ENSEIRB-MATMECA

Filière informatique, 2ème année

Réseau

Rapport de projet

CookieTorrent

COOK Clement
CHAMBRE Jonathan
COMBES Gaetan
GRALL Alexis
REAVAILLE Nathan

Mai 2016

Table des matières

1 Protocole **3**

2 Organisation et Utilisation **3**
2.1 Organisation des sources . . . 3
2.1.1 Partie Client . . . 3
2.1.2 Partie tracker . . . 4
2.2 Utilisation . . . 4

3 Le Client **5**
3.1 Le parseur . . . 5
3.2 Threads . . . 5
3.2.1 Le TrackerHandler . . . 6
3.2.2 Le ClientHandler . . . 7
3.2.3 Le userHandler . . . 7
3.3 Gestion des fichiers . . . 8
3.4 Traitement des requêtes . . . 8
3.5 Gestion des téléchargements . . . 9
3.5.1 Envoi des Interested et Traitements des bufferMap . . . 9
3.5.2 Algorithme de téléchargement . . . 9

4 Le Tracker **10**
4.1 Le parseur . . . 10
4.2 Gestion des clients et des fichiers . . . 10
4.3 Traitement des requêtes . . . 11
4.4 Threads . . . 12

Introduction

1 Protocole

Le protocole utilisé est celui proposé dans le sujet a quelques modifications près. Nous avons mis en annexe un schéma résumant les échanges Client/Tracker et Client/Client. Le Client. Le Tracker a été codé en C et le client en Java. Voilà ce qui est fonctionnel :

- Coté tracker :
  - L'ensemble des requêtes définies dans le sujet sont fonctionnelles ;
  - Le tracker utilise un threads pool et gère la connexion de plusieurs clients en parallèle ;
  - Le tracker affiche des logs dans la console ;
- Coté client :
  - L'ensemble des requêtes définies dans le sujet sont fonctionelles ;
  - Le client utilise plusieurs pool de threads pour gérer en parallèle les différentes fonctionnalités ;
  - Il affiche des logs dans la console ;
  - Le téléchargement de fichiers est fonctionnel (fichiers texte et binaire) ;
  - Enfin il gère le cas où un fichier n'est pas présent en totalité sur le réseau.
  - Le client gère plusieurs téléchargements en parallèle ;
  - Le client intègre un affichage graphique ;

Les problèmes restants :

- Coté client :
  - Coté protocole les requêtes have périodiques n'ont pas été implémentées ;
  - Seuls l'ip et le port du tracker sont configurables dans le client ;
  - Le client ne gère pas la déconnexion d'un client lors d'un téléchargement ;
  - Il ne gère pas non plus la déconnexion du tracker. Il n'est pas capable de s'y reconnecter ;
  - L'affichage ne se lance que si le client se connecte au tracker ;
  - Vitesse de transfert faible ;

2 Organisation et Utilisation

2.1 Organisation des sources

Les sources sont découpées en deux parties : Client et Tracker

2.1.1 Partie Client

Nous avons réalisé une documentation détaillée du code du client grâce à javadoc, la commande pour générer cette documentation vous est donnée dans la partie Utilisation de notre rapport. Cette documentation décrit l'ensemble des classes et méthodes de la partie

java du code. Néanmoins voici quelques détails sur l'organisation des fichiers sources dans le dossier Client.

— L'adresse et le port du tracker peuvent être configurés à partir du fichier config.ini ;
— Le dossier shared contient les fichiers que l'on souhaite partager (cf Partie 3.4). C'est le fichier SharedFile qui se charge de la gestion des fichiers partagés ;
— Le parseur se trouve dans le dossier Parse ;
— Les traitements de ces requêtes sont implémentés dans le dossier RequestTreatment ;
— La partie de gestion des téléchargements se trouve dans le dossier DownloadManager ;
— Le main du client se trouve dans le fichier Client.java. Les fichiers TrackerHandler, ClientHandler et UserHandler gèrent les différentes interfaces du client, ils correspondent chacun à un thread (cf partie 3.3).
— Quelques tests ont été implémentés, ils sont regroupés dans le dossier Test ;
— Les fichiers dans Tools sont les outils élémentaires au fonctionnement du client, comme le hachage des fichiers en MD5, les classes représentant un fichier, un peer... ;
— Le code de l'interface et des traitement associés se trouve dans le dossier interface ;

