Amirkaveh Chervin
Laporte Adrien

ENSEEIHT - Toulouse INP
Étudiants 2ème année
Filière SN spécialité SIL

Rapport Hidoop V0

Chervin Amirkaveh & Adrien Laporte

Table des matières

Introduction

Dans ce projet, nous découvrons les applications concurrentes appliquées au calcul intensif et au traitement de données. Dans ce but, nous avons implanté la partie Hidoop du projet en particulier nous avons implémenté les interfaces suivantes : JobInterfaceX, Daemon ainsi que CallBack que nous avons créé. Nous avons également implémenté les formats à travers une classe abstraite et deux classes représentant les formats.

1. Implémentation de JobInterface

1.1 Interface JobInterface

L'interface JobInterface contient uniquement trois procédures : un setter pour le nom d'entrée du fichier et un pour son format ainsi que la procédure startJob. Cependant, cette inferface n'étant pas assez complète nous avons utilisé l'interface JobInterfaceX qui contient 4 setters de plus et 6 getters qui correspondent aux attributs suivants :

- inputFormat : le format d'entrée
- inputFname : le nom du fichier d'entrée
- nbReduce : le nombre de ***reduce***
- nbMap : le nombre de ***map***
- outputFormat : le format de sortie
- outputFname : le nom du fichier de sortie

Nous avons choisi de ne pas utiliser le *SortComparator* car nous nous n'y voyons pas d'utilité actuellement dans notre projet.

1.2 Classe Job

Tout d'abord nous avons implémenté le constructeur Job() où nous donnons les valeurs par défaut des différents attributs de la classe. Dans la version actuelle nous avons choisi de mettre 3 *maps* et 1 reduce. On a défini également des noms d'entrée et de sortie et les formats qui seront LINE en entrée et KV en sortie.

Ensuite nous avons implémenté la méthode ***startJob(MapReduce mr)*** qui est la méthode la plus importante de cette classe. Elle se divise en deux parties distinctes :

- Une partie dédiée au lancement des map*s* sur les serveurs distants, ce qui permet le traitement des fragments de fichiers.
- Une autre partie consacrée au lancement du *reduce* qui permet l'assemblage des des résultats de map en local

Le lancement de la fonction map sur l'ensemble des serveurs distants nécessite d'abord de se connecter à ces derniers. On utilisera pour cela l'architecture RMI et notamment Naming.lookup. On crée ensuite un ***CallBack*** qui transmet une ***cycleBarrier*** instanciée précédemment. Elle a pour but de synchroniser les exécutions des maps sur les serveurs distants. Ainsi la prodécure attend la fin de l'exécution de tous les maps avant de passer à la partie reduce.

Avant de lancer la fonction ***reduce***, il nous faut d'abord collecter l'ensembe des résultats issus des maps. Pour cela on lance la méthode ***HdfsRead*** de la classe ***HdfsClient*** qui a en paramètres le nom des fichiers à récupérer. Une fois ces fichiers en local, on pourra faire appel à la méthode reduce. Il faudra cependant créer préalablement des ***Formats*** et les ouvrir afin qu'ils soient utilisés par la fonction. Enfin, on n'oubliera pas de fermer les formats ouverts.

Par la suite on définit très simplement les getters et les setters sans oublier pour les setters de lancer des exceptions lorsque le paramètre d'entrée n'est pas correct. Par exemple le nombre de reduce est forcément positif ou le format d'entrée ne peut pas être nul.

2. Implémentation de Daemon

2.1 Interface Daemon

L'interface Daemon est associée au côté serveur du RMI. Cette interface étend donc logiquement la classe Remote, en tant que serveur RMI. La seule méthode présente dans cette interface est la méthode runMap. Cette méthode permet de lancer map sur chaque Daemon. En tant que méthode appelée à distance, elle doit lever l'exception RemoteException.

2.2 Classe DaemonImpl

La classe DaemonImpl implémente l'interface Daemon, et en tant qu'implémentation de celle-ci, elle étend la classe UnicastRemoteObject. Elle a un attribut, qui est le nom du serveur Daemon, ainsi que deux méthodes : runMap et main. La méthode runMap doit, comme indiqué précédemment, lancer le map. Pour cela, on doit ouvrir un reader et un writer, puis effecteur notre map sur le Mapper m placé en paramètre de notre méthode. Nous n'oublions pas d'effectuer l'appel au finish du CallBack afin d'attendre la fin de ses exécutions.

La méthode main, elle, permet de créer effectivement les Daemon. Nous avon choisi un port afin de créer un Registry puis nous avons appelé un contructeur de Daemon. Pour finir, nous avons bindé un URL avec le Daemon que nous avns créé.

3. Formats

3.1 Classe FormatImpl

La classe FormatImpl est une classe abstraite qui implante l'interface Format. Notre choix de de faire de cette classe une classe abstraite était motivé par notre obligation de faire des méthodes read, et write différentes suivant notre Format (KVFormat ou LineFormat). Mais aussi de notre volonté à synthétiser notre code et à éviter d'alourdir nos fichiers KVFormat et LineFormat avec des méthodes qui seront indentiques dans les deux cas.

Ainsi, nous avons décidé d'implémenter la classe FormatImpl en tant que classe abstraite avec comme seules méthodes abstraites les méthodes read et write car elles sont les deux seules méthodes qui diffèrent suivant le Format

Les méthodes qu'on a implanté ici étaient les suivantes : open, close, getIndex, getFname et setFname. Les trois dernières méthodes sont des getters et des setters classiques appliqués sur des attributs de notre classe.

3.2 Classes KVFormat et LineFormat

Nous nous intéressons maintenant aux classes KVFormat et LineFormat. Comme vu précédemment, ces deux classes étendent la classe FormatImpl. En ce sens, dans ces classes, il nous faut uniquement implanter les méthodes read et write.

Dans le cadre de la classe KVFormat, nous avons décidé de lire la ligne avec le BufferedReader buffR que nous avons en attribut de notre classe FormatImpl de manière protected. Nous avons donc, sans problème, pu l'appeler et lire le contenu de ce buffer, puis nous avons split la ligne selon les KV. Pour la fonction write, nous avons également fait appel à un autre attribut de la classe FormatImpl, le FileWriter FileW. Dans le cadre de la classe LineFormat, nous avons effectué le même appel à FileW que précédemment, à l'exception près que nous n'avons pas écrit les mêmes éléments de notre KV record. Pour ce qui est de la méthode write, nous avons utilisé le même principe que la méthode read

sauf que, une fois le contenu du buffer obtenu, nous avons créé un KV à l'aide du constructeur KV.

Conclusion

Pour conclure, nous avons implanté les classes Job, Daemon, FormatImpl, KVFormat et LineFormat, ainsi que l'interface Daemon. Nous n'avons cependant pas pu tester nos classes car, en plus de ne pas pouvoir tester notre lien avec HDFS, nous n'avons pas eu le temps de tester les Formats, ce que nous aurions pu faire. Pour ce qui est de notre CallBack, il pourrait grandement être amlioré par la suite, cependant, cette première version nous semblait fonctionnelle et nous avons voulu garder des modifications pour une future itération, lorsque nous aurons un HDFS avec lequel tester.