Filière Informatique

Projet Réseau

Application d'échange de fichiers en Pair à Pair

Otavio Flores Jacobi
Abdelouahab Laaroussi
Julien Miens
Tanguy Pemeja
Maxime Rosay

Encadrant :
Joséphine Calandra

Version du 24 mai 2020

Contents

1 Introduction **2**

2 Management du projet **2**

3 Architecture **2**

4 Tracker **3**
- 4.1 Pool de thread . . . 4
- 4.2 Fonctionnement du tracker . . . 4
- 4.3 Parser les commandes reçues . . . 5
  - 4.3.1 Announce . . . 5
  - 4.3.2 Update . . . 5
  - 4.3.3 Getfile . . . 5
  - 4.3.4 Look . . . 5
  - 4.3.5 Quit . . . 6
- 4.4 Base de données . . . 6
  - 4.4.1 Recensement de fichiers . . . 6
  - 4.4.2 Stockage de pairs . . . 7

5 Pairs **7**
- 5.1 Architecture . . . 7
  - 5.1.1 Shell . . . 7
  - 5.1.2 Parser et ParsingResult . . . 8
  - 5.1.3 Peer . . . 8
  - 5.1.4 Config . . . 8
  - 5.1.5 MD5 . . . 9
  - 5.1.6 DownloadManager et MultiThreadServer . . . 9
  - 5.1.7 OtherPeers . . . 9
  - 5.1.8 BufferMap . . . 10
- 5.2 Communications Pairs-Tracker . . . 10
  - 5.2.1 Announce . . . 10
  - 5.2.2 Look . . . 10
  - 5.2.3 Getfile . . . 11
  - 5.2.4 Update . . . 11
- 5.3 Communications Pairs-Pairs . . . 11
  - 5.3.1 Interested . . . 11
  - 5.3.2 Getpieces . . . 12
  - 5.3.3 Have . . . 12

6 Utilisation d'un tracker à distance **12**
- 6.1 AWS EC2 . . . 13
- 6.2 Mise en œuvre . . . 13

7 Conclusion **13**

1 Introduction

Le projet de Réseau consiste en la réalisation d'un système de transfert de fichiers *pairs-à-pairs* utilisant un *tracker* afin de centraliser l'information et indiquer aux pairs à quels autres pairs s'adresser pour obtenir des fichiers.

Le P2P est une technique utilisée dans le but de mieux répartir l'utilisation du réseau et surtout de réduire la charge de travail dans une structure centralisée.

Dans ce projet, nous constatons que notre serveur centralisé (tracker) n'est utile que pour faire connaître à un "pair" l'emplacement et les fichiers des autres, mais le tracker n'est pas responsable du téléchargement ou de l'envoi d'un fichier, ce qui réduit considérablement la charge de cette entité. De plus, le fait de pouvoir obtenir des données de plusieurs sources en même temps permet également d'équilibrer la charge de chaque pair.

Ce projet engendre des problèmes théorique comme les problèmes de sécurité lorsque nous connectons deux hôtes.
Mais d'autres problèmes avec une perspective plus pratique se posent. Aujourd'hui par exemple, la plupart de nos réseaux (ipv4) utilisent le protocole NAT. Nous devons avoir accès à notre routeur et permettre à certains ports d'être atteints directement (en contournant le NAT) pour que le P2P fonctionne, sinon nous ne serions pas en mesure d'établir une connexion directe avec quelqu'un derrière une table de NAT.

2 Management du projet

Afin de mieux répartir les tâches, nous avons divisé notre groupe en deux parties : deux personnes pour le `tracker` (C) et trois pour la partie java (pairs).
Pour les premières versions, les deux équipes travaillaient avec des `mocks`, l'équipe du pair a fait un `mock` pour le `tracker` et vice-versa. Après que la première version du tracker ait été fonctionnelle, le groupe des pairs a commencé à utiliser la vraie version du `tracker`, ce qui a permis le débuggage et l'apport d'améliorations au système.

Chaque groupe était indépendant dans sa manière de travailler, mais nous devions communiquer afin de fusionner les informations. Pour cela, nous avons utilisés notre serveur `Discord` et fait des appels réguliers avec notamment notre encadrante pendant les heures de cours.

3 Architecture

Le système est composé de deux entités : un `tracker` recensant les informations de chaque pair puis les pairs en-eux même.

