



APPLICATIONS TCP-IP

APPLICATION D'ÉCHANGE DE FICHIERS EN PAIR À PAIR  
(FILESHARE)

RAPPORT INTERMÉDIAIRE DE PROJET

*Johan* CHATAIGNER  
*Emeric* DUCHEMIN  
*Laurent* GENTY  
*Dylan* HERTAY  
*Lucas* TROCHERIE

Dépôt Thor n° free-REZO

Sous la responsabilité pédagogique de  
Tidiane SYLLA

Année 2019-2020 - Semestre 8

## 🗨️ Objectifs du rapport

🗨️ Ce rapport est un rapport intermédiaire de projet ayant pour but d'expliciter l'état d'avancement de celui-ci pour le groupe 3-1. Le but du projet est de réaliser un réseau d'applications **peer** travaillant en collaboration pour se partager et télécharger des fichiers, le tout sous le contrôle d'un **tracker**. La liste des spécifications est donnée dans le sujet du projet.

## 1 🗨️ Organisation et méthodologie

L'organisation de ce projet se fait à l'aide de l'outil *Trello*, qui permet de découper le projet en tâches, répartir ces tâches à des membres de l'équipe mais aussi de suivre l'avancement de celles-ci. A cet outil s'ajoutent des réunions vocales régulières sur le serveur *Discord* de projet de réseau, où l'avancement de chaque membre et l'avancement global dans le projet sont étudiés en collaboration avec M.Tidiane Sylla, notre référent pour ce projet. Ces réunions permettent également de remplir le tableau *Trello* avec les affectations des membres à des tâches ainsi que l'ajout de nouveaux objectifs pour la réussite du projet.

## 2 🗨️ Implémentation actuelle

### 2.1 Tracker

🗨️ de représenter le tracker, nous avons pris une base d'exercice donné en amont du projet ayant le rôle d'un serveur TCP-IP. Dans ce projet, le but est donc de pouvoir mettre en place un serveur qui pourra recevoir des communications TCP grâce à des *sockets*. Le tracker va donc ouvrir un port que l'on donnera en paramètre et accueillir les pairs qui vont s'y connecter. Le but ici est donc de travailler avec les types de *sockets* en C :

- `struct sockaddr_in` pour représenter une *socket*
- écriture dans un descripteur de fichier (*socket*)

A savoir, que dans notre cas ici, la *socket* a une **famille** et un **type** qui ne changent pas : il s'agit d'une *socket* de la famille `AF_INET` et de type `SOCK_STREAM`. En effet, nos communications sont en TCP.

Le tracker va donc devoir être compilé et exécuté comme sur la Figure 1 : `build/tracker <port>`. De fait, une connexion va s'ouvrir au port donné. Si aucun port n'est donné, alors le fichier `config.ini` sera lu et le port par défaut sera celui de ce fichier. En revanche, si aucun port n'est donné et que le fichier n'est pas rempli ou n'existe pas, alors le programme s'arrête.

```
root@DESKTOP-UFPIR05:/mnt/d/Documents/Enseirb/free-Rezo/tracker# build/tracker 10000
THPOOL_DEBUG: Created thread 0 in pool
THPOOL_DEBUG: Created thread 1 in pool
THPOOL_DEBUG: Created thread 2 in pool
THPOOL_DEBUG: Created thread 3 in pool
THPOOL_DEBUG: Created thread 4 in pool
Network Interface Name :- eth0
```

FIGURE 1 – Lancement du tracker

Comme nous pouvons le voir, plusieurs *threads* sont créées par le tracker. Nous avons utilisé la bibliothèque de `threadpool`. De fait, nous créons 5 *threads* (valeur qui pourra être changée par la suite par une règle de pré-compilation). Chaque *thread* va attendre une tâche qui va lui être donnée : quand

le tracker recevra une nouvelle connexion depuis un pair, il va ajouter "du travail" aux *threads* qui vont ensuite piocher la connexion entrante et la traiter depuis le départ.

