Rapport de Projet Réseaux

Réalisation d'un Client Bittorent en Java

Équipe 5 : Youssef Boulkhir, Asaad Belarbi, Pierre Cheylus

Phelma 3A 2021-2022
Filière Systèmes Embarqués et Objets Connectés

Superviseur école :
Olivier Alphand
olivier.alphand@univ-grenoble-alpes.fr

Table des matières

1 Introduction

L'bjectif de ce projet était de réaliser un client Java implémentant le protocole Bittorent selon sa spécification 1.0 et permettant ainsi de faire des transferts de fichiers pair à pair pour des torrents mono-fichier.
Dans ce rapport nous présenterons d'abord les fonctionnalités implémentées dans notre client en détaillant quelques points clefs de notre implémentation ainsi que l'architecture objet que nous avons utilisée. Ensuite nous présenterons les performances réseaux en se basant sur un scénario test de référence et enfin nous présenterons notre méthode de tests et validation du fonctionnement du client ainsi que notre organisation du travail au cours de ce projet.

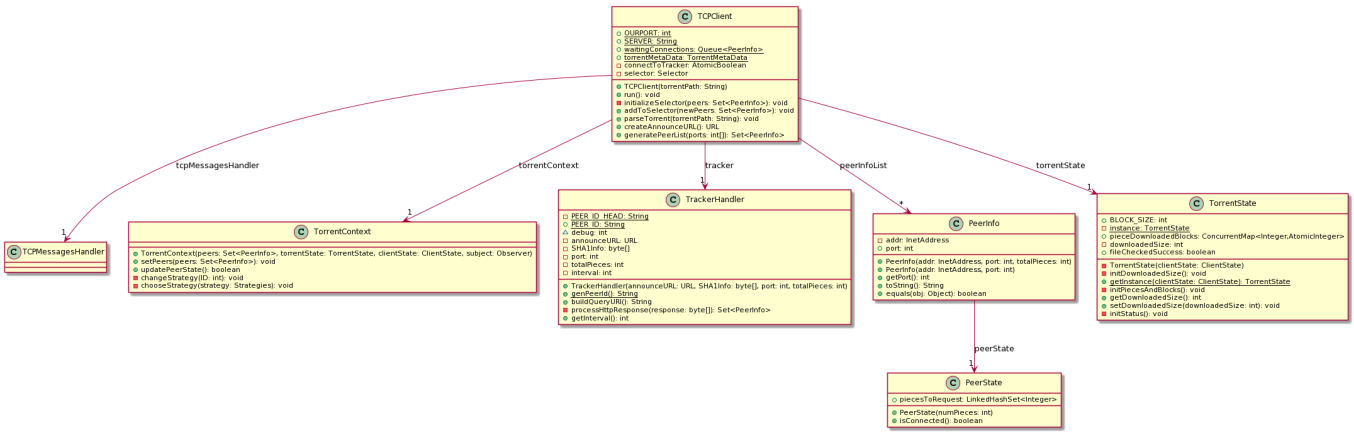
2 Implémentation

2.1 Tableau des fonctionnalités

Sprint	Fonctionnalité	avancement (précision)
Sprint 1	Tracker (HTTP Req URLEncodé/Réponse debéncodée)	OK
Leecher 0%	Socket bloquante + java.io	OK
Leecher 0%	Ecriture pièce sur disque	au fur et à mesure
Leecher 0%	Plusieurs messages bittorent au sein d'un segment TCP	OK
Leecher 0%	Message inconnu traîté	KEEPALIVE
Leecher 0%	Sélection de pièces	RARESTFIRST,ENDGAME,RANDOM
Seeder 100%	Bitfield créé en fonction du test	OK
Seeder 100%	Gestion pièces et blocs	OK
	Clients supportés	Aria2C, Vuze, Qbittorent ...
Sprint 2	Support Multi-Leecher	OK
	Support Multi-Seeder	OK
	Bytebuffer	1 par message
	Sélection de pièces	Strategies
	Evaluation de performances	local, distribué
	Resume calculé à partir du fichier partiel	OK
	Scénario multi-Leecher supporté	OK
	Gestion concurrente	Selector
	Scénario Leech et Seed en même temps	OK
Sprint 3	Design patterns	Observer, Stratégie
	Sélection de pièces	Random, RarestFirst, ENDGAME
	Message supporté	tous sauf PORT
	Clean Code, Factorisation, Archi. Objet, . . .	Ok
Avancé	Test automatisé	Scripts
	Calcul débit, barre progression	Calcul débit, progression
	Compilation automatique	Maven
	Taille de pièce (32K et plus)	Toutes les tailles
	Extensions	Aucune
	Tracker contacté à intervalle régulier	OK
	Gestion des déconnexions des peers	OK à tous instants
	Tracker contacté à intervalle régulier	OK
	Changement dynamique des strategies	OK
	Selection des peers	Supression des peers inutiles
	Timeout pour les peers inactifs	Envoie KEEPALIVE et Timeout

