



APPLICATIONS TCP-IP

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

RAPPORT FINAL 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

Table des matières

1	Organisation et méthodologie	2
2	Implémentation finale	2
2.1	Architecture globale	2
2.2	Tracker	3
2.2.1	Sockets	3
2.2.2	Exécution et threads	3
2.2.3	Implémentation des commandes	4
2.2.4	Stockage des fichiers : table de hashage	6
2.3	Pair	8
2.3.1	Communication avec le tracker	8
2.3.2	Communication entre pairs	9
2.3.3	Structure de données	10
2.3.4	Interface graphique	11
3	Évaluation et tests	13
4	Documentation	14
5	Problèmes rencontrés	14
6	Améliorations et objectifs futurs	15
6.1	Pair	15
6.2	Tracker	15
7	Conclusion	15

Introduction

Ce rapport est le rapport final de projet **FileShare** Pair à Pair pour le groupe 3-1.

Le but du projet est de réaliser un réseau d'applications **pair** travaillant en collaboration pour se partager et télécharger des fichiers, le tout sous le contrôle d'un **tracker**. Une liste des spécifications a été fournie et nous allons l'explicitier tout au long de notre rapport.

1 Organisation et méthodologie

L'organisation de ce projet s'est faite à l'aide de l'outil *Trello*, qui permet de découper le projet en tâches, de répartir ces tâches à des membres de l'équipe et de suivre l'avancement de celles-ci.

Le découpage a été fait comme suit :

- Laurent, Dylan et Johan se sont occupés du développement du **Tracker**
- Emeric et Lucas se sont occupés du développement du **Pair**

Le *Trello*, maintenu à jour toutes les semaines, recense donc toutes les fonctionnalités en cours de développement sur chacun des modules mais aussi celles qui ont déjà été faites.

A cet outil s'ajoutent des réunions vocales régulières sur le serveur *Discord* de projet de réseau, dans lesquelles 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 et d'ajouter de nouveaux objectifs pour la réussite du projet.

2 Implémentation finale

Dans cette partie nous allons aborder les choix que nous avons fait pour le rendu final et l'architecture adoptée. Ces choix seront justifiés afin de mettre en avant les avantages que notre implémentation possède par rapport à d'autres.

2.1 Architecture globale

Le projet se divise en deux grandes parties : **tracker** et **pair**. C'est pourquoi l'arborescence à la racine du projet est la suivante :

```
free-Rezo/  
├── README.md  
├── tracker/  
└── peer/
```

Ci-dessous les arborescences respectives de ces deux parties :

```

tracker/
├── dox
├── Makefile
├── src/
│   ├── hash_table.c
│   ├── hash_table.h
│   ├── port_table.c
│   ├── port_table.h
│   ├── queue.h
│   ├── thpool.c
│   ├── thpool.h
│   ├── tracker.c
│   ├── tracker.h
│   ├── utils.c
│   └── utils.h
└── tests/
    ├── test_hash_table.c
    └── test_port_table.c

```

```

peer/
├── Makefile
├── config.ini
├── seed.dat
└── src/
    ├── AddFileSender.java
    ├── AnnounceToTracker.java
    ├── ButtonListener.java
    ├── ConfigTrackerSender.java
    ├── DatFileParser.java
    ├── FileManager.java
    ├── GetFileTracker.java
    ├── GetPiecesPeer.java
    ├── GetPiecesSender.java
    ├── HavePeer.java
    ├── InterestedPeer.java
    ├── PeerConfig.java
    ├── ReceiveFromPeer.java
    ├── SendToPeer.java
    ├── Sender.java
    ├── UpdateAnnounce.java
    ├── UpdateHave.java
    ├── buttonListenerSender.java
    ├── gui.java
    ├── guiRunner.java
    ├── loading.java
    └── lookTracker.java

```

2.2 Tracker

Un **tracker** est un serveur qui aide à la communication entre les **pairs**. Dans le partage de fichiers en pair à pair, un pair (client) va envoyer une requête au *tracker* afin de savoir où peut se trouver le fichier qu'il recherche et également indiquer au *tracker* les fichiers qu'il possède.

2.2.1 Sockets

Ici, 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 spécifiera 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, 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`.

2.2.2 Exécution et threads

Le *tracker* va donc devoir être compilé et exécuté comme sur la Figure 1 : `build/tracker <port>`. De fait, une connexion va s'ouvrir sur le 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 sur la Figure 1, plusieurs *threads* sont créés 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 processus principal du *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 une structure `struct socket_ip` détaillée sur la Figure 2 :

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

FIGURE 2 – Structure de socket-ip

De fait, le *tracker* possédant une seule donnée de type `struct socket_ip` va pouvoir la donner aux *threads* dans la routine afin qu'ils la traitent.

Ainsi, chaque *thread* peut avoir 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.