Le tracker gère donc seulement l'ajout des connexions à l'ensemble du travail à faire. Une routine d'un *thread* ne peut avoir qu'un seul argument passé en paramètre et c'est donc pourquoi nous avons fait le choix de modéliser une connexion par un structure `struct socket_ip` :

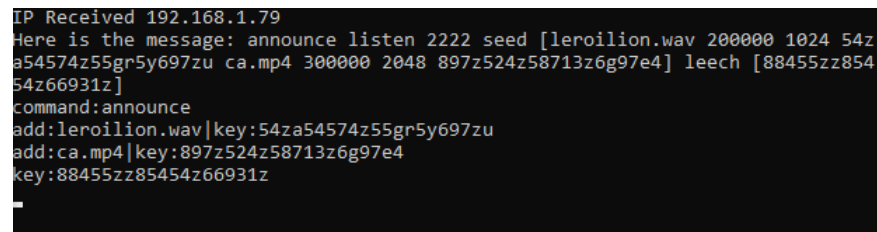
```
1 | typedef struct socket_ip {
2 |     int sockfd; /* Description de la socket */
3 |     char* ip; /* adresse ip du pair */
4 | } socket_ip;
```

De fait, nous pouvons donner à chaque *thread* l'accès à la *socket* pour pouvoir écrire dessus par exemple, mais aussi à l'adresse IP du pair dans la mesure où les ajouts dans notre table de hashage se font grâce à cette donnée.

Un *thread* va donc prendre la tâche à faire et la traiter. Pour ce faire, nous avons mis en place un *parseur* qui va lire la commande donnée et effectuer le traitement associé :

- announce : ajouter les fichiers *seed* dans notre table de hashage au port associé
- look : rechercher des fichiers selon des critères (nom et taille)
- getfile : récupérer une liste de pairs possédant le fichier associé à la clé donnée
- update : ajouter les nouveaux fichiers dans la table de hashage

**Announce** Comme nous pouvons le voir sur la Figure 2, le tracker reçoit la première commande d'un pair, affiche l'IP entrante et va décomposer les paramètres.



```
IP Received 192.168.1.79
Here is the message: announce listen 2222 seed [leroilion.wav 200000 1024 54z
a54574z55gr5y697zu ca.mp4 300000 2048 897z524z58713z6g97e4] leech [88455zz854
54z66931z]
command:announce
add:leroilion.wav|key:54za54574z55gr5y697zu
add:ca.mp4|key:897z524z58713z6g97e4
key:88455zz85454z66931z
```

FIGURE 2 – Announce d'un pair

Si *seed* est donné en paramètre alors il faudra mettre les arguments entre crochets. Attention, le format est stricte, 4 arguments doivent être donnés **par ajout** :

- nom du fichier
- taille du fichier
- taille d'envoi de pièce du fichier
- clé du fichier

On pourra enchaîner les fichiers entre les crochets, en revanche il faut faire attention : si deux fichiers sont en *seed* : le premier est composé d'assez d'arguments mais pas le deuxième, **le premier sera quand même ajouté à la table de hashage** et un message d'erreur s'affichera sur le terminal de la connexion entrante.

De plus, si une taille (de fichier ou d'une pièce) n'est pas un entier, alors un message d'erreur s'affichera aussi : seuls les entiers sont autorisés pour ces paramètres.

Lors de la lecture, les ajouts et données importantes seront affichés sur le terminal du tracker, mais aussi dans le fichier `log` qui nous permettra de garder une trace de l'exécution du serveur TCP.

Si tout s'est bien passé, un seul message `> ok` s'affichera.

**Look** Sur la Figure 3, on peut voir qu'un pair demande à trouver un fichier en particulier. Le *thread* va chercher dans sa table de hashage un fichier aux critères donnés :

— nom : **seulement égalité**

— taille en octets : **<, > et égalité**

Si un fichier (ou plusieurs) correspond à la demande, alors un message de type :

> list [filename1 length1 piecesize1 key1 ...]

```
IP Received 192.168.1.79
Here is the message: look [filename="ca.mp4" filesize>"1000"]
command:look
filename:ca.mp4
size:1000
comparator:>
```

FIGURE 3 – Look d'un pair

**Getfile** En ce qui concerne cette commande sur la Figure 4, une simple recherche dans la **table de hashage** va s'effectuer.

```
IP Received 192.168.1.79
Here is the message: getfile 897z524z58713z6g97e4
command:getfile
key:897z524z58713z6g97e4
```

