Ecole Nationale Supérieure d'Informatique et de Mathématiques Appliquées

Implémentation d'un client BitTorrent

Projet 3A SEOC

Etudiants :

Rémi DEPREUX

Gloria EJ-JENNANE

Audrey LENTILHAC

Encadrant :

Olivier ALPHAND

Table des matières

1 Introduction

Ce document constitue le rapport du projet Client BitTorrent et propose une description de l'implémentation logicielle effectuée et de l'organisation globale de ce projet.

Ce projet a été réalisé par l'équipe 2 constituée des étudiants en 3ème année SEOC : Rémi Depreux, Gloria Ej-Jennane et Audrey Lentilhac dans le cadre du Projet SEOC de dernière année à l'ENSIMAG.

Nous aborderons dans ce document les points suivants:

- Fonctionnalités du système
- Architecture logicielle du système
- Algorithmes pertinents
- Performance du système
- Validation du système
- Organisation globale du projet

Le manuel de l'utilisateur pour lancer l'application se trouve dans le fichier Readme.md sur le dépôt.

2 Partie 1 : Implémentation du projet

2.1 Fonctionnalités de l'application

L'application BitTorrent supporte les fonctionnalités suivantes :

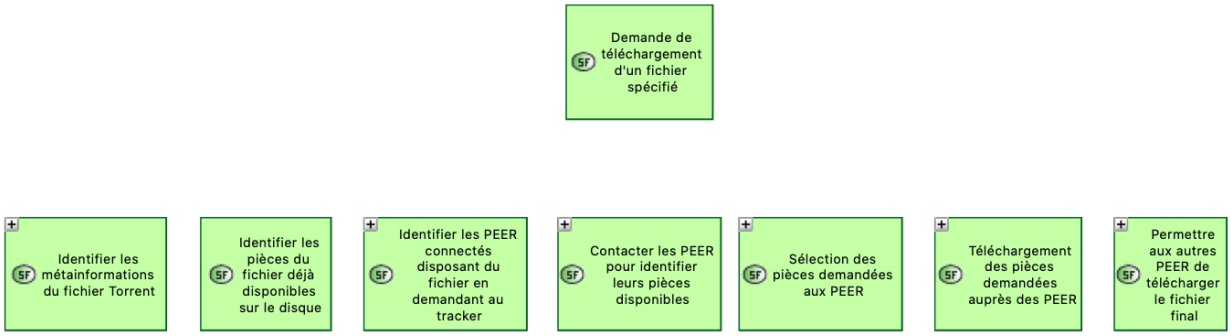

Figure 1: Fonctionnalités générales du système

La figure ci-dessous présente les fonctionnalités implémentées pour la fonctionnalité générale "Identifier les métainformations du fichier Torrent" :

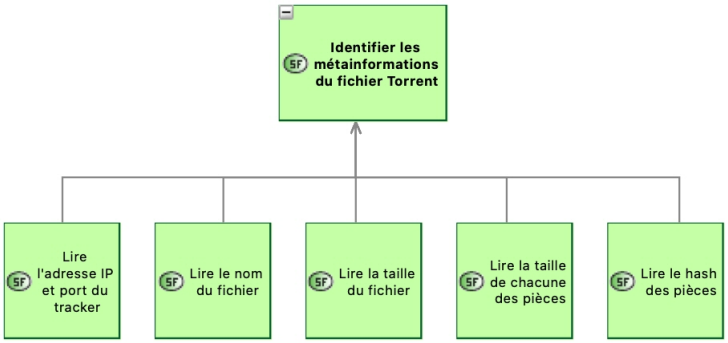

Figure 2: Fonctionnalités liées à l'Identification des métainformations du fichier Torrent

La figure ci-dessous présente les fonctionnalités implémentées pour la fonctionnalité générale "Contacter les peers pour identifier leurs pièces disponibles" :

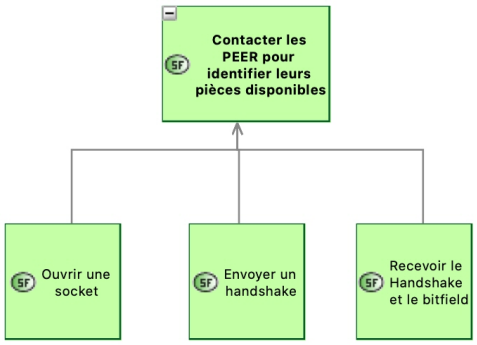

Figure 3: Fonctionnalités liées au contact des peers pour connaître leurs pièces

La figure ci-dessous présente les fonctionnalités implémentées pour la fonctionnalité générale "Sélection des pièces demandées aux peers" :

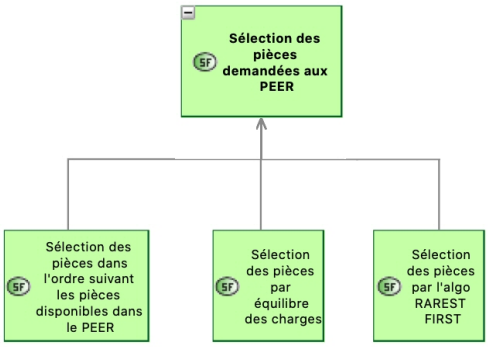

Figure 4: Fonctionnalités liées à la sélection des pièces

La figure ci-dessous présente les fonctionnalités implémentées pour la fonctionnalité générale "Téléchargement des pièces demandées aux peers" :

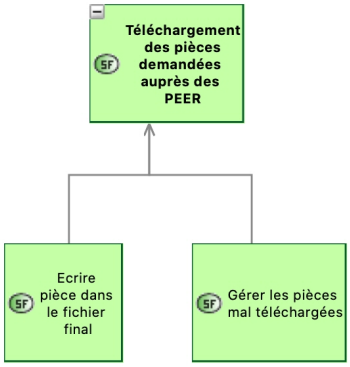

Figure 5: Fonctionnalités liées au téléchargement des pièces

La figure ci-dessous présente les fonctionnalités implémentées pour la fonctionnalité générale "Permettre aux autres peers de télécharger le fichier final" :