2.2.3 Implémentation des commandes

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 donné
- **update** : ajouter les nouveaux fichiers dans la table de hashage

Announce Comme nous pouvons le voir sur la Figure 3, le *tracker* reçoit la première commande d'un pair, affiche l'IP entrant et va décomposer les paramètres.

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

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

```

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 3 – Commande **Announce** d'un pair

— clé du fichier

On pourra enchaîner les fichiers entre les crochets, en revanche il faut faire attention : si deux fichiers sont en *seed* : si 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 importants 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 Le pair a la possibilité de questionner le *tracker* en lui transmettant des critères et le *tracker* lui répondra avec une liste de fichiers satisfaisant ces critères. Sur la Figure 4, on peut voir qu'un pair demande à trouver un fichier remplissant différents critères tels que :

— nom : **seulement égalité**

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

Pour ce faire, nous allons effectuer une recherche sur la liste de fichiers en comparant nos critères aux données. Si un fichier (ou plusieurs) correspond à la demande, alors un message du type suivant sera envoyé :

```
> 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 4 – Commande **Look** d'un pair

Getfile De plus, un pair peut demander au *tracker* de récupérer les informations relatives à un fichier en particulier, notamment le port et l'IP des pairs associés.

Comme montré sur la Figure 5, 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

```

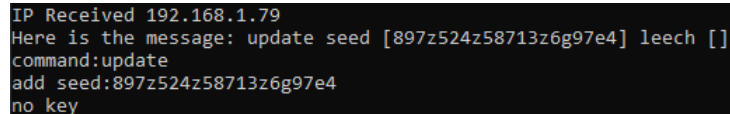
FIGURE 5 – Commande **Getfile** d'un pair

La valeur de la clé donnée en paramètre va être *hashée*, et pour chaque fichier dans cet index, le *tracker* va trouver le fichier correspondant à la clé donnée et renvoyer la liste de *seeders* (de pairs possédant le fichier) en tant que réponse.

Si le fichier n'existe pas, nous allons renvoyer un message d'erreur au pair demandant.

Update Dès lors qu'un pair se connecte au *tracker*, il annonce des nouveaux fichiers. Périodiquement, il va pouvoir faire des mises à jour au *tracker* en lui notifiant des fichiers qu'il possède à intervalles réguliers (intervalles indépendants entre chaque pairs).

Le pair va donc annoncer au *tracker* les fichiers qu'il possède et ceux qu'il télécharge. Un exemple est montré sur la Figure 6 dans lequel le pair va notifier le *tracker* des nouveaux fichiers qu'il a en sa possession et des nouveaux fichier qu'il télécharge.



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

FIGURE 6 – Update d'un pair

Sur le terminal du *tracker*, les nouveaux fichiers *seed* apparaissent (on peut en mettre plusieurs à la suite). Il faut faire attention car la commande *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.

Nous avons donc créé un dictionnaire (représenté grâce à une table de hashage) :

- clé : IP
- valeur : port

Et ce dictionnaire permet de recenser les couples (uniques) IP:port, par défaut, grâce à la première connexion de chaque pair. En effet, comme nous l'avons dit plus tôt, quand un pair effectue la commande *update*, il donne seulement les clés et donc les informations concernant le fichier doivent déjà exister.

Nous avons donc créé la structure représentée sur la Figure 7 nous permettant de gérer les ports par défaut :

```
1 struct peer {
2     char* IP; /** adresse IP du pair */
3     int port; /** port par défaut stocke durant le premier announce */
4     SLIST_ENTRY(peer) next_peer; /** prochain pair dans le dictionnaire */
5 };
```

FIGURE 7 – Structure de port par défaut

De fait, dès lors qu'un pair se connecte grâce à un *announce*, le port avec lequel il se connecte sera considéré comme son port par défaut et sera rajouté dans cette structure. Quand il voudra rajouter de nouveaux fichiers grâce à un *update*, il s'agira du port trouvé s'il est dans le dictionnaire des ports par défaut.

Les *threads* possèdent une flexibilité concernant le format des commandes effectuées : le *parsing* est assez permissif concernant la syntaxe (les espaces peuvent être doublés par exemple). Le seul point à vraiment respecter est le nombre de paramètres à donner : il doit être multiple de 4 pour la commande *announce* par exemple (le format est rappelé en commentaire de chaque fonction).

2.2.4 Stockage des fichiers : table de hashage

Nous avons stocké les données dans une table de hashage dans laquelle les fichiers sont conservés sous forme de structure comme sur la Figure 8 :

Nous avons fait le choix de représenter chaque fichier par une structure dans la mesure où il est plus facile de manipuler des pointeurs de *struct* et de pouvoir ensuite les échanger, les modifier, itérer dessus, ...

