Projet réseau 3A SEOC 2020/2021 : Client Bittorrent
Rapport

Equipe 01

Valentin Marull
Thomas Gaillot
Muhammad Mohd Najib

Supervisé par Oliver Alphand

21 janvier 2021

1 Introduction

Lorsqu'on télécharge un fichier sur internet, peu importe le format, généralement c'est un échange entre 2 machines (client et serveur). Cependant, il existe une manière plus efficace pour se faire : utiliser un protocole peer-to-peer, qui permet à plusieurs machines de communiquer entre elles, comme le Bittorrent, dont nous avons réalisé un client de zéro.
Ce rapport a donc pour objectif d'exposer notre travail au cours de ce projet.

2 Partie technique

2.1 Vu d'ensemble de l'implémentation et détails

Résumé des fonctionalité suportées :

Sprint	Fonctionnalité	Avancement
Sprint 1	Tracker (HTTP Req URLEncodé/Réponse de-béncodée)	OK
Leecher 0%	Socket bloquante + java.io	OK
Leecher 0%	Ecriture pièce sur disque	Au fur et à mesure
Leecher 0%	Message inconnu traité	OK
Leecher 0%	Sélection de pièces	Aléatoirement
Seeder 100%	Sélection de pièces : Stratégie	Incrémental
Seeder 100%	Bitfield créé en fin du torrent	OK
Seeder 100%	Gestion pièces et blocs	OK
	Clients supportés	Vuze, aria2c
Sprint 2	Support Multi-leecher (socket non bloquante OP_ACCEPT)	OK
	Support Multi-seeder	OP_CONNECT
	Bytebuffer	1 pour l'envoi, 1 pour la réception
	Sélection de pièces : Conditions Sprint 2 respectées	Pas fini
	Eval. de perf	OK
	Resume calculé à partir du fichier partiel	KO
	Scénario multi-leecher supporté	KO
	Gestion concurrence Thread	KO
Sprint 3	Designs patterns	Machine à état
	Sélection de pièces	Pas fini
	Message supporté HAVE, CANCEL, KEEPALIVE	OK
	Clean Code, Factorisation, Archi. Objet, ...	OK
Avancé	Tests automatisés	
	Taille de pièce (16K et +)	OK
	Tracker contacté à intervalle régulier	KO