FIGURE 4 – Getfile d'un pair

**Update** Pour la Figure 5, le pair va notifier le tracker des nouveaux fichiers qu'il a en sa possession et des nouveaux fichier qu'il leech.

```
IP Received 192.168.1.79
Here is the message: update seed [897z524z58713z6g97e4] leech []
command:update
add seed:897z524z58713z6g97e4
no key
```

FIGURE 5 – Update d'un pair

Sur le terminal du tracker, nous voyons qu'il apparaît les nouveaux fichiers *seed* (on peut en mettre plusieurs à la suite). Il faut faire attention car l'update concerne les fichiers qui sont déjà présents dans la table de hashage. En effet, vu que l'on donne nos fichiers au début de notre connexion avec notamment la taille du fichier mais aussi la taille d'un envoi de pièce, il faudrait aussi rajouter ces informations dans cette commande pour pouvoir traiter ce cas.

A **l'heure actuelle ce n'est pas le cas**, et donc la mise à jour de nouveaux fichiers ne marche que si les clés existent déjà.

Les *threads* **possèdent une plus ou moins flexibilité** concernant les commandes effectuées : des espaces peuvent être disposés à droite à gauche en double entre les arguments cela marchera quand même. Le seul point à vraiment respecter est le nombre de paramètres à donner : multiple de 4 pour announce par exemple.

## 2.2 Stockage des fichiers : table de hashage

Nous avons stocké les données dans une table de hashage dans laquelle nous stockons des **fichiers** sous forme de structure.

```
1 struct file {
2     char *key; /* clef "unique" */
3     char *name; /* nom du fichier */
4     int length; /* longueur totale du fichier */
5     int piecesize; /* taille de la piece */
6     int nb_owners; /* nombre total de possesseur */
7     SLIST_HEAD(, owner) owners; /* liste des possesseurs */
8
9     SLIST_ENTRY(file) next_file; /* le pointeur vers le prochain element */
10 };
```

La table de hashage est un tableau dont l'indice correspond à une clé et celle-ci est faite grâce à une fonction de hashage prenant en paramètre la clé donnée par le *peer* (md5). Comme une fonction de hashage peut provoquer des collisions c'est-à-dire que pour une clé différente elle peut renvoyer le même indice. **Du coup, chaque tableau contient une liste de fichiers.** Nous avons choisi d'utiliser la bibliothèque *bsd queue* qui implémente des structures et permet de manipuler des listes facilement. Nous avons choisi des listes simplement chaînée car vu qu'il n'était pas nécessaire de supprimer des éléments (ce n'est pas mis dans l'énoncé que nos pairs peuvent supprimer leur fichiers) mais juste de lire et ajouter des éléments ce qui suffisait amplement.

Cette table de hashage réduit grandement le temps de recherche des fichiers par le *tracker* lorsqu'on lui donne la clé. Cependant lorsque le *tracker* a besoin de rechercher les fichiers portant un certain nom ou ayant une taille spécifique alors on perd son efficacité et il faut alors rechercher dans tous les fichiers présents dans la table de hashage.

La création ainsi que la mise à jour des fichiers s'effectuent tous deux grâce à la fonction `hash__add()` qui modifie la valeur des éléments à l'intérieur du fichier. Plus particulièrement, en ce qui concerne les possesseurs de fichier, il y a une structure dédiée :

```
1 struct owner {
2     char* IP; /* adresse IP du possesseur du fichier*/
3     int port; /* port d'ecoute */
4     SLIST_ENTRY(owner) next_owner; /* prochaine element de la liste*/
5 };
```

Le stockage des **peers** possédant le fichier de clé *key* s'est fait sous forme de liste aussi car nous ne savons pas la quantité de *peer*. De plus, un **peer** peut avoir un fichier et le partager sur plusieurs ports **du coup** nous avons considéré qu'un propriétaire de fichier était le couple : IP + port. La modification et la lecture de ces données est beaucoup plus simple à obtenir, en effet, il suffit de parcourir la liste et s'il y manque un élément on peut l'ajouter.

## 2.3 Peer

L'application **Peer** est écrite en **Java** et possède plusieurs parties. Cette application doit répondre à plusieurs attentes, elle doit à la fois pouvoir communiquer avec le Tracker présent dans le réseau et avec les autres peers dans le même réseau.