```

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(, seeder) seeders; /** liste des possesseurs */
8     LIST_HEAD(, leecher) leechers; /** liste des personnes qui le telecharge */
9     SLIST_ENTRY(file) next_file; /** le pointeur vers le prochain element */
10 };

```

FIGURE 8 – Structure de fichier

La table de *hashage* est un tableau dont l'indice correspond à une clé et celle-ci est construite grâce à une fonction de *hashage* prenant en paramètre la clé donnée par le *pair* (calculée en **md5**). Comme une fonction de *hashage* peut provoquer des collisions, elle peut pour une clé différente, renvoyer le même indice et par conséquent chaque tableau contiendrait une liste de fichiers. Nous avons choisi d'utiliser la bibliothèque **Queue BSD** qui implémente des structures et permet de manipuler des listes facilement. Nous avons opté pour des listes simplement chaînées car il n'était pas nécessaire de supprimer des éléments mais juste de lire et d'ajouter des éléments.

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. En effet, il faut dans ce cas rechercher parmi tous les fichiers présents dans la table de *hashage*.

La création ainsi que la mise à jour des fichiers s'effectuent toutes 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 représentée comme sur la Figure 9 :

```

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

```

FIGURE 9 – Structure de Seeder

Le stockage des *pairs* possédant le fichier de clé *key* est réalisé sous forme de liste car la quantité de *pair* n'est pas connue à l'avance. De plus, un *pair* peut avoir un fichier et le partager sur plusieurs ports. Nous avons donc 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 un élément manquant peut facilement y être ajouté.

Concernant la gestion des personnes qui téléchargent les fichiers, c'est-à-dire les **leechers**, le même principe que pour la gestion des pairs a globalement été adopté (Figure 10).