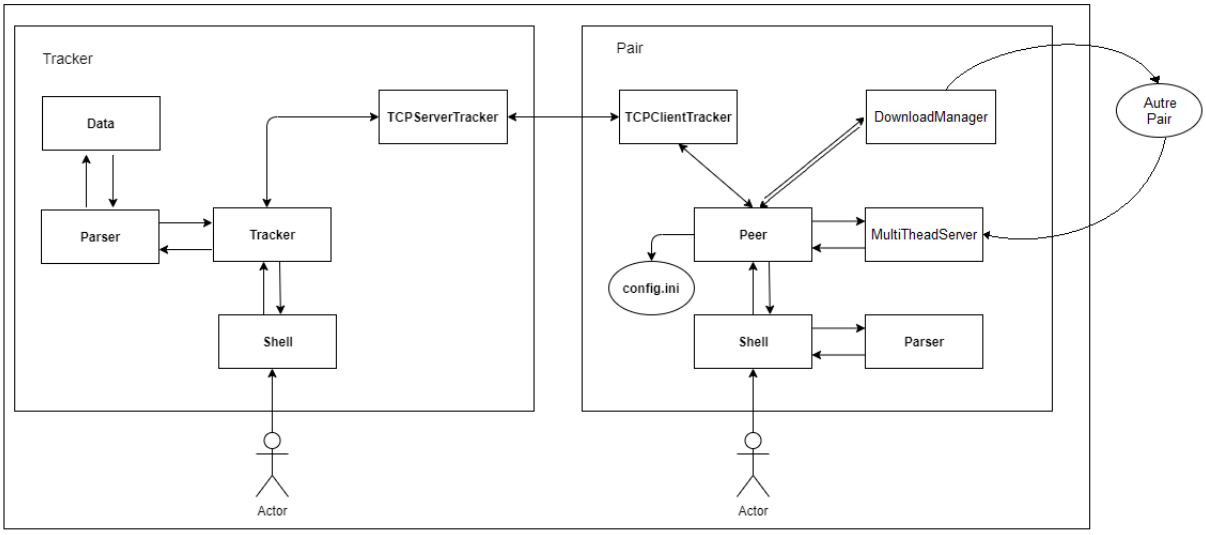

Figure 1: Architecture global du projet

Sur la figure 1 ci-dessus, nous pouvons voir le principe de fonctionnement de notre projet dans son ensemble. Le shell sert d'interface pour l'utilisation de notre `tracker` et de nos `pairs`.

Le `tracker` s'occupe de la gestion des informations des différentes pairs grâce à `Data` en passant par un `Parser`, qui s'occupe de vérifier l'intégrité des commandes fournies.

À l'instar du tracker, le parsing des commandes de nos `pairs` s'effectue directement lors de la communication avec le shell. De plus, chacun des pairs doit pouvoir communiquer avec les autres. Pour se faire, nous avons 2 modules `MultiThreadServer` et `DownloadManager` qui permet cette communication pair-pair.

4 Tracker

Le *Tracker* a pour mission de recevoir des requêtes des pairs lui indiquant les parties de fichiers qu'ils possèdent et celles qu'ils demandent. Il doit ensuite leur communiquer quels autres pairs possèdent les parties de fichiers demandées.

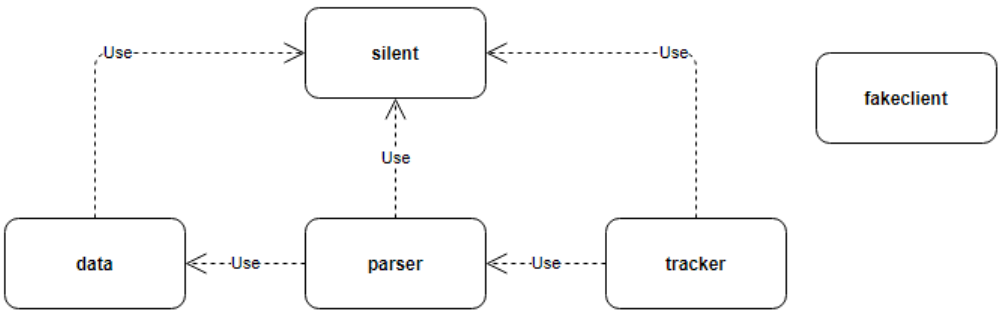

Figure 2: Architecture global du tracker

Pour lancer le `tracker`, il faut utiliser la commande:

```
$ ./tracker [-v] [-p port_number]
```

Les différentes options facultatives sont les suivantes:

- `-v`: active le mode non silencieux[1] du `tracker`. Les logs sont alors affichés directement dans le terminal de ce dernier.
- `-p`: permet de spécifier un numéro de port au `tracker`. Par défaut, il s'agit du port 25000 si l'option n'est pas renseignée. De plus, si le port en question est déjà en cours d'utilisation, le `tracker` va incrémenter ce dernier jusqu'à en trouver un non utilisé.

4.1 Pool de thread

Le *tracker* utilise un pool de threads[2] pour traiter les requêtes qu'il reçoit en parallèle. C'est au tracker de choisir un thread pour lui associer une connexion entrante, ainsi chaque thread attend activement d'avoir reçu du travail avant de poursuivre son exécution[3].
Une fois la connexion entrante terminée (déconnexion d'un pair), le thread repart en attente active. La connexion entre le tracker et le pair peut être uniquement terminée par le pair.