### 2.3.1 Communication avec le Tracker

Nous avons implémenté la fonction de communication entre le Tracker et le pair à l'initialisation, annonce. Nous avons pour l'instant un **pair** qui annonce les fichiers dont il dispose dans un dossier seed. Ce dossier est au même niveau que notre exécutable dans l'arborescence. Lorsque le tracker est mis en route, il communique donc avec le tracker et lui envoie immédiatement tout le contenu de ce dossier. **Cet** implémentation sera discutée dans le paragraphe Améliorations et Objectifs futurs.

**Announce** En ce qui concerne cette commande sur la Figure 6, on va annoncer le port sur lequel on écoute ainsi que le fichier que l'on possède plus précisément le titre, la taille, le nombre de parties et la clef. Le tracker va aussi obtenir notre adresse IP en même temps.

```
Please enter the message: announce listen 1111 seek [toto 1200 2 123456789]
> ok
```

FIGURE 6 – Announce d'un pair

**Look** Cette commande sur la Figure 7 va demander au tracker les fichiers ayant les critères demandés. On obtient alors le titre du fichier, la taille du fichier, les parties de ce fichier ainsi que la clef du fichier. On pourra alors utiliser la commande *getfile* pour le télécharger par la suite.

```
Please enter the message: look [filename="toto" filesize>"1000"]
> list [toto 1200 2 123456789]
```

FIGURE 7 – Look d'un pair

**Getfile** Cette commande sur la Figure 8 va demander au tracker les peers possédant ce fichier afin de pouvoir le télécharger ultérieurement. On va donc recevoir l'IP et le port des peers possédant le fichier.

```
Please enter the message: getfile 123456789
> peers 123456789 [192.168.1.24:2222 192.168.1.24:1111]
```

FIGURE 8 – Getfile d'un pair

### 2.3.2 Communication entre Peers

Il y a différentes manières de communiquer pour un peer avec un de ses pairs. Il peut le faire avec les commandes suivantes :

1. *interested* : cette commande permet d'indiquer à un autre peer son intérêt pour un fichier. La réponse indique quel partie du fichier le peer a en mémoire.
2. *getpieces* : cette commande permet de demander des pièces à un autre peer
3. *have* : cette commande permet périodiquement à un peer d'indiquer aux autres peers présent sur le réseau l'évolution du téléchargement d'un fichier

Pour l'instant, tout les fichiers qui sont destinés au téléchargement sont stockés dans un fichier *seed* situé à la racine du dossier **peer**. Le Peer lance un thread qui s'occupe de la classe **Buffermap**. A l'heure actuelle, le buffermap est un tableau de booléen. Cette implémentation est loin d'être optimale mais permet de manipuler le buffermap de manière simple.

Le programme enregistre les fichiers qu'il a en sa possession dans une liste sous la forme de couple (hashMd5, buffermap), appelée **fileManager**. Cette liste est ensuite mise à jour au fur et à mesure des requêtes.

Pour ce qui est de l'analyse des requêtes, le peer lance pour l'instant un thread à part qui écoute sur un port, analyse les requêtes qu'il reçoit et effectue les modifications correspondantes dans le *fileManager*. Cette dernière fonctionnalité n'est pas encore complètement implémentée mais est très proche de ce stade.

De même, pour l'instant, le *peer* lance un nouveau thread de manière systématique pour ouvrir une socket sur un port et envoyer une requête à un autre thread.



### 3 Problèmes rencontrés

Un des **majeurs problèmes** rencontrés est le fait que durant la phase de tests nous sommes à distance les uns des autres. Il nous est donc très difficile de faire communiquer nos applications entre elles. Nous arrivons à les faire tourner en local, mais nous ne maîtrisons pas les communications au travers de la France **=)**.

D'autres problèmes se situent notamment au niveau de l'écoute du tracker avec le nombre de connexions maximales. En effet, nous avons rencontrés quelques problèmes concernant l'accès concurrentiel des pairs à notre tracker car notre tracker n'accepte que les pairs en local et donc tester les accès concurrents est difficile.

