Projet Réseaux - Rapport

Clément Bertin
Pierre Chevallier
Corentin Le Métayer
Maëlle Toy-Riont–Le Dosseur

Mai 2020

Table des matières

1 Introduction

L'objectif du projet consiste en la mise en place d'un système d'échange de fichiers en mode pair-à-pair. Nous avons réalisé une version centralisée avec un tracker permettant d'enregistrer les informations transmises par chaque peer, ainsi que les intéractions entre peers, et de les restituer à la demande des peers.

2 Organisation

Afin de mettre en place une dynamique de travail à distance, nous avons créé un groupe de communication en ligne sur **Discord**, qui nous a permis de nous répartir le travail et d'établir un planning des tâches à effectuer, ainsi qu'un historique des questions à poser et des réponses obtenues. Nous avons aussi utilisé ce groupe de discussion afin de s'aider en cas de problème. L'utilisation d'un **Trello** nous a permis d'établir une vision des tâches à effectuer sur le court et long terme.

Le travail a d'abord été divisé en trois parties : une personne sur l'implémentation du **tracker**, deux personnes sur l'implémentation du **peer**, et une personne sur la mise en place des **fichiers de configuration**, la compilation et l'exécution des peers et du tracker. Dans un second temps, nous avons établi une architecture générale pour les fichiers et les protocoles des requêtes, puis nous avons implémenté les différentes requêtes en suivant le plus possible l'architecture imaginée.

3 Architecture des fichiers et dossiers

Pour l'architecture générale du code, nous avons créé 3 dossiers séparés *central*, *distributed* et *blockchain* pour chaque version.
Pour la version centralisée :

- le code du **tracker** et du **peer** sont séparés dans des dossiers différents.
- le dossier **build** contient le code compilé.
- le dossier **Config** contient les fichiers de configuration du *tracker* et des *peers*.
- le dossier **Logs** contient les logs pour chaque élément du programme.

4 Implémentation des fichiers configurations et log

4.1 Fichiers de configurations

La configuration du tracker et des peers peut être effectuée grâce à la commande `make configuration`.

4.1.1 Interface graphique

Afin de faciliter la configuration, nous avons mis en place une interface graphique simple grâce à l'utilitaire *bash dialog*. Il s'agit d'un utilitaire présent sur la majorité des distributions Debian et qui peut être installé sur d'autres distributions linux.

Bash dialog permet la création d'une interface graphique dans le terminal de l'utilisateur. Pour inclure dialog dans le projet, nous avons utilisé un script bash donnant les mêmes possibilités que cet utilitaire. De cette façon, un utilisateur ne possédant pas l'utilitaire pourra tout de même utiliser notre interface.

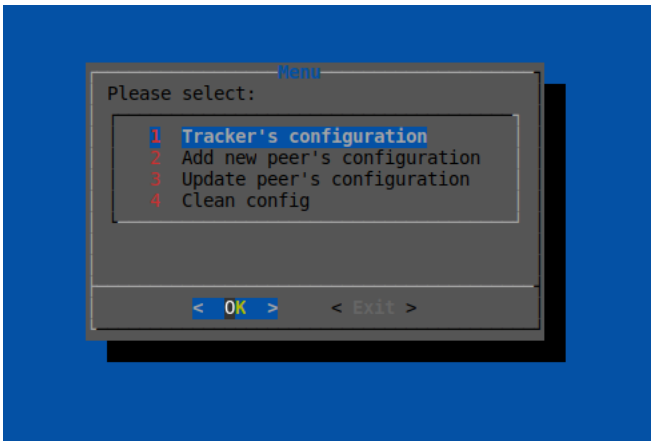


FIGURE 1 – Interface graphique pour la configuration des peer et du tracker

L'interface de configuration est partagée en 4 parties : l'écriture des configurations du tracker, l'écriture et la mise à jour des configurations de chaque peer et la suppression de fichiers de configuration. La configuration du tracker et de chaque peer produit, dans le dossier Config, un fichier `config_tracker.ini` ou `config_peerNuméroDePort.ini`. On retrouve dans ces fichiers les informations des peers et tracker que l'utilisateur peut changer au choix avec l'interface graphique ou directement dans le fichier. Le changement à la main par l'utilisateur est cependant déconseillé puisqu'avec l'interface graphique nous effectuons des tests évitant les erreurs de configuration.

4.1.2 Gestion des erreurs de configuration

Nous avons mis en place des **tests** afin de vérifier que les entrées de l'utilisateur soient correctes lors de la **configuration**.

Nous vérifions que le port saisi par l'utilisateur se situe dans la bonne plage (entre 1024 et 65 535) et qu'aucun autre peer ou tracker n'est déjà configuré sur ce port. Une évolution des configurations pourrait être de vérifier que le numéro de port est bien disponible, c'est à dire que rien n'est actuellement connecté à ce port. D'autre tests permettent de vérifier que l'utilisateur ne fait pas de fautes de frappes.