2.2 Architecture

2.2.1 UML



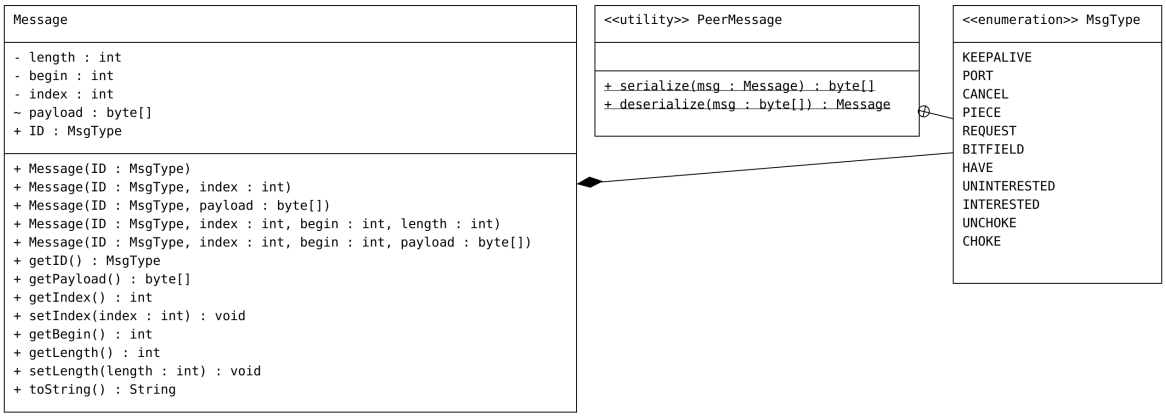
Un diagramme UML synthéthique de l'architecture de notre programme Java peut être trouvé ci-dessus. En effet, la classe TCPClient est la classe principale de ce projet. Elle comprend entre autre les informations de connexion TCP, les métadonnées du torrent, les connexions en attentes et le selector. Elle comprend également le TCPMessagesHandler qui s'occupe du traitement des messages TCP envoyés et reçus, le TrackerHandler permettant la communication avec le tracker afin de générer la liste des pairs sous forme de PeerInfo, le TorrentState et le TorrentContext qui permettent de garder les informations générales sur le torrent en cours tel que les pièces présentes ou manquantes, celles qui ont été demandées et celles qui sont en cours de téléchargement, etc.
Nous avons utilisé quelques design patterns utiles pour ce projet, tels que le Singleton pour toutes les classes dont on a besoin d'une instance unique, Stratégie pour le choix d'algorithme de sélection des pièces, et Observer pour notifier le contexte du patron stratégie des changements sur les informations et les connexions des pairs, par exemple à la réception d'un message HAVE.

2.2.2 Fonctionnement

Au démarrage du client, le fichier torrent est parsé par le TorrentFileHandler, qui génère un TorrentMetaData contenant toutes les données nécessaires pour le fonctionnement du client.
Ensuite, grâce à l'URL d'announce extraite des métadonnées, on peut contacter une première fois le Tracker via le TrackerHandler pour générer la liste des PeerInfo contenant les addresses et les ports des autres clients sur le torrent. L'état de chaque pair est stocké en mémoire dans un PeerState (choked, unchoke, interested, etc.).
La gestion des messages Bittorrent se fait via le TCPMessagesHandler, qui lit les messages recues, les envoies à NIODownloadHandler où se trouve notre machine à état, afin de mettre à jour les informations dans nos structures de données concernant les pairs. On gère la stratégie de téléchargement grâce au TorrentContext : celui-ci choisit une stratégie de séléction des pièces à télécharger selon l'état actuel des pairs et les pièces que l'on possède déjà, entre autre.
L'écriture sur le disque et la vérification des pièces se fait via le LocalFileHandler, qui génère également le Bitfield client à envoyer aux nouveaux pairs.

