INTERGICIEL & SYSTÈMES CONCURRENTS

EVALUATION DE HDFS
HIDOOP V0

November 13, 2017

Alexys Dussier & Marie Elisabelar
ENSEEIHT
Informatique et Mathématiques Apppliquées

Sommaire

Introduction

1 PARTIE TECHNIQUE

1.1 Tests & Bugs

Les serveurs et le client fonctionnent en local. Les serveurs démarrent correctement. On part du principe qu'il n'y a qu'un seul client qui envoie des fichiers. Nous avons remarqué que certaines commandes faisaient planter le premier serveur à être contacté par le client, notamment :

- demande de lecture d'un fichier qui n'existe pas sur le serveur;
- demande d'écriture d'un fichier qui n'existe pas en local sur le client.

Ces commandes lèvent des exceptions qui terminent le serveur. Il pourrait être pertinent de changer le traîtement de l'exception par un message d'erreur envoyé au client dans le terminal. Néanmoins, le serveur gère sans soucis plusieurs fichiers, du moment qu'ils n'ont pas le même nom. Si ce n'est pas le cas, les anciens fragments sont écrasés. Une possible amélioration serait de demander confirmation de l'écrasement des données.

Nous avons essayé d'envoyer différents fichiers Line sur le serveur. Si le fichier contient au moins une ligne, il est fragmenté sans problèmes (voir en annexe pour un exemple de lecture des fragments). En revanche, lors de la lecture du fichier, le fichier résultat est différent de celui envoyé au début : il contient un retour à la ligne supplémentaire à la fin. Si le fichier à écrire est vide, il est tout de même fragmenté et chaque serveur se voit attribué un fragment contenant une ArrayList vide. Si à notre échelle cela ne pose pas de problème, on peut facilement imaginer qu'à une échelle de plusieurs centaines de clients, de tels fragments vides peuvent prendre beaucoup de place pour rien. Il pourrait être pertinent, lors de l'écriture d'un petit fichier, de limiter le nombre de serveurs sur lesquels il sera fragmenté. De plus, lors de la lecture d'un fichier vide, aucun fichier résultat n'est créé.

Lorsqu'on envoie un fichier au format KV, il est fragmenté correctement (sauf encore une fois si le fichier est vide). Néanmoins, lors d'une demande de lecture, le fichier renvoyé n'est pas généré correctement : une Exception de Stream Corrompu (StreamCorruptedException) indique qu'il y a eu une erreur lors de l'écriture des fragments dans le fichier résultat. Nous avons relevé cette exception en exécutant notre programme LectureKV (voir en annexe). Le format KV n'est donc pas utilisable pour le moment.

1.2 Validation

Le code fourni fonctionne, bien que ce ne soit pas en réparti et uniquement en local. Ainsi on peut écrire un fichier en le fragmentant, puis le lire en concaténant les fragments, et on peut aussi détruire les fragments. Tout cela fonctionne globalement avec le format Line, mais pour le format KV, un bug persiste au niveau de la lecture, le fichier lu ne correspond pas au fichier écrit et est donc illisible, comme dit précédemment.

1.3 Performance

Le système utilise des sockets, faits pour transporter de grandes quantités de données, ce qui est un choix performant. Cependant, pour communiquer entre leurs différents processus, ils ouvrent et ferment autant de sockets que d'objets à transferer. On pourrait envisager de limiter les ouvertures et fermtures de sockets pour améliorer laperformance globale.

Dans leur fonction write() de HDFSClient, le programme close() puis open() à nouveau le fichier pour revenir au début. Cela dessert la performance, et on pourrait préférer les méthodes mark() et reset(), qui permettent de revenir à un endroit donné du fichier.

Dans la classe Format, dans open(), on utilise un système de communication à distance pour récupérer le chemin relatif du fichier, ce qui est très lourd pour ce petit problème. Il aurait fallu stocker cette variable en local plus tôt dans l'algorithme, ou créer une fonction de conversion locale. Cependant, comme ce n'était pas demandé de gérer l'arborescence dans la V0, cette amélioration ne peut être prise que positivement.

1.4 Qualité du Code

Le code n'est pas commenté ni aéré, ce qui est dommageable pour sa relecture et ses futures améliorations. On remarque aussi certaines répétitions de code qui seront mentionnées par la suite.

2 SYNTHÈSE

2.1 Correction

Nous avons pu voir dans la partie Validation que le produit fonctionnait plutôt correctement.

Nous pouvons toutefois noter que si plusieurs Clients cherchent à accéder en simultané à un même Serveur, comme leur protocole de communication utilise plusieurs Messages dans un ordre préci, il y aura des problèmes de concurrence, mais c'est normal car il s'agit de la V0.

2.2 Complétude

Il fallait faire une application répartie sur plusieurs machines. Leur application ne fonctionne que sur une même machine en local. Ce n'est pas tout à fait ce qui était demandé.Cependant, l'aspect réparti des serveurs est simulé car chacun est identifié par un numéro de port, utilisé pour les contacter.

