Projet BitTorrent

Auteurs : **Da Silva Andrade David**
Magnin Antoine

Classe : **ITI-Jour 3ème année 2014-2015**

1 Table des matières

2 L'objectif du projet

Ce projet a pour but l'implémentation d'un client BitTorrent simplifié, il doit simplement se connecter à un tracker (protocole THP) dont l'adresse est fournie dans un fichier « .torrent ». Une fois l'adresse de ce tracker extraite il faut effectuer une requête afin d'obtenir les IP et les ports des peers qui ont les pièces du torrent. Une fois ces IP stockés il faut démarrer la communication avec les peers, pour ce faire on utilise le protocole PWP, celui-ci permettra de savoir qu'elles peers possèdent quels pièces, ainsi que la demande de ces dernières. Le client doit pouvoir garder une trace de chaque pièce que possèdent les peers, ainsi qu'effectuer des logs de ceux-ci dans un format ASCII et CSV.

Il n'est pas obligatoire que le client soit capable de récupérer la totalité d'un fichier partagé. Elle devra cependant déterminer sur quels peers se trouve quels morceaux de fichiers, récupérer les deux premiers morceaux (pieces) du fichier et vérifier que la fonction de hachage SHA-1 sur les morceaux récupérés correspond bien aux résultats indiqués dans le fichier des métadonnées .torrent.

Notre client est un *bad leecher*, c'est-à-dire qu'il ne partagera pas les pièces qu'il a en sa possession.

3 Approche théorique du protocole

3.1 La terminologie

Peer : un pair est un nœud dans le réseau qui participe au partage de fichier. Il peut agir simultanément en tant que serveur ou en tant que client avec les autres nœuds du réseau.

Neighboring peers : Pairs à qui un client a un point actif à point TCP.

Client : Un client est un agent utilisateur qui agit comme un pair pour le compte d'un utilisateur.

Torrent : Un torrent est le terme pour le fichier (un seul fichier torrent) ou un groupe de fichiers (multifichiers torrent) que le client télécharge.

Swarm : Un réseau de pairs qui opèrent activement sur un torrent donné.

Seeder : Un pair qui a une copie complète d'un torrent.

Tracker : Un tracker est un serveur centralisé qui contient des informations sur un ou plusieurs torrents et des réseaux de pairs associés. Il fonctionne comme une passerelle pour les pairs dans un réseau de pairs.

Metainfo file : Un fichier texte qui contient des informations sur le torrent, par exemple, l'URL du tracker. Il a généralement le .torrent d'extension.

Peer ID: Une chaîne de 20 octets qui identifie le poste. Comment l'ID des pairs est obtenu en dehors de la portée de ce document, mais un poste doit faire en sorte que l'ID de pairs qu'il utilise a une très forte probabilité d'être unique dans le réseau de pairs.

Info hash: Un hash SHA1 qui identifie de manière unique le torrent. Elle est calculée à partir de données dans le fichier de métadonnées.

3.2 L'encodage

Dans BTP / 1.0 le fichier métadonnées et toutes les réponses du Tracker sont encodés dans le format de bencoding. Le format spécifie deux types scalaires (entiers et chaînes) et deux types de composés (listes et dictionnaires).

```
dictionary = "d" 1*(string anytype) "e" ; non-empty dictionary
list       = "l" 1*anytype "e"          ; non-empty list
integer    = "i" signumber "e"
string     = number ":" <number long sequence of any CHAR>
anytype    = dictionary / list / integer / string
signumber  = "-" number / number
number     = 1*DIGIT
CHAR       = %x00-FF                    ; any 8-bit character
DIGIT      = "0" / "1" / "2" / "3" / "4" /
             "5" / "6" / "7" / "8" / "9"
```

Les entiers sont codés en préfixant une chaîne contenant la représentation en base dix du nombre entier avec la lettre "i" et postfixant avec la lettre "e". Par exemple l'entier 123 est codé comme i123e.

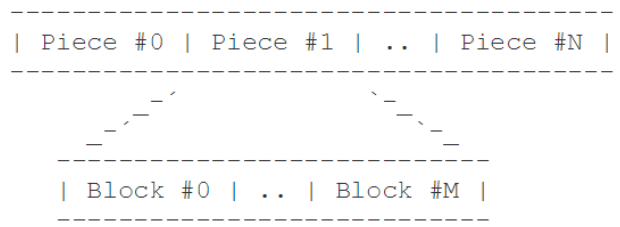
Les chaînes sont encodées en faisant précéder le contenu de la chaîne avec la longueur de la chaîne suivi de deux points. Par exemple la chaîne "announce" est codé comme "8: announce".