2.3 Traitement de messages

La géneration et lecture des message se fait à travers la classe PeerMessage, elle renvoie une Classe Message qui contient tous les informations du message reçu ou à envoyé, ID, payload, index ....



La machine à état se trouvant dans la classe NIODownloadHandler, permet de faire les traitement necessaire après chaque message envoyé ou reçu, provenant dans la classe TCPMessageHandler.

2.4 Algorithme de sélection des pièces

Nous avons implémenté trois stratégies / algorithmes de sélection des pièces à télécharger :
— Rarest First
— Random Piece
— EndGame

Lors du démarrage du client, la stratégie par défaut est le Rarest First. On bascule ensuite en Random Piece une fois que tous les pairs possèdent toutes les pièces qui nous manquent, puis on termine en EndGame une fois qu'il reste trois pièces à télécharger.

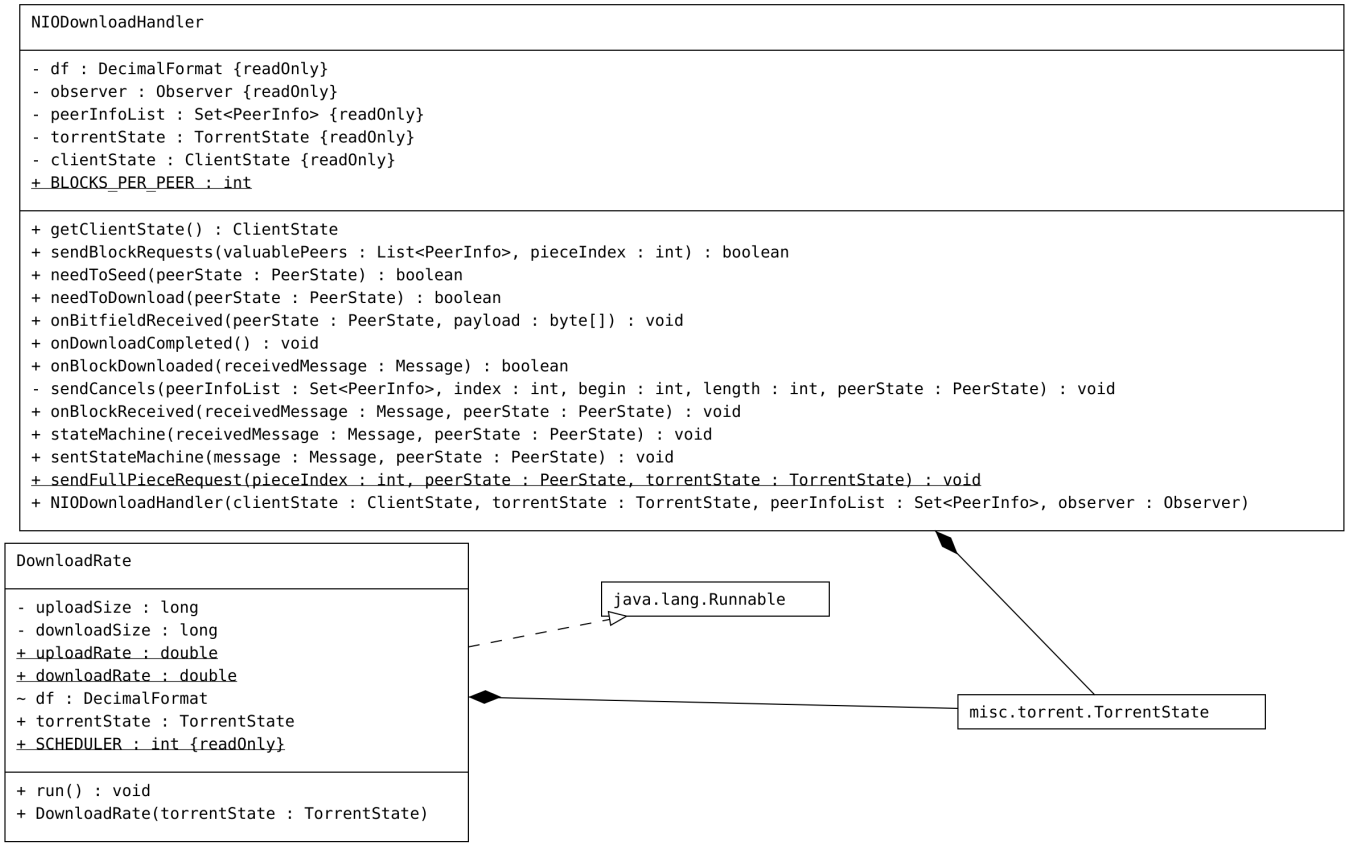
En RarestFirst, on commence tout d'abord par utiliser le bitfield local afin d'obtenir une liste des pièces qui nous intéressent. Ensuite, à partir des bitfields des pairs reçus lors de l'établissement de la connexion, on établit les occurences de chaque pièce afin de définir les pièces les plus rares. On télécharge ainsi toutes les pièces ayant la plus faible occurence, en choisissant des pairs aléatoires possédants la pièce souhaitée afin d'éviter la congestion. On continue ainsi jusqu'à ce que tous les pairs possèdent toutes les pièces qui nous manquent.