2.1.2 Partie tracker

Dans le dossier Tracker :

— Le fichier tracker.c est le fichier principal contenant la fonction main du Tracker.
— Le fichier config.ini permet de configurer l'adresse et le port d'écoute du Tracker.
— Le fichier config.c est le module permettant de lire le fichier config.ini pour en extraire les informations.
— Le fichier link.c contient une implémentation de liste chaînée utilisée par le Tracker.
— Le fichier parser.c contient le parseur du Tracker pour interpréter les commandes reçues sur la socket.
— Le fichier client.h contient la structure que le tracker remplit et garde en mémoire pour chaque client connecté.
— Le fichier file.c est le module permettant de gérer les listes de fichiers gardés en mémoire par le Tracker.
— Le fichier functions.c est le module qui exécute les commandes **announce**, **look**, **getfile**, ... sur le Tracker, et répond au client sur la socket.

2.2 Utilisation

Pour exécuter notre projet :

1. Dans src/Tracker lancer la commande make all pour compiler les sources du Tracker et ./tracker pour l'executer.
2. Dans src/Client lancer la commande `./CookieClient.sh` Ce script se charge de créer les fichiers nécessaire au fonctionnement du client, de compiler et d'executer le client.
3. Have fun !

Pour générer la documentation javadoc, dans le dossier ./src :

```
javadoc -private -d Documentation Client/Tools/* Client/*.java Client/DownloadManager/*
Client/Parser/* Client/Interface/*.java Client/RequestTreatment/* Client/Tools/*
```

Pour l'afficher, toujours dans ./src (Avec firefox ici, mais vous pouvez utiliser votre navigateur préféré) :

```
firefox ./Documentation/overview-summary.html
```

Vous pouvez retouver ces infirmation dans le fichier README.txt à la racine du projet.

3 Le Client

La difficulté supplémentaire du client est qu'il joue plusieurs rôles dans le protocole P2P. Un rôle de client dans ses échanges avec le tracker et avec les clients auxquels il demande des fichiers, mais aussi un rôle de serveur pour les clients qui veulent télécharger un de ses fichiers en partage. Il faut donc gérer des connexions à au minimum 3 sockets : une pour initier la connexion au tracker, une en écoute pour les éventuels clients et une autre pour initier des connexions avec d'autres clients, d'où l'utilité des threads.

3.1 Le parseur

Il interprète la requête et selon son type, crée les objets appropriés pour qu'elle puisse être traitée. Ces objets sont ceux définis dans tools : le `CookieFile` (représentation dans le client d'un fichier), le Peer (représentation d'un autre client), etc... Le parseur effectue également des contrôles sur la syntaxe des requêtes avec l'exception `BadRequestException`. Il vérifie entre autre que les requêtes sont complètes et retourne une exception `LengthRequestException` si ce n'est pas le cas.

3.2 Threads

Le client utilise des threads pour gérer les différentes connexions avec les clients et le tracker mais aussi les interactions avec l'utilisateur. Ces threads sont gérés grâce à un thread pool de la classe `Executors.newCachedThreadPool`. Grâce à cette classe, la gestion des threads est simplifiée. Le main va générer différentes tâches qu'il va donner au `ThreadPool`. Il va se charger de créer un nouveau thread et de le supprimer une fois qu'il a fini. Le `ThreadPool` n'est pas de taille finie. C'est-à-dire qu'il peut y avoir autant de threads que l'on veut. (Dans la suite du rapport nous parlerons pour simplifier de création d'un thread. Il faut comprendre ici créer un tache et l'ajouter au `ThreadPool`. Aucun thread n'est crée "à la main").

Il y a 3 threads principaux, un pour chaque interface du client, que l'on a appelé dans le projet des handlers. Le `TrackerHandler` initie la connexion avec le tracker et lit dans la socket les requêtes du tracker pour les traiter. Le `ClientHandler` gère les différentes connexions des clients. Enfin l'`UserHandler` sert d'interface entre l'utilisateur et le client.