La structure des threads est la suivante:

```
struct thread_args {
  int socket_value;                    // Identifiant de socket representant
                                       // l'interconnectivite tracker-pair

  pthread_t thread;                    // Permet de stocker le thread

  int status;                          // Indique la disponibilite du thread

  int filled_status;                   // Permet de savoir quand tous les champs
                                       // sont bien remplis et pour que le thread
                                       // puisse ensuite s'executer

  int exit_status;                     // Indique s'il faut liberer le thread

  char client_addr[INET_ADDRSTRLEN];  // Renseigne l'adresse IP du pair connecte
};
```

Chaque thread bouclera sur la même fonction suivante en pseudo-code:

```
/* Attente de travail */
/* Lire tant que la connectivite est etablie */
read(thread.socket, buffer);      // lecture de la socket
output = parser(buffer);          // envoie au parser pour interpretation
write(thread.socket,output);      // ecriture sur la socket
/* Si la connexion est terminee, repartir au debut*/
```

Cela leur permet ainsi de lire et d'écrire sur une socket dédiée à un pair spécifique de manière continue jusqu'à déconnexion de ce dernier.

4.2 Fonctionnement du tracker

L'algorithme utilisé par le `tracker` en pseudo-code est le suivant :

```
/*  Initialisation de la base de donnee */
/*  Lancement des threads (pool de thread) */

socket = listen();                   // ouverture d'une socket d'ecoute
                                     // pour les connexion entrantes
/* Boucle executee par le tracker */
while (1) {
    newsocket = accept(socket);      // nouvelle connexion
    thread = free_thread_id();       // recuperation d'un thread libre
    thread.socket = new_socket;      // association du socket au thread
}
close(socket);                       // on ferme la socket dediee aux connexions
                                     // entrantes
exit_threads();                      // on termine tous les threads
```

[1] Présent dans le module Silent
[2] Nombre de threads limité à 5
[3] A savoir lire / écrire dans une socket prédéfinie

Le `parser` va ensuite interpréter la commande et stocker les différentes informations collectées dans la base de données du `tracker`.

4.3 Parser les commandes reçues

Le second module dont il est question dans cette partie dédiée au `tracker` est le `parsing` des commandes entrantes.

4.3.1 Announce

Cette commande permet au pair d'être recensé par le `tracker`.
La commande `announce` s'écrit sous la forme universelle suivante:

```
$ < announce listen [port_number] seed [filename size piece_size key ...] leech [
    key1 key2 ...]
$ > ok      // ou nok
```

Notre `parser` va vérifier l'intégrité de notre commande suivant les critères suivant :

1. L'ordre et la présence des arguments de la commande
2. La conformité des informations fournies dans seed : bloc de 4 données, bonne utilisation des crochets[4]
3. L'encadrement de la liste des clés du leech par des crochets

4.3.2 Update

Cette commande permet au pair d'actualiser ses données personnelles sur ses fichiers possédés et en cours de téléchargement.

```
$ < update seed [key1 key2 ..] leech [key1 ..]
$ > ok      // ou nok
```

Notre `parser` va vérifier l'intégrité de notre commande suivant les critères suivant :

1. L'ordre et la présence des arguments de la commande
2. La bonne utilisation des crochets dans l'encadrement des listes des clés de seed et leech

4.3.3 Getfile

Cette commande permet au pair de récupérer une liste d'adresse IP avec leur port associé correspond à une `key` déterminée.

```
$ < getfile key
$ > peers key [IP:port_number ..]
```

Lorsque notre `parser` va détecter le mot-clé `getfile`, il va vérifier s'il y a bien une clé qui suit et ignorer les arguments qui suivent.

4.3.4 Look

Cette commande permet de récupérer des informations de fichiers (nom, taille, découpage et `key`) selon différents critères: nom du fichier, taille du fichier (>,< ou =) ou adresse IP d'un pair.

```
$ < look [criterion ..]
$ > list [...]
```

Pour la commande `look`, notre `parser` va vérifier l'intégrité de notre commande suivant les critères suivant :

[4]Dans notre implémentation, il n'est pas possible d'avoir des crochets dans le nom des fichiers

1. L'ordre et la présence des arguments de la commande
2. La bonne utilisation des crochets dans l'encadrement de nos critères de recherche
3. Des critères de recherche valides parmi notre liste de critère implémentés : filename=, filesize>, filesize=, filesize< et peerip=.

Ce dernier critère est un ajout de notre groupe afin de pouvoir récupérer des fichiers d'un pair spécifique, identifié par son adresse ip. Lorsque ce critère est activé, on diminue la complexité de notre fonction de recherche en ne sélectionnant que les fichiers appartenant à ce pair.