```

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

```

FIGURE 10 – Structure de Leecher

Comme nous pouvons le voir sur la Figure 10, cette structure est semblable mais différente à celle des *seeders* : en effet, il a été considéré que lorsqu'un pair télécharge un fichier (quand il est *leecher*), il ne possède **pas encore** le fichier (il n'est pas encore *seeder*). De fait, quand un pair aura fini le téléchargement d'un fichier et l'annoncera au *tracker* grâce à la commande *update* dans le champ *seed*, le pair sera supprimé de la liste des personnes qui téléchargent le fichier donné (on va le supprimer de la liste des *leechers* dans le fichier) et il sera ajouté dans la liste des *seeders*.

Par conséquent, il est préférable d'avoir des listes doublement chaînées en ce qui concerne la liste des *leechers* dans un fichier.

2.3 Pair

L'application **pair** 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 pairs dans le même réseau. De plus, une interface graphique a été implémentée pour faciliter l'interaction entre l'utilisateur et l'application.

2.3.1 Communication avec le tracker

Nous avons implémenté la fonction de communication entre le *tracker* et le pair à l'initialisation, **announce**. Le pair annonce les fichiers qui sont spécifiés dans le fichier **seed.dat**. Ce fichier se trouve à la racine du dossier **peer**. Lorsque le pair est mis en route, il communique donc avec le *tracker* et lui envoie immédiatement tous les fichiers présents dans **seed.dat**, ce sont les fichiers qu'il met à la disposition de la communauté. Cette implémentation sera discutée dans la partie Améliorations et Objectifs futurs (partie 6).

Announce En ce qui concerne cette commande visible sur la figure 11, le pair va annoncer le port sur lequel il écoute ainsi que les fichiers qu'il possède. Plus précisément, il va préciser pour chaque fichier son titre, sa taille, son nombre de parties et sa clé. Le *tracker* va également obtenir par l'envoi de cette commande l'adresse IP du pair.

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

FIGURE 11 – Commande **Announce** d'un pair

Look Cette commande présentée sur la figure 12 va demander au *tracker* les fichiers ayant les critères demandés. Le pair obtient alors le titre du fichier, la taille du fichier, la taille des parties de ce fichier ainsi que la clé du fichier. On pourra alors utiliser la commande **getfile** pour obtenir des informations nécessaires au téléchargement par la suite.

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

FIGURE 12 – Commande **Look** d'un pair

Getfile Cette commande sur la Figure 13 va demander au *tracker* les pairs possédant ce fichier afin de pouvoir le télécharger ultérieurement. Le pair va donc recevoir l'adresse IP et le port d'écoute des pairs possédant le fichier. Cette commande **getfile** est ensuite immédiatement suivie d'un appel à la commande **interested**. De fait, le pair vient d'obtenir les informations requises à la demande de pièces aux pairs, il peut donc passer à la récupération des pièces venant des autres pairs sur le réseau grâce à la commande **getpieces**.

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

FIGURE 13 – Commande **Getfile** d'un pair

2.3.2 Communication entre pairs

De la même manière que pour le *tracker*, les pairs doivent pouvoir communiquer entre eux à l'aide de différentes règles. D'un point de vue architectural, un pair écoute en permanence sur un port comme un serveur, défini sur le port 15000 par défaut. Un pair établit ensuite des connexions temporaires avec d'autres pairs afin de transmettre des commandes, des informations et des morceaux de fichier. Dans cette partie, nous imaginerons deux pairs communiquant entre eux nommés Alice et Bob. Nous nous placerons du point de vue d'Alice.