Autre chose importante, le client doit périodiquement envoyer une mise à jour des fichiers en seed et en leech. Nous avons fixé ce temps à 30 secondes. Pour faire cela, nous avons utilisé la classe `SchedulerThreadPool`. Cette classe va, toutes les 30 secondes, créer

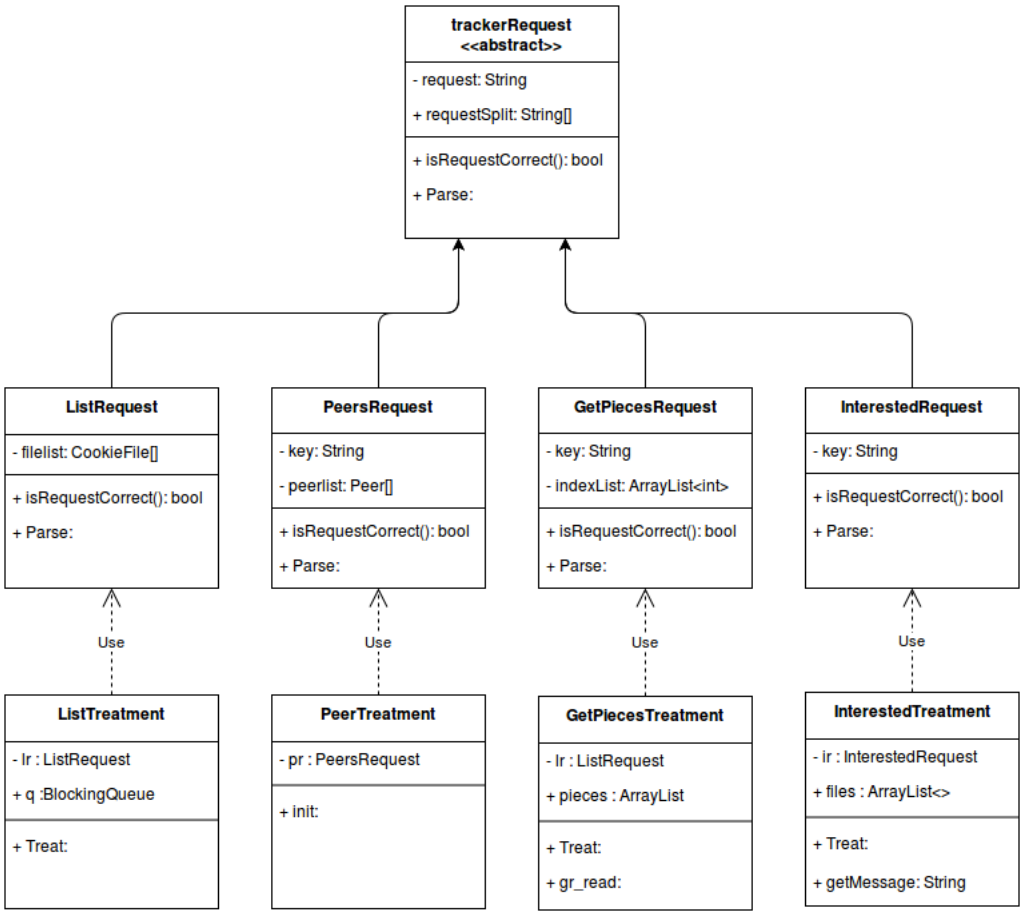

FIGURE 1 – Diagramme de classe partiel du parseur et du traitement des requêtes

un thread et exécuter la classe `Runnable updateTask` qui va envoyer la requête update au tracker.

Nous avons également fixé un timeout avec le tracker sur la socket de une minute, pour détecter un problème.

3.2.1 Le TrackerHandler

Le `TrackerHandler` gère une partie des échanges avec le tracker. Lors du lancement du client, il envoie la requête `"announce"`. Il donne son port d'écoute et les fichiers qu'il partage (qu'ils soient en seed ou en leech) au tracker. Puis, il attend un `"ok"` comme réponse du tracker. Il reste ensuite en écoute, et assure le traitement des réponses du tracker.

3.2.2 Le ClientHandler

Le `ClientHandler` assure le coté serveur des échanges client/client. A chaque fois qu'un client se connecte (à chaque accept sur la `SocketServer` du Client), un thread `ClientHandler` est créé (Une tache est ajoutée au threadpool). Il traite les requêtes reçues et se termine. La fermeture de la connexion nous a posé problème. Le client n'a pas moyen de savoir qu'un client a fini son téléchargement, ou qu'il s'est déconnecté. Nous avons donc posé un timeout de 10 secondes sur les sockets clients. Cela permet, même en cas de perte/fin de connexion, de fermer le thread en cours.