Les listes sont un nombre arbitraire d'éléments qui sont « bencodé » précédés de la lettre "l" et postfixé avec la lettre "e". Il en résulte que les listes peuvent contenir des listes imbriquées et des dictionnaires. Par exemple "li2e3: foo" définit une liste contenant le nombre entier "2" et la chaîne "foo".

Les dictionnaires sont un nombre arbitraire de paires clé / valeur délimités par la lettre "d" au début et à la lettre "e" à la fin. Toutes les touches sont bencoded cordes tandis que la valeur associée peut être tout élément bencoded. Par exemple " d5:monthi4e4:name5:aprile" définit un dictionnaire contenant les associations: "month" => "4"et "name" => "april". Toutes les clés de dictionnaire doivent être triées.

3.3 Les pièces et les blocs

Chaque pièce représente une série de données dont il est possible de vérifier en utilisant un hachage SHA1 pièce. Lors de la distribution des données en pièces PWP (Peer Wire Protocol) sont divisés en un ou plusieurs blocs, comme le montre le schéma ci-dessous:



Le nombre de pièces dans le torrent est indiqué dans le fichier de métadonnées. La taille de chaque pièce dans le torrent reste fixe et peut être calculée selon la formule suivante:

```
fixed_piece_size = size_of_torrent / number_of_pieces
```

La taille d'un bloc est définie une valeur choisie lors de l'implémentation qui ne dépend pas de la taille fixe de la pièce. Une fois une taille fixe définie, le nombre de blocs par pièce peut être calculé en utilisant la formule:

```
number_of_blocks = (fixed_piece_size / fixed_block_size)
                 + !!(fixed_piece_size % fixed_block_size)
```

3.4 Le fichier de métadonnées

Le fichier métadonnées fournit au client des informations sur la localisation du tracker ainsi que le torrent à télécharger. Outre la liste des fichiers qui se traduira par le téléchargement de torrent, il répertorie également comment le client doit séparer et vérifier les pièces individuelles qui composent le torrent complet.

3.5 Le tracker HTTP

Le protocole HTTP Tracker (THP) est un mécanisme simple pour introduire des peers entre eux. Un tracker est un service HTTP qui doit être contacté par un peer pour rejoindre un réseau de peers. En tant que tel le tracker constitue le seul élément centralisé en BTP / 1.0. Un tracker ne fournit pas par lui-même l'accès à toutes les données téléchargeables. Un tracker de peers repose sur l'envoi des demandes régulières. Il peut supposer qu'un peer est mort s'il manque une requête.

3.6 Le protocole « Peer Wire »

Le but du PWP est de faciliter la communication entre peers voisins dans le but de partage de fichiers. PWP décrit les étapes prises par un peer après qu'il a lu dans un fichier métadonnées et contacté un tracker pour recueillir des informations sur d'autres peers avec lesquels il peut communiquer. PWP est posée sur le dessus du protocole TCP et gère toute sa communication en utilisant des messages asynchrones.

4 Approche pratique de l'implémentation du client BitTorrent

4.1 Vue globale

Pour la mise en place de notre client nous avons opté pour l'utilisation du Java comme langage de développement, il apporte une plus simple implémentation des sockets que le C, ainsi qu'une plus grande facilité pour la gestion des chaînes de caractères.

4.2 Le traitement du fichier de métadonnées

Comme vue dans l'approche théorique, le fichier contenant les métadonnées est encodé en Bencode. Afin de décoder son contenu nous avons utilisé une librairie qui se nomme bee-encode :

https://code.google.com/p/bee-encode/

Cette librairie nous permet de simplement d'extraire les informations du fichier « .torrent » en passant le flux de donnée (*InputStream*) extrait de l'ouverture du fichier « .torrent ». Les dictionnaires sont vus comme un type Map sous Java, les entiers sont des BigInteger et les listes sont des ArrayList.