## 4 Améliorations et objectifs futurs

### 4.1 Peer

Nous souhaiterions mettre en place un environnement ou une interface graphique qui permettraient d'interagir avec le pair. Cet interface permettrait par exemple de fixer les différentes valeurs, telles que le numéro de port ou l'Ip du tracker, ou encore de demander l'envoi de requête par exemple aux pairs ou au tracker. Cette interface fournirait une abstraction pour les utilisateurs, et ce serait un outil de débogage utile. Elle permettrait aussi à terme d'afficher les logs écrits par l'exécution de notre programme.

Nous n'avons pas encore implémenté certaines fonctions de communications des pairs. Ainsi il faudrait encore que nous codions les fonctions look (dont nous avons déjà les structures de traitement interne prêtes), la fonction qui permettrait de périodiquement annoncer l'état de nos fichiers au tracker. Il nous faudrait alors coder une routine en java qui s'effectuerait tout les x temps.

Nous souhaiterions aussi mettre en place un système de stockage de l'état de fichier. Nous pourrions par exemple mettre en place un système d'écriture dans un fichier .dat où le lieu du fichier, son état, les différentes pièces dont on dispose, sa taille..., serait stockés. On n'aurait alors pas besoin d'alourdir notre programme avec des structures de données de stockage, et on pourrait de plus garder, d'un lancement à l'autre, les informations relatives à un fichier.

Pour le moment nous faisons un announce sur tous les fichiers présents dans le dossier seed de la racine. Nous voudrions pouvoir proposer à l'utilisateur de choisir les fichiers de sa machine qu'il laisse à disposition du tracker et des autres utilisateurs.

### 4.2 Tracker

Actuellement concernant la connexion d'un pair, sont stockés : la socket (entier) et l'adresse IP. Chaque *thread* sera par la suite délégué à devoir *parse* la commande rentrée. Cependant, dans le cas d'un *update*, on donne seulement les nouvelles clés que l'on possède par exemple. A aucun moment nous avons connaissance du port dans lequel sont disponibles les nouveaux fichiers.

Le prochain objectif est donc de faire en sorte de pouvoir garder en mémoire le port donné dans la première commande durant la connexion (*announce*) et potentiellement le réutiliser quand un *update* sera fait ou bien nous devons imposer le fait de donner un port dans la commande *update*, cependant cela changerait le cahier des charges.

Cependant, nous avons considéré qu'un propriétaire de fichier était un couple "tait l'adresse IP et port ce qui implique qu'une même adresse IP peut avoir plusieurs ports (donnés dans des *announce* différents) et donc il faudra aussi faire le choix d'en choisir un parmi les différentes possibilités.

De plus, lors des *leech* on ne fait rien. On n'ajoute pas l'utilisateur en tant que propriétaire en devenir du fichier. En effet, à l'heure actuelle, savoir que tel pair est en train de télécharger un fichier

ne nous apporte rien dans la mesure où quand il l'aura téléchargé complètement, il le notifiera grâce à la commande *update*.

Nous allons discuter afin de savoir si ajouter le pair en tant que propriétaire en devenir est une bonne chose dans la mesure où cela nous ferait gagner du temps dans la mesure où nous ne l'ajoutons pas par la suite.

En revanche, cela impliquerait de changer la structure d'un fichier et donc de notifier que tel utilisateur a le fichier en cours de téléchargement (donc il le possède partiellement) mais pas encore prêt à être téléchargé par d'autres pairs.

Une autre amélioration concernant le tracker serait de nettoyer le *parseur* dans la mesure où le code peut être grandement simplifié afin d'éviter les variables inutiles. De plus, à l'heure actuelle, l'écriture concurrente est très basique (écriture des messages dans l'ordre d'affichage) en revanche, par la suite, l'affichage sera plus fourni avec notamment l'heure et le pair ayant réalisé la commande.

Une autre amélioration qui semble importante, serait de gérer le cas de la connexion non locale. En effet, comme nous l'avons dit précédemment, les pairs peuvent se connecter au tracker seulement en local et le tracker n'accepte pas encore les connexions externes au réseau. Le but ici serait de pouvoir obtenir l'adresse publique de notre machine (si c'est faisable) en C, et d'accepter les connexions externes.

Enfin, nous écrirons un panel de test au fur et à mesure du projet permettant de vérifier que nos fonctionnalités de la table de hashage et les commandes entrées par le client soient bonnes. Actuellement, nous avons implémentés des tests sur la table de hashage qui vérifie chacune des fonctions dans le fichier *hash\_table.c*.