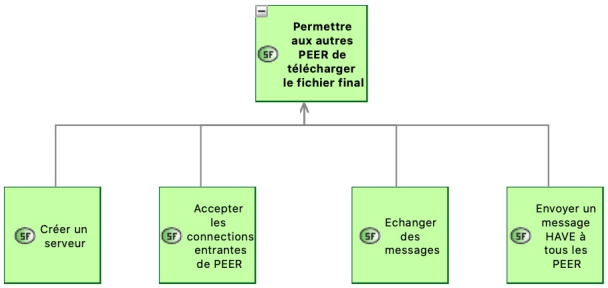

Figure 6: Fonctionnalités liées au seeding

2.2 Fonctionnalités non supportées de l'application

L'application ne supporte pas les messages de type CANCEL et KEEPALIVE. Les tests n'ont pas pu être automatisés par manque de temps et du fait que l'application ne supporte pas Aria2c lorsque celui-ci est leecher.

Un résumé des fonctionnalités implémentées et non implémentées se trouve en Annexe A

2.3 Architecture de l'application

L'architecture logicielle du système est présentée ci-dessous :

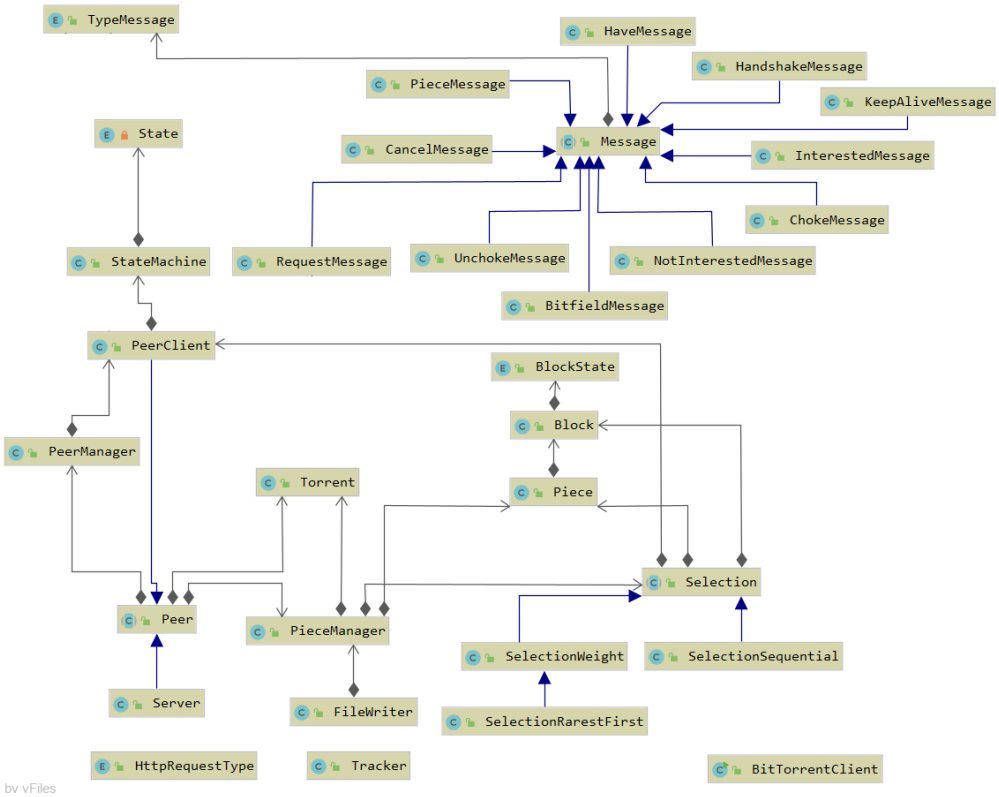

Figure 7: Diagramme de classe de l'application

Le point d'entrée du système correspond à la classe BitTorrentClient. La classe PeerManager gère les différents objets de la classe PeerClient correspondant aux peers. La classe Server gère la partie Seeder du système. La classe PieceManager gère les pièces du fichier. La classe FileWriter est en charge de l'écriture des pièces dans le fichier. La classe abstraite Message et ses différentes classes filles s'occupent de la gestion des différents types de message.
Les diagrammes de classes spécifiques aux différents packages sont en Annexe.

Le diagramme de séquence suivant présente l'enchainement des méthodes lors d'un téléchargement complet d'un fichier avec un peer :