Une fois passé en Random Piece, on sélectionne aléatoirement une pièce et des pairs afin de continuer le téléchargement jusqu'à ce qu'il ne reste que 3 pièces à télécharger.
Enfin, en EndGame, on va demander à tous les pairs toutes les pièces et blocs manquants. Lorsqu'on reçoit un block, en envoie des cancel à tous les autres Peers.

2.5 Fonctionnalités

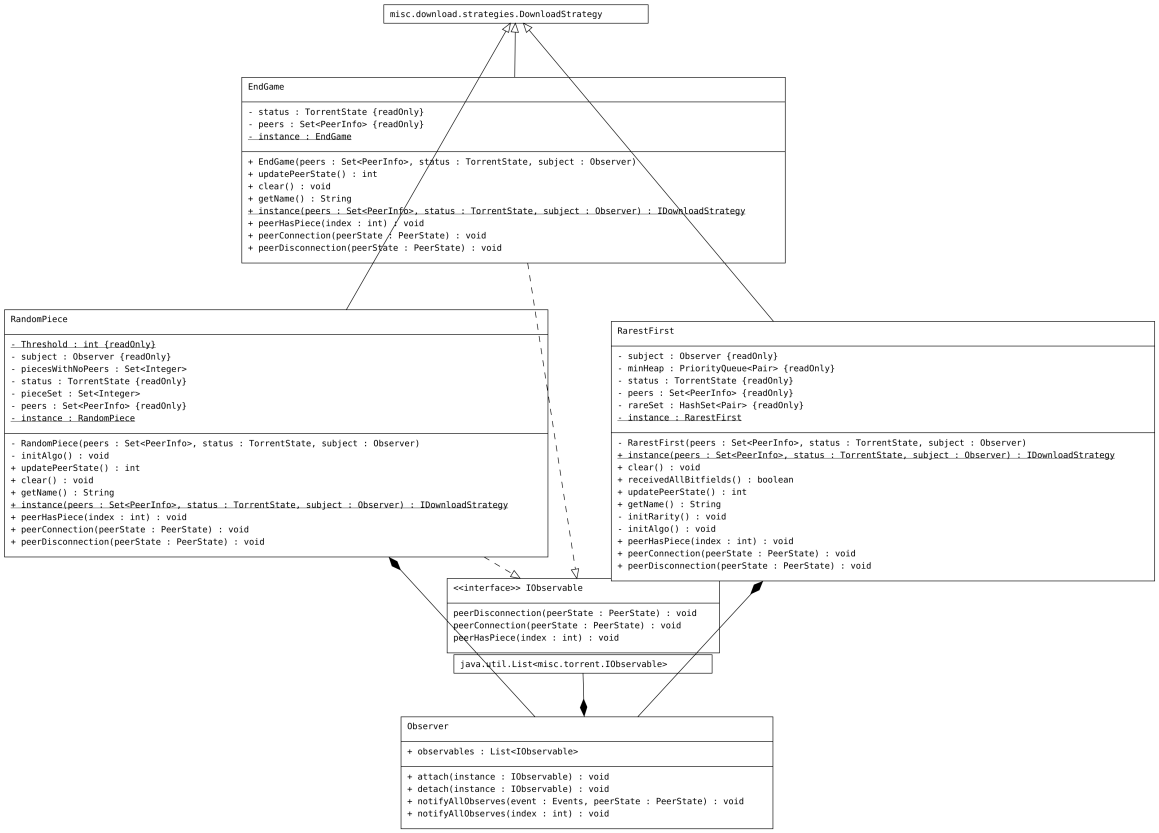
2.5.1 Calcul du débit

On lance un thread, dans la classe DownloadRate, qui calcule le debit du télechargement et de l'Upload après un délai d'une seconde. Pour cela, on utilise les données (nombre de bytes downloaded/uploaded) provenants de TorrentState, et qui sont mise à jour par NIODownloadHandler à chaque fois qu'on reçoit ou envoie une pièce.