Pour extraire toutes les informations utiles du fichier nous avons créé une classe « Metafile », celle-ci permet simplement en lui passant un fichier au constructeur d'extraire toutes les données et de les rendre accessibles via des accesseurs. Voici la liste des paramètres accessibles ainsi que leurs types :

- String announce
- ArrayList<String> announce_list
- String comment
- String createdBy
- Date creationDate
- Map<String, ?> info
- private String name
- Integer piece_length
- byte[] pieces
- String source
- Integer length (si c'est un seul fichier)
- ArrayList<Map<String, ?>> files (si c'est un torrent de type multifile)

Les fichiers torrents qui contiennent plusieurs fichiers (multifiles) peuvent être testés via une méthode de la classe Metafile qui se nomme « isSingleFile() » et qui retourne un boolean à faux si c'est le cas.

4.3 Allocation de l'espace disque et requête au Tracker

Une fois les informations du contenu (taille des données) du torrent extraites des métadonnées via la classe Metafile, il faut allouer l'espace disque où l'on mettra ensuite toutes les pièces téléchargées avec notre client. Pour cela nous utilisons une classe qui se nomme « Torrent », celle-ci prend 3 paramètres pour le constructeur. Le premier est l'objet de type Metafile précédemment créé. Le deuxième est le numéro du port avec lequel on désire communiquer avec le tracker afin d'effectuer la requête des peers. Et finalement, le troisième, c'est le peer ID de notre client, celui-ci peut être choisi aléatoirement, il faut simplement qu'il soit composé de 20 caractères.

Si le fichier est de type « single file » on va simplement créer un seul fichier comportant le nom du torrent. Si c'est un torrent dit « multifiles », un dossier comportant le nom du torrent sera créé, il contiendra ensuite toute l'arborescence du torrent.

L'infohash est également calculé dans le constructeur de cette classe, il sera nécessaire pour effectuer la requête au tracker. Il est accessible à tout moment via un accesseur « getInfoHash() » qui retourne celui-ci sous forme d'un tableau de bytes.

Pour la requête vers le tracker on utilise la méthode « request » de la classe Torrent, celle-ci va effectuer une requête de type GET à l'URL descrite ci-dessous:

```
URL url = new URL(torrent.getAnnounce() +
                "?info_hash=" + this.infoHashEncoded +
                "&peer_id=" + this.peerID +
                "&port=" + this.port +
                "&uploaded="      + this.uploaded +
                "&downloaded=" + this.downloaded +
                "&left=" + this.left +
                "&event=" + this.event +
                "&key=12345" + "&compact=1");
```

L'infohash est celui qui a été calculé précédemment. Le peer ID c'est celui qui à été passé lors de la construction de l'objet, idem pour le port. Les champs uploaded et downloaded sont à 0, car on part du principe que c'est la première fois qu'on télécharge le fichier (donc pas de pièces présentes sur le disque) et aussi que notre client est un bad leecher, donc qu'il n'a pas uploadé de pièces. Le champ left correspond à quantité totale de bytes que le peer (notre client dans ce cas) nécessite pour finaliser sont téléchargement. Le champ event dans notre cas est utilisé avec la valeur « started », celle-ci doit toujours être mise si c'est la première fois qu'un peer se connecte au tracker.

Les deux derniers champs, key et compact, ne sont pas toujours nécessaires, mais afin de faire en sorte que notre requête fonctionne vers le plus grand nombre de trackers possibles nous avons décidé de les garder. Le paramètre key est une clé qui est censée prouver l'authenticité de l'auteur de la requête, dans notre cas nous avons juste mis une valeur fixe. Le compact est pour décrire sous quel format nous demandons la réponse qui contient les peers. Si le compact vaut 0, la réponse sera sous la forme d'un dictionaire contenant l'IP des peers, leur port ainsi que l'ID de ceux-ci. Si le compact est à 1 la réponse sera simplifiée, les peers seront « encodés » sous la forme de 4 bytes pour l'adresse IP, suivi de 2 bytes pour le port. Chaque peer sera donc espacé de 6 bytes. L'avantage de l'utilisation du compact c'est que la trame de réponse se retrouve considérablement raccourcie, car l'ID du peer n'est pas présent. Si on désire vraiment le connaître, il faudra l'extraire lors du handshake avec le peer en question.

La méthode request retourne un dictionnaire contenant la réponse complète.

4.4 La liste des peers

Afin de pouvoir stocker chacun des peers (ainsi que leurs informations relatives) obtenus en réponse par le tracker, nous avons développé deux classes : peer et peers. La classe "Peer" constitue un peer simple, avec des accesseurs retournant l'adresse IP, le port ou l'ID. La classe "Peers" est instanciée en prenant la réponse du tracker en paramètre. Avec cette dernière, elle récupère le nombre de seeders et de leechers ainsi que l'intervalle de temps entre les requêtes vers le tracker. Chaque tuple "adresse IP : port" forme un objet de type "Peer" et est stocké dans un tableau.

