Projet de Réseau
Rapport intermédiaire

Song Chen
Rouxel Quentin
Abid Rachid
Gueye Papa Magueye

25 mars 2012

Introduction

Le but du projet est de développer une application pour le partage de fichiers en mode pair à pair (Peer-to-Peer). Pour le partage de fichier en P2P, un pair est à la fois serveur et client des autres pairs. La première partie de ce projet consiste à développer conjointement un client et un serveur (tracker). Le tracker a pour fonction de récupérer, gérer et redistribuer l'information entre les différents pairs, ces derniers utilisant ces informations pour communiquer entre eux.

Chapitre 1

Tracker

1.1 Tracker

Le tracker consiste à faire le lien entre les différents pairs, il récupère les informations de chaque client, les stocke et les redistribue au besoin. Le langage choisi pour implémenter cette application est le C. Le travail effectué sur le tracker s'articule en 3 parties :

- le serveur TCP
- la base de données
- le parseur de requêtes
- le traitement des requêtes

1.1.1 La base de données

La base de données repose essentiellement sur une liste de structures "`struct mainTrackerElement`". Chaque `mainTrackerElement` de cette liste correspond à un fichier et est repéré par sa clé md5, car en effet, c'est le seul paramètre qui permet d'identifier un fichier de manière unique et qui permet ainsi le moins de redondance d'informations dans la base. Ci-dessous le code de cette structure :

```
struct mainTrackerElement {
        struct link* peers;
        char *keyP;
        int size;
        int pieceSize;
        struct link* noms;
};
```

- `struct link *peers` liste les différents pairs sous forme de chaîne de caractères du type [$IP1 :$PORT1 ...].
- `char *keyP` est la clé md5 correspondant au fichier.
- `int size` est la taille du fichier en octets.

- `int pieceSize` est la taille des pièces en octets.
- `struct link *noms` liste les différents noms de ce fichier sous forme de chaîne de caractères.

1.1.2 Le parseur de requêtes

Pour le parseur de requêtes le choix a été fait d'utiliser lex/yacc. L'utilisation d'une grammaire a été privilégié pour sa flexibilité - il est par exemple possible d'inverser les paramètres, ne pas se soucier des espaces etc. - et étant donné que le sujet de projet était sujet à des mises à jour, les modifications du code probables étaient ainsi rendues plus simples.
L'une des difficultés était de faire accepter en entrée du parseur une chaîne de caractères de type `char*` car yacc n'est pas prévu pour de manière native. Il a donc fallu coder une fonction `readInput()` pour simuler la lecture dans un fichier. Ensuite, il a fallu spécifier un lexeur (avec *lex*) pour analyser les mots de la requête et enfin une grammaire qui en parcourant l'arbre syntaxique, rempli une structure de données qui contiendra toutes les informations incluses dans chaque requête. Ci-dessous la structure en question :

```
struct commandLine {
        enum type type;

        // announce:
        int port;
        int filesNumber;
        char* fileNames[MAX_FILES_NUMBER];
        int lengths[MAX_FILES_NUMBER];
        int pieceSize[MAX_FILES_NUMBER];
        char *keys[MAX_FILES_NUMBER];

        // look:
        char *fileName;
        int supTo;
        int infTo;

        // getfile:
        char *getKey;

        // update:
        int seedKeysNumber;
        int leechKeysNumber;
        int isSeeder;
        int isLeecher;
        char* seedKeys[MAX_FILES_NUMBER];
        char* leechKeys[MAX_FILES_NUMBER];
};
```

Chaque bloque correspondant à un type de requête (`announce`, `getfile`, `getfile` ou `update`). Cette structure est déclarée dans le programme en tant que variable globale afin d'être accessible par *yacc*. Elle fait partie des données sur lesquelles il faut mettre un mutex pour gérer les accès concurrents des différents clients.

1.1.3 Le serveur TCP

Cette partie repose simplement sur 3 fonctions :

- `cree_socket_stream()` consiste à initialiser la connexion en créant la socket (*socket*()) et la liant à un point de communication (*bind*()) défini par l'adresse locale et le port d'écoute.
- `serveur_tcp()` écoute sur la port prévu (*listen*()) `socket_contact` et créé un thread (*pthread_create*()) par connexion entrante.
- `traite_connexion()` est la fonction de thread : elle prend en paramètre le descripteur de socket retourné par (*accept*()). Cette fonction reçoit une requête grâce à *recv*() puis lance le parseur par `yyparse()`. A ce moment là, elle lance la fonction `cloneCommandLine()` qui clone la structure contenant les données de la requête afin de libérer cette ressource partagée et lance la fonction de traitement appropriée en fonction du type de la requête.