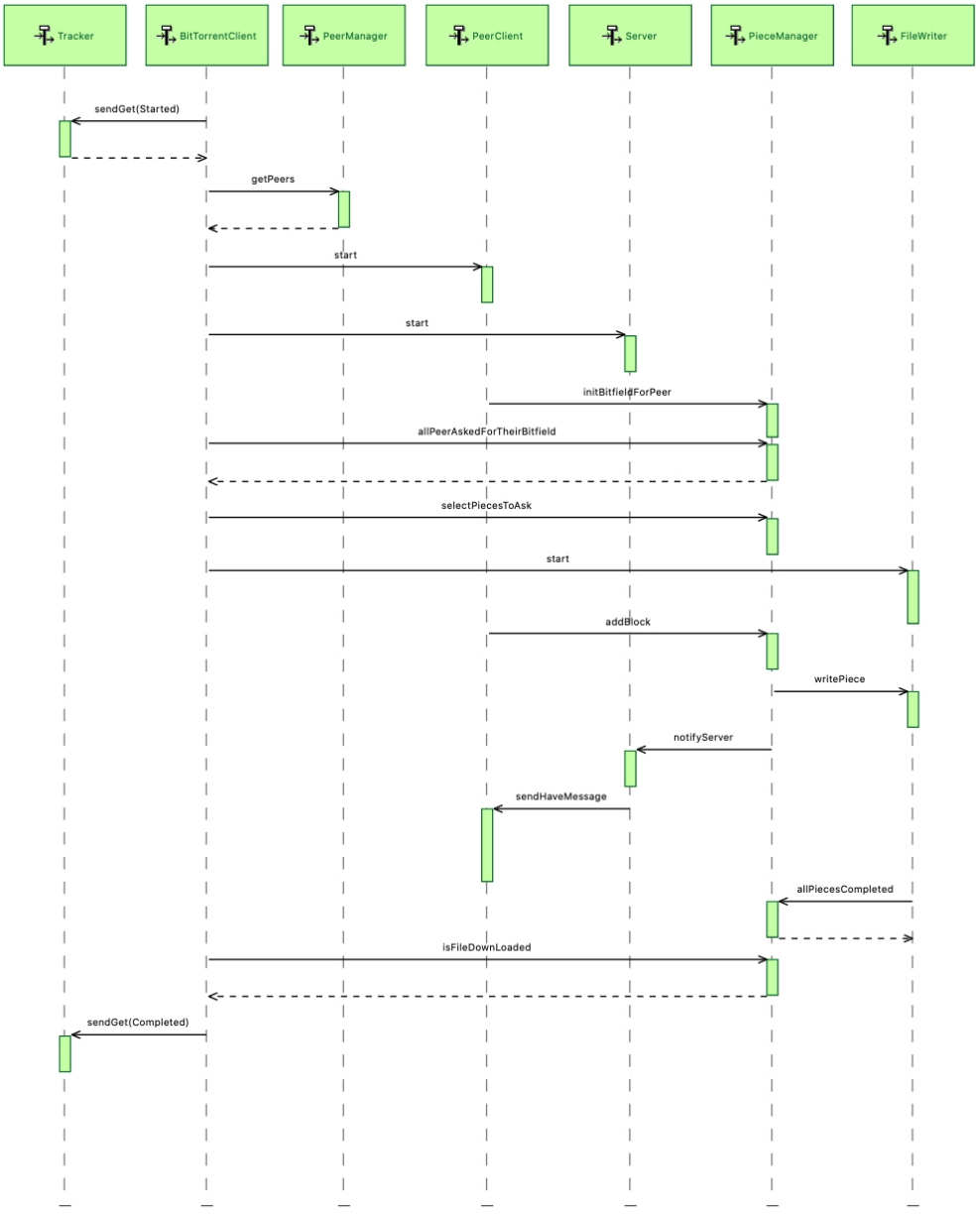

Figure 8: Diagramme de séquence d'un téléchargement de fichier auprès d'un peer

BitTorrentClient contacte le tracker pour connaître les peers connectés. Une fois, les peers connus,

un thread par peer est démarré. Le thread du server démarre également. Un échange de messages s'effectue afin de connaître les pièces que possèdent le peer. Une fois que le système a récupéré tous les bitfields des peers, la sélection des pièces s'effectue via le PieceManager. Le peerClient est alors informé via un booléen de PieceSelection que la sélection est terminée et qu'il peut demander des pièces au peer. Une fois une pièces entière récupérée, le pieceManager envoie via une queue au FileWriter la pièce à écrire dans le fichier. Une fois le fichier téléchargé complètement, BitTorrentClient envoie un message COMPLETED au Tracker.

2.4 Algorithmes

2.4.1 Traitement des messages

Le traitement des messages côté leecher utilise une machine à état avec 4 états pour différencier les cas où le client est choke/unchoke et interested/uninterested. Afin de traiter avec performance les messages reçus lors du leeching, un bytebuffer est alloué directement (AllocateDirect) et la socket TCP/IP est chargée au maximum pour augmenter le débit de téléchargement. Si les messages sont tronqués car le bytebuffer est plein alors celui-ci est compacté pour permettre la reconstruction du message lors de la prochaine lecture sur la socket.

Côté serveur (la partie *seeding* de notre application), la gestion des connexions avec les autres *peers* et les messages sont traités à l'aide d'un *Selector*. On le retrouve dans la classe *Server*, avec les méthodes qui gèrent l'ouverture de *socket*, l'acceptation de connexion avec d'autres peers et la réception ou l'envoi de messages à ceux-ci.

Pour ce faire, le *Selector* va se référer à un ensemble de clefs, chacune associée à une *socket* sachant que l'on dispose d'une *socket* par *peer*. Comme ces *sockets* sont susceptibles d'être utilisées pour recevoir ou envoyer des messages, les clefs gérées par le *Selector* possèdent des attributs spécifiants les opérations que l'on souhaite effectuer sur les *sockets* : lire et écrire le plus souvent.

Une fois que l'on dispose de ces *sockets* et de ces clefs, notre programme va itérer sur ces dernières en regardant les opérations souhaitées. Si on cherche à lire, on va alors récupérer l'*attachment* de la clef [1] sous forme de *ByteBuffer* que l'on va parser pour récupérer le message. Selon le type de message, nosu répondons sur la même *socket* avec le message correspondant aux spécifications du protocole BitTorrent. Par exemple : si l'on reçoit un message *Request*, on s'empresse d'envoyer la *Piece* correspondante.

À tout cela s'ajoutent quelques détails : dont notamment la vérification d'absence d'incohérence vis à vis du protocole dans les messages que l'on reçoit, notamment avec la méthode *checkMessageConsistency*. Par exemple, on s'assure d'avoir bien reçu un message *Interested* avant de répondre à un message *Request*.

2.4.2 Sélection des pièces

Plusieurs algorithmes de sélection ont été implémentés pour ce système:

- Un algorithme de sélection séquentiel
- Un algorithme de sélection par charge

[1] Autrement dit, le contenu reçu dans la *socket*

- Un algorithme de sélection par charge et les pièces plus rares d'abord

Cette classe intéragit avec la classe PieceManager en charge des pièces. Pour gérer les différentes pièces à télécharger et gérer la concurrence des accès aux données, les dictionnaires de type ConcurrentHashmap et Hashtable sont utilisés permettant une gestion des pièces simple et thread-safe.

2.4.3 Traitement du resume

La reprise du téléchargement dans le cas où le fichier a déjà été téléchargé (mais pas forcément en entier) est gérée de la façon suivante : lors du téléchargement du fichier, dès qu'une pièce est complète, c'est à dire qu'on a téléchargé les données de tous les blocs qui la compose, on indique que la pièce est téléchargée dans un fichier que l'on qualifie de fichier de configuration. Ce fichier porte le même nom que le fichier torrent auquel on ajoute "-config".