3.2.3 Le userHandler

Le `UserHandler` est une interface entre le client et l'utilisateur de `CookieTorrent`. Il permet à l'utilisateur de regarder si un fichier est disponible chez d'autres clients (requête `look`) grâce à plusieurs critères comme le nom du fichier ou sa taille, de choisir les fichiers qu'il souhaite télécharger et de gérer les fichiers en partage. Nous avons réalisé une interface graphique grâce au package `swing`.

FIGURE 2 – Interface du client

Enfin nous avons ajouté deux autres `ThreadPool` pour gérer les téléchargements pa-

rallèles et pour envoyer plusieurs requêtes à la fois. Nous rentrerons plus dans les détails sur ces threads par la suite.

3.3 Gestion des fichiers

Nous avons implémenté quelques fonctions qui permettent à l'utilisateur de gérer les fichiers en partage. Tous les fichiers qu'il souhaite partager doivent être présents dans le dossier `shared`. A partir de l'interface il peut ajouter ou supprimer un fichier. Tous les fichiers en partage sont affichés dans la fenêtre principale de `CookieTorrent`. Néanmoins il faut redémarrer le client pour que ces changements soit pris en compte par le tracker. En effet le protocole ne permet pas d'ajouter un fichier qui n'est pas présent sur le réseau après le `announce` de début de connexion. Ces fichiers en partage sont initialisés grâce au fichier Shared.state et y sont sauvegardés après la fermeture du client.

3.4 Traitement des requêtes

list La requête `list` est la réponse du tracker à la requete `look`. Le traitement de cette requête consiste à créer les objets `CookieFile` (cf Documentation) pour chaque fichier reçu dans la requête. Enfin pour que l'utilisateur puisse voir le résultat de cette requête nous devons envoyer ces `CookieFiles` au thread de l'interface. Nous utilisons une instance de la classe `BlockingQueue` pour envoyer les `CookieFiles` à l'`UserHandler` qui affiche les différents fichiers. Un message d'erreur s'affiche s'il n'y a aucun fichier dans la réponse.

peers La requête peers contient les ip et port des peers qui possède le fichier (où une partie) que l'on souhaite télécharger. Le traitement de cette requête consiste à créer une nouvelle socket pour chaque peer. Cette solution n'est pas entièrement satisfaisante, si le nombre de peer est important pour ce fichier, la consommation de ressource est importante.

Interested Le traitement de la requete `insterested` est plus simple, il suffit de retourner au client le `bufferMap` du fichier concerné.

have Les requêtes have n'ont pas de traitement particulier. Les bufferMap sont simplement stockés dans un ArrayList. Cet ArrayList de buffer est ensuite traité pour constituer une `RepartitionStructure` dont nous verrons les détails plus tard.

getpieces Le traitement de cette requête consiste à lire `pieceSize` octets dans le fichier concerné avec le bon offset pour chaque index de la liste. Pour cela nous utilisons un reader de la classe `RandomAccessFile`. Lors de ce traitement il est nécessaire de poser un `ReadLock`. Les pièces que nous envoyons sont de type String. Nous avons encodé les données binaires en Base64.

data De la même manière que pour getpieces il s'agit ici d'écrire la pièce dans le fichier avec le bon offset. Nous utilisons la même classe, `RandomAccessFile`, pour cet opération en posant un verrou `WriteLock`. Il faut aussi mettre à jour le bufferMap de ce fichier.

3.5 Gestion des téléchargements

Un téléchargement est un processus en plusieurs phases. La première consiste a récolter les bufferMap des peers qui possèdent le fichier (ou une partie). Il faut ensuite déterminer a quel peers prendre quelles pièces afin d'avoir un fichier complet à la fin du processus. Cette partie utilise deux `ThreadPool`. Le premier pool, instancié dans le `trackerHandler` permet de lancer plusieurs téléchargements en parallèle. Le second permet que l'envoi des `interested` et `getpieces` aux peers se fassent en parallèle. Ces threads nous ont posé des problèmes de concurrence, notamment au niveau de l'écriture et de la lecture dans les fichiers. Nous avons pour éviter ces problèmes utilisé des `ReadWriteLocks`. Un autre problème d'accès concurrent persiste. Nous avons pu le régler uniquement en ajoutant un délai entre les `getpieces`...