Les fonctions de traitement sont au nombre de 4 et chacune correspond à un type de requête :

- `announce_tracker()`
- `look_tracker()`
- `getFile_tracket()`
- `update_tracker()`

Chacune de ces fonctions effectue le traitement associé au type de la requête (mises à jour, recherche, modifications etc.).

Chapitre 2

Pair

2.1 Motivations du choix du langage : Java

Clairement, le plus gros du travail d'implémentation de ce projet se situe au niveau du client. Le client a d'une part les fichiers sur le disque à gérer mais également plus de messages différents dans le protocole à implémenter. Pour cette raison, nous avons préféré utiliser pour le développement du pair un langage de haut niveau tel que Java afin de nous simplifier la tâche et de laisser l'emploi du C au tracker, plus simple.

2.2 Utilisation du buffermap

Pour que le pair puisse connaître les pièces disponibles d'un fichier temporaire, un buffermap est mis en place dans la structure de chaque fichier. Il est une séquence de bits dont le nombre correspond au nombre des pièces du fichier. Chaque bit indique la présence de la pièce correspondante dans le buffer.

En considérant le nombre de bits par rapport à la taille d'un fichier, le buffermap dans notre structure est implémenté comme un tableau. On constate ensuite que pour une pièce de fichier, il existe trois états différents pour lesquels on doit effectuer différents traitements :

- soit cette pièce n'existe pas localement. Dans ce cas là, le pair demande à son voisin leur disponibilité sur cette pièce.
- soit cette pièce est en train de téléchargée. Donc le pair n'a pas besoin de faire le demande sur cette pièce mais il ne peut pas non plus de fournir des données de cette pièce, car ces dernières ne sont pas complètes.
- soit elle est finie de télécharger ou est déjà disponible dans le disque. Elle est prêt d'être envoyée au voisin.

Pour la raison qu'on doit coder ces 3 cas dans une case de tableau, un bit binaire n'est pas suffisant. C'est pourquoi on définit un tableau d'entier.

2.3 Détection de la fin de transmission

Une autre problématique technique est de détecter la fin d'une transmission de message. En effet, le protocole autorise à transférer des données binaires. Il est alors impossible de convenir d'un symbole correspondant à la fin de la transmission. Puisque le protocole n'indique pas la taille des données binaires, il faut à partir des données connues calculer la taille des données à lire. De ce fait, la détection de la fin de la transmission est spécifique à chaque message du protocole.

2.4 Stockage des fichiers temporaires

Les gestions des données temporaires, c'est à dire des pièces d'un fichier en cours de téléchargement est également un problème difficile. Dans une première approche, nous pensions écrire dans un répertoire approprié sur le disque chaque pièces dans un fichier séparé. Cependant, vu la taille réduite de chaque pièce et le grand nombre de pièces potentielles pour les grand fichiers (plusieurs millions) nous avons choisi une autre approche afin de ne pas saturer le système de fichier.

Un unique fichier temporaire est créé par fichier en cours de téléchargement. Les pièces sont écrites les unes à la suite des autres en binaires dans ce fichier (pas dans l'ordre des données mais dans l'ordre de réception des pièces).

Bien entendu, ces fichiers comportent un header spécifique regroupant les informations essentielles du fichier en cours de téléchargement (taille, clef et taille des pièces) mais également la disposition des pièces dans le fichier.

Conclusion

À ce jour, la totalité des fonctionnalités du tracker demandées par le sujet sont implémentées, un client a été implémenté en C pour réaliser différents tests. Il s'avère que tout semble fonctionner néanmoins il reste quelques détails à régler tels que des fuites mémoires, la pose de mutex pour gérer les accès concurrents, remettre un peu d'ordre dans le code, commenter etc.. En ce qui concerne le client, le réalisation n'est pas encore terminée. En effet il a été apporté une grande importance et un grand soin à la gestion de fichiers afin d'avoir une application qui pourrait réellement être déployée sur le réseau internet. Cependant il s'avère que ce n'est pas le but réel du projet, c'est pourquoi une autre direction a été prise ce weekend pour rendre le client fonctionnel le plus rapidement possible.