Lors du lancement d'un peer sur un certain numéro de port :
- Si aucun fichier de configuration associé à ce port n'est trouvé, un message d'erreur s'affiche et le programme est terminé.
- Si un fichier de configuration est trouvé, les différents champs sont séparées dans un tableau de String, puis chaque élément est récupéré au bon format (int, String, etc.).

4.2 Fichiers de log

Nous avons mis en place un dossier contenant les **logs** de chaque peer et du tracker, afin de conserver un historique complet des requêtes envoyées et reçues. Ces fichiers de logs sont créés lors de l'éxecution d'un peer ou d'un tracker et correspondent à leur numéro de port.

Les informations dans les fichiers logs sont plus complètes que sur le terminal pendant l'exécution : elles permettent de savoir à quel moment les requêtes ont été reçues ou envoyées, de connaître l'adresse et le port de l'expéditeur, ainsi que celle du destinataire.

Ces fichiers nous ont principalement permis de s'assurer que les requêtes implémentées étaient correctes.

5 Implémentation des Peer et du Tracker

5.1 Tracker

L'implémentation du tracker se base sur un modèle classique de "serveur" en utilisant les fonctions de base d'interaction client/serveur en mode TCP. Un pool de thread permet de gérer les multiples connexions : quand le tracker reçoit une demande de connexion d'un peer, il ouvre une *socket* pour ce peer qui est confié à un thread disponible du pool. Ce thread exécute alors le traitement permettant d'écouter les requêtes du peer, de les traiter et d'y répondre tant que la connexion n'est pas fermée côté peer. Plusieurs fonctions de parsing pour les requêtes entrantes ont été ajoutées. Un thread fonctionne à part dès le lancement du tracker et gère l'affichage des connexions au tracker.

Nous avons choisi d'utiliser une table de hachage afin de garder en mémoire les données partagées par les peers. Cette table de hachage contient comme entrées les fichiers partagés et leurs informations, et les clés de cette table sont les clés générées par md5sum. Chaque entrée de la table de hachage contient : le *nom* et la *taille* du fichier, la *taille des pièces*, l'*adresse IP* et le *port* des peers possédant ce fichier. Le code pour la table de hachage est celui de la librairie **uthash**.

```
  struct my_struct {
  const char* id;   /* key, file MD5 */
  char* file_name;
  char* file_size;
  char* pieces_size;
  char* access;      /* IP:Port ' ' delimite les adresses */
  UT_hash_handle hh; /* makes this structure hashable */
};

```

5.2 Peer

Concernant le peer, nous avons séparé le fonctionnement *serveur* (permettant de recevoir et gérer les requêtes) du fonctionnement *client* (permettant d'envoyer les requêtes et de gérer les réponses). Nous avons donc créé deux threads : un pour les connexions *entrantes* et l'autre pour *l'envoi de messages*. Comme pour le tracker, un pool de thread garde la connexion tant que la socket n'est pas fermée.

Au niveau de la structure des fichiers, nous avons choisi de définir des classes spécifiques à chaque requête, afin d'avoir une meilleure visibilité lors de l'implémentation :

- RequestAnnounce
- RequestGetfile
- RequestGetPieces
- RequestHave
- RequestInterested
- RequestLook
- RequestUpdate

La classe **Peer** permet de lancer le thread gérant les connexions et d'afficher le prompt des actions utilisateur.

La classe **ClientHandler** permet de gérer l'envoi et la réception de message d'un peer ou d'un tracker.

La classe **Command** permettant de regrouper toutes les informations nécessaires au peer, ainsi que des fonctions utiles (liste des fichiers téléchargés, buffermaps, etc.).

6 Implémentation des requêtes

La partie suivante présente les implémentations des requêtes accompagnées de schémas explicatifs. Ces schémas nous ont servis à mettre en évidence les axes utiles pour l'implémentation.

6.1 Announce

Le schéma que nous avons conçu pour la requête **announce** se trouve sur la figure 2.