Le fichier de configuration est généré à chaque téléchargement d'un nouveau fichier. C'est un fichier texte qui indique, pour chaque pièce, si celle-ci a déjà été téléchargée ou non auparavant (1 si la pièce a été téléchargée, 0 sinon). Ainsi, lorsque l'on reprend un téléchargement qui a été mis en pause, il suffit de lire ce fichier.

Lors de la lecture du fichier, si on constate qu'une pièce a déjà été entièrement téléchargée, on la récupère dans le fichier de destination, et on l'ajoute au pieceManager. Ainsi, la pièce ne sera pas téléchargée une nouvelle fois, et le pourcentage de téléchargement pourra être calculé en fonction.

2.5 Performances

Expérience MacOS sur un fichier de 2,5 Go	Débit
Notre client (0%) Vs 1 Vuze (100%)	10 Mo/s
Notre client (100%) Vs 1 Vuze (0%)	30 Mo/s
Notre client (100%) Vs 4 aria2c (0%)	Non réalisé
Notre client (0%) Vs 4 aria2c (100%)	Non réalisé

Pour améliorer les performances du système, nous utilisons des Direct Buffer pour lire et écrire sur les sockets des différents peers et nous chargons au maximum la socket en calculant la quantité maximale de pièces que l'on peut demander en une seule fois.
Le système utilise plusieurs threads afin de paralléliser au mieux les tâches lorsque cela est possible. En effet, le système possède un thread par peer, un thread pour le serveur, un thread pour l'écriture dans le fichier et un thread pour le calcul de la sélection des pièces.

2.6 Validation

Pour valider le système, nous avons mis en place des tests unitaires en java nous permettant de valider en terme de logiciel les différentes fonctionnalités.
Pour valider les fonctionnalités sur le réseau, nous n'avons pas pu automatiser les tests avec Aria2c car la prise en main d'Aria2c était difficile et est arrivée trop tardivement dans le projet. Les différentes fonctionnalités ont cependant étaient validées au moyen de tests avec Vuze, QBittorrent et Transmission avec des fichiers plus ou moins lourds sur 3 OS distincts: Ubuntu, Windows et MacOS.

Les principaux bugs résolus sont:

- les selectors et les socketchannels
- la gestion des bytebuffers
- la concurrence des threads

2.7 Bonnes pratiques

2.7.1 Orienté objet

La conception logicielle a été réfléchie en début de projet en privilégiant une conception orientée objet. Celle-ci n'a quasiment pas changé au cours des semaines de projet.
La factorisation a été faite au cours du développement lorsque nous avons remarqué que certaines classes pouvaient hériter de la même classe et ainsi augmenter la factorisation du code.

2.7.2 Documentation

Un Readme permet de présenter le lancement du système et la JavaDoc est présente dans le dossier doc/javadoc/.

2.7.3 Design pattern

Pour gérer les états choke/unchoke et interested/uninterested d'un peer côté leecher, une machine à états est implémenté de la manière suivante :

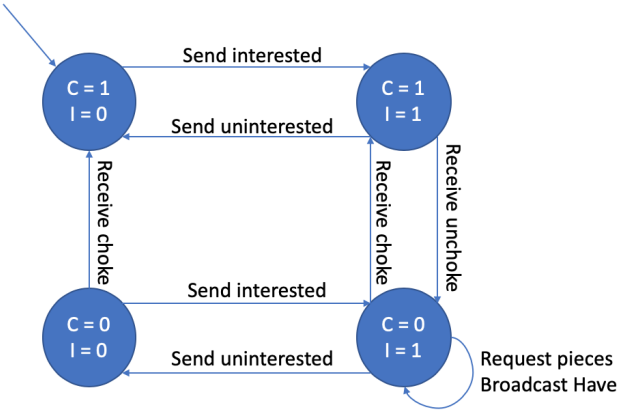

Figure 9: Machine à états choke/unchoke et interested/uninterested

Le patron de conception Observateur est implémenté dans le système entre la classe PieceManager et la classe Server afin de notifier le Server de la réception d'une pièce entière intègre afin que celui-ci envoie un message Have à tous ces clients connectés.

3 Partie 2 : Organisation du projet

3.1 Méthode

Pour ce projet, nous avons suivi la méthode Scrum avec 4 sprints.
Pour cela, nous avons opté pour la méthode de travail suivante :

- Ajouter des cartes sur le workflow GitLab.
- Estimer le poids des tâches par rapport à leur difficulté entre 1 (facile) et 4 (difficile)
- Chacun choisit les cartes qu'il ou elle souhaitait traiter[2]
- Chacun avance sur ses cartes de son côté, avec de temps en temps des mises au point tous ensemble, à distance ou en physique à l'occasion de la messe hebdomadaire du Mardi. [3]
- Nous avons utilisé des branches pour gérer les phases complexes du projet afin que la branche master soit toujours opérationnelle.

3.2 Poids des tâches

Le tableau suivant présente les tâches globales et leur poids suivant les différents sprints :

Tâche Sprint 0	Poids	Tâche Sprint 2	Poids
Lecture de la spec BitTorrent	4	Multiclients	3
Configurer opentracker	1	Message Have	3
Faire communiquer 2 Vuzes	4	créer un fichier de configuration	2
Capture Wireshark	1	Parser fichier pour récupérer les pièces	3
Explication payload des messages	1	Sélection des pièces Séquentiel	3
Diagramme temporel	2	Sélection des pièces Par charge	2
Architecture logicielle	3		
Tâche Sprint 1	**Poids**	**Tâche Sprint 3**	**Poids**
Parser metadata fichier torrent	2	Logger	1
Socket non bloquante	3	Machine à états leecher	3
Construire les messages	3	Contact tracker régulièrement	2
Construire fichier à partir des pièces	2	Charger la socket	2
Dialoguer avec les peers	3	Réception performante des pièces	2
Socket serveur	3	Sélection des pièces Rarest pieces	2
Traiter les messages reçus	3	Rapport de projet	1
Tracker	3		

3.3 Répartition des tâches

En a découlé la répartition des tâches suivante, tout *Sprint* confondu :

- Gloria
  - Dialogue avec le tracker
  - Mise en place de la socket serveur

[2] Les éventuels conflits étaient résolus à l'occasion de duels non-létaux
[3] Bien évidemment, ces mises au point donnaient souvent lieu à des déboggages plein de rebondissements.

