

Rapport final du projet Hidoop

Armelle Jezequel
Loïs Roth
Paul Lacaille
Yannis Benabbi
Alexandre Chatain

15 janvier 2020

Table des matières

1	Revue de la partie HDFS	3
1.1	Structure	3
1.1.1	Le Client HDFS	3
1.1.2	Le Serveur HDFS	3
1.1.3	Le Daemon HDFS	3
1.2	Choix de conception	3
1.2.1	Configuration des daemons	3
1.2.2	Extensibilité	4
1.2.3	Enregistrement des fragments	4
1.2.4	Tests	4
1.3	Synthèse	4
2	Partie Hidoop	5
2.1	Petit manuel d'utilisation	5
2.2	Structure	5
2.3	Choix de conception	5
2.3.1	paramètres de la communication RMI	5
2.3.2	parallélisation des map	5
2.3.3	construction du callback	6
2.3.4	ajout d'un nouveau format de fichier	6
2.3.5	conventions de nommage	6
2.4	Tests	6
2.5	Synthese	7

3 Conclusion

7

2ÈME ANNÉE - SCIENCES DU NUMÉRIQUE



1 Revue de la partie HDFS

1.1 Structure

La partie HDFS se compose de trois parties :

- Le client envoie au serveur la commande qu'il veut effectuer sur un certain fichier.
- Le serveur attend les requêtes de clients et fait le lien entre le client et les daemons des noeuds du cluster. C'est le point d'entrée du cluster.
- Les daemons attendent eux les requêtes du serveur afin de les traiter.

1.1.1 Le Client HDFS

L'exécution du client se fait avec la commande :

- `java hdfs.ClientHDFS <commande> <nomFichier>`
- `<commande>` : 1 = `hdfsWrite` (upload), 2 = `hdfsRead` (download), 3 = `hdfsDelete`

Le client va créer une socket pour se connecter avec le serveur et lui envoyer la commande et le nom du fichier passés en argument. Ensuite, si la commande est 1 (upload), le client va lire le contenu du fichier et petit à petit l'envoyer au serveur. Si la commande est 2 (download), le client va attendre que le serveur lui envoie le contenu du fichier et l'écrire petit à petit dans le fichier. Si la commande est 3 (delete), le client s'arrêtera là car il n'aura plus rien à faire.

1.1.2 Le Serveur HDFS

Le serveur va attendre les requêtes des clients mais avant cela, il faut d'abord initialiser l'environnement. La classe `ServerHDFS` contient un objet static `HashMap` afin de conserver la localisation des fragments de chaque fichier sur les daemons. Ce registre est enregistré dans le fichier `registre.ser` dès qu'il est mis à jour. Il faut donc le charger à chaque fois que le programme `ServerHDFS` est lancé. Le serveur peut ensuite attendre les clients. Lorsqu'un client se connecte, le serveur va ensuite se connecter à son tour aux daemons lancés et leur envoyer la commande et le nom du fichier concerné qu'il a lui-même reçu du client.

Si la commande est un upload, le serveur va recevoir le contenu du fichier par le client, le fragmenter et envoyer les fragments aux daemons. Si la commande est un download, le serveur va aller chercher les fragments du fichier dans le bon ordre avant de les envoyer aux clients.

1.1.3 Le Daemon HDFS

L'exécution du daemon se fait avec la commande :

- `java hdfs.daemon.DaemonHDFS <identifiant>`
- `<identifiant>` : numéro d'identifiant du daemon. Ce numéro doit être compris entre 0 et la taille - 1 des tableaux de configuration de la classe `config/Project.java`

Le Daemon va attendre les requêtes du serveur. Lorsqu'il en reçoit une, il va ouvrir les canaux de communication avec le serveur pour recevoir la commande et le nom du fichier concerné. Si la commande est un upload, le daemon va recevoir les fragments et les enregistrer. Si la commande est un download, le daemon va lire les fragments enregistrés et les envoyer au serveur. Si la commande est un delete, le daemon va supprimer le fichier contenant les fragments du fichier.

1.2 Choix de conception

1.2.1 Configuration des daemons

Afin que les daemons des deux parties soient cohérents, ils utilisent tous la configuration définie dans le fichier `config/Project.java`. Il suffit ainsi de modifier ce fichier si on veut plus ou moins de daemons et leur adresse.

1.2.2 Extensibilité

Pour que les programmes soit extensible, Nous avons décidé d'abstraire certain comportement en commençant par la fragmentation du contenu du fichier. L'interface `ChoixFragmenteurI` déclare une méthode retournant un objet de type `Fragmenteur` à partir d