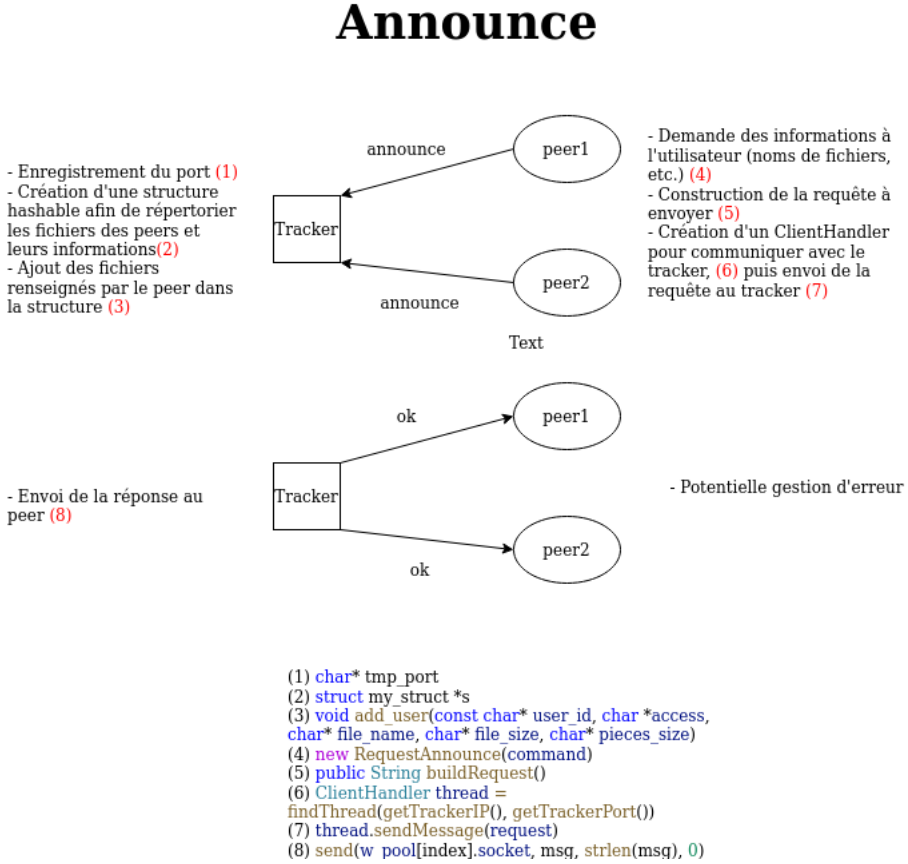


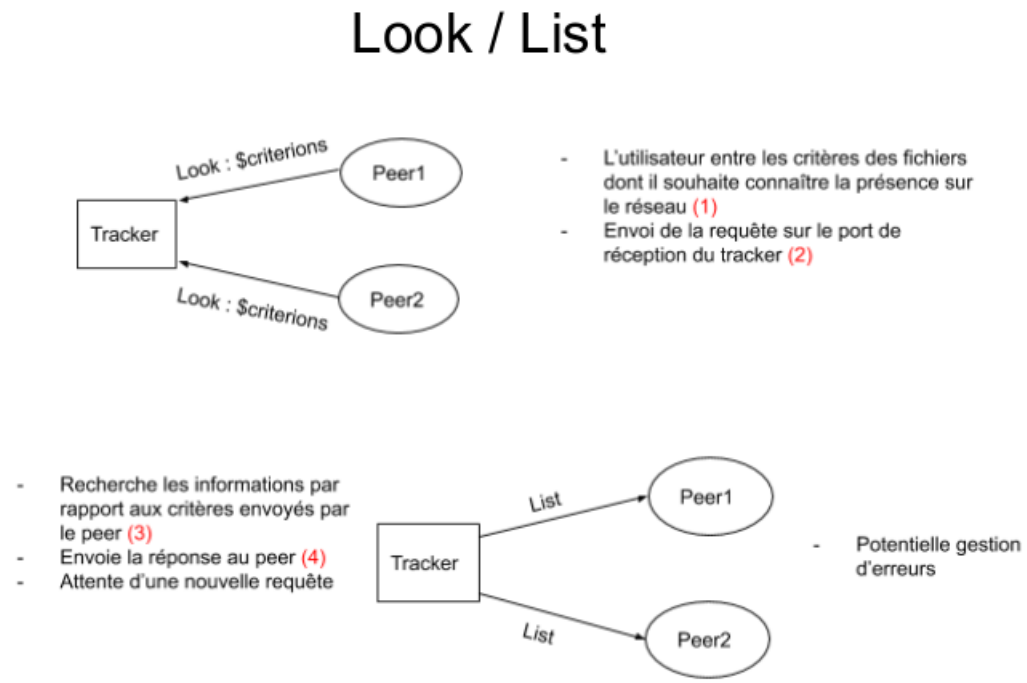
FIGURE 2 – architecture de la requête **announce**

Quand l'utilisateur entre la commande pour déclencher la requête **announce**, il doit d'abord saisir les fichiers qu'il souhaite déclarer au tracker, et le peer vérifie que ce fichier existe. Si c'est le cas, il récupère les informations sur le fichier à envoyer au tracker, et envoie la requête à celui-ci.

Chaque fichier renseigné au tracker ajoute un buffermap rempli de "1" pour le peer pour ce fichier (car il le possède entièrement) dans la table des buffermaps contenant les fichiers complets du peer. Dès que le tracker reçoit cette requête, il parse le message reçu pour isoler les différentes composantes d'entrée de la table de hachage puis il enregistre ces données et renvoie "Ok" au peer. Le parsing est réalisé avec les caractères "[ ]" et les différents espaces de la requête.