2.5.2 Gestion des connecions, déconnections

Les connections (réception des Bitfields) et déconnections de peers sont gérés par notre application, par le bias d'un observer pattern, qui notifie toutes les classes concernées. De plus, les messages desquels le Peer déconnecté devait s'en occuper sont re-transmis à d'autres Peers. La méthode chargée de cette gestion est +removePeer(PeerInfo peerInfo) dans la classe misc.download.TCPMessageHandler.



2.5.3 Analyse de l'utilité des Peers

Lors de la connection avec chaque Peers, ainsi que lors de la fin du télechargement, notre programme analyse le bitfield du Peer en question, et détermine si il y'a un besoin de maintenir la connection avec lui. Deux methodes sont utilisées pour cela, misc.download.NIODownloadHandlerneedToSeed et misc.download.NIODownloadHandlerneedToDownload. Suite à ceci, celon les cas on peut procéder à l'enovie d'un Choke, ou Not interested, ou terminer la connection avec le peer.

2.5.4 Timeout des peers

Chaque fois qu'on reçoit une message de la part d'un Peer, on stocke le temps de la réception de ce message. De ce fait, si un Peer ne répond pas pendant une longue durée (10 secondes) on lui envoie un message KeepAlive, et si plus de 15 secondes se sont écoulées sans avoir de réponse de la part du Peer, on coupe la connection avec celui-ci. La méthode concernée est misc.download.TCPMessagesHandlerendOfStream

2.5.5 Fast et Slow mode

Du fait de certains problèmes que nous avons rencontrés, notamment le bloquage de notre client par un Peer qui n'arrive pas à gerer un nombre de réquetes/bloques, nous avons élaboré deux mode

de télechargement

— Slow mode : le nombre de réquetes envoyés est limité à 5, de même pour le nombre de réquetes reçues (cas ou on est Seeder).

— Fast Mode : on peut envoyer et recevoir un plus grands nombre de requètes, cependant, certains clients comme vuze sont sucéptible de se déconnecter.