4.3.5 Quit

Cette commande bonus permet de conserver une intégrité des données présentes dans la base de données. Quand un pair se déconnecte manuellement, il utilise cette commande et ses données personnelles seront supprimées de la base.

```
$ < quit
$ > Good bye!
```

4.4 Base de données

La base de données est constituée de deux tableaux contenant pour le premier des structures `seed` et le second des structures `peer`.
Nous avons préféré utiliser des tableaux plutôt qu'une fonction de hashage par question de simplicité de codage et de complexité temporelle. En effet, nous accédons et modifions nos tableaux directement à partir d'indice[5] permettant ainsi de créer des liste d'indices pour y référer plus rapidement.

Il faut noter que chaque fichier est représenté de manière unique par sa `key` et que chaque pair est identifiée par son adresse IP.

4.4.1 Recensement de fichiers

La structure `seed` permet de stocker le détail d'un fichier (nom, taille, ...). Elle dispose de plusieurs champs listés ci-dessous:

```
    struct seed {
        char filename[MAX];  // renseigne le nom du fichier

        int len;             // renseigne la taille du fichier

        int piece_size;      // renseigne le decoupage du fichier

        char key[MAX];       // renseigne la cle associee au fichier

        int status;          // indique si cette case du tableau de fichiers est
                             // libre ou non

        int owners[MAX];     // renseigne la liste des pairs (par ID) possedant
                             // le fichier

        int owners_count;    // renseigne le nombre de possesseurs
    };
```

Pour modifier ces champs, nous avons des `accesseurs` prenant, pour la plupart, l'indice du tableau d'une `seed` en argument.

[5] Ces accès sont en temps constant

De plus, nous avons deux fonctions de recherche de fichiers (alias `seed`) par `key` ou par nom de fichier.
La première fonction retourne l'indice du tableau où se trouve la `seed` recherchée ou `SEED_NOT_FOUND` si elle n'existe pas dans la base de donnée.
La seconde prend un argument en plus du nom de fichier recherché, il s'agit d'un tableau d'indice (au préalable vide). La fonction de recherche va ajouter dans ce tableau tous les indices trouvés puis retourner le nombre d'indices obtenu. Si la valeur retournée vaut zéro, alors elle retourne `FILENAME_NOT_FOUND` indiquant que ce nom de fichier n'existe pas dans la base de donnée.

4.4.2 Stockage de pairs

La structure `peer` permet de stocker le détail d'un pair (IP, port, ...). Elle dispose de plusieurs champs listés ci-dessous:

```
1   struct peer {
2       char client_ip[INET_ADDRSTRLEN];   // indique l'adresse IP du pair
3
4       int port;                          // indique le numero de port du
5                                          // pair
6
7       int files[MAX];                    // precise la liste (par ID) de
8                                          // fichiers que le pair possede
9
10      int files_count;                   // renseigne le nombre de fichiers
11                                         // que possede un pair
12
13      int leechs[MAX];                   // precise la liste (par ID) de fichiers
14                                         // en telechargement
15
16      int leechs_count;                  // renseigne le nombre de fichiers en
17                                         // cours de telechargement
18
19      int status;                        // indique si cette case du tableau de
20                                         // pairs est libre ou non
21  };
```

Cette structure fonctionne de la même manière que `seed` avec des `accesseurs`.
De plus, nous avons aussi une fonction de recherche de pair par adresse IP, elle fonctionne de manière similaire à la recherche de `seed` par `key` en retournant l'indice du tableau où se trouve le pair ou `PEER_NOT_FOUND` le cas échéant.

5 Pairs

Les pairs sont les éléments clés d'un système de partage en peer-to-peer. Le pair a été codé en Java et doit interagir avec le tracker pour connaitre l'état du réseau et ainsi découvrir les autres pairs qui possédent les fichiers qui l'intéressent. Nous commmencerons ici par aborder l'architecture du pair avec une description de ces classes principales, puis nous aborderons l'implémentation des commandes pair-tracker avant d'expliquer les commandes pair-pair.

5.1 Architecture

Le pair est constitué d'un ensemble de classe comme indiqué sur la figure 3. La classe `Main` n'y est pas représentée, celle-ci lance le Shell avec un Peer et un Parser en argument.