Table 1 – Fonctionnalités

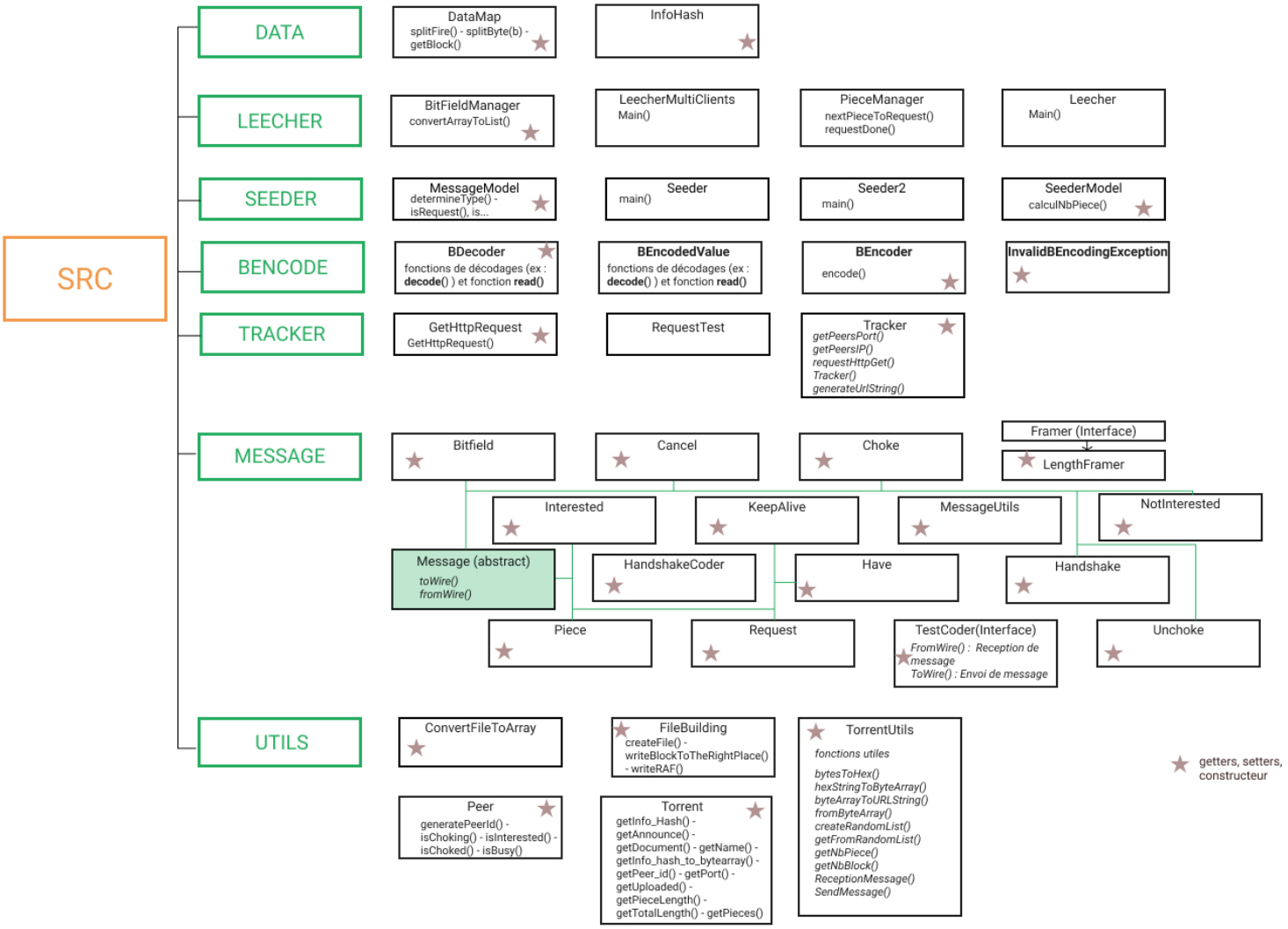


Figure 1 - Diagramme UML de classes

2.2 Fonctionnement du Leecher

La première phase de l'exécution du Leecher consiste à initialiser les objets nécessaires à l'exécution du programme. On fait alors appel à plusieurs classes utiles du projet tel que Utils.Torrent, utils.TorrentUtils ou encore des classes importées (ex : java.nio.channels), ce qui nous permet de :

— récupérer l'InfoHash et le Peer ID ;
— récupérer le nombre de pièces et de blocks du fichier ;
— définir la communication avec le tracker et récupérer les peers et les ports ;
— se connecter au premier canal ouvert.

Ainsi, la communication peut commencer.