2.5.6 Contacte du Tracker à intervalle réguliers

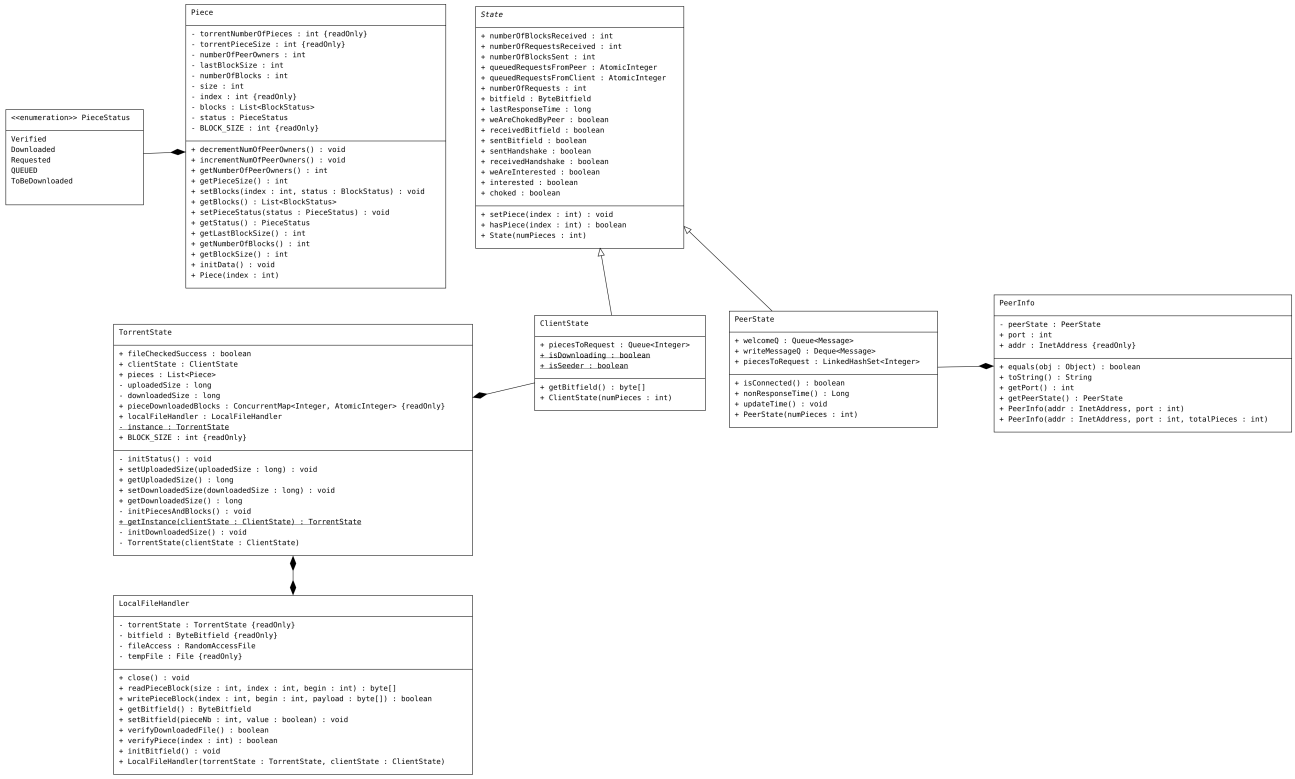
Le tracker est contacté à un intervalle réguliers. La taille de l'intervalle est donnée en réponse de la part du Tracker. Cela nous permet de chercher si il y'a d'autres Peer qui sont annoncés par le tracker.

2.5.7 Vérifications des pièces au debut/fin du télechargement

à la fin et au debut du télechargement, on verifie l'intégrité des pieces télechargées, avec les Hash présents dans le meta info du torrent. Cela nous permet de re-telecharger des pieces dans le cas de lors non-validité.

2.5.8 Suivi de l'état des peers et du torrent

Plusieurs informations cocnernant l'état des peers, ainsi que ceux du torrent télchargées sont stoqués et mises à jour au fur et à mesures du télechargement. On peut connaitre à chaque instant des informations tels que, le nombre de pieces envoyés et reçues de la part de chaque Peer, l'état du peers (interested, choked...)..., le débit du télachargement du torrent...



2.6 Structures de données

Pour chaque Peer, nous avons utilisé deux files circulaires (writeMessage, WelcomeQ) pour stoquer les messages que nous devons lui envoyer lors de la prochaine opération Write. Le choix d'une Deque, relève du fait que nous avons jugés que certains messages sont plus prioritaires que d'autres, notamment les Choke/Unchoke, Have .. qui doivent êtres envoyés avant les réquetes/Pieces.
Au niveau du rarest First, nous avons utilisé un MinHeap, vu que l'on a besoin de trouver les m pieces ayant l'occurence minimal. Cela est fait à l'aide d'un MinHeap en une compléxité temporelle de O(log(N)), ce qui est plus efficace que de faire le trie des occurences à chaque fois O(Nlog(N)).
En ce qui concerne la liste des Peer, nous avons optés pour un Set au lieu d'une liste, afin d'éviter d'avoir des doublons au niveau des Peers.