5.1.1 Shell

C'est la classe en charge d'interagir avec l'utilisateur. Elle possède un objet de classe Peer auquel elle envoie les commandes après les avoir parsées afin d'en vérifier la validité. Comme c'est le point d'entrée du programme, elle se charge aussi de démarrer le Timer permettant l'envoie périodique de la commande `update` au tracker. Son algorithme pseudo-code est le suivant :

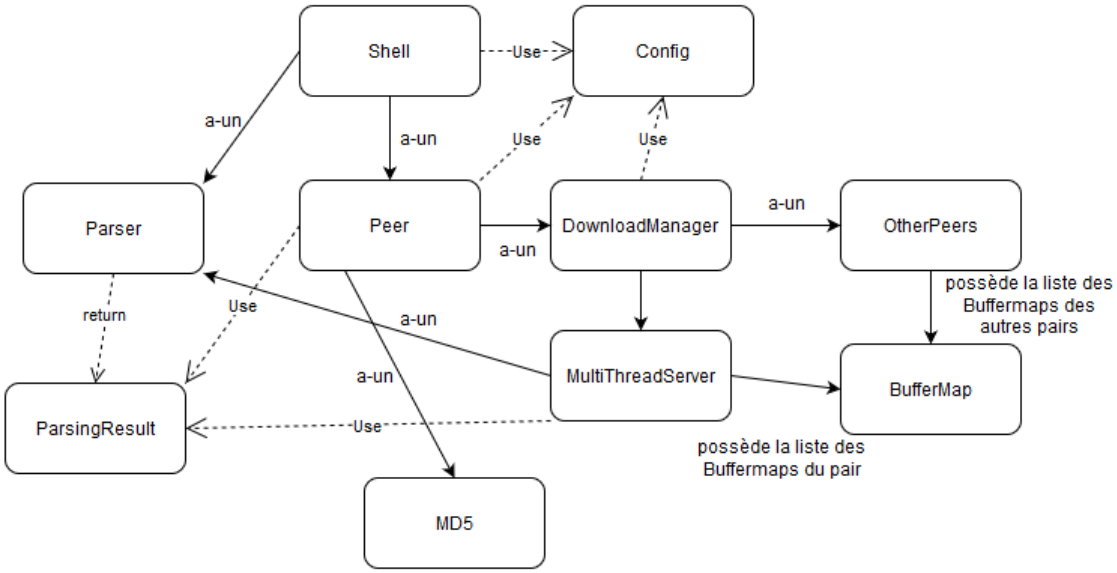

Figure 3: Diagramme de Classes du Pair

```
while (true) {
    newcmd = readLine();
    r = parser.parse(newcmd);
    peer.run(r);
}
```

Il s'agit donc de la boucle principale attendant les commandes utilisateurs.

5.1.2 Parser et ParsingResult

Le `parser` prend en entrée la chaîné de caractère tapée dans le prompt et l'interprète en constituant un objet de type `ParsingResult`. Cet objet se comporte comme un dictionnaire, qui possédera en plus un champ type pour indiquer le type de la commande (`announce`, `look`, ...). Ensuite, selon son type on pourra alors invoquer des `getter` pour récupérer les informations relatives à chaque requête. Par exemple pour un requête de type `announce`, on pourra appeler la méthode `getPort()` afin d'obtenir le port spécifié. On peut ainsi vérifier que la commande entrée est correcte avant de l'envoyer au tracker, ce qui est effectué par la méthode `run` du pair.

5.1.3 Peer

La classe Peer reçoit des commandes sous forme de `ParsingResult` puis les retransmet au tracker ou au DownloadManager, selon que la commande est destiné au tracker ou à un autre pair. Le détail des commandes est abordé dans les sections suivantes. Cette classe sert surtout d'aiguillage pour les commandes entrantes provenant de l'utilisateur.

De plus, cette classe implémente la méthode pour générer une commande `announce` automatiquement. Cette méthode parcours le dossier `./files/` dans lequel se trouvent les fichiers à partager et dans lequel on sauvegardera les fichiers reçus. Lors de l'envoie de l'`announce` (qu'il soit automatique ou non), on récupère alors la liste des `seeds` et on la passe au DownloadManager pour qu'il remplisse la liste de nos propres buffermaps avec.

5.1.4 Config

On utilise le module java `Properties` afin de charger le fichier `config.ini`. Il est ensuite simple de récupérer les informations contenues dans ce fichier. Voici un exemple d'utilisation de ce module :

```
prop = new Properties();
prop.load(new FileInputStream("config.ini"));
int port = Integer.parseInt(prop.getProperty("tracker-port"));
```

La classe `Config` que nous avons implémenté (cette classe a été créée en utilisant une technique appelée *singleton*) utilise donc cela pour récupérer le fichier. La classe interdit son instanciation multiple (constructeur privé) pour ne charger qu'une seule fois le fichier.

Le fichier `config.ini` contient les informations relatives aux paramètres de fonctionnement du pair, à savoir :

- l'adresse IP du tracker et son numéro de port
- le numéro de port d'écoute du pair
- l'envoie automatique ou non du announce initial
- la taille par défaut des morceaux de fichiers
- la période entre deux envoies d'update au tracker.

5.1.5 MD5

Ce module a pour but de générer une clé unique permettant d'identifier chaque fichier. Pour ce faire, le fichier est parcouru par bloque de 1024 octets en mettant à jour la clé au fur et à mesure.