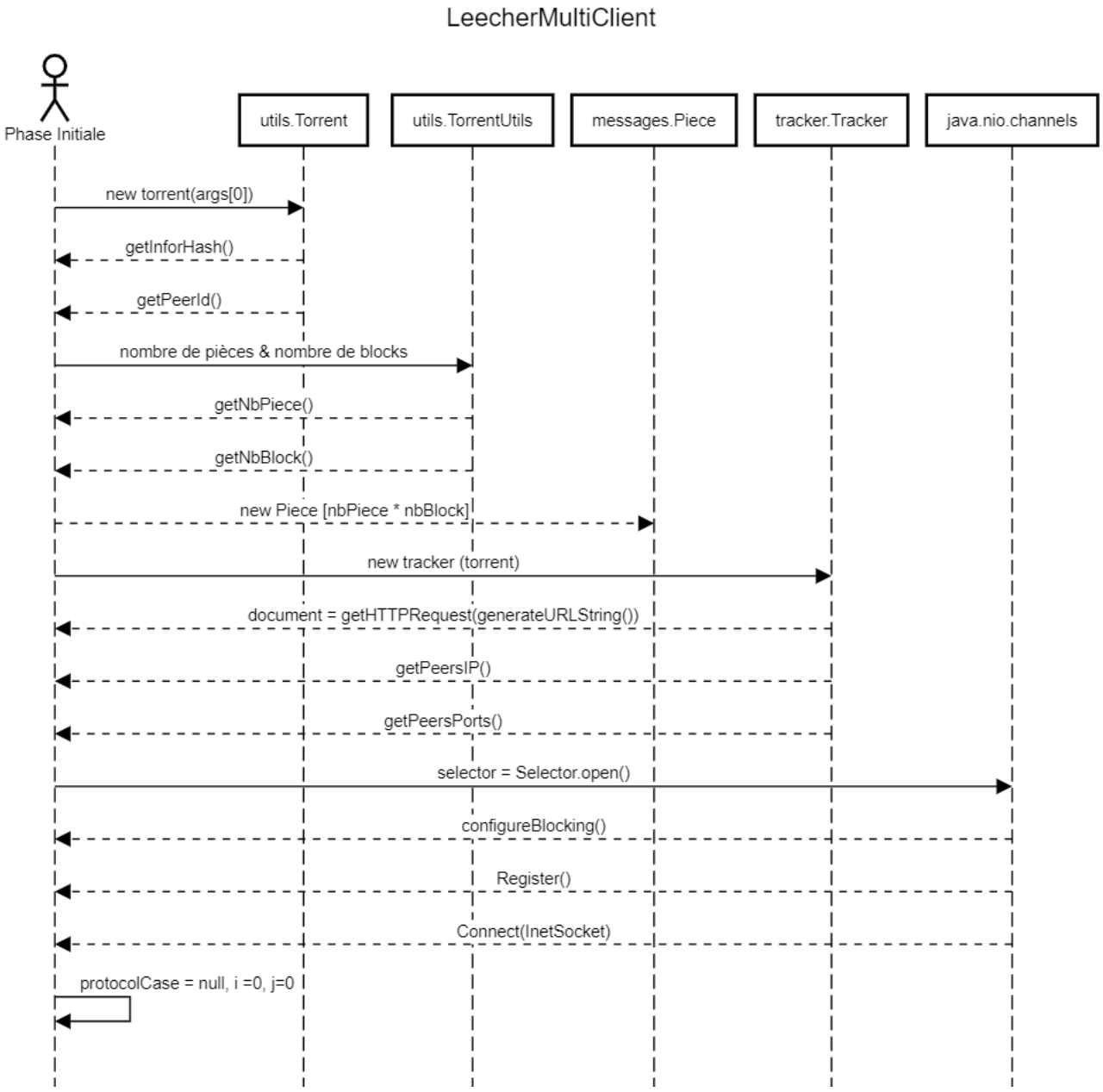


Figure 2 - Diagramme séquentiel : Initialisation Leecher Mutli Clients

Lorsqu'un canal est prêt, on le ratache au peer, puis on alloue un buffer, qui va être chargé de récupérer des messages.
On débute ensuite la communication par un envoi d'HANDSHAKE, avec la méthode toWire(), wrap() et write(). On passe ensuite dans l'état d'attente de réception de message (key.isReadable()), où la réponse de notre HANDSHAKE doit être reçue.
On reçoit également le BITFIELD, comportant des informations telle que les pièces détenues par le peer. On passe ensuite dans le troisième état (key.isValid() key.isWritable()), où l'on envoi un message d'intêret au peer (INTERESTED).
On attends ensuite la réponse devant être UNCHOKE, puis l'on choisi entre deux types de requêtes pour obtenir nos fichier : la requête de gros fichier ou la requête d'un petit fichier. Le diagramme suivant illustrera ces requêtes, permettant d'obtenir le fichier en question.
Une fois le fichier obtenu, on envoi un message de fermeture (NOT INTERESTED) et on organise les pièces reçues.