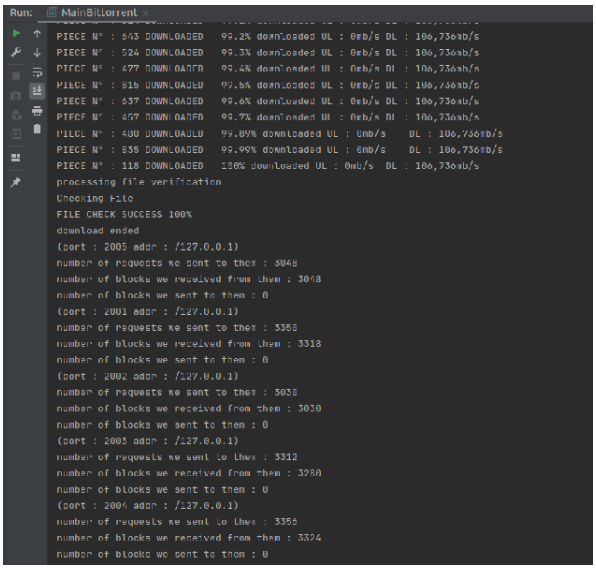
3 Performances

3.1 Scénario de Test

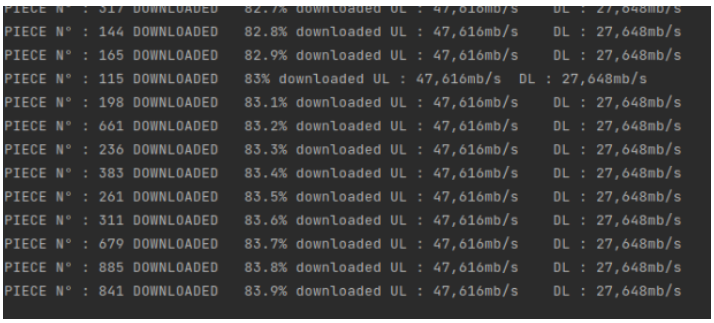
Nous avons fait des tests sur notre machine local, ainsi qu'en distribué sur les machines de l'Ensimag. Pour le calcul du debit, notre client est capable de le faire, et les valeurs obtenues étaient cohérente avec ceux affichées par vuze, aria et wireshark. ci dessous quelques exemples de résultas

obtenus par notre application.

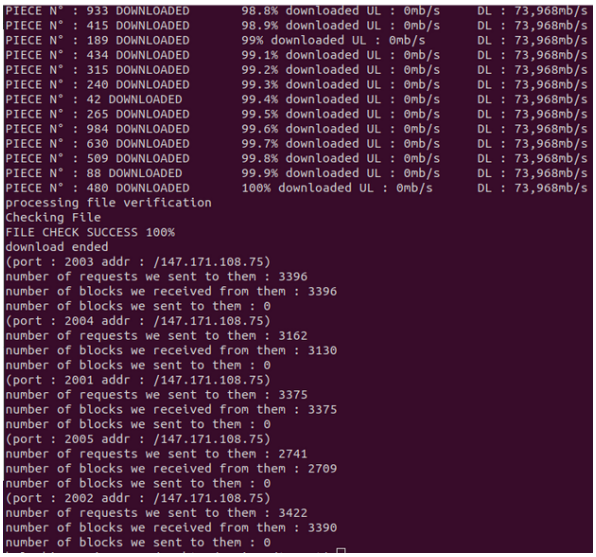
Scénario : en local avec notre client en mode leecher et 5 aria en mode Seeder.



Scénario : en local avec notre client en mode leecher et 5 aria en mode seeder+leecher, sans communication entre eux.



Scénario : en distribué, notre client leecher et 5 aria en mode seeder.



3.2 Tableau de performances

3.2.1 En distribué, sur les machines D259 et D260, torrent de taille 3GO

Notre client	vs	Mode	Debit
seeder	Aria leecher	fast	120 Mb/s
leecher	Vuze seeder	slow	10 Mb/s
leecher	Aria seecher	fast	60 Mb/s
leecher	5 Aria Seeder	fast	80 Mb/s

3.2.2 En local : torrent de taille 250 MB

Notre client	vs	Mode	Debit
seeder	Aria2c leecher	low	15 Mb/s
seeder	Aria leecher	fast	100 Mb/s
seeder	Vuze leecher	slow	6 Mb/s
leecher	Aria Seeder	fast	100 Mb/s
leecher	5 Aria Seeder	fast	110 Mb/s
leecher	5 Aria Seeder	slow	6 Mb/s
leecher	5 Aria leecher+seeder	fast	UL :12 Mb/s DL :5 Mb/s
leecher	5 Aria leecher+seeder	fast	UL :50 Mb/s DL :30 Mb/s
seeder	4 Aria leecher	fast	UL : 300 Mb/S

4 Tests & Validation

4.1 Scripts

Pour tester notre application, nous avons implementés quelques scripts, ainsi que deux programmes en GOlang, l'un permettant de génerer un fichier avec une taille et des nombres de pieces définies, et l'autre permettant de génerer le fichier meta info .torrent. Ensuite nous avons écrits des scripts en bash pour automatiser tous le processus.