6.2 Look

Le schéma initial de l'architecture de la requête **look** se trouve sur la figure 3.




```
(1)  new RequestLook(command,filename,filesize)
(2)  command.sendToTracker(request)
(3)  USER_ITER(file){
     ....
     strcat(msg, file->file_name); strcat(msg, " ");
     strcat(msg, file->file_size); strcat(msg, " ");
     strcat(msg, file->pieces_size); strcat(msg, " ");
     strcat(msg, file->id);
     }
(4)  send(w_pool[index].socket, msg, strlen(msg), 0)
```

FIGURE 3 – Architecture de la requête **look**

Si l'utilisateur entre la commande **look**, il doit saisir les critères demandés par rapport au fichier qu'il cherche à télécharger. Il est nécessaire qu'il y ait au moins un critère à demander pour envoyer la requête. Une fois ces critères entrés, il suffit ensuite de les mettre dans la requête et de l'envoyer au tracker.

Dès que le tracker reçoit cette requête, il récupère les différents critères et recherche les fichiers correspondant à ces critères dans sa table de hachage en la parcourant, selon l'égalité au niveau des noms. Le tracker construit une réponse avec la clé des fichiers correspondant au critères et l'envoie au peer.

6.3 Getfile

L'architecture pour la requête **getfile** se trouve sur la figure 4.

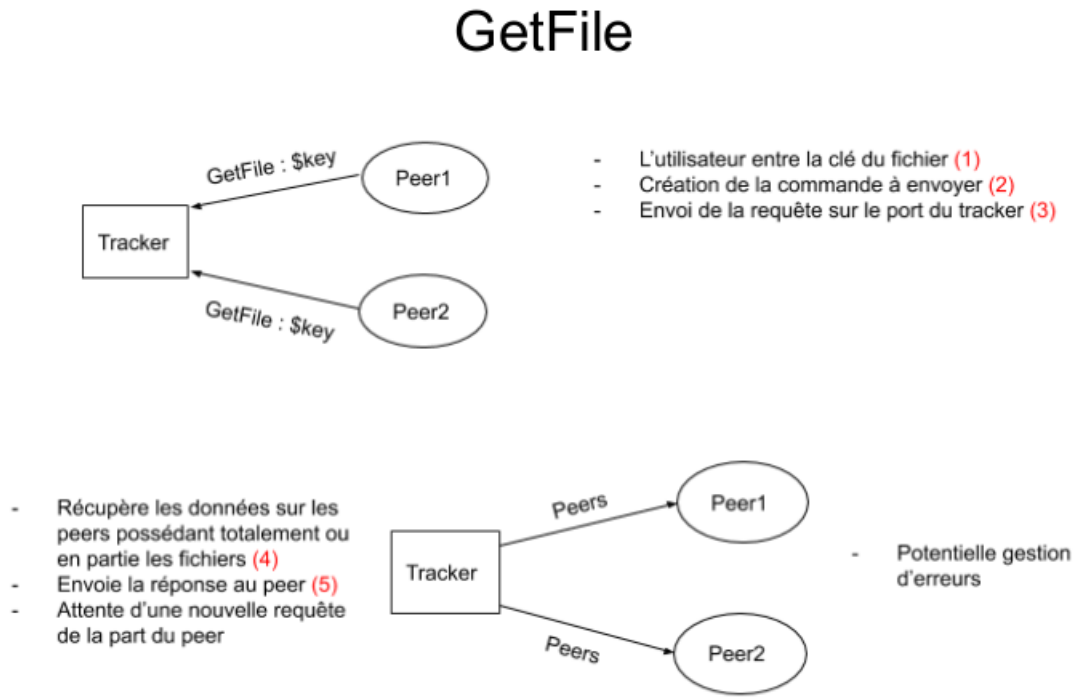


FIGURE 4 – Architecture de la requête **getfile**

L'utilisateur peut entrer la commande **getfile** pour demander la liste des peers possédant un fichier. Dans ce cas, le peer demande à l'utilisateur la clé md5 du fichier qu'il souhaite, puis construit la requête et l'envoie au tracker.

Le tracker reçoit la requête et extrait la clé du fichier demandé. Avec les fonctions de la bibliothèque de la table de hachage le tracker récupère les couples adresses IP/port correspondant aux peers qui possèdent ce fichier. La requête réponse est construite en fonction des couples trouvés et est envoyée au peer.

6.4 Interested

La figure 5 représente l'architecture de la requête **interested**.