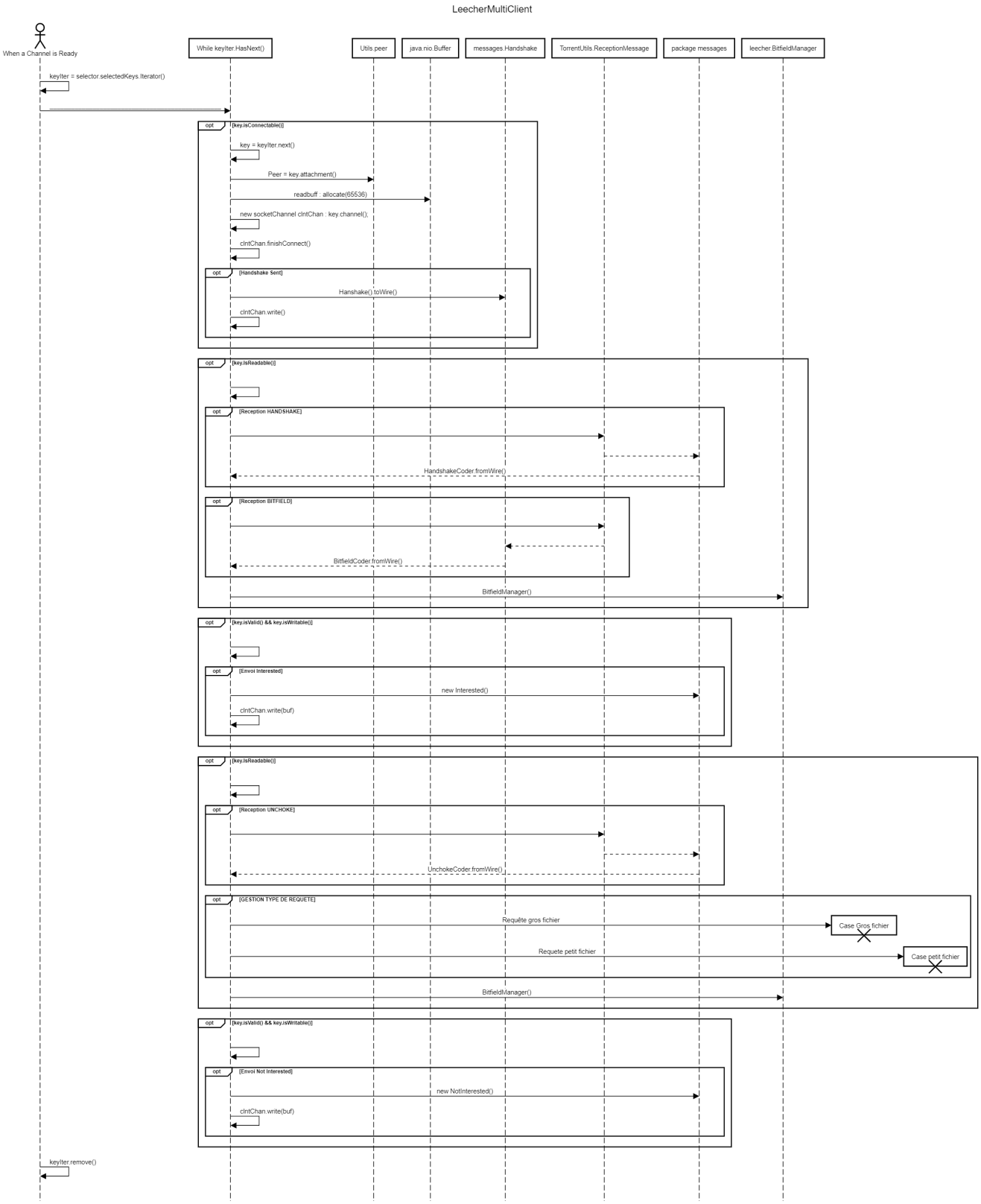


Figure 3 - Diagramme séquentiel : Communication avec un peer

La figure 3 montre l'enchaînement d'une communication entre un peer et le leecher, dans la structure (Connexion - Attente de réception - Envoi de message).
Ainsi, les envois de message sont obligatoirement réalisés dans l'état .isValid() .isWritable() tandis que la réception dans l'état .isReadable() (sauf pour le premier HANDSHAKE, réalisé dans l'état .isConnectable()).
Comme expliqué dans le diagramme précédent, la communication s'effectue par un échange

d'HANDSHAKE, puis réception de BITFIELD, envoi d'INTERESTED, et réception d'UNCHOKE. Ensuite, la phase de demande de fichier et de sa récupération dépend de la taille du fichier.