Le module permet aussi de vérifier qu'une clé et un fichier, donné par son chemin, correspondent bien.

5.1.6 DownloadManager et MultiThreadServer

Le DownloadManager est en charge de traiter les réponses du tracker et d'envoyer des requêtes aux autres pairs. Il possède un objet `OtherPeer` qui contient une liste des buffermaps des autres pairs avec leurs adresses IP et ports respectifs.

Il travail en collaboration avec un MultiThreadServer en charge de recevoir les requêtes des autres pairs. C'est objet a à sa connaissance les fichiers et des buffermaps possédés par le pair. A son initialisation, un thread est lancé dont la mission est de prendre les connexions entrantes puis d'y répondre.

5.1.7 OtherPeers

Cette classe contient les informations sur les fichiers et les buffermaps des autres pairs. Elle possède deux listes:

- la liste des couples `IP:port`
- la liste des map entre une clé et un buffermap

Chaque autre pair est donc identifié par un indice qui est le même pour les deux listes et à partir d'un numéro de pair et d'une clé, on peut retrouver le buffermap de l'autre pair.

La méthode `newPeer` prend en argument un couple `IP:port` et soit créer un nouveau pair si l'adresse est inconnu, soit retourne l'indice du pair s'il était déjà connu, cela permet d'éviter de créer des doublons dans les listes. L'indice retourner peut alors servir à ajouter ou modifier la liste des buffermaps de ce pair en fonction des données reçues.

Cela implique une complexité linéaire pour trouver ou créer un pair, mais une complexité constante ensuite pour obtenir un buffermap à partir d'une clé et d'un pair.

5.1.8 BufferMap

Cette classe contient le détail d'un fichier, à savoir :

- le nom du fichier
- la clé
- la taille des parties de ce fichier
- la taille du fichier
- le buffermap, i.e. des booléens pour savoir quelles parties sont possédées
- un booléen pour savoir si toutes les parties sont possédés
- la liste de ces parties sous forme de tableau de `byte`, cette liste reste vide dans le cas d'un buffermap d'un autre pair.

Le buffermap est utilisé dans deux cas de figures. Tout d'abord pour les fichiers réellement possédés, ou en cours de téléchargement, par le pair, et d'un autre côté pour stocker les buffermaps des autres pairs et ainsi savoir à qui on peut demander telles parties. Dans ce second cas la liste des parties en `byte` reste vide car il ne s'agit pas de nos fichiers.

Une fois que le buffermap possède toutes les pièces d'un fichier, il procède alors à la sauvegarde du fichier dans le répertoire local `./files/`.

5.2 Communications Pairs-Tracker

Le pair communique avec le tracker afin d'obtenir les informations de connexions des autres pairs, à savoir les adresses IP et les ports d'écoutes, selon les fichiers dont il a besoin. Les commandes envoyer au tracker sont généralement tapées à la main dans le Shell, sauf pour les commandes automatiques. Les commandes passent d'abord par le parser afin de vérifier leur validité et afin de déterminer à qui elles sont destinées, avant d'être retransmises. Une fois parsées, les commandes se redirigées vers `peer.run(r)` qui les aiguillera vers la méthode adaptée à leur traitement. Ces méthodes retournent la chaine de caractères reçue du tracker pour l'afficher dans le Shell, après traitement de la réponse si besoin.

5.2.1 Announce

La commande `announce` se fait au début de la connexion, afin d'indiquer au tracker notre port d'écoute, les fichiers que l'ont possède et les fichiers que l'on désire.

- commande : `announce listen $Port seed [$Filename1 $Length1 $PieceSize1 $Key1 $Filename2 $Length2 $PieceSize2 $Key2 ...] leech [$Key3 $Key4 ...]`
- réponse : `ok`

La commande est aiguillée vers la méthode `handleAnnounce` dans peer. Cette méthode commence par récupérer les `seeds` afin de constituer ses propres buffermaps. Ensuite la commande est envoyée au tracker, la réponse de celui-ci est retournée pour être affichée dans le Shell.

Cette commande peut-être automatisée au lancement de l'application en le précisant dans le fichier de configuration `config.ini`. Dans ce cas la méthode `getAutoAnnounceCommand` de peer permet de générer la chaine de caractère correspondante à la commande, puis celle-ci est redigirée comme-ci elle avait été tapée dans le Shell.

5.2.2 Look

La commande `look` est une requête effectuée auprès du tracker par le pair afin d'obtenir les détails d'un ou plusieurs fichier à partir d'un critère. Le tracker implémente plusieurs critères, à savoir la recherche par nom de fichier, par taille de fichier ainsi que par adresse IP. Le tracker répond alors avec les éléments tels que la taille et la clé du fichier...