3.5.1 Envoi des Interested et Traitements des bufferMap

La première phase consiste à envoyer à chaque peer une requête `interested` pour connaître son bufferMap. Cette collecte est effectuée par un pool de thread. Les bufferMap sont ensuite traités et regroupés dans une structure que l'on a appelé `RepartitionStructure` dont voici le schéma : (Pour un fichier de 10 pieces)

Dans le schéma, nous pouvons voir que notre client dispose déjà des pieces 2, 5 7, 9 et 10. Le peer 1 possèdes les pièces 1, 4 et 8. Personne ne possède la piece 6. La construction de cette structure a une complexité en $\Theta(NbPieces * NbPeers)$. Elle est implémentée avec des ArrayList.

3.5.2 Algorithme de téléchargement

L'algorithme de téléchargement que nous avons mis en place reste simple. Nous parcourons la `RepartitionStructure`. Pour chaque piece le client tire aléatoirement un peer parmi ceux qui possèdent la pièce et lui envoie un getpiece pour cette pièce. Arrivé à la fin de la structure, on teste si le fichier est complet et on vérifie sa clé. Si tout concorde, le téléchargement prend fin. Sinon on met à jour notre structure avec notre bufferMap et les nouveaux bufferMaps des peers.

Cet algorithme n'est pas optimisé. Un meilleur algorithme consisterait à mesurer la bande passante de chaque peer connecté pour essayer de maximiser son débit descendant. Néanmoins cet algorithme a le mérite de répartir les pièces téléchargées entre les peers disponibles.

Dans le cas où le fichier que l'on souhaite télécharger n'est pas présent en totalité sur le réseau, le client est capable de détecter que les pièces qu'il lui manque ne sont pas accessibles. Dans ce cas il arrête le téléchargement et toutes les secondes il essaye de le reprendre en envoyant un `getfile` au tracker.

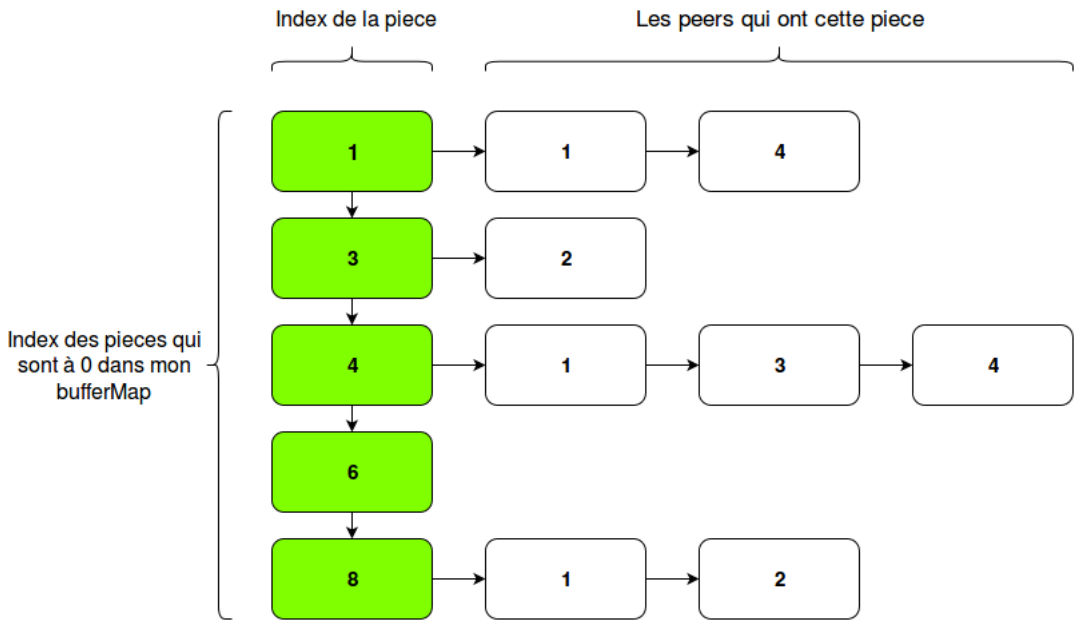

FIGURE 3 – Structure de Repartition

Enfin la déconnexion d'un peer en cours de téléchargement déclenche une levée d'exceptions qui ne sont pas gérées.