Dans le cas d'un petit fichier, de taille totale inférieure à une taille d'une pièce, on envoie qu'une seule requête, ayant pour réponse la réception de la piece, et l'envoi du message HAVE, validant la réception.

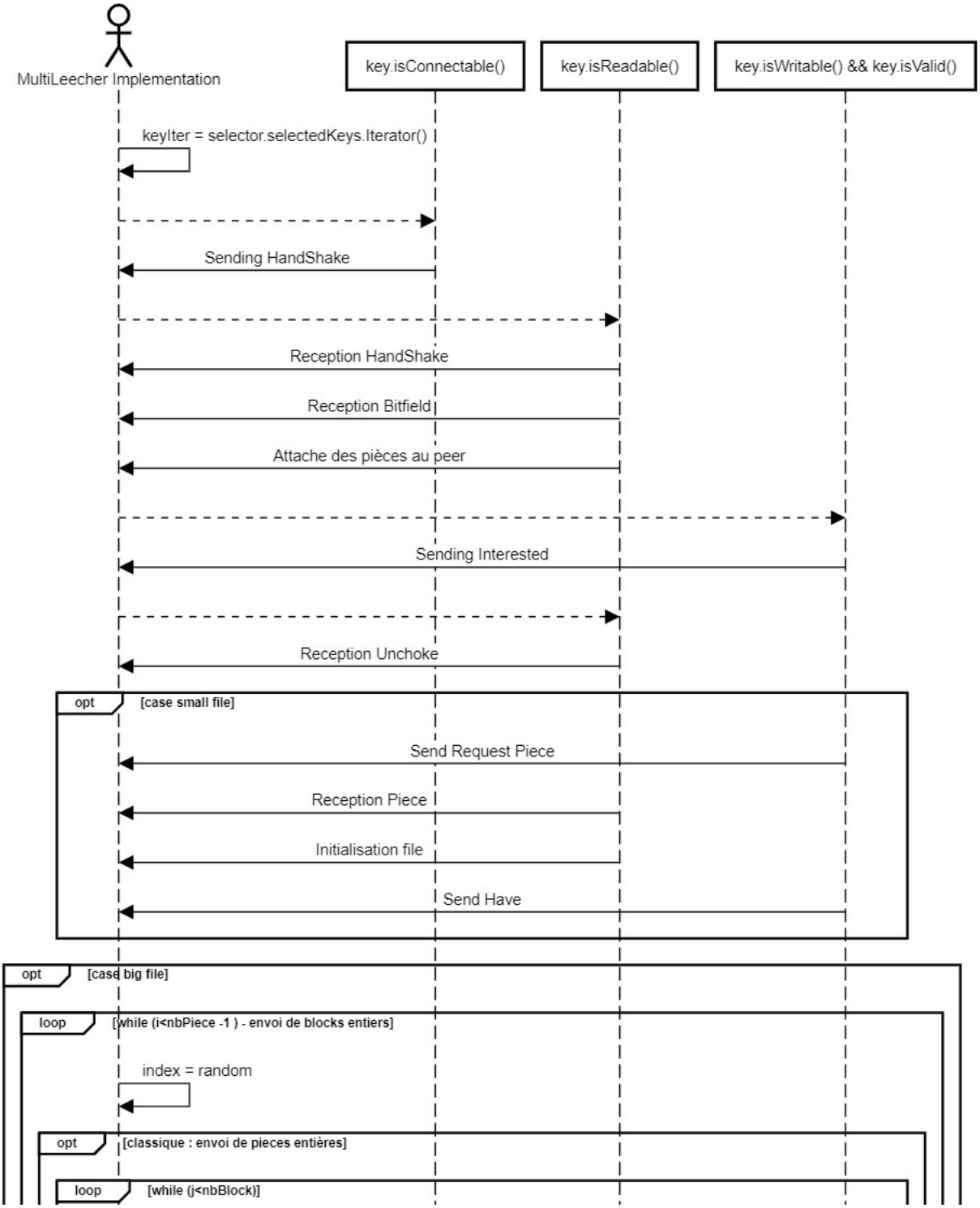
Dans le second cas, le fichier à envoyer doit être composé de plusieurs pièces, et potentiellement de plusieurs blocks. On envoie ces pièces dans un ordre d'index aléatoire, complète : c'est-à-dire que la longueur de la pièce demandée est égale à la longueur maximale.
A chaque réception d'une pièce, on envoi un message de validation HAVE.
On définit enfin trois cas finaux, pour la dernière pièce à envoyer :