4.2 Problèmes rencontrées

Parmi les problèmes rencontrés :

— Selector : Au niveau du selector, dans certains cas, des bloquage entre le OP_READ et OP_WRITE bloquaient le telechargement.

— Tracker : Dans cetains cas, le tracker rendait en réponse des ports invalides ou negatives

— Blacklists : Quand on traite plusieurs messages à la fois, il se peut que Vuze nous bloque, et que Aria se deconnecte. Après avoir cherché la cause de ces incidents, il s'est avéré qu'on fait, tous ces application utilise JAVA, et dans certains moments la mémoire de la JVM se sature, et donc aria2c est automatiquement fermé par le système d'éxploitation, tandis que Vuze bloque notre peer. Pour remedier à ce problème, nous avons mis en place deux modes de telechargement : Slow et fast.

— Usage de mémoire : L'usage de la mémoire de notre application, pour un torrent de 250mb, s'élève jusqu'à 6GO, ceci est probablement du à la manière dont on enregistre tous les les informations sur toutes les pièces dans des HashMap.

5 Organisation du travail

5.1 Organisation générale

Nous avons décidé de séparer le travail dès le début en le découpant en tâches distinctes et en se répartissant les tâches au fur et à mesure des différents sprints.
Des exemples de tâches peuvent être : Extraction des métadonnées du fichier Torrent, communication avec le tracker (avec intervalle régulier), gestion des messages, gestion des connexions TCP, mise en place des algorithmes de sélection des pièces, mise en place des structures de données pour la gestion des informations des pairs, mise en place de la machine à états, écriture locale sur le disque, tests Java, tests réseaux, etc.
Cela a été assez utile pour que chacun puisse avancer indépendemment des autres mais cela a aussi pu rendre la tâche finale plus compliquée lorsqu'il a fallu agencer les différents éléments les uns avec les autres. En effet, un travail commun afin de se mettre d'accord plus facilement sur l'architecture objet à mettre en place pour faire fonctionner le client aurait pu permettre moins de confusion lorsque l'on a cherché à articuler les différentes parties les unes avec les autres.

5.2 Conclusions personnelles

5.2.1 Youssef Boulkhir

Le projet est à comprendre comme il faut avant de commencer à coder ; le principe du protocole mérite bien d'être bien compris. Le partage des tâches dans ce projet était d'une importance inouï puisque les parties sont liées :par conséquent, j'ai appris comment voir le code des autres et le comprendre avant de commencer une tâche. Ceci étant, le travail personnel est aussi important et pourtant l'entraide est inévitable

5.2.2 Asaad Belarbi

Ce projet m'a permis d'approfondir mes notions sur la programmation réseaux et orienté objet. En travaillant en groupe, nous avons créé une application, en commençant par la conception de son architecture, la compréhension du protocole Bittorrent, le développement des divers fonctionnalités et les tests de validité. Ce que j'ai retenu, c'est que l'étape de conception est la plus importante, surtout quand on travaille sur une application complexe, car si le code n'est pas bien structuré dès le début, on trouve beaucoup de difficultés a le re structurer après, et on se retrouve avec plusieurs dépendances entre les classes. De plus, les patrons de conceptions sont très intéressants et doivent être utilisés en priorité.

5.2.3 Pierre Cheylus

Le protocole réseau Bittorent a toujours été pour moi un symbole de grande liberté de l'Internet actuel de par sa spécificité de ne pas télécharger depuis une entité centralisée. J'ai trouvé très intéressant de pouvoir apprendre et comprendre les ficelles d'un protocole de téléchargement pair-à-pair tel que celui-ci parce que cela m'a permis de réaliser la complexité que pouvait avoir un mécanisme qui semble au premier abord assez simple. J'ai trouvé très intéressant la façon que ce protocole a de lier l'encodage avec le BEncode, la communication avec des serveurs Web pour le tracking des torrents, la communication TCP avec les pairs et toute la gestion des pièces et des différents messages entre autres.
Enfin, j'ai trouvé intéressant comme pour le projet GL de développer un tel projet en communicant et en s'organisant en équipe dans une logique de répartition des tâches afin de diviser la charge de travail. J'ai trouvé que la communication entre les différentes parties du projet a été assez laborieuse et que l'on aurait pu gagner en clarté avec une meilleure communication, cependant je suis satisfait du résultat final.