4 Le Tracker

4.1 Le parseur

Le parseur permet d'interpréter les commandes reçues sur la socket. Quand le tracker reçoit un message, il appelle la fonction `parse`, en lui fournissant en paramètre la chaîne de caractères reçue.

Le parseur parcourt la chaîne de caractères, et pour chaque information rencontrée, il remplit la structure `command`. Cette structure contient les informations nécessaires au traitement de chacune des commandes (type de la commande, listes des fichiers pour `announce`, numéro de port, ...).

Une fois la structure remplie, elle est retournée au tracker, qui peut facilement avoir accès à toutes les données.

4.2 Gestion des clients et des fichiers

Le rôle du tracker est de retenir l'ensemble des fichiers présents sur le réseau, et d'informer les clients de leur localisation.

Pour cela, nous avons donc choisi d'implémenter une liste chaînée globale 'client_lnk', qui contient tous les clients connectés au tracker. A chaque connexion, une structure client

Algorithm 1 Téléchargement d'un fichier

```
while File is not complete do
  Repartition ← getPeersBufferMaps()
  for all pieces in Repartition do
    peer ← Rand()
    send(getpiece, pieceIndex, peer)
  end for
end while
if Correct Key then
  End Download
else {Bad key}
  Corrupted file
end if
```

est créé, et insérée dans la liste. Cette structure possède elle-même une liste chaînée de fichiers. Ainsi pour chaque client, on peut connaître les fichiers qu'il partage.

La structure client contient l'id de la socket du client, son adresse ip, son port d'écoute, et la liste de ses fichiers. C'est l'id de la socket de connexion qui identifie un client de manière unique.

La structure d'un fichier contient son nom, sa taille, la taille dans laquelle il est découpé, et sa clé.

4.3 Traitement des requêtes

Lorsque le tracker reçoit la structure command, il la redirige vers la fonction correspondante en fonction du type de la commande (`announce`, `look`, `getfile` ou `update_seed`). Ces fonctions s'occupent de traiter la commande, et de répondre au client sur la socket.

announce Cette fonction inscrit le port d'écoute du client dans sa structure, puis ajoute à la liste de ses fichiers l'ensemble des fichiers annoncés (seed ou leech). Pour les fichiers leech, seule la clé est renseignée, et le nom du fichier est par défaut '*'. Quand tout est mis à jour, la fonction écrit sur la socket du client '`ok`'.

look Cette fonction parcourt l'ensemble des fichiers de chaque client, et crée une liste chaînée des fonctions respectant le premier critère demandé dans la commande (recherche par nom ou par taille). Puis, dans le cas où plusieurs critères sont demandés, il parcourt à nouveau cette liste chaînée et retire pour chaque critère les fichiers qui ne le satisfont pas (à l'aide d'une fonction auxiliaire `criterion_respected`. Quand tous les critères ont été pris en compte, la fonction écrit sur la socket `list [xxx]` où xxx est la liste des fichiers respectant l'ensemble des critères. Cependant, pour rendre l'utilisation plus intuitive, le programme client ne permet pas de préciser plusieurs critères de recherche.

getfile La fonction getfile parcourt l'ensemble des fichiers de chaque client, et retourne sur la socket `peers $key [$ip1:$port1 ...]` contenant l'ensemble des adresses et ports des clients possédant le fichier demandé.

update_seed Cette fonction permet de tenir régulièrement au courant le tracker des fichiers que possède chaque client. Le protocole ne permet pas à un client de rajouter un nouveau fichier sur le réseau (car seules les clés des fichiers sont précisées dans cette commande), en revanche, un client peut posséder de nouveaux fichiers qu'il a téléchargés chez d'autres clients (ou seulement une partie). Pour gérer cela, on réutilise la fonction `announce`, qui permet de mettre à jour la base de données.

4.4 Threads

Il a été choisi d'utiliser un module de thread pool et de démarrer un thread parmi ce thread pool lorsqu'un message est reçu par le serveur. Ainsi avec quelques threads seulement, plusieurs clients sont gérés sans latence apparente.

Un problème du thread pool est qu'un client peut créer des bouchons sur le thread pool en envoyant une multitude de requêtes. Pour palier à ce deuxième problème, un seul thread par client peut être utilisé pour le traitement d'un message reçu.