— soit elle est entière, et on envoi la même requête que précédemment,

— soit elle est plus petite qu'un block, et on envoi une requête pour la taille restante du fichier,

— soit elle est plus grande qu'un block mais plus petite que nBBlock, et dans ce cas on envoi des requête de block entier plus une requête du block final.

LeecherMultiClient - Machine à état



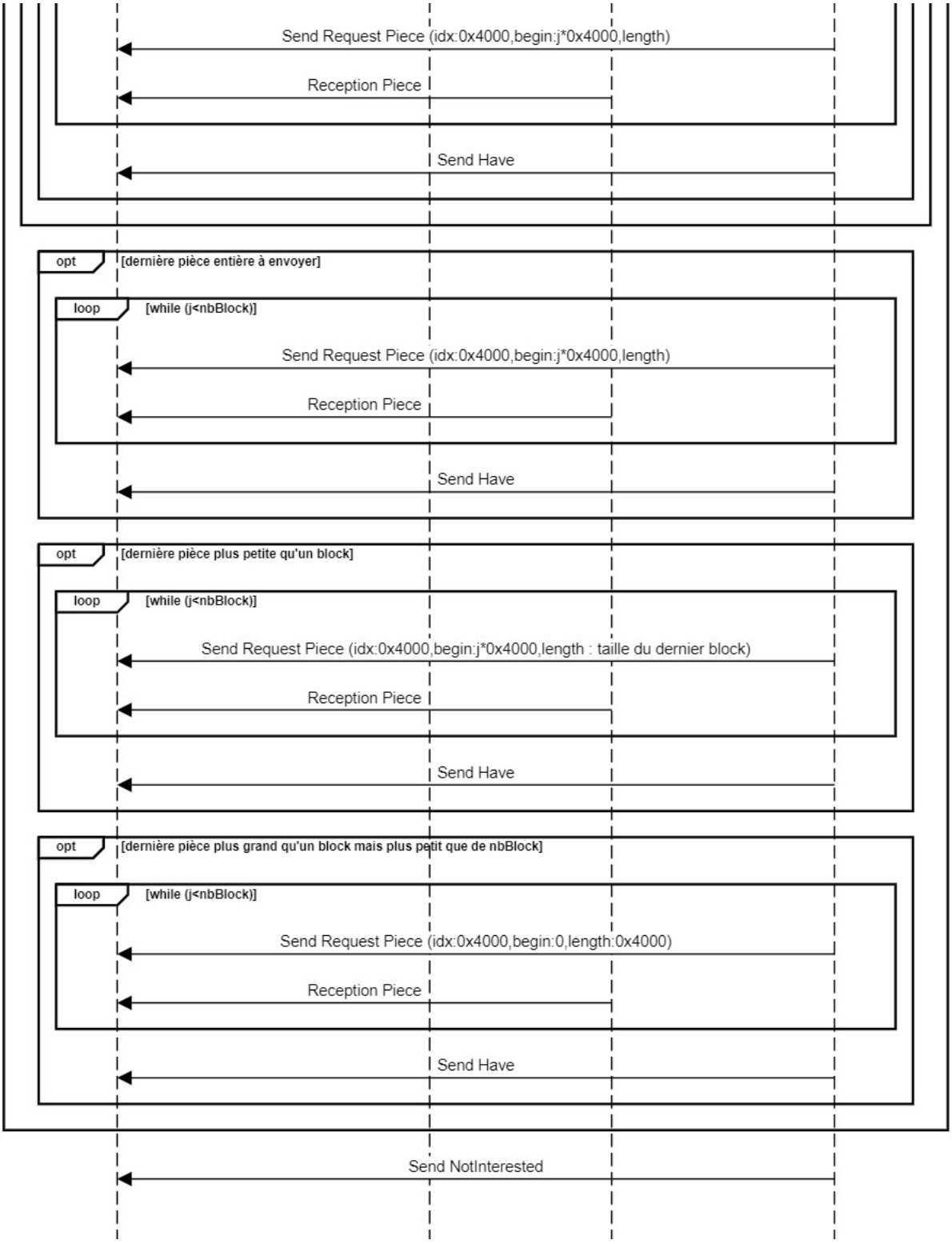


Figure 4 - Diagramme séquentiel : Machine à état

Pour la stratégie de sélection des pièces et leur gestion :
Dans un premier temps, pour le leecher avec un seul Peer, nous récupérons le nombre de pièces et le nombre de blocs par pièce du torrent, avec les méthodes getNbPiece et getNbBlock de la classe TorrentUtils.
Avec le nombre de pièce, on créer ensuite une liste d'entiers de 0 à nbPiece, dans un ordre aléatoire. C'est dans cet ordre que seront téléchargées les pièces.

Pour le cas du multi client :

— Sauvegarde de l'état de chaque Peer :
On peut envoyer une demande de Piece d'un Peer ssi le Peer n'est pas occupé par un autre client. Donc, à la réception de requête Unchoke, on sait que le Peer n'est pas occupé par un autre client. Dans ce cas-là, on peut faire des envoie de Request.
Pour réaliser ceci, nous avons créé l'objet Peer en Java, qui stocke les données de chaque peer. Un peer est constitué par des attributs suivants :

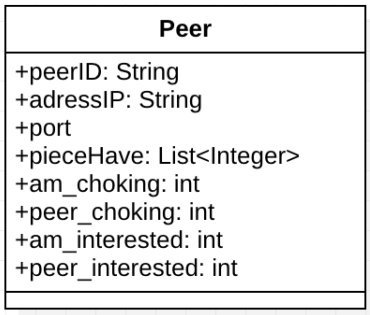


Figure 5 - Attributs de classe Peer

En sauvegardant l'état de chaque Peer, on peut s'assurer que les peers ne sont pas occupé le transfert de fichier et la communication point à point entre les clients ne sera pas interrompu. Ainsi, on a l'information sur la liste de Pièce disponible pour chaque Peer.

— Sauvegarde de la liste de pièces disponible de chaque Peer :
Dans chaque objet Peer, nous avons créé l'attribut "pieceHave". Cet attribut correspond à la liste pièce disponible pour un client et il peut-être la liste de pièces téléchargée dans le cas d'un Leecher. Cet attribut est mise à jour à la réception de requête Bitfield de chaque Peer. Nous avons créé la classe BitFieldManager pour gérer les requêtes BitField reçues par notre programme. Il convertis le payload de Bitfield en une liste d'entière qui correspond aux indexes des pièces disponible dans chaque Peer.

— Algorithme de sélection de pièce :
La stratégie de sélection de pièce à envoyer est dépendante de la liste de pièces déjà téléchargées par le client/leecher et la liste de pièces disponible pour chaque Peer/seeder. A chaque réception de requête Unchoke et Piece, nous allons faire une demande de Piece au Peer correspondant en envoyant la requête Request. Nous avons également créé la classe PieceManager qui sert à implémenter l'algorithme de sélection. Dans cette classe, on peut créer des différents algorithme de selection. Nous n'avons pas le temps d'implementer d'autre algorithme mais l'algorithme que nous avons créé est suffisant pour le fonctionnement de programme. Cette classe contient la liste de pièces demandé (requested) pour assurer qu'on ne fait pas la demande d'une même pièce à deux Peer différent.
A la réception de requête Unchoke, nous allons parcourir pieceHave de ce Peer et on cherche la pièce qui n'est pas encore demandé ou la pièce qui n'est pas dans la liste

"requested". Si on trouve cette pièce, on va retourner l'index pour ensuite envoyer la requête Request. Si on ne trouve pas la pièce correspondant, cela veut dire que ce Peer n'a pas la pièce qu'on souhaite avoir. Dans ce cas, on va retourner l'index -1 pour montrer qu'on n'est plus intéressé avec ce Peer.

Nous n'avons pas eu le temps de terminer le Leecher Multi-Clients. En effet nous n'avons pas implémenté la gestion des pièces pour plusieurs peers. Le leecher multi-client fonctionne donc que dans le cas d'un seul peer et dans le cas où il y a plusieurs peers avec chacun une pièce différente.

