons également pas eu le temps de faire en sorte que le nombre de threads soit déclaré dans le fichier config.ini et de développer l'interface graphique.