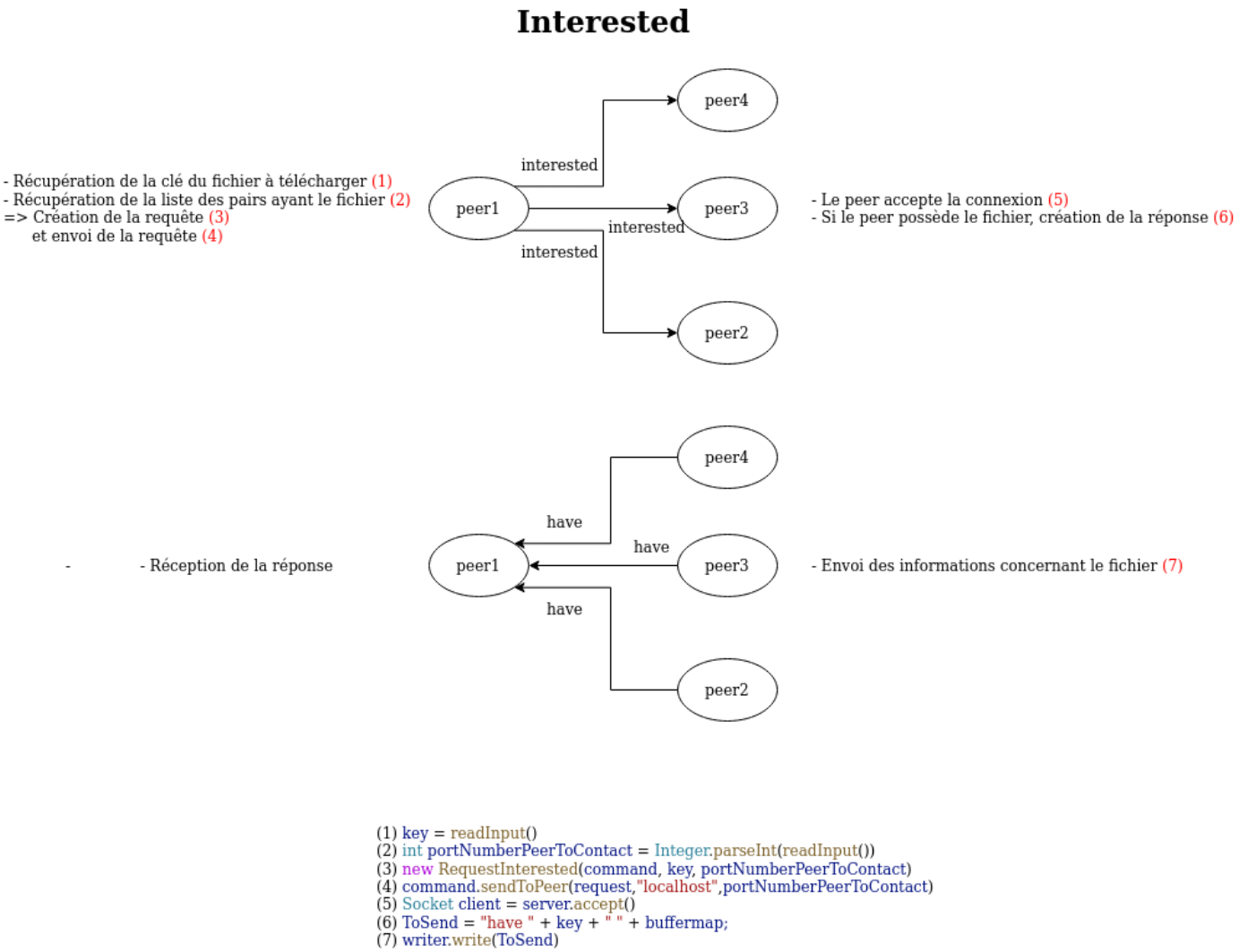


FIGURE 5 – Architecture de la requête **interested**

Si l'utilisateur entre la commande **interested**, le peer demande la clé md5 du fichier qu'il souhaite, et le port du peer qu'il souhaite contacter. Le peer envoie cette requête, puis le peer contacté va chercher le buffermap correspondant à la clé renseignée dans la requête, soit dans la table de hachage des fichiers complets, soit dans celle des fichiers en train d'être téléchargés. S'il trouve une buffermap correspondante, alors il répond avec celle-ci, sinon, il répond "0".

6.5 Getpieces

L'architecture de la requête **getpieces** se trouve sur la figure 6.