2.3 Performances

Scénario de test : Pour réaliser ces tests, nous utlisions ces 3 clients avec un fichier de 1Go, en local (nous n'avons pas réussi à faire fonctionner les tests entre deux machines de l'ensimag car le tracker ne renvoyait pas le peer dans sa réponse du GET).

Commandes à effectuer :

java -jar Leecher_Mono.jar <file.torrent>
java -jar Seeder.jar <file> <file.torrent>
java -jar Leecher_Multi.jar <file.torrent> (N'est pas terminé)

Expérience sur un fichier de 3Go	Débit
Client (0%) VS Aria2c(100%)	35 Mo/s
Client (100%) VS Vuze(0%)	40 Mo/s
Client (0%) VS 3 Aria2c (50%, 50%, 100%)	

TABLE 2 – Performances

Pour le premier test, le résultat a été obtenu avec aria2c (Wireshark donne cependant un débit supérieur, de 55 Mo/s).
Le deuxième test, on peut observer le débit directement sur Vuze.
Le troisième test n'a pas été effectué, car le leecher Mutli Clients n'est pas opérationnel.

Ces tests ont été réalisés sur la configuration suivante :

- Macbook pro 2015
- Processeur 2,2 GHz Intel Core i7 quatre coeurs
- Mémoire 16 GO 1600Mhs DDR3
- Disque dur : SSD 251 GO, vitesse de liaison : 8 GT/s
- Réseau : 1Go/s de capacité de connexion

2.4 Tests et validation

Dans un tel projet, il est nécessaire de tester les fonctionnalités implémentées au fur et à mesure. Nous avons commencé par tester nos classes de messages, et donc le leecher (envoie et réception des messages à la suite) par la même occasion, avec Vuze en seeder et en localhost (sans le tracker) et nous regardions sur Wireshark si tout se déroulait correctement. Nous avons ensuite testé le Tracker avec une méthode qui y faisait appel et nous regardions sur Wireshark si la réponse était présente et conforme à nos attentes. Puis nous avons testé notre seeder (qui nécessite le bon fonctionnement du tracker) avec Vuze en leecher, et toujours avec Wireshark.

Nous avons enfin fait de même pour le leecher Multi-Clients, sauf que nous avons utilisé aria2c pour avoir plusieurs seeder (et avoir plus de fiabilité qu'avec Vuze qui fonctionne de manière irrégulière). Pour le Multi-Clients, nous avons utilisé le script test_local.sh, qui permet de lancer plusieurs aria2c en seeder en parallèles.

Au cours de ce projet nous avons plusieurs bugs résolus, et des difficultés/subtilités qui ont coûté du temps, voici les principales :

— Opentracker n'a pas fonctionné pendant un moment. Cela a été résolu en récupérant une ancienne version compilée d'Opentracker.
— Au début lorsque nous développions les messages et que nous les testions, nous avons eu un problème avec le message Handshake car nous n'avions pas géré correctement le info_hash, et donc le seeder n'arrêtait la connexion.
— Dans le cas de plusieurs peers, nous avons eu un problème avec les ports récupérés, qui étaient tout le même. Le problème venait du code de récupération de ces ports dans une liste (le dernier récupéré effaçait les autres).
— Nous avons eu un problème pour l'écriture du fichier sur disque. A la base nous écrivions sur disque toutes les pièces d'un coup, à la fin du programme. Ce n'était pas optimisé, mais le problème a été que lorsque nous avons introduit le téléchargement des pièces dans un ordre aléatoire, cette technique ne fonctionnait plus. Nous avons donc utilisé un autre type de donnée pour le fichier (RandomAccessFile), ce qui nous a permis d'écrire au bon endroit après chaque téléchargement de pièce.

2.5 Bonnes pratiques

Pour ce qui est des étapes d'architecturation et de factorisation, nous avons commencé par créer une classe pour chaque type de messages, le tout dans le package messages. Chaque message contient un constructeur, des getters et setters, et deux autres méthodes : toWire et fromWire qui permettent respectivement de coder et décoder un message sous la bonne forme (les messages sont envoyés et reçus sous forme de ByteArray).
Nous avons ensuite créer une classe Torrent avec plusieurs méthodes pour récupérer toutes les informations que contiennent un Torrent, et une classe TorrentUtils pour toutes les méthodes utiles à la factorisation du code, le tout dans un package utils. Puis nous avons créé 3 packages pour chaque les 3 grosses parties du projet : leecher, seeder, et tracker, qui regroupe toutes les classes relatives.

En terme de design pattern, on a utilisé une machine à état pour le leecher, chaque été correspondant à un message du protocole Bittorent, avec des états différents pour le message request, selon si c'est la dernière pièce (i.e le dernier index) ou pas.
On a essayé de faire le plus possible de Clean Code en ayant des noms le plus explicite possible pour les classes, les attributs et les méthodes mais aussi séparant bien chaque fonctionnalité, et en commentant le plus possible le code.
En ce qui concerne git, nous utilisions principalement la branche master sauf si nous avions quelque chose à tester et qui ne fonctionnait pas à priori. Nous avons également essayé de mettre des messages pour les commit les plus explicites possibles. Pour notre gestion de la méthode SCRUM, cf la partie 3.1.