4.5 Le gestionnaire de peers

Ce gestionnaire est implémenté dans la classe PeerManager.java. L'intérêt de développer une telle classe est double : gérer les connexions vers tous les peers et gérer le téléchargement de toutes les pièces. Ainsi la classe est décomposée en deux parties : une méthode "updatePeers" dont le rôle est de tenter une connexion (handhsake) vers chaque peer, puis un thread "manager" dont la tâche, est de maintenir le nombre de peer actifs au maximum (définit par une constante nommé MAX_PEERS_CONNECTIONS et qui est passée au constructeur de la classe). L'expression "peer actif" détermine un peer qui effectue un téléchargement de pièces.

La méthode "updatePeers" prend en paramètre un objet de type "Peers", et pour chaque objet de type "Peer" stocké dans ce tableau, elle crée un objet de type "PeerConnection" (dont le contenu est décrit dans le chapitre suivant). L'attribut "handshaken" de cet objet informe si l'initialisation de la connexion avec le peer distant s'est déroulée avec succès ou non. Dans l'affirmative, l'objet est stocké dans un tableau contenant tous les "PeerConnection" prêts pour le téléchargement.

Le thread "manager" est lancé dès l'instanciation de la classe "PeersManager". Il passe en revue le tableau contenant les objets "PeerConnection" et vérifie si la connexion est toujours existante. Dans l'affirmative, il lance le téléchargement si le nombre de peer actifs est inférieur à la limite. Sinon, l'objet est simplement retiré du tableau.

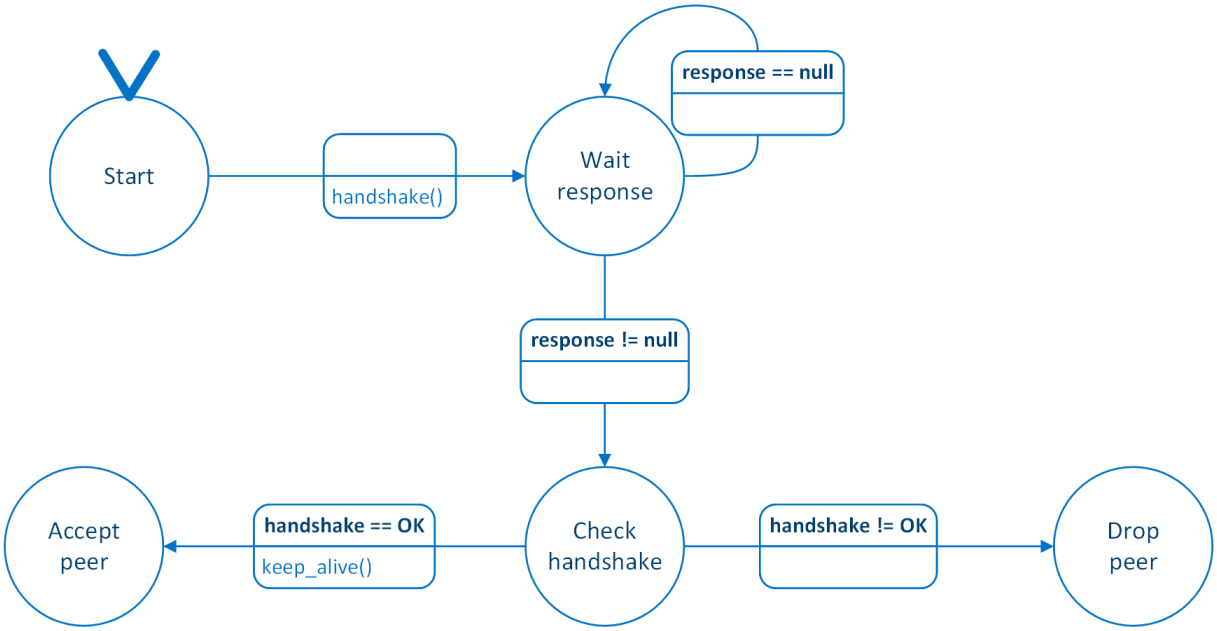
4.6 Connexion au peer et téléchargement d'une pièce