  - Gestion de la reprise de téléchargement
  - En conflit permanent avec Windows
- Rémi
  - Traduction des messages en objet Java
  - Gestion des messages côté Seeder
  - Implémentation de fonctionnalités mineures
  - Tentative étalée sur des dizaines et des dizaines d'heures de faire fonctionner l'envoi de *HaveMessage* après réception de pièce.
  - Tentative d'apprendre à coder en Java. [4]
- Audrey
  - A peu près tout le reste

3.4 Planning réel du projet

La figure ci-dessous présente le diagramme de Gantt du projet :

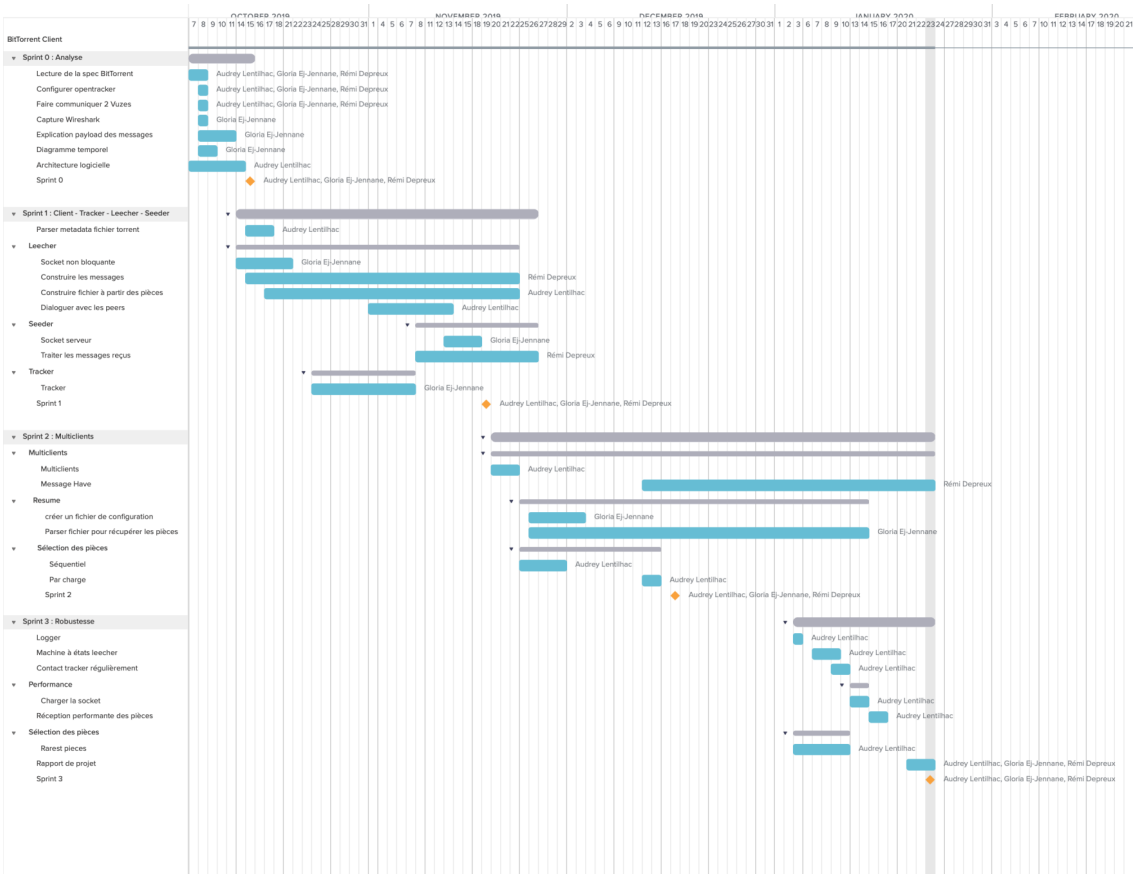

Figure 10: Répartition des tâches et avancement du projet

[4] Pour la petite histoire, il a finalement appris à maîtriser le try/catch : il en est encore émerveillé !

3.5 Difficultés rencontrées

Que serait un projet sans problème d'organisation ? Peu intéressant ! En ce sens, notre projet s'est avéré vraiment intéressant. Pour éviter les redondances, on vous renvoie directement aux paragraphes ci-dessous où vous retrouverez les difficultés que l'on a pu rencontrer.

3.6 Retours personnels

3.6.1 Retour personnel de Rémi Depreux

Ce projet a été l'occasion pour moi de renforcer mes compétences, quelques peu fragiles vu le peu que j'ai fait jusqu'ici, en *Java*. C'était intéressant de décortiquer les spécificités d'un protocole, mais j'avoue que je pensais être davantage confronté à des notions de réseau. J'ai tout de même apprécié tenter de résoudre des problèmes en parcourant attentivement les trames *Wireshark* et en parcourant sans cesse avec l'outil de déboggage notre programme. [5]

Finalement, je regrette de ne pas m'être plus investi dans ce projet. Non pas que je ne me sois pas investi : je ne l'ai juste pas été suffisamment à certains moments où d'autres avançaient. Forcément cela a donné lieu à un certain déphasage et donc des lacunes de compréhension qui accentuaient la difficulté d'appréhender de façon autonome certaines problématiques au rythme du groupe. Bref, à partir d'un certain moment il valait mieux que l'on m'indique quoi faire car je manquais de compréhension globale pour prendre des initiatives.

3.6.2 Retour personnel de Gloria Ej-jennane

Le fait de travailler sur un projet en partant "from scratch" est très satisfaisant je trouve. La plupart des projets à l'Ensimag jusqu'ici ont été des projets où l'on avait toujours une base de code. Là au moins, on a une vraie réflexion à avoir sur l'architecture de l'application. Ce projet m'a permis d'utiliser mes compétences en Java, mais aussi de les améliorer, surtout en ce qui concerne la gestion des sockets. C'est une notion que je n'avais jamais vue auparavant. Ce projet m'aura également appris que travailler sur Windows, ce n'est pas forcément une bonne idée.