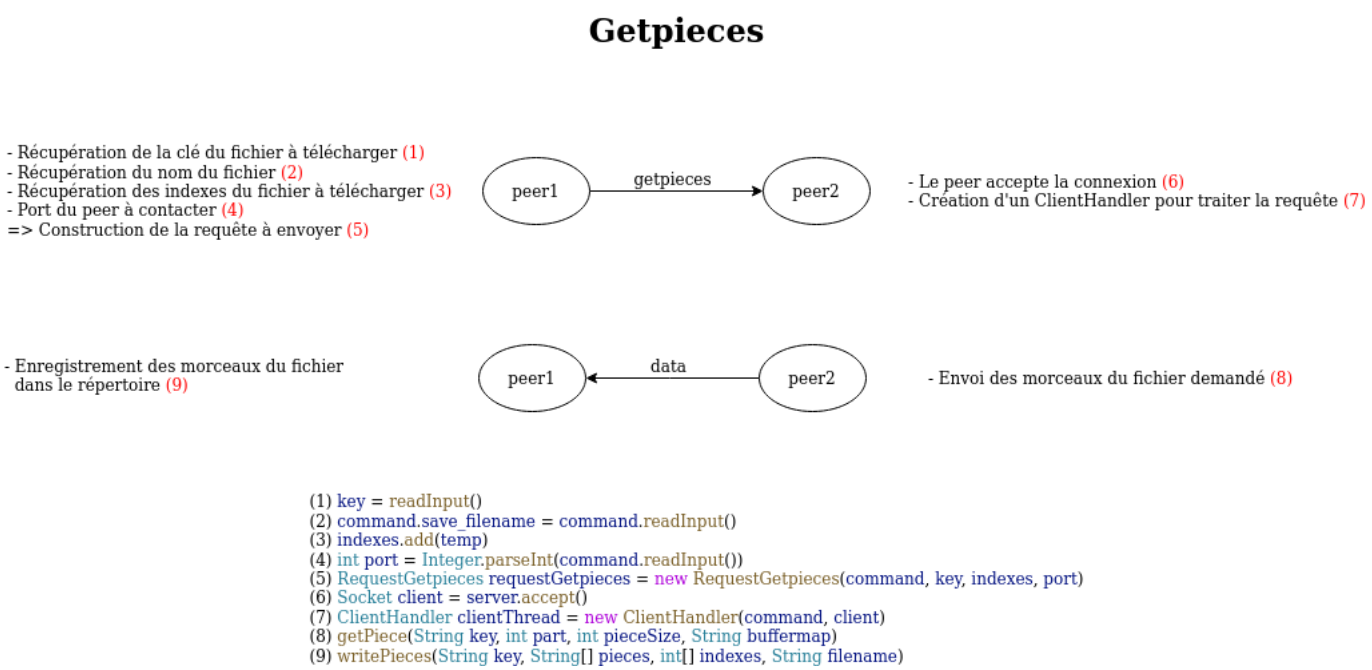


FIGURE 6 – Architecture de la requête **getpieces**

Dès que l'utilisateur entre la commande **getpieces**, le peer demande le port du peer à contacter, le nom du fichier à l'enregistrement, la clé du fichier, ainsi que les index souhaités. Il envoie ensuite la requête, et le peer contacté cherche alors les différents morceaux de fichiers correspondants.

Pour cela, il cherche d'abord s'il possède complètement le fichier ou non. S'il ne le possède pas entièrement, il va directement chercher les morceaux de fichier dans la table de hachage stockant les morceaux de fichiers en train d'être téléchargés. Sinon, il va chercher dans le fichier enregistré sur le disque. Ce peer va construire la réponse au fur et à mesure des morceaux trouvés, puis va l'envoyer au premier peer.

A la réception de la réponse, le peer va enregistrer les morceaux de fichier dans une table de hachage et mettre à jour son buffermap. Une fois que tous les morceaux ont étés enregistrés, si le buffermap est rempli de "1", on enregistre toutes les pièces dans un fichier sur le disque, puis on transfère le buffermap dans la table de hachage des fichiers complets. Si on reçoit un nouveau morceau pour ce fichier, alors on transfère à nouveau le buffermap dans la table de hachage des fichiers en train d'être téléchargés.

6.6 Have

La figure 7 représente l'architecture de la requête have.