L'utilisation de threads implique des problèmes d'accès concurrentiels. Ainsi des erreurs peuvent survenir lors de la déconnexion d'un client. En effet un thread est dédié à la détection de la réception d'un message et parcourt la liste des clients. Lorsqu'un client est supprimé de la liste, le thread qui la parcourt peut obtenir une erreur de segmentation.

Pour remédier à cela un verrou a été posé sur la liste sous la forme d'un entier. Malheureusement ce verrou n'est pas complètement sûr car entre la vérification du verrou et le verrouillage un autre thread peut faire de même. Néanmoins ce verrou permet de rendre plus rare une erreur qui l'était déjà.

L'attente du déverrouillage de la liste par un thread est une attente active car ce temps est supposé assez faible.

Annexes

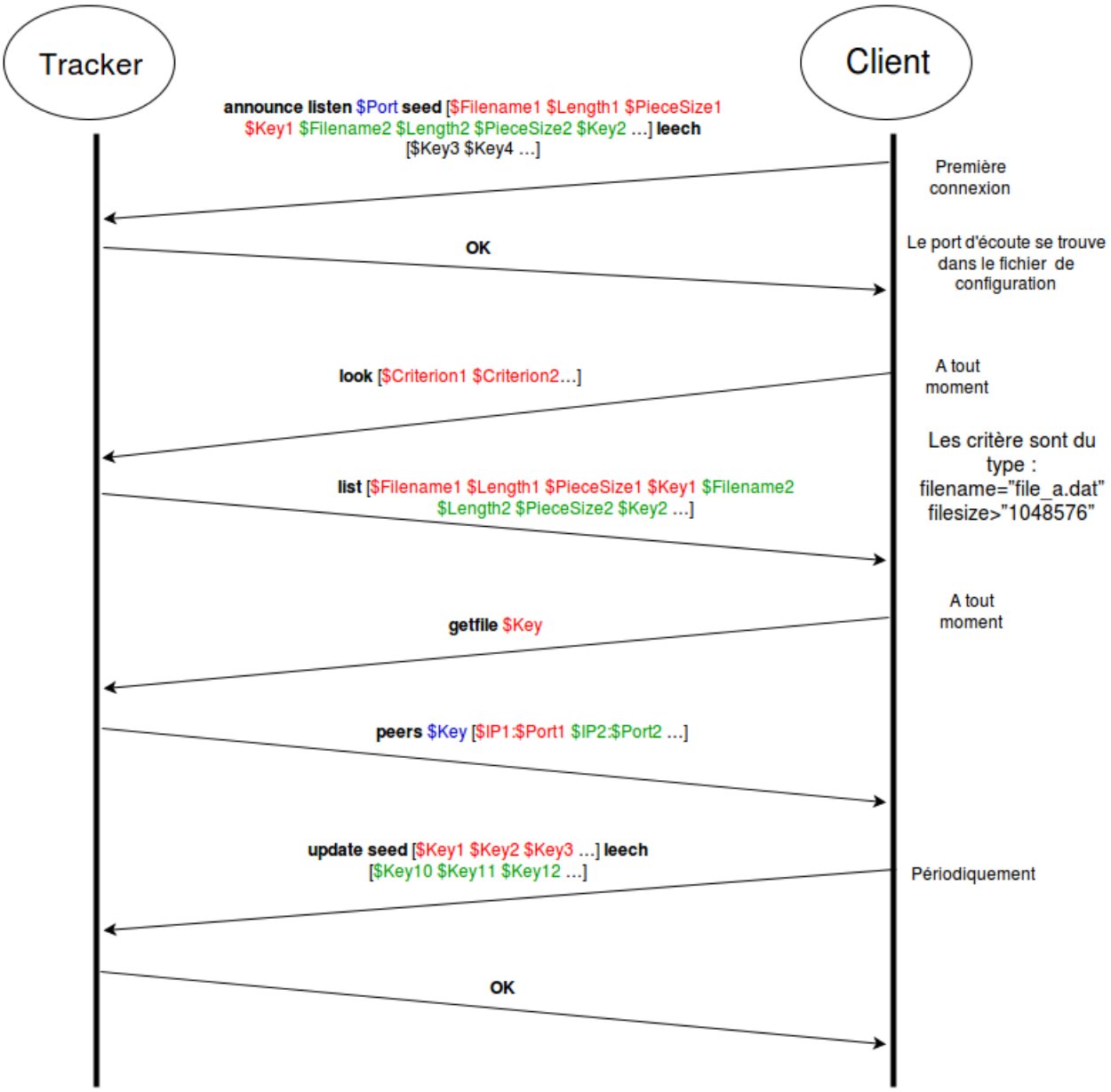

FIGURE 4 – Echanges CLient / Tracker

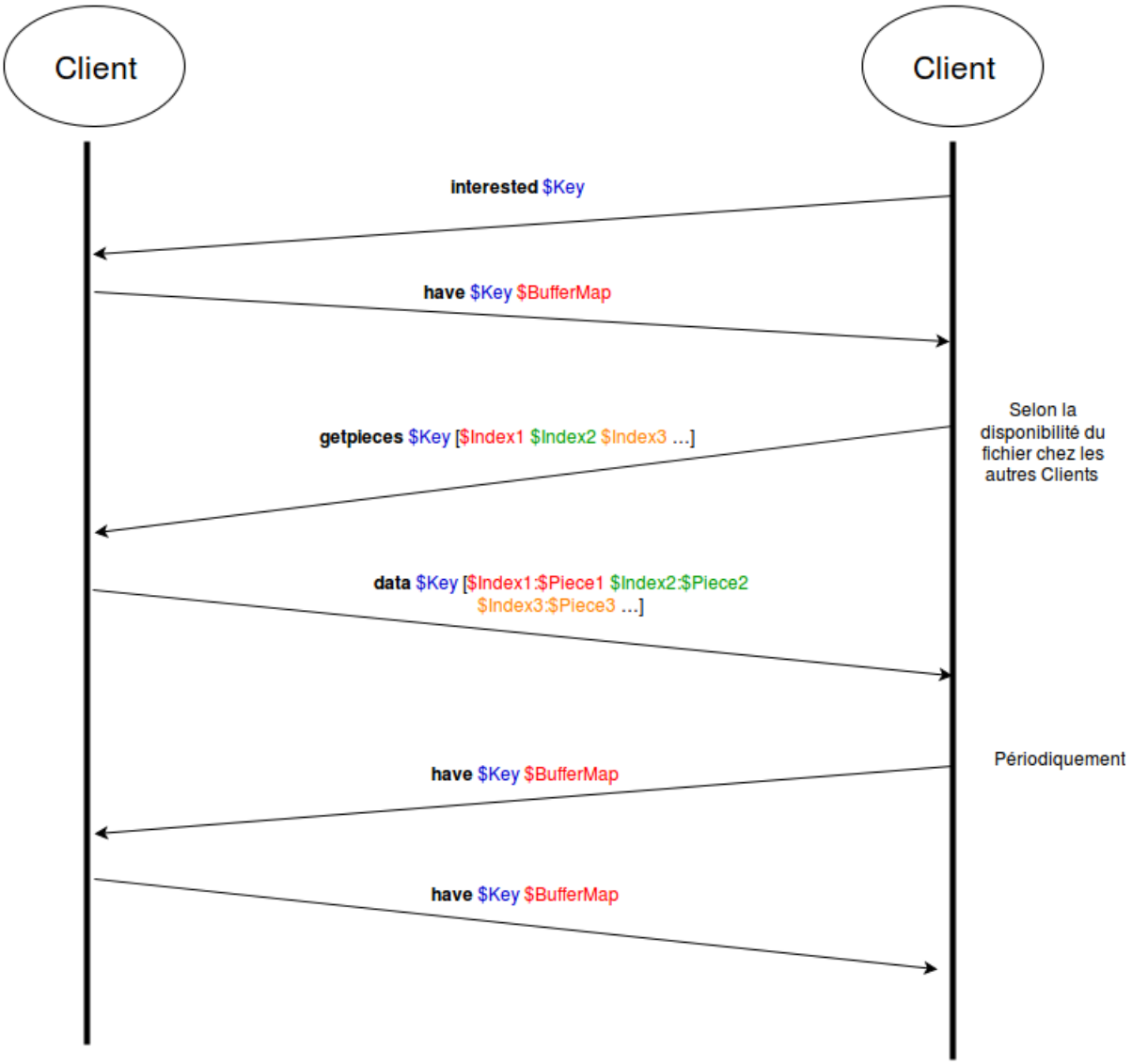

FIGURE 5 – Echanges Client / Client