Concernant la répartition du travail, celle-ci a parfois été inégale, pour la simple et bonne raison que nous n'avançons pas tous à la même vitesse. Pour notre groupe, la difficulté principale a été de trouver un rythme qui convient à tout le monde. Malgré cette difficulté, l'ambiance dans le groupe est restée saine et agréable.

3.6.3 Retour personnel de Audrey Lentilhac

Pour ma part, j'ai appris à mieux utiliser les threads. Le plus difficile a été de faire en sorte que le projet soit robuste. Ce n'est pas facile de s'arrêter quand on avance sur les différentes tâches et on rentre dans un cercle vicieux où il y en a un qui avance et les autres sont perdus sur ce qui se passe.

Au niveau du projet, je pense que faire ce projet en C++ et avec des sockets UDP pourrait être intéressant. Je pense aussi qu'ajouter un rendu du diagramme de classes prévisionnel au Sprint 0 permettrait à de nombreux groupes de ne pas avoir à casser leur code suivant les sprints.

[5] J'aurais probablement encore plus apprécié si j'avais réussi à résoudre les problèmes que j'avais.

4 Conclusion

Ce projet quelque peu tumultueux, à l'image de notre relation avec les clients Vuze ou QBitTorrent, nous aura mené tant bien que mal au bout. À quelques détails près, [6] notre client est fonctionnel avec un débit plutôt satisfaisant et notre organisation (bien qu'imparfaite) de travail a fait que nous n'avons pas été confronté au phénomène bien connu chez les Ensimag : passer les quelques derniers jours du projet à coder [7]. En tant qu'étudiant avec plusieurs années d'expérience, je trouve cela déjà pas si mal !

Au revoir *ByteBuffers* en tout genre, *byte arrays*, *Sockets non bloquantes*, *Selector* bien-aimé et ces adorables *Threads* qui sont si détestables quand il s'agit de débugger.[8] Au revoir également à *opentracker*. Ce fût un plaisir d'écrire à peu près 13 265 fois chacun [9]

```
1      $ ./opentracker -i 127.0.0.1 -p 6969
```

Wireshark, tu as su être un ami en toutes circonstances. Parfois tes propos étaient crus. On se serait bien passé de tes **TCP FULL WINDOW**, ou des **CONTINUATION DATA**. Mais en définitive, tu ne faisais que nous dire la vérité. Et pour ça, on te remercie !

IntelliJ, tu es un outil bien sympa même si tu es propriétaire avec ta version *ultimate* disponible gratuitement en tant qu'étudiant que nous ne serons bientôt plus. Simplement, ton outil pour générer les diagrammes *UML* est vraiment pas fou on va dire.

Du reste, adieu Vuze et QBitTorrent. La plaie est encore trop ouverte pour espérer renouer le dialogue un jour. Audrey ne s'est toujours pas remise de votre attitude inadmissible. Donc on va s'en tenir à *µTorrent*.

C'est sûrement avec une larme [10] que s'achève notre dernier projet à l'ENSIMAG. Rythmé par des difficultés : ne serait-ce que pour faire en sorte de re-faire fonctionner le client après un pull plein de surprises, ponctué de quantité démesurée de travail pour un aspect minime qui s'avérera être obsolète d'ici la fin du projet.. En tous cas, c'était un bon projet. On aura appris des choses, et on ne verra plus jamais le téléchargement **légal** de Torrent de la même façon. [11].

PS : Après relectures collectives, nous nous sommes rendus compte que cette conclusion n'était pas très joyeuse, mais en vrai nous sommes contents du projet. Si c'était à refaire, nous modifierions certaines choses, mais nous restons satisfaits de notre résultat !

[6] Aria, désolé
[7] Du moins, pas la personne qui est entrain d'écrire ces lignes
[8] Il aurait été trop simple de pouvoir passer d'un *thread* à l'autre facilement pendant le déboggage
[9] Automatiser, kesako ?
[10] De joie ou de tristesse, ça reste à déterminer
[11] Vous nous auriez vu, émerveillés face au téléchargement des différentes pièces de la dernière saison de *Grey's Anatomy*, on était sur le point d'ouvrir *Wireshark*

5 Annexes

A Résumé des fonctionnalités

Fonctionnalité	Avancement
Multi-clients	OK
Resume calculé à partir du fichier original	OK
Sélection de pièces : Conditions Sprint 2 respectées	OK
Sélection de pièces : Stratégie	Rarest First
Plusieurs messages bittorrent au sein d'un segment TCP	OK
Designs patterns	Observer / State Machine
Message HAVE, CANCEL, KEEPALIVE	HAVE
Tests automatisés	KO
Clients supportés	Vuze, QBittorrent, Transmission
Scénario multi-leecher supporté	OK
Taille de pièce (32K et +)	OK
Tracker contacté à intervalle régulier	OK