Tout le protocole PWP est implémenté à travers la classe "PeerConnection". Du moins, ce que nous avions besoin en tant que "bad leecher". Nous avons donc développé une fonction procédant au "handshake", un thread qui écoute tous les messages reçus (lecture sur le socket) ainsi qu'un deuxième thread qui traite chacun de ces messages. A cela s'ajoute un autre thread qui récolte des statistiques sur le téléchargement et les écrit dans un fichier de logs, puis un thread qui permet d'envoyer des messages de type "keep-alive" afin d'être sûr que la connexion est toujours active. Finalement, la fonction principale, elle aussi threadée, envoie des requêtes de blocs successives de manière à télécharger des pièces une par une. L'objet "PeerConnection" s'exécute de manière entièrement parallélisée.

A l'instanciation de la classe, la méthode "handshake" est appelée, puisqu'il s'agit de la première étape du protocole PWP. Il permet de s'assurer que la communication est possible avec le peer distant, en s'échangeant un message bien précis. L'issue de cet échange (après l'analyse du message) détermine l'état de l'attribut "handshaken", qui est utilisé par le "PeersManager" comme expliqué auparavant.

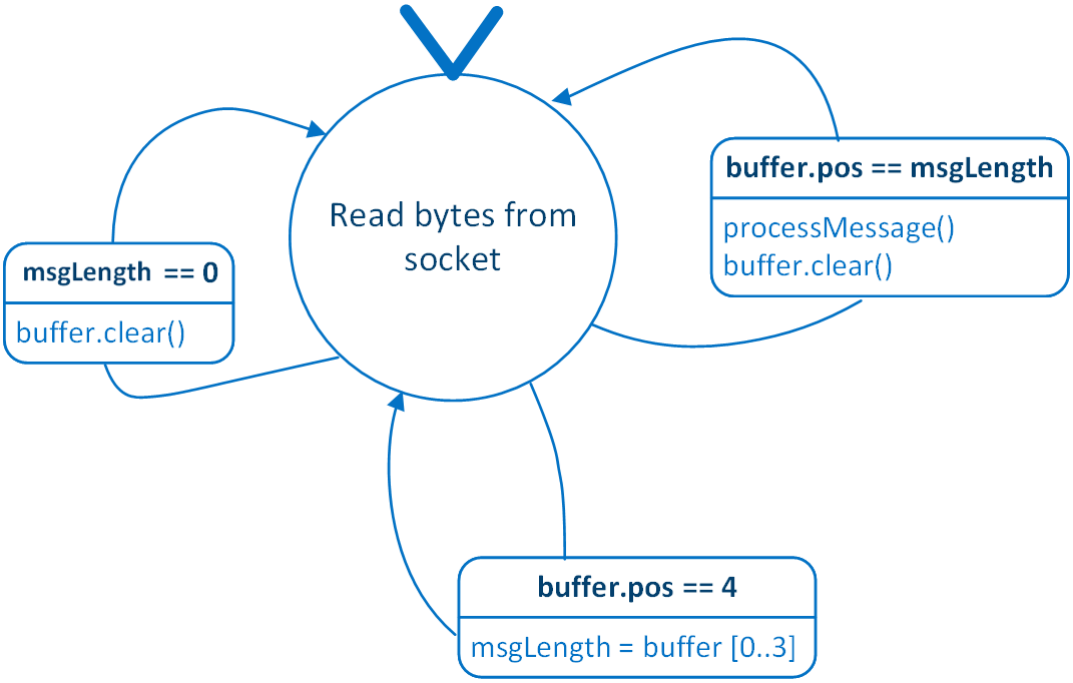
Si on lance un demande de blocs, puis on attend la recéption de ceux-ci et que pendant l'envoi des blocs le peer connecté décide de se déconnecté notre client restera bloqué en écoute. Le seul moyen de savoir à 100% si un peer est toujours connecté c'est de lui envoyer quelque chose. C'est là où rentre en jeu le « keep-alive », ce petit message de 4 bytes permet simplement de tester si la communication est toujours active. On a donc implémenté un thread qui tourne boucle en attente passive (sleep) et qui se réveille après un certains temp (2min selon la RFC) afin d'envoyer ce « keep-alive », si l'envoi échoué on considpre donc le peer s'est déconnecté. Si la communication est coupée, on désactive donc ce peer (changement de l'attribut « isAlive » à false), ce PeerConnection sera ensuite « drop » par le PeerManager la prochaine fois qu'il testera le flag « isAlive ».