Interested Cette commande permet à Alice de manifester son intérêt pour un fichier. Elle est envoyée aux pairs présents sur le réseau qui possèdent le fichier en question. Cette commande peut être envoyée spontanément par l'utilisateur, mais elle est également envoyée automatiquement après la réception de la commande **peers** de la part du *tracker* (réponse à la commande **getfile**). En effet après un **getfile**, toutes les informations nécessaires à la communication ont été reçu. Le programme va donc automatiquement montrer son intérêt aux pairs ayant le fichier désiré afin de rendre le processus plus simple pour l'utilisateur. Cependant, s'il y a eu un problème ou que le pair n'était plus connecté, l'utilisateur peut vouloir renvoyer la requête **interested** aux pairs voulus. Il a alors la possibilité de le faire simplement depuis l'interface graphique fournie.

[illegible]FIGURE 14 – Commande **Interested** d'un pair

Un exemple de l'utilisation de la commande **interested** se trouve sur la figure 14, où le *hash* du fichier correspond à celui d'un fichier au format **PDF**. Pour simplifier la visualisation des *buffermaps*, ces derniers sont envoyés sur les figures présentes dans ce rapport sous une forme compréhensible, c'est-à-dire comme une suite de 0 et de 1. Cependant, dans l'application, les *buffermaps* sont envoyés sous forme de **String**, avec une pièce représentée par un seul bit sur le type **char**. Le *buffermap* est donc transmis et apparaît donc comme une suite de caractères **UTF-8** peu compréhensible.

Getpieces Cette commande permet à un pair (Alice) de demander la récupération de morceaux de fichier venant d'un autre pair (Bob). Elle est envoyée au pair de Bob dont Alice est sûr qu'il possède le fichier en question ou partie de ce fichier. Les parties demandées sont également nécessairement possédées par le pair de Bob. De la même manière que ce qui est fait pour **interested**, l'envoi de cette requête se fait automatiquement après la réception de la réponse à la requête **interested**. Néanmoins l'utilisateur conserve la possibilité d'envoyer lui même cette requête directement au pair de son choix sur le port de son choix (s'il rentre des données fausses il n'obtiendra simplement aucun retour exploitable).

[illegible]FIGURE 15 – Commande **Getpieces** d'un pair

Un exemple de l'utilisation de **getpieces** se trouve sur la figure 15, utilisée ici pour récupérer le premier morceau du fichier PDF mentionné dans le paragraphe précédent.

Have Cette commande est envoyée dans deux scénarios différents :

- Le premier est la réponse à la commande **Interested** de Bob, auquel cas Alice répond avec la commande **Have** qui inclut la version de son *buffermap* pour ce fichier.
- Le deuxième est l’envoi périodique de cette commande aux autres pairs présents sur le réseau lors du téléchargement d’un fichier, pour que ceux-ci puissent connaître l’état d’avancement de celui-ci.

[illegible]FIGURE 16 – Commande **Have** d'un pair

Un exemple de l'utilisation de la commande **have** se trouve sur la figure 16, où Alice envoie son *buffermap* concernant le fichier PDF mentionné précédemment à Bob. Bob répond avec son *buffermap*.

Data Enfin, cette dernière commande est une réponse à la commande **Getpieces** envoyée par Bob dans laquelle Alice envoie les pièces demandées par Bob sous forme de **String**. La conversion d'un fichier en String s'est faite à l'aide de la méthode **readAllBytes()** existante nativement en **Java**, et la découpe en séparation et concaténation de chaîne de caractères.

2.3.3 Structure de données

Au lancement de l'application, le pair se déclare au *tracker* et lance un *thread* qui se charge d'écouter sur le port défini par l'utilisateur (ou le port par défaut), à la manière d'un serveur. La base de données est mise à jour dès la réception d'une commande ou dès la réception d'une réponse.

Une classe **fileManager** a été écrite. Cette classe a pour but de stocker toutes les données nécessaires au bon fonctionnement de l'application et à leur manipulation. De manière à ne pas avoir de doublon de classe de stockage de données et également pour que différents *threads* manipulent facilement la base de données. Cette classe a été écrite sous forme de **singleton**. Il n'y a donc qu'une instance de cette classe dans l'application.