- commande : `look [$Criterion1 $Criterion2 ...]`
- réponse : `list [$Filename1 $Length1 $PieceSize1 $Key1 $Filename2 $Length2 $PieceSize2 $Key2 ...]`

La méthode chargée de son traitement est `handleLook`. La réponse du tracker est alors redirigée vers le DownloadManager afin qu'il crée le buffermap associé au fichier avec sa clé, sa taille, sa taille de pièces et son nom, en vu de son téléchargement futur. Cette méthode est très importante car si le buffermap du fichier n'est pas créé, on ne peut pas stocker les parties de ce fichier lors d'un `getpieces` par exemple.

5.2.3 Getfile

La commande `getfile` sert à récupérer la liste des pairs en capacité de fournir des parties de fichiers pour un fichier donné par sa clé. Sans elle les pairs seraient incapables d'interagir entre eux.

- commande : `getfile $Key`
- réponse : `peers $Key [$IP1:$Port1 $IP2:$Port2 ...]`

C'est la méthode de peer `handleGetFile` qui se charge d'effectuer la requête. La réponse du tracker est ensuite retransmise au DownloadManager afin qu'il actualise sa liste d'autres pairs. Cette liste est stockée dans l'objet `OtherPeers` décrit précédemment.

Le DownloadManager actualise sa liste en ajoutant, ou récupérant s'il existait déjà, un pair dans sa base de données, puis il lui ajoute un buffermap vide, initialisé avec le nom, la clé, la taille et la taille de pièces que l'on a obtenus au préalable avec la commande `look`. Si cela n'a pas été effectué avant, alors on ne peut pas créer les buffermaps vides pour chaque pair et la commande échoue.

5.2.4 Update

La commande `update` est envoyée à intervalles réguliers pour prévenir le tracker de l'évolution de ses `seeds` et de ses `leechs`.

- commande : `update seed [$Key1 $Key2 $Key3 ...] leech [$Key10 $Key11 $Key12 ...]`
- réponse : `ok`

La commande devant être envoyée périodiquement, un timer a été mis en place avec les classes Java `Timer` et `TimerTask`. La période entre chaque émission est fixé dans le fichier de configuration. La méthode se trouve dans la classe `Peer` et se nomme `sendUpdateCommand`. Elle fonctionne de la même manière que `getAutoAnnounceCommand` sauf qu'elle n'envoie que les clés des buffermaps. La commande est ensuite directement envoyée au tracker sans passer par `peer.run(r)` car la tâche est effectuée en parallèle de l'exécution normale.

5.3 Communications Pairs-Pairs

Nous allons maintenant aborder les commandes entre pairs. Ces commandes permettent l'échange de parties de fichiers directement entre les pairs. Cette phase ne peut se faire qu'après avoir collecté les informations nécessaires auprès du tracker.

5.3.1 Interested

La commande `interested` sert à indiquer à un autre pair que l'on souhaite connaitre son l'état de son buffermap en vu d'un téléchargement futur. Elle ne peut être envoyée que si l'on connait déjà l'adresse des pairs qui possèdent tout ou partie d'un fichier correspondant à la clé utilisée. Ces adresses sont obtenues via une commande `getfile` transmise au tracker précédemment.

- commande : `interested $Key`

- réponse : `have $Key $BufferMap`

La commande est tapée au shell puis après parsing, la méthode `handleInterested` est appelée. Cette méthode récupère les buffermaps des autres pairs en demandant au DownloadManager de le faire. Pour cela le pair appelle la méthode `getBufferMapFromPeers` qui enverra la commande `interested` à tous les pairs pour lesquels il sait qu'ils possèdent le fichier (Cet envoi est effectué par le Socket TCP qui relie un pair à un autre). Les réponses sont ensuites analysées pour mettre à jour la liste des buffermaps des autres pairs. Cette liste se trouve dans l'objet `OtherPeers` décrit précédemment en section 5.1.7.

5.3.2 Getpieces

La commande `getpieces` permet de demander des parties de fichiers, et donc ce procéder au téléchargement. A l'instar de la commande `interested`, il faut connaitre l'adresse IP et le port d'un pair possédant les parties du fichiers avant de pouvoir envoyer la requête.

- commande : `getpieces $Key [$Index1 $Index2 $Index3 ...]`
- réponse : `data $Key [$Index1:$Piece1 $Index2:$Piece2 $Index3:$Piece3 ...]`

Avec les commandes précédentes, nous savons quel pair a chaque morceau (et à quel adresse IP et quel port l'autre pair écoute), et nous pouvons demander à ce pair les données. Encore une fois, la classe `OtherPeers` est très utile ici, car nous pouvons simplement lui demander "quel paire a le fichier associé à cette clé" (C'est exactement ce que fait la fonction `getIPPortFromKey`) et ensuite on peut demander les morceaux que nous voulons.