B Package message

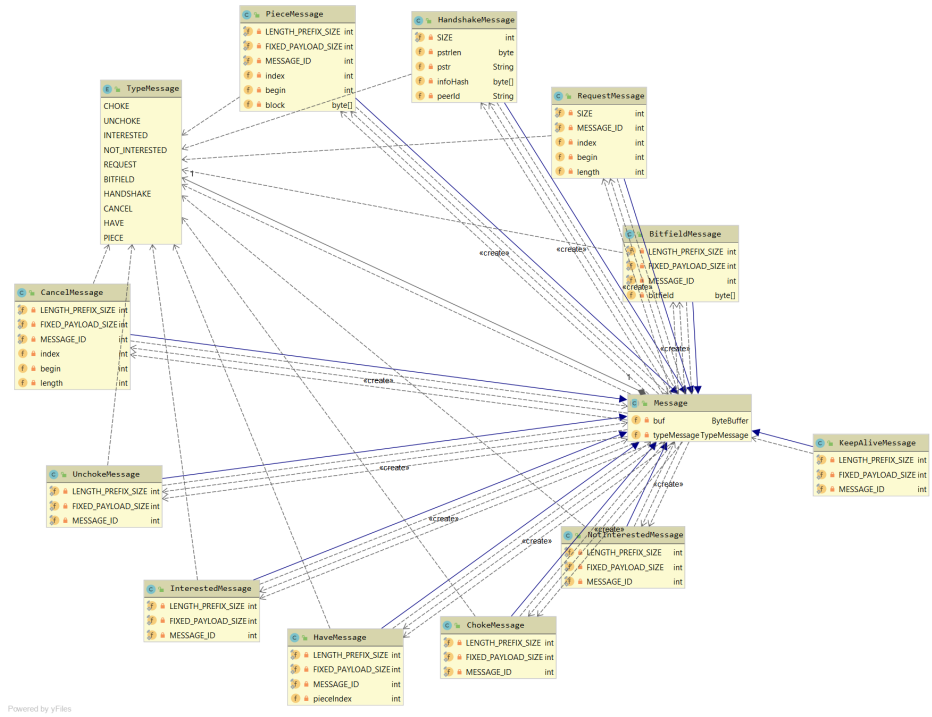

Figure 11: Diagramme de classes sur le package message

C Package peers

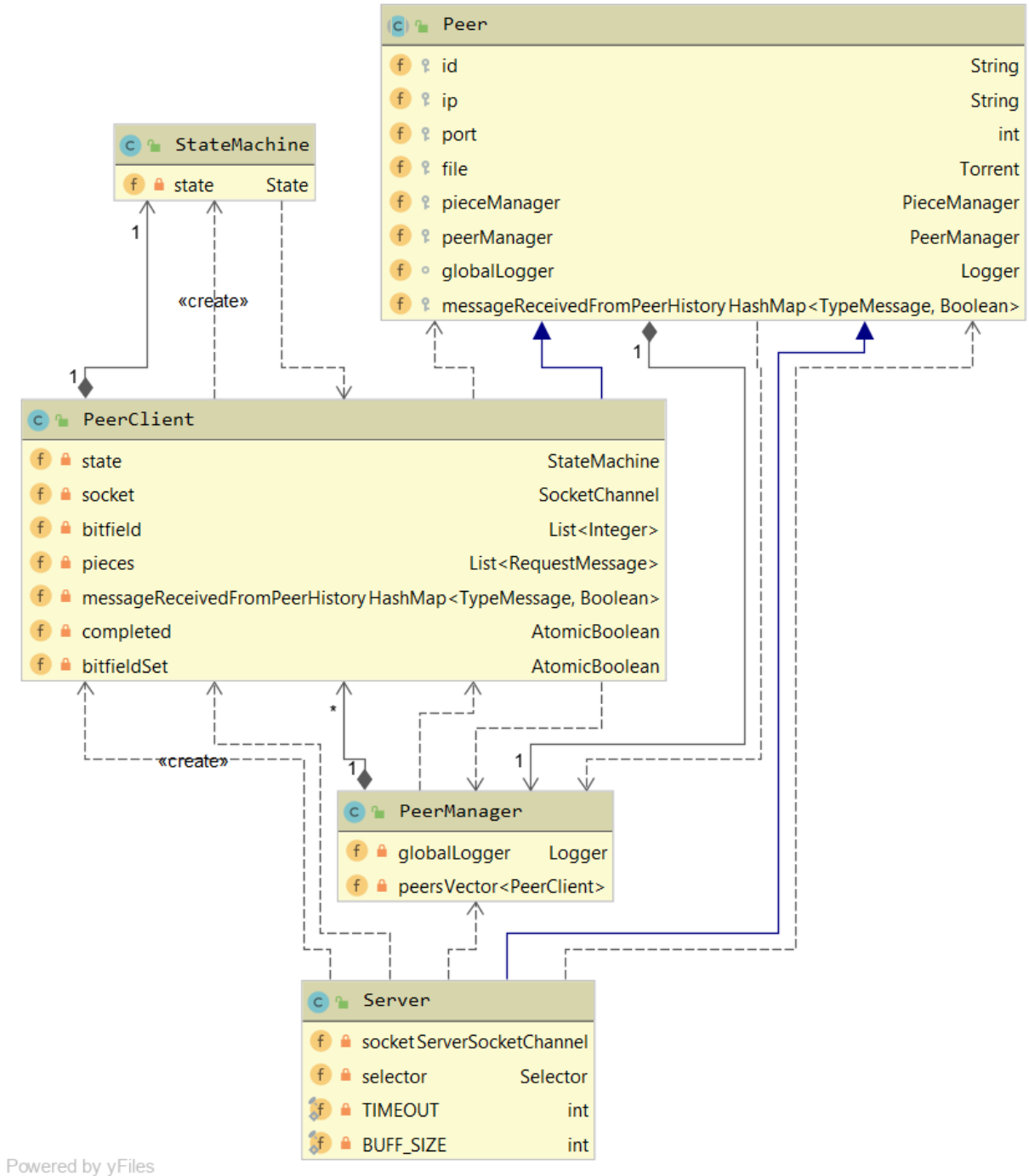

Figure 12: Diagramme de classes sur le package peers

D Package pieces

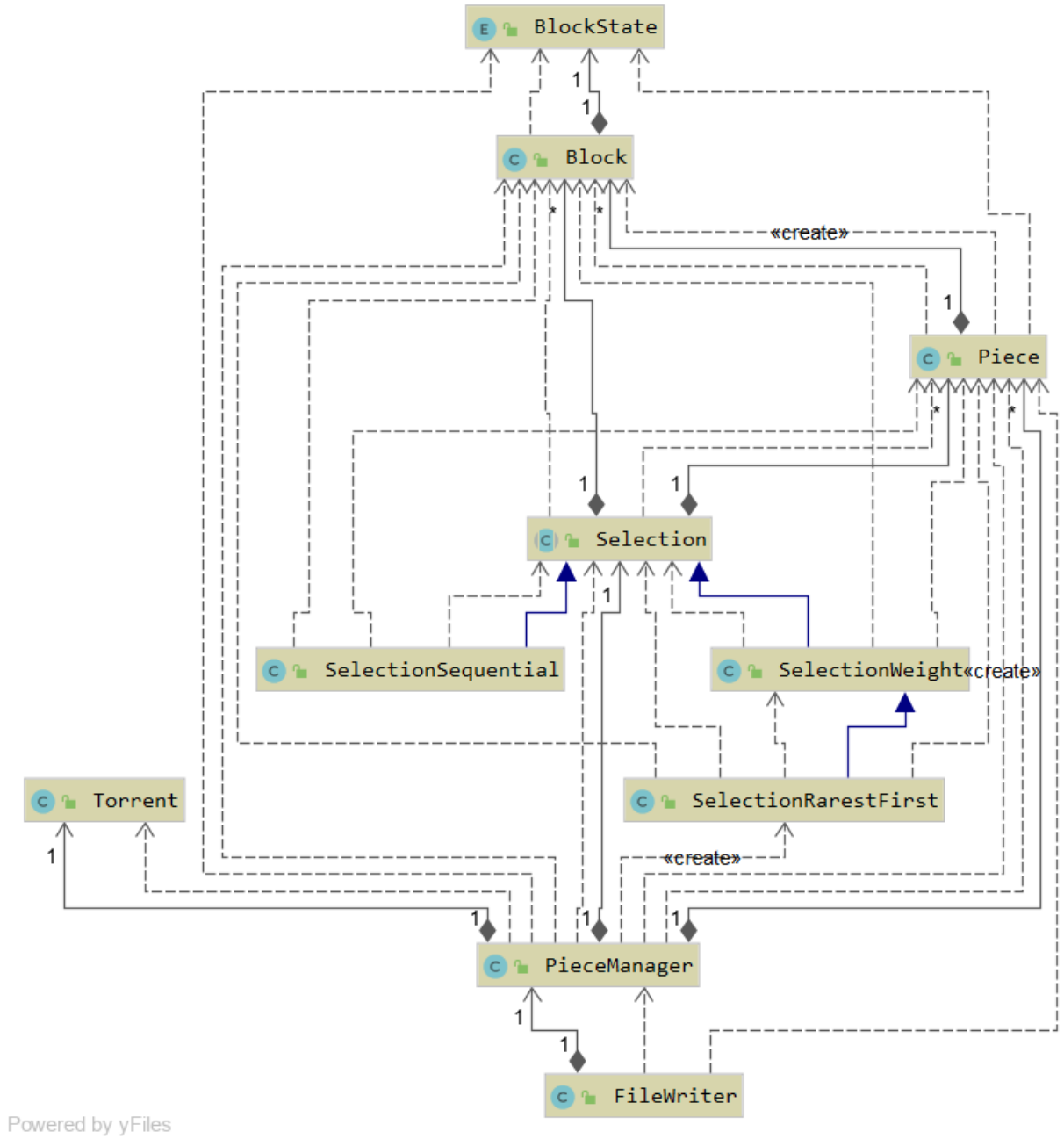