Pour être identifié, le **hash** de chaque fichier est calculé grâce à l'algorithme **md5**. A chaque fichier est également associé un **buffermap**. Il s'agit d'une suite de bits dont la longueur correspond au nombre de morceaux présents une fois que le fichier a été découpé. Le *buffermap* est stocké sous la forme d'un tableau de **booleans** dans notre implémentation. Cela est loin d'être idéal mais permet une manipulation simple de l'objet *buffermap*. Dans une commande, le *buffermap* est envoyé sous forme de **String**. L'écriture binaire du *buffermap* correspond alors à l'écriture binaire de la **String**, c'est-à-dire que si un utilisateur venait à écouter les communications entre deux pairs, il verrait une chaîne de caractère **UTF-8**. Dans cette chaîne un bit représente réellement une pièce du fichier.

Au lancement de l'application, 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. Il enregistre également dans une autre liste appelée **fileMatch** les couples (**hashmd5**, **filePath**) de manière à pouvoir retrouver un fichier rapidement. En effet, par exemple, le découpage d'un fichier n'est réalisé que lors de l'envoi d'un ou plusieurs morceaux à un autre pair, il était donc nécessaire de pouvoir rapidement faire un rapprochement entre le *hash* d'un fichier et son emplacement.

Une dernière structure de données a été écrite contenant des couples (**hashMd5**, **peers**), appelée **peerManager**. Cette structure de données a pour but de pouvoir mémoriser quels autres pairs sur le réseau ont totalité ou partie d'un fichier. Un pair est identifié sous forme de **String** par **IP@:port**.

2.3.4 Interface graphique

Pour faciliter les communications entre un pair et un utilisateur, une interface graphique a été écrite en utilisant le *framework* **Swing**. Cette partie de l'implémentation du pair a été très chronophage de par la complexité et l'aspect un peu obscur de ce *framework*.

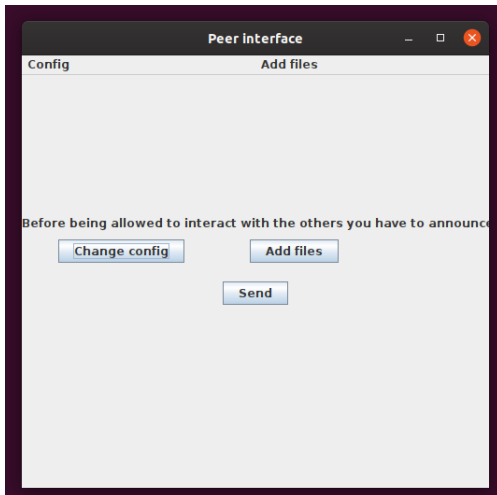


FIGURE 17 – Page de démarrage de l'application

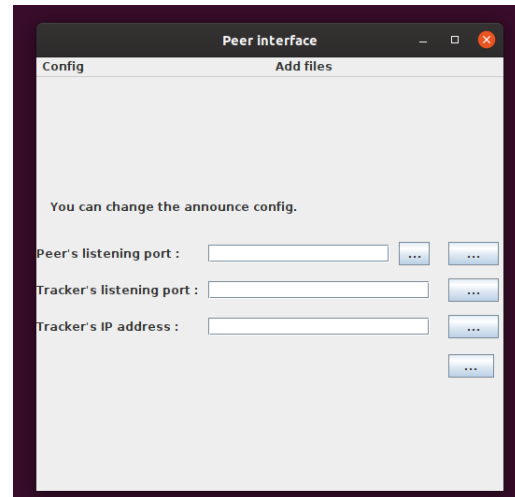


FIGURE 18 – Configuration de l'application

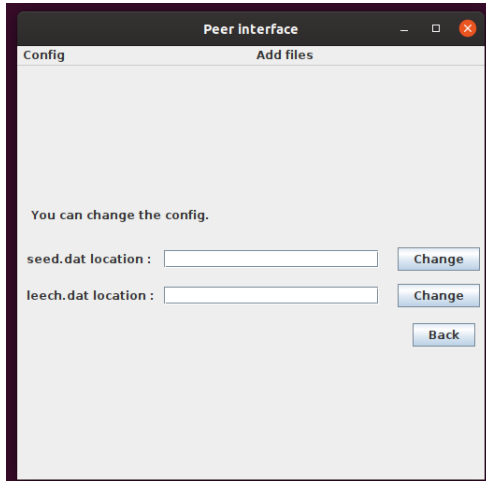


FIGURE 19 – Demande et mise à disposition de fichiers

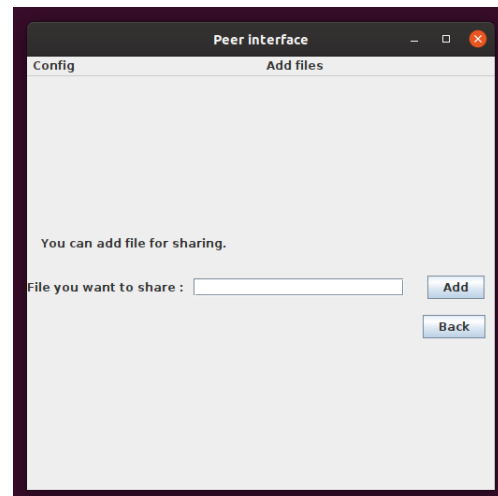


FIGURE 20 – Ajout de fichiers dans l'application

La page de démarrage de l'application est visible sur la figure 17. Plusieurs choix s'offrent à l'utilisateur. Le premier est la possibilité de changer la configuration d'un pair. Le deuxième est la possibilité d'ajouter des fichiers dans la base de données du pair, de manière à enrichir l'ensemble des fichiers partagés. Enfin, l'utilisateur a la possibilité d'appuyer sur le bouton *Send* de manière à se déclarer au *tracker*.

La page de configuration est visible sur la figure 18. L'utilisateur peut changer différents paramètres de l'application.

- Le premier est le port d'écoute du pair. Cette modification est nécessaire pour avoir plusieurs pairs en local.

- Le deuxième est le port d'écoute du *tracker*.
- Le dernier est l'adresse IP du *tracker*, qui doit être connue à l'avance.

L'utilisateur possède également la possibilité sur la page visible sur la figure 19 de configurer l'emplacement des fichiers **seed.dat** et **leech.dat**. Dans ces fichiers se trouvent les chemins relatifs des fichiers mis à disposition par le pair et ceux qui intéressent ce dernier.

Enfin, la page pour ajouter des fichiers dans l'application est visible sur la figure 20. L'utilisateur peut ajouter des fichiers en écrivant le chemin relatif vers ceux-ci.

Une fois que l'utilisateur a appuyé sur le bouton *Send* de la page de démarrage, l'utilisateur est désormais accueilli sur la page d'accueil, visible sur la figure 21.

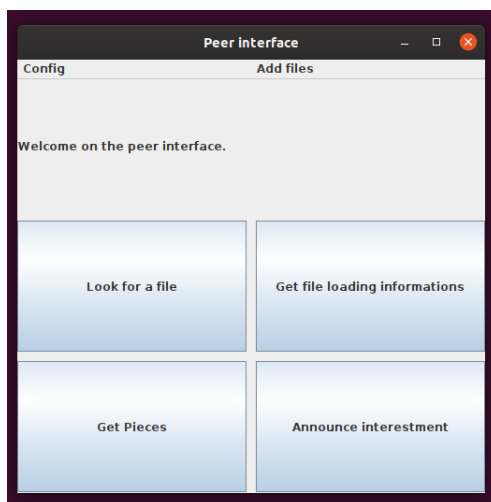


FIGURE 21 – Page d'accueil de l'application

Quatre choix sont offerts à l'utilisateur, chacun correspondant à une commande spécifiée dans le sujet.

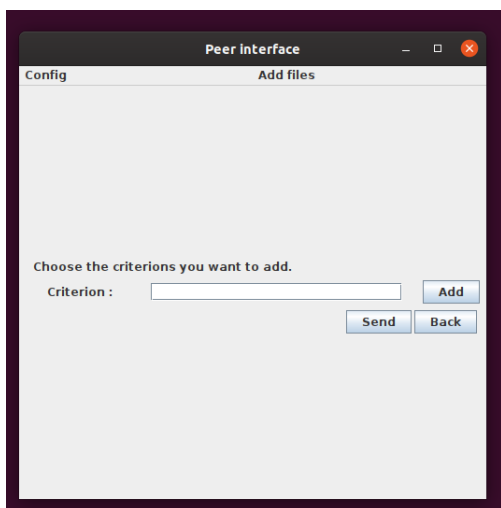


FIGURE 22 – Demande manuelle look au tracker

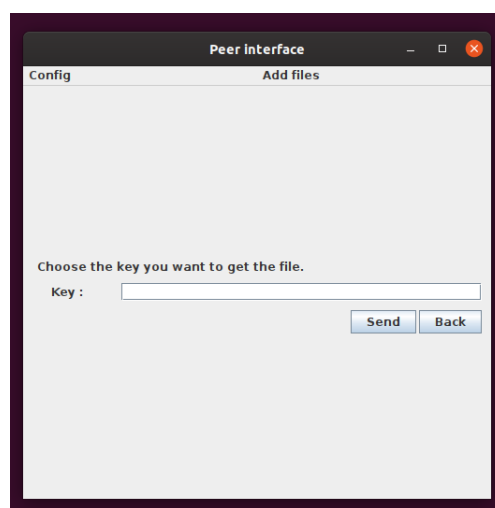


FIGURE 23 – Ajout de fichiers dans l'application

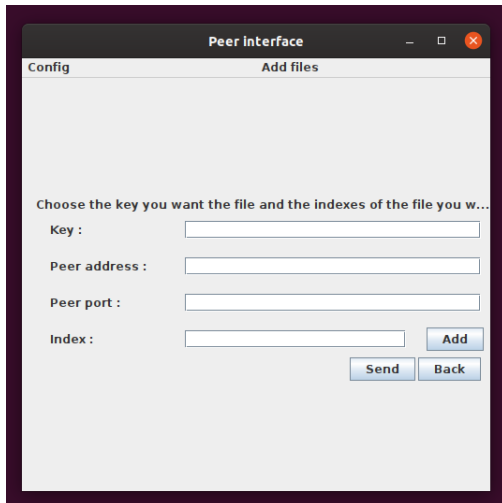


FIGURE 24 – Ajout de fichiers dans l'application

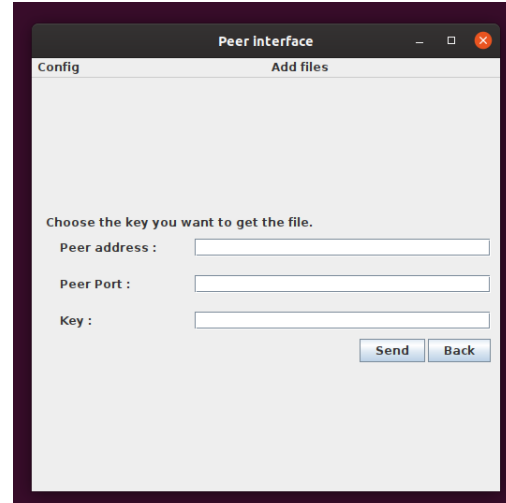


FIGURE 25 – Ajout de fichiers dans l'application

Le premier correspond à demander la commande **look** au *tracker*, dont l'interface est visible sur la figure 22. L'utilisateur peut rentrer manuellement un *criterion* qui sera transmis au *tracker*.