3 Partie personnelle

3.1 Organisation du travail

Dans un tel projet, l'organisation du travail entre les différents membres de l'équipe est primordial.

Pour ce projet nous avons utilisé les méthodes agiles. En effet, le travail a été organisé en 4 sprints. Un premier sprint d'une semaine pour découvrir le sujet, un deuxième sprint d'environ 7 semaines pour le mono client (leecher 0% et seeder 100% avec la gestion du tracker), un troisième sprint d'environ 6 semaines pour le multi-client, et enfin un quatrième sprint d'une semaine pour finaliser le projet. Nous avions un SCRUM master (Valentin) qui organisait régulièrement des réunions quotidiennes (sur Discord) afin de savoir qui a fait quoi et qui va s'occuper des prochaines tâches. Nous avons mis en général 2 personnes sur une "grosse" tâche et une autre sur une tâche moins compliquée afin d'équilibrer correctement le travail, mais cela n'a pas empêché de s'entraider lorsque besoin.

Pour savoir ce que nous avions à faire, nous avons réalisé des user stories, c'est-à-dire que nous avons mis sur des "post-it" (sur trello), chaque fonctionnalité pour chaque sprint avec leur utilité, ce qui permet de ne pas s'égarer et de développer ce qui est vraiment attendu.

3.2 Les principales difficultés d'organisation rencontrées

Au cours du projet, nous avons rencontré quelques difficultés d'organisation, en plus des difficultés techniques.
En effet, une des principale difficultés rencontrées est le fait de ne pas avoir fini nos tâches en même temps et d'être bloqué pour la suite, car on avait besoin d'une fonctionnalité par terminer. Par exemple, nous avons eu ce souci lorsque le leecher Mutli Client était terminé, mais pas la gestion des pièces.

3.3 Les retours personnels de chaque membre de l'équipe

Retour de Valentin :

Ce projet a pour moi était très enrichissant autant sur le plan technique que le sur le plan social. Premièrement, le sujet était très intéressant, mêlant à la fois la programmation orientée objet et réseaux avec des notions nouvelles, j'avais vraiment l'impression de travailler sur un réel projet d'ingénierie informatique et cela m'a permis de beaucoup m'améliorer sur la conception d'un logiciel. J'ai réalisé à quel point c'est un important d'avoir une bonne architecturisation, une bonne factorisation, des noms explicites et des commentaires quand on travail à plusieurs car cela permet d'y voir beaucoup plus clair sur l'ensemble du projet et d'être plus efficace. Cela m'a également permis d'en apprendre plus sur la gestion d'un tel projet en équipe et de voir ce qui est efficace ou pas en terme de gestion d'équipe.
Pour moi le plus difficile a été la compréhension du protocole au début, mais une fois compris cela a été beaucoup plus simple pour moi. J'ai eu également pas mal de problèmes techniques (Opentracker qui ne fonctionnait pas en trautre) qui m'ont pas mal ralenti sur l'avancement du projet, car avec un peu plus de temps nous aurions pu terminer le projet.

Retour de Thomas :

Ce projet informatique de longue durée m'a permis de comprendre de nombreux enjeux sur le développement d'un projet de groupe, telle qu'une organisation devant être rôdée ainsi

que des objectifs devant être précis et répartis dans le temps pour garder une motivation de groupe et un travail efficace. J'ai ainsi beaucoup appris au niveau technique, puisque ce projet de client Bittorrent a été enrichissant au niveau programmation, réseau, et également sur l'optimisation et la factorisation d'un code comportant de nombreux fichiers. Des difficultés sont apparues pendant ce projet puisqu'à distance pendant cette longue durée, il a fallu rester motivé, et les interactions étaient fréquentes, mais étaient aussi coûteuses en temps.

Retour de Muhammad :

Ce projet est le deuxième projet de longue durée en informatique pour moi. Cela me permet de revoir et appliquer les notions que j'ai appris pendant mes stages et pendant le projet de génie logiciel. La différence est que cette fois-ci, j'ai l'occasion de travailler sur un projet qui me passionne et que j'ai choisi moi-même au début de l'année. Le projet est intéressant parce qu'il y a le côté réseaux. Donc, cela m'a appris le protocole point à point (bittorrent) et on peut étudier l'avantage et l'inconvénient de ce type de protocole par rapport à d'autres protocoles que j'ai appris. Ensuite, il y a également l'aspect de développement logiciel et la gestion de projet. J'ai beaucoup apprécié l'atelier Clean Code qui montre l'importance d'une bonne architecture de logiciel pour pouvoir découper les fonctionnements de logiciel pour avoir un développement plus flexible et efficace.