Dans notre architecture, le responsable de la connexion et du téléchargement des données est la classe `DownloadManager` qui ouvrira une connexion TCP avec le `MultiThreadServer` de l'autre pair. Celui-ci créera une thread responsable de l'obtention des données du fichier et répondra ensuite au pair qui a demandé les données.

5.3.3 Have

La commande `have` est envoyée à la réception de nouvelles parties de fichier aux autres pairs afin de les informer du changement de buffermap.

- commande : `have $Key $BufferMap`
- réponse : `have $Key $BufferMap`

Cette commande est réalisée avec l'appel à la méthode `sendHaveToAll` du DownloadManager. La commande est envoyée à tous les pairs ayant déjà montré leur intérêt pour la clé du fichier. Ceux-ci réponde alors eux aussi avec leur buffermap pour ce fichier, ce qui permet leur mise à jour pour le pair local. Les autres pairs reçoivent et traitent la requête via leur MultiThreadServer.

6 Utilisation d'un tracker à distance

Après avoir eu la première version de notre projet qui fonctionnait localement, nous voulions aller plus loin : avoir notre système pour fonctionner sur Internet, avoir tous les défis de l'utilisation d'un vrai réseau.

Notre idée initiale était de déployer notre tracker sur un serveur distant et ensuite chacun de nous utiliserait un pair pour se connecter entre nous. Le principal problème de cette approche est que nous utilisons le protocole NAT et que nous n'avons pas accès à notre routeur pour ouvrir directement un port. Nous avons donc décidé de changer un peu notre approche.

Au lieu d'utiliser un pair dans chacun de nos ordinateurs, nous avons le tracker et un pair dans un endroit distant (mais ils sont tous deux dans les mêmes sous-réseaux non masqués par la NAT) et nous pouvons utiliser les pairs dans nos ordinateurs pour télécharger des données de ce pair distant.

6.1 AWS EC2

Afin de déployer notre application, nous avions besoin d'une machine distante sur laquelle nous pouvions configurer notre application pour qu'elle fonctionne. En cherchant des fournisseurs, nous avons trouvé Amazon Elastic Compute Cloud (AWS EC2). EC2 est un service Web qui fournit une capacité de calcul sécurisée et redimensionnable dans le cloud, ce qui signifie que nous sommes en mesure de déployer notre tracker et un pair sur différents IPS/ports et de nous y connecter à distance.

6.2 Mise en œuvre

En effet, pour atteindre cet objectif, on crée une instance EC2 pour déployer le tracker, on a plusieurs choix pour le système d'exploitation à utiliser sur l'instance, on choisit Ubuntu puisqu'on est familiarisé à l'utiliser. Après la création de l'instance on crée un fichier .pem sur la console AWS qui sera la clef pour s'identifier pendant la connexion en SSH. En appliquant ces étape on est prêt à lancer le traker puisqu'il suffit de cloner le projet et de le compiler avant de le lancer. Mais même si notre traker va s'exécuter sans problèmes on n'arrivera pas à interagir avec lui de l'extérieur à cause des règles de sécurité de AWS, donc il faut revenir au console et ajouter une nouvelle règle entrante qui va concerner le port où le traker et en train d'écouter et définir le type du protocole (dans notre cas Custom TCP), ainsi le traker est prêt à l'utilisation.
De même façon on peut créer une instance EC2 pour déployer un peer puisque nos ordinateurs personnels ne peuvent pas être accessible de l'extérieur du réseau local sans configurer le router (qui est impossible dans notre cas puisqu'on utilise Wifi fournit par le campus).

Faire cette partie supplémentaire a été vraiment enrichissant pour le groupe. En utilisant l'outil EC2, une grande partie des connaissances de réseau ont été mise à l'épreuve: Nous avons dû comprendre la configuration DNS de base de l'AWS EC2, nous avons dû utiliser des sous-groupes de réseaux AWS pour ouvrir les bons ports d'accès, enfin nous avons dû comprendre pourquoi notre connexion P2P ne fonctionnait pas (problème de NAT). Nous avons alors conclu que même non obligatoire, c'était une partie très intéressante du projet, qui nous a fait comprendre encore plus dans la pratique plusieurs concepts.

7 Conclusion

Finalement, notre équipe est assez satisfaite du travail accompli et de l'implication de chacun de ses membres. Les conditions dans lesquelles nous avons travaillé étaient assez difficiles, mais nous estimons que nous avons bien relevé le défi.
Ce projet nous a permis de mieux nous familiariser avec les différents concepts vus en cours de réseaux et de les mettre en application à travers l'implémentation de cette application. En ce qui concerne la gestion de ce projet, nous avons pu assurer une bonne communication interne, un bon rythme de travail et un bon avancement global du projet malgré les circonstances exceptionnelles que nous avons vécues.