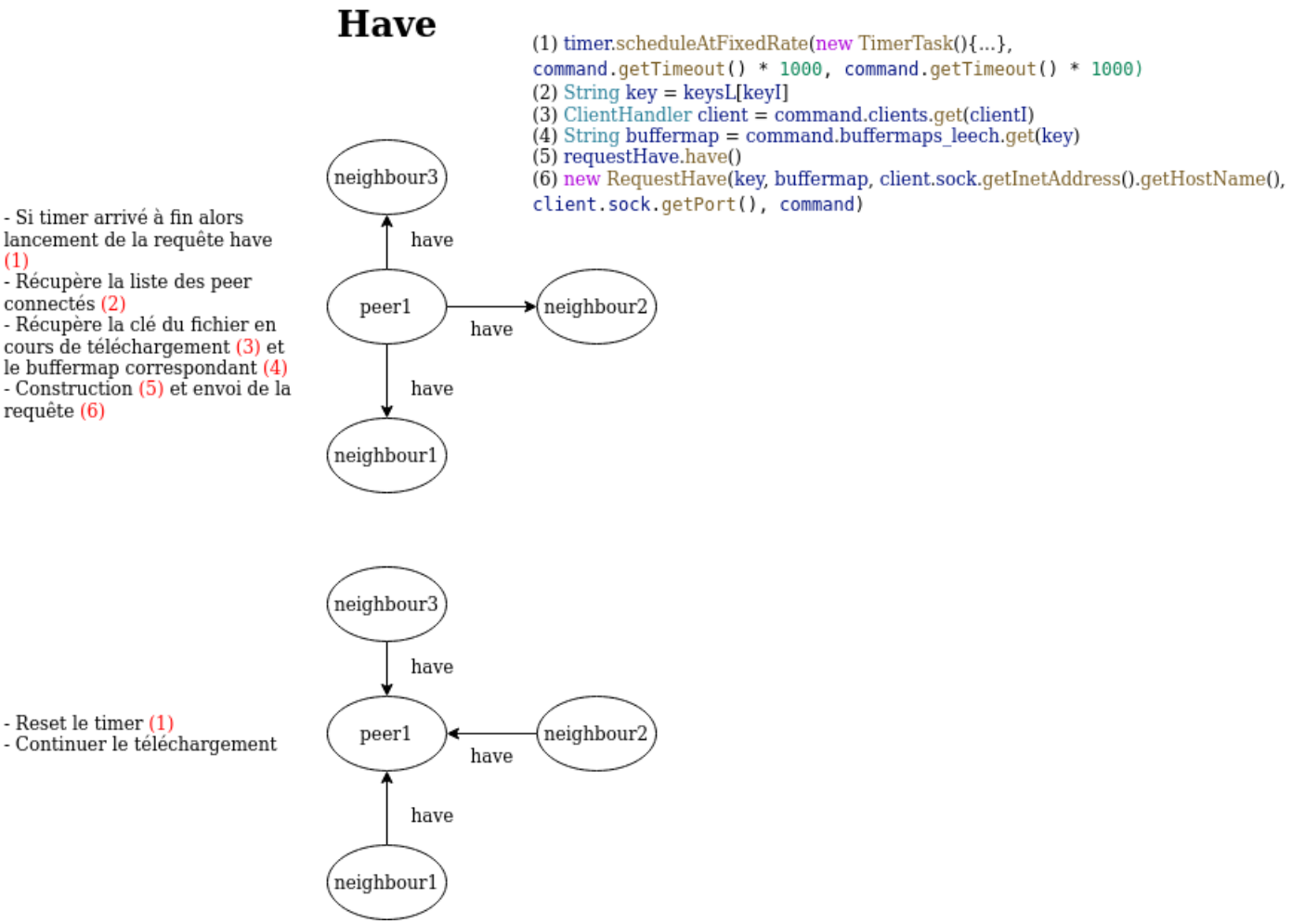


FIGURE 7 – Architecture de la requête **have**

L'utilisateur ne peut pas déclencher manuellement la requête **have** car elle se déclenche périodiquement. En revanche, il peut indiquer la période d'envoi de cette requête dans le fichier config du peer. Cette requête récupère la liste des peers connectés, puis la liste des clés de fichiers possédés par le peer. Il envoie ensuite à chaque peer les clés et son buffermap correspondant. A la réception de cette requête, le peer contacté répond avec son buffermap correspondant à la clé renseignée.

6.7 Update

Le schéma que nous avons conçu pour la requête **update** se trouve sur la figure 8.