Machine d'état pour le handshake :



Lorsque le handshake est terminé et validé, il faut impérativement écouter les messages. Le thread "RecveiveMessages" est dédié à cette tâche et lit en continu les données qui arrivent sur le socket. Les données sont découpées pour reconstituer les messages envoyés par le peer, et chacun de ces messages est traité par le thread "ProcessMessage" selon la RFC.

Machine d'état pour la réception des messages (PWP receiver) :

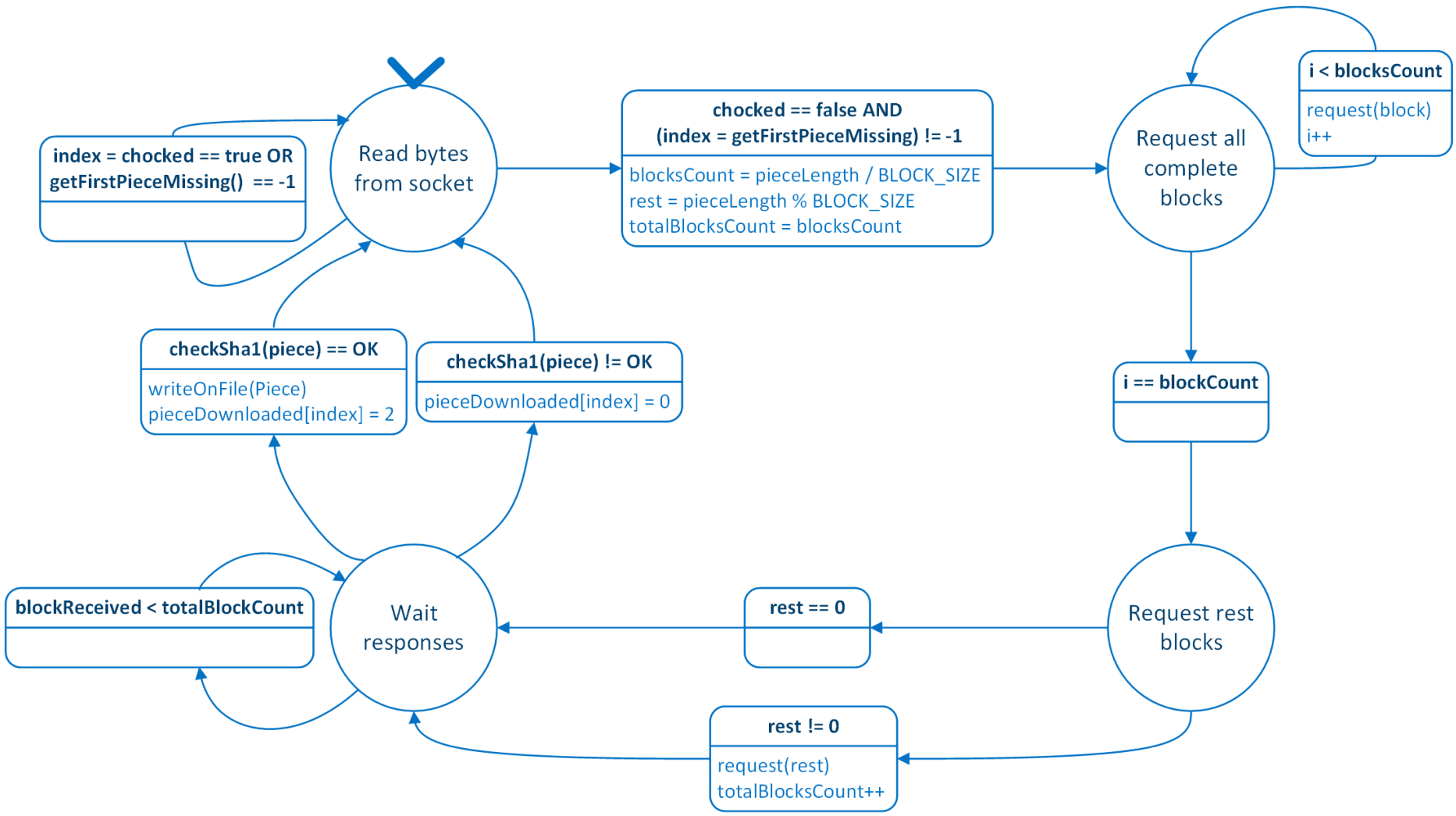


On clear le buffer quand le msgLength vaut 0 (lors de la réception d'un message de type « keep-alive »).