Le deuxième correspond à l'interface visible sur la figure 23. L'utilisateur peut rentrer manuellement un *hash* de fichier et la commande **getfile** sera envoyée automatiquement au *tracker*.

Le troisième permet à l'utilisateur de récupérer manuellement des morceaux de fichier, comme cela est visible sur la figure 24. L'utilisateur doit rentrer le fichier qui l'intéresse, les coordonnées du pair en question et enfin spécifier les morceaux de fichier qui l'intéressent. La commande **getpieces** ainsi composée sera envoyée au pair demandé.

La quatrième et dernière possibilité permet à l'utilisateur de manifester son intérêt pour un fichier. L'interface pour réaliser cette commande est visible sur la figure 25. L'utilisateur a simplement à rentrer les coordonnées d'un pair et la clé du fichier en question, et la commande **interested** sera encore une fois envoyée automatiquement.

3 Évaluation et tests

Afin de s'assurer du bon fonctionnement de notre implémentation, différents tests ont été effectués :

- tests sur la table de *hash* des fichiers
- tests sur les *seeders* et *leechers*
- tests sur le dictionnaire des pairs
- tests sur l'exécution des commandes

Pour les tests concernant la table de *hashage*, deux fonctions ont été implémentées et ont été utiles pour corriger certains problèmes mais également pour avoir une vision globale des pairs et des fichiers mis à disposition à un instant précis :

- `hash__print()` qui va afficher tous les fichiers disponibles (sous le format : `indice | nom, clé seeders [IP1:port1 ...] leechers [IP1:port1 ...]`)
- `hash__peer_print()` qui va afficher tous les pairs qui se sont identifiés au *tracker* ainsi que les fichiers qu'ils sont en train de télécharger et ceux qu'ils possèdent (sous le format : `IP:port seed [clé1 ...] leech [clé1 ...]`)

Les éléments suivants ont également été testés grâce à l'interface graphique :

- tests sur les structures de données des pairs

- tests sur les communications entre pairs
- tests sur l'analyse des commandes transmises
- tests sur la découpe de fichiers

4 Documentation

La partie *tracker* du projet est commentée de sorte à pouvoir générer une documentation avec **Doxygen**. Pour la créer, il suffit de lancer **make doxygen** dans le dossier **tracker/**. Cela génère un dossier **html/** dans lequel on retrouve le fichier **index.html** à lancer dans un navigateur (Figure 26).

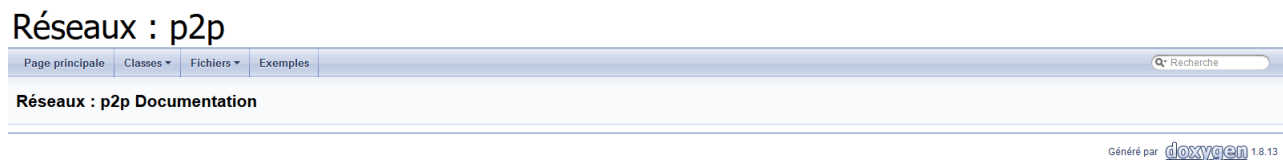


FIGURE 26 – Page d'accueil de la documentation

L'onglet *Classes* redirige vers la documentation de toutes les structures présentes dans le code telles que **seeder** et **leecher**. L'onglet *Fichiers* permet lui d'accéder à la liste des fichiers sources. Il suffit ensuite d'en choisir un pour visualiser sa documentation.

La partie pair est également documentée grâce à **Doxygen** et se lance de la même manière que pour le *tracker*. L'onglet *Classes* redirige vers la documentation de toutes les classes présentes dans le code source, comme par exemple la classe **FileManager**.

5 Problèmes rencontrés

Un des problèmes majeurs rencontrés est le fait que durant la phase de développement, le groupe est resté séparé. Il a donc été très difficile de faire communiquer les applications entre elles. Celles-ci tournaient sur une même machine locale, mais un manque de maîtrise des communications à travers la France nous a empêché de tester l'application sur plusieurs machines en réseau.

Un problème a également été rencontré concernant le *tracker* : afin de pouvoir arrêter ce dernier nous étions obligés d'utiliser un signal d'interruption (Ctrl-C). Cependant, en faisant cela, toutes les ressources allouées par le programme n'étaient pas libérées et un gestionnaire de signaux a dû être ajouté : **signal_handler**. Quand le signal d'interruption est reçu, les ressources mémoires du *tracker* sont libérées et l'exécution du programme est terminée.

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 celui-ci n'accepte que les pairs en local et donc tester les accès concurrents est difficile.