Figure 13: Diagramme de classes sur le package pieces

E Package tracker

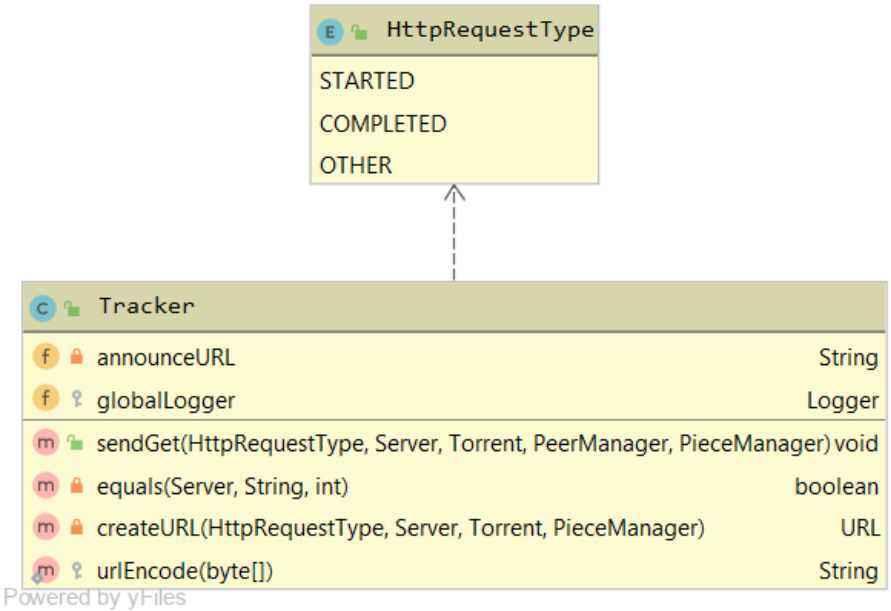

Figure 14: Diagramme de classes sur le package tracker

F Classe BitTorrentClient

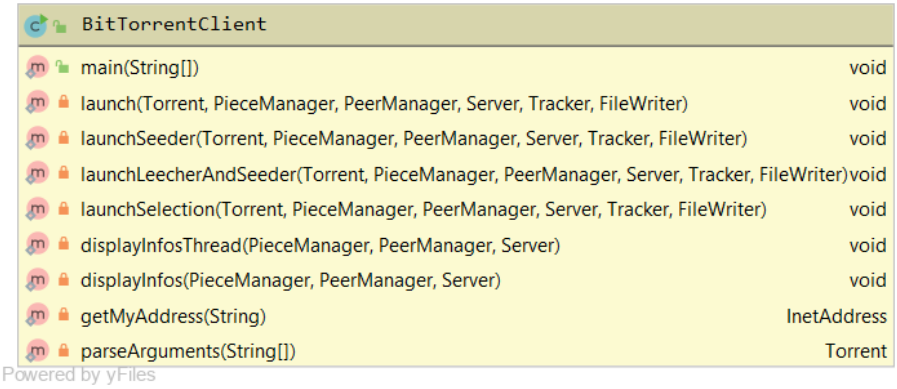

Figure 15: Classe BitTorrentClient