Update

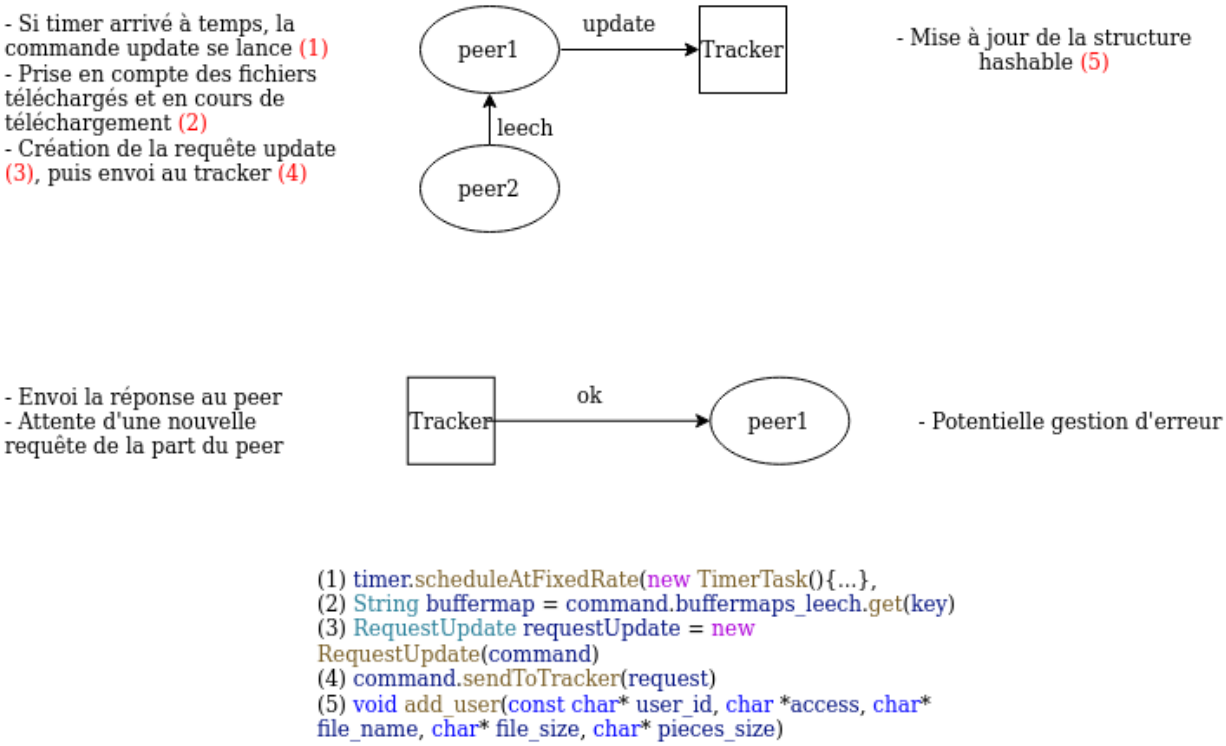


FIGURE 8 – Architecture de la requête **update**

Comme la requête **have**, la requête **update** est périodique et ne peut être déclenchée par l'utilisateur. Cette requête est déclenchée avec la même fréquence que **have**.
Le peer récupère la liste des clés correspondant aux fichiers complets, ainsi que la liste des clés des fichiers en cours de téléchargement. Ensuite, le peer construit la requête et l'envoie au tracker. À la réception de cette requête, le tracker parse le message entrant et met à jour les données de ce peer dans sa table de hachage.

6.8 Download

Nous avons aussi ajouté une requête utilisateur permettant de télécharger un fichier automatiquement.

Pour cela, nous demandons d'abord le nom du fichier à l'enregistrement, le nom du fichier à acquérir et la taille minimale de fichier.
Ensuite, le programme enchaîne plusieurs requêtes. Tout d'abord **look** pour obtenir la clé du fichier, puis **getfile** pour obtenir tous les peer possédant ce fichier qu'il stocke dans une liste.
Puis le programme envoie la requête **interested** à chaque élément de cette liste afin de récupérer toutes les buffermap. Le programme décompose chaque buffermap afin de savoir quels morceaux demander à quels peers.
Lorsque le plan des index à demander aux peers est récupéré, il suffit d'envoyer une requête **getpieces** à chaque peer, en prenant soin de ne pas dépasser la taille maximum de message.

6.9 Pool de threads

Nous avons mis en place pour le peer comme pour le tracker un pool de threads permettant de garder la connexion jusqu'à la fin de l'exécution. Cela permet de gagner du temps sur la réception et l'envoi de messages.

L'implémentation côté tracker a été assez simple à déployer, le tracker ne faisant que recevoir des messages et envoyer des réponses. En revanche, la mise en oeuvre côté peer a été plus délicate, du fait de l'envoi de requêtes et de la réception de réponses. Nous avons donc créé des fonctions permettant de retrouver le thread concerné lors de l'envoi des messages, ou d'en créer un nouveau si le thread n'existe pas.

7 Difficultés rencontrées

Voici une liste des difficultés techniques que nous avons rencontrées au cours du projet, et comment elles ont été résolues.

7.1 Manipulation des chaînes de caractères

Le tracker est écrit en C, ce qui nécessite manipuler régulièrement des chaînes de caractère. Ces manipulations ont présenté une certaine difficulté technique. En effet, les chaînes de caractères sont assez délicates à utiliser en C, car à leur utilisation s'ajoute une gestion subtile de la mémoire. A titre d'exemple, le parsing et la création des réponses aux requêtes ont été particulièrement compliquées à gérer et ont entrainé des bugs. Nous avons néanmoins réussi à corriger les erreurs en utilisant des outils de debug tout au long du projet.

7.2 Update côté tracker

Avant l'automatisation de l'ensemble des requêtes, nous avons été confrontés à une erreur particulièrement compliquée. Lors du téléchargement d'un fichier suivi d'un update, il se produisait un segfault du côté du tracker. Nous avons d'abord essayé de résoudre les bugs grâce à l'utilisation de *valgrind* et *gdb*, mais leur utilisation dans ce cas précis n'était pas probante. Nous avons donc plutôt décidé d'afficher des variables à différents endroits du code, qui ont servi de "points d'arrêts", ce qui nous a aidé à résoudre le problème.

8 Conclusion

Malgré ces difficultés, nous avons réussi à créer une application tout à fait fonctionnelle qui répond aux attentes. De plus, nous avons préféré avoir une version finale stable plutôt qu'une version instable remplissant tous les objectifs. En utilisant cette base, nous aurions ensuite pu travailler sur les versions "distributed" et "blockchain" afin d'obtenir une version plus complète.

Ce projet a été pour nous l'occasion de mettre en pratique ce qui nous a été enseigné en cours de réseaux, ainsi que d'approfondir nos connaissances dans ce domaine.

De plus, cela a été pour nous l'occasion d'apprendre le travail à distance, et d'apprendre l'organisation de groupe sur un projet dense.