Un autre problème rencontré se trouve dans la transmission des *buffermaps* entre les pairs. L'encodage bit à bit sur l'objet **String** oblige la longueur du *buffermap* transmis à être un multiple de 8, et un pair ne peut ainsi pas être sûr d'avoir la totalité d'un fichier. Ce problème a été corrigé en réécrivant la totalité d'un fichier dès réception de nouvelles pièces de celui-ci.

6 Améliorations et objectifs futurs

6.1 Pair

L'application pair peut être améliorée sur différents axes. Le premier est l'utilisation de tableaux de booléens en mémoire pour stocker les *buffermaps*. Ce choix est loin d'être idéal pour l'occupation mémoire de l'application, et a été pris au début du projet de par la simplicité de manipulation de ce type d'objet. Cependant, ils pourraient être remplacés par un type différent comme un tableau de `char` ou par l'utilisation du type `String` qui, couplée à des opérations bits à bits, permettrait de gagner en espace mémoire et également en performance.

De plus, la transmission des *buffermaps* pourrait également être améliorée en prenant la convention de toujours terminer un *buffermap* par un 1. Cela permettrait de lever l'ambiguïté sur la longueur d'un *buffermap* dans l'utilisation du type `char`.

L'interface graphique aurait d'une manière générale pu être plus complète, avec l'ajout d'éléments agréables pour l'expérience utilisateur comme un tutoriel par exemple ou encore l'ajout du réglage de plus d'éléments de configuration. D'autres aspects pourraient également être améliorés. Un explorateur de fichiers pourrait être ajouté de manière à éviter la mauvaise saisie du chemin d'un fichier par l'utilisateur. De plus, un item de validation pourrait être ajouté de manière à lui signifier que son entrée a été prise en compte par l'application.

Un autre aspect d'amélioration serait une meilleure protection de l'application. En effet, les *buffermaps* et les morceaux de fichiers transmis sont passés en clair sur le réseau, un attaquant n'aurait aucun problème pour récupérer les données, la seule difficulté serait de convertir la chaîne de caractères récupérée en un *buffermap* exploitable par un attaquant. Une amélioration bienvenue serait donc le chiffrement des *buffermaps* et des morceaux de fichier transmis. Aucune protection n'a également été ajoutée pour vérifier que l'interlocuteur d'un pair ou d'un *tracker* est bien d'un de ces deux types. Un attaquant pourrait très facilement se faire passer pour un pair en envoyant des requêtes seules. Un moyen d'identification serait donc également bienvenu pour la sécurité de l'application.

6.2 Tracker

Une autre amélioration concernant le *tracker* serait de nettoyer le *parseur*. 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), par la suite, l'affichage serait 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, une amélioration possible serait d'effectuer plus de tests : en plus de tester les différentes fonctions des structures, il serait peut être optimal de tester des scénarios entiers de connexion et de voir si à chaque fin de scénario la situation finale est attendue (par exemple un scénario de communication d'échanges entre 3 pairs).

7 Conclusion

Pour conclure, ce projet nous a permis de découvrir comment une application de téléchargement pair à pair fonctionne. En effet, notre implémentation nous a permis de voir le fonctionnement global de ce type d'application mais également de découvrir les problématiques qu'elle soulève. Nous ne pensions pas que le téléchargement en parallèle d'un fichier tout comme la gestion simultanée des différents pairs pouvant se connecter au tracker constitueraient de véritables challenges. De plus, des fonctionnalités comme le lancement du *tracker* sur une adresse IP

non privée est une fonctionnalité que nous ne pensions pas difficile, mais qui s'avère l'être. La communication et le développement se sont fait en local durant ce projet mais pouvoir lancer le *tracker* sur une machine distante et se connecter depuis chez soi sur celui-ci avec notre pair aurait été une situation heureuse.

Nous sommes plutôt satisfaits des objectifs atteints durant le développement de ce projet malgré le fait que nous aurions aimé aller plus loin dans les objectifs avancés au lieu de se focaliser sur une très bonne version de base.