Le peer est rendu actif avec l'appel de la méthode "start" (appel de la méthode "run" threadée). Dès lors, les requêtes pour télécharger les pièces commencent. A chaque itération d'une boucle infinie, il vérifie d'être autorisé à envoyer des requêtes (flag "chocked") et s'assure qu'une pièce manquante localement est disponible chez le peer distant. Si ces conditions sont remplies, alors une requête pour chaque bloc constituant la pièce en question est envoyée. Seulement lorsque tous les blocs sont reçus, on vérifie le SHA-1 de la pièce entière. Si l'issue du test est positive, on écrit la pièce en question sur le fichier. Le téléchargement de la prochaine pièce peut commencer.

Dès que l'objet "PeerConnection" devient actif, le thread "writeLogs" est lancé. Son rôle consiste à récolter les données (date, adresse IP, port, version du soft, vitesse de téléchargement, pièces manquantes) du peer distant, et de les inscrire dans un fichier de logs toutes les 10 secondes.

Machine d'état pour la requête des pièces (PWP sender) :



5 Les fonctionnalités

5.1 Implémentées et fonctionnelles

- L'extraction des données du fichier « .torrent »
- Création de la requête au tracker
- Envoi de la requête au tracker et réception de la réponse
- Mise à jour des infos du tracker à un intervalle donnée obtenue dans la réponse
- Extraction des peers de la réponse du tracker
- Établissement des connexions vers les peers via un handshake
- Réception des messages de type PWP
- Traitement des messages reçus (bitfield, have, choke, unchoke, interested, uninterested et piece)
- Téléchargement des pièces.
- Possibilité de configurer le nombre de téléchargements des pièces simultané (une pièce par peer)
- Maintiens de la connexion à un peer géré par un keep-alive
- Inscription des logs dans un fichier
- Vérification de l'intégrité des pièces téléchargées (sha1)
- Sauvegarde des pièces reçues dans un fichier sur le disque (uniquement single file)

5.2 Incomplètes

- La sauvegarde sur le disque ne supporte pas les torrents multifiles
- Traitement des messages incomplet (cancel, request et port)
- Actualiser le nombre de pièces téléchargées dans la requête au tracker

5.3 Manquantes

Ces fonctionnalités n'étaient pas demandées dans le cahier des charges, mais elle pourrait être implémentée dans un prochain travail :

- Plusieurs torrents actifs par client
- Interface graphique
- Mise en pause/reprise du téléchargement
- Limite de débit de téléchargement (bits/s)
- Client en seeder

6 Bugs rencontrés

8 décembre 2014

Problème : fichier torrent.java
Lorsque l'on essaye de créer une arborescence composée de dossiers, les dossiers ne sont pas créés. Même avec la méthode "mkdirs", ils devraient être créés.

Solution : Il suffit d'ajouter un "./" dans le path du fichier

Reproduction :
Dans le fichier "Torrent.java" qui contient le constructeur de la classe, remplacez les lignes ligne :

```
String filePath = "." + File.separator + torrent.getName() + File.separator;
// Get the file path
for (byte[] data : path) {
	filePath = filePath + File.separator + new String(data);
}
```

Par le code :

```
String filePath = "";
// Get the file path
for (byte[] data : path) {
	filePath = filePath + File.separator + new String(data);
}
```

Aucune arborescence de fichier n'est créée par le mkdirs qui est contenu dans la fonction. Tous les fichiers sont créer à la racine.

9 décembre 2014

Problème :
Pas de réponse correcte de la part des peers lors de notre handshake.

Solution :
Notre peer envoyait un infohash erroné, dans la RFC il est spécifié que c'est le même infohash qui est envoyé lors de la requête au tracker, mais en réalité c'est n'est pas toute à fait le même. Lors de la requête au tracker l'infohash est transformer en hexa puis il est "URLEncodé", dans notre cas c'était évident qu'il ne fallait pas faire l'encodage de l'URL, mais on ne savait pas qu'il ne fallait pas le transformer en hexa.
Donc pour faire une requête correcte l'info hash doit être envoyé en brut, c'est à dire directement via un tableau de byte.