2.3 Pertinence

Dans la classe HDFSClient, certains attributs sont déclarés en static alors qu'ils ne sont pas communs à tous les objets de cette classe, il s'agit d'une erreur de conception, il faudrait plutôt mettre ces attributs en tant que variables temporaires dans le main.

Dans la classe HDFSClient, dans la fonction read(), on ouvre et on ferme un fichier autant de fois que l'on écrit dedans, ce qui est très lourd. Il faudrait ouvrir ce fichier une seule fois, puis écrire tout ce qu'on a à écrire, puis enfin le fermer.

2.4 Cohérence

Dans HDFSClient, dans la fonction write(), on a une duplication de code immense pour faire la distinction entre le type KV et le type Line. Il faudrait soit créer des méthodes génériques (qui de plus pourraient s'adapter à encore plus de types de formats), soit faire des switch uniquement aux endroits nécessaires.

Dans Format, des Messages sont en attributs alors qu'ils ne définissent pas un Format. On pourrait croire qu'un Format contient des Messages, ce qui n'est pas le cas, il les utilise juste. On aurait pû uniquement les définir dans la fonction les utilisant (ici open()).

Dans une grande partie des fonctions, on utilise des catch de plusieurs exceptions à la fois sans faire de distinctions entres celles-ci, engendrant par là une duplication importante du code. On pourrait faire un catch sur uniquement l'exception "Exception e".

Toujours sur la classe Message, on réalise des open() et des close() à chaque send() et à chaque receive(). Si cette méthode permet de simplifier la vie du programmeur, cela diminue grandement la performance car on ouvre et ferme une nouvelle connexion (socket) à chaque message. On aurait aimer voir des méthodes open() et close() dans la classe Message que le programmeur aurait dû appeler avant de faire ses send() et receive().

Les méthodes send() et receive() de la classe Message sont surchargées. Cependant ces surcharges ne sont pas pertinentes car elles implémentent des fonctions différentes, en effet l'une est faite pour envoyer un message à un serveur, et l'autre un message à un client, ce qui est différent. On aurait préféré renommer ces méthodes en send2Client() et send2Serveur(). Ou alors on aurait pû créer une unique méthode send() qui aurait pris en paramètre le nom de la machine à qui envoyer le message (et le port si besoin est).

Dans les Formats, chaque write() modifie l'état de la mémoire en retenant dans une ArrayList le KV que l'utilisateur veut écrire en mémoire. Puis c'est uniquement lorsque le close() est appelé que tout le fichier est écrit en mémoire dure. C'est une très bonne idée d'implémentation car cela permet d'éviter d'écrire plein de fois en mémoire dure, ce qui aurait été très long.

Toujours dans les Formats, pour le stockage de leurs fichiers en dur, ils utilisent writeObject() pour écrire d'abord le type du Format, puis pour écrire une ArrayList contenant l'ensemble des KV à écrire. Il s'agit d'une implémentation propre, simple et efficace qui sera facilement réutilisable. Ce qui est très positif.

3 AUTRES PISTES D'AMÉLIORATION

Voici d'autres pistes d'amélioration pour leur code :

Point à travailler	Problème engendré	Amélioration proposée
La création de la classe Message était une très bonne idée puisqu'elle permettait de bien séparer ce qui relevait de la communication inter-processus de ce qui relevait du code en lui-même. Cependant, pour pouvoir l'utiliser il faut créer un objet Message pour chaque type d'objet que l'on souhaite envoyer	Cela nous force à dupliquer du code pour créer un Message pour chacun des types que l'on veut envoyer	On aurait aimé voir les méthodes send() et receive() de Message attendre des Objects en paramètres et les voir être cast dans le type attendu lors de la réception
Dans Format, mise du port du serveur en paramètre du constructeur	La classe Format ne devrait pas avoir à s'occuper des systèmes de communication (mauvais placement de l'information)	Ils devraient créer une autre classe pour gérer le port, ou l'enregistrer à une couche supérieure à celle du Format

ANNEXES

Code de lecture des fragments Line

```
package hdfs;
import ...
public class LectureLine {
    public static void main (String args[]) {
        File f = new File("titi.txt2");
        FileInputStream fis = null;
        try {
            fis = new FileInputStream("titi.txt2");
            ObjectInputStream ois = new ObjectInputStream(fis);

            Format.Type type = (Format.Type)ois.readObject();
            ArrayList<String> array = (ArrayList<String>)ois.readObject();

            for (String o : array){
                System.out.println(o);
            }
            ois.close();
            fis.close();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Le code pour lire des fragments KV est le même, à l'exception de l'extension du nom de fichier et le type d'élément de l'ArrayList.

Code de lecture d'un fichier KV non fragmenté :

```
package hdfs;
import ...
public class LectureKV {
        public static void main(String[] args) {
                File f = new File("toto.kv");
                FileInputStream fis = null;
                int nbKV = 4;
                try {
                        fis = new FileInputStream("toto.kv");
                        ObjectInputStream ois = new ObjectInputStream(fis);

                        KV kv = null;
                        for (int i = 0; i<nbKV; i++){
                                kv = (KV) ois.readObject();
                                System.out.println(kv);
                                        }

                        ois.close();
                        fis.close();

                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```