Reproduction :

Dans le fichier "ClientBitTorrent.java" remplacez le code suivant :

```
PeersManager pm = new PeersManager(peers, metafile.getPieces(),
torrent.getInfoHash(), peerID);
```

Par le code :

```
PeersManager pm = new PeersManager(peers, metafile.getPieces(),
javax.xml.bind.DatatypeConverter.printHexBinary(torrent.getInfoHash()).getB
ytes(), peerID);
```

Vous verrez que plus aucun peer ne répondra correctement. Vous aurez des "End of file exception" lorsque vous recevrez quelque chose.

14 janvier 2014

Problème :
Certains blocs ne se téléchargent pas correctement.

Solution :
Lors de la réception d'un bloc, on calculait la taille dynamiquement (car le dernier bloc d'une pièce pouvait potentiellement être plus petit que la taille fixe définie dans le code). Lors de ce calcul, nous récupérions la valeur du champ "length" et en déduisant la longueur de l'en-tête nous obtenions la taille du bloc. Là où nous avons fait faux, c'est d'avoir déduit la longueur du champ "length" aussi, alors que celui n'est pas comptabilisé dans sa propre valeur. Par conséquent, tous les blocs avaient une taille trop petite de 4 bytes.

Reproduction :

Dans le fichier PeerConnection.java, remplacer la ligne :

```
// 1 (id) + 4 (index) + 4 (begin) = 9
int blockLength = ByteBuffer.wrap(msgLength).getInt() - 9;
```

Par :

```
// 4 (length) + 1 (id) + 4 (index) + 4 (begin)
int blockLength = ByteBuffer.wrap(msgLength).getInt() - 13;
```

7 Répartition du travail

Description de l'activité	Pourcentage de travail	
	Da Silva	Magnin
Recherche d'une librairie beeencode	50	50
Classe Metafile.java		
Lire le fichier	100	0
Récupérer les infos via la librairie bee-encode	100	0
Classe Torrent.java		
Création de l'arborescence des fichiers	20	80
Construction de la requête vers le tracker	20	80
Envoi de la requête vers le tracker	0	100
Classe Peers.java		
Extraction des informations du trackers et accès par getters	100	0
Classe Peer.java		
Classe contenant les propriétés d'un peer	100	0
Classe PeerManager.java		
Création du manager	50	50
Version finale du manager	0	100
Classe PeerConnection		
Mise en place du handshake	50	50
Confirmation de la réponse correcte au handshake	80	20
Sous-classe SendMessages	0	100
Sous-classe ReceiveMessage	0	100
Requêtes des blocs	50	50
Contrôle de l'intégrité des pièces	50	50
Général		
Rapport	50	50
Machines d'états	50	50
Vidéo (écriture et réalisation)	50	50

8 Conclusion

L'ensemble de ce projet était très intéressant, dans la mesure où tout le travail de recherche nécessaire avant l'implémentation nous a permis de découvrir le protocole BitTorrent (presque) entièrement. Par ailleurs, c'était aussi une occasion pour nous de mettre en pratique les connaissances acquises en cours de programmation à travers un projet un peu plus conséquent que d'habitude. Même si l'expérience de travailler en groupe n'était pas nouvelle, c'était la première fois qu'il fallait y accorder un soin particulier : la répartition efficace des tâches. Dans l'ensemble, nous pensons avoir réussi à le faire correctement et sommes satisfaits de la manière dont nous avons procédé.

D'un point de vue plus technique, le ressenti global n'était pas à la hauteur de nos attentes. Nous avions le choix du langage pour implémenter ce client BitTorrent, et nous nous sommes penchés sur du Java. Avec du recul, nous aurions sans doute utilisé Python pour les quelques avantages qu'il procure par rapport à Java (plus grande flexibilité en général). Dans l'ensemble, nous avons passé davantage de temps à devoir résoudre des problèmes de langage que des problèmes d'algorithme.

Finalement, nous avons malgré tout réussi à développer un client pouvant télécharger un fichier entier, et dont les fonctionnalités dépassent un petit peu ce qui était demandé dans le cahier de charges. Par conséquent, nous sommes fiers d'avoir relevé le défi avec succès, et notre souvenir sur ce travail restera très bon.

9 Sources

Pour toutes les définitions théoriques du protocole :
https://code.google.com/p/bee-encode/
http://www.bittorrent.org/beps/bep_0003.html
https://wiki.theory.org/BitTorrentSpecification
http://jonas.nitro.dk/bittorrent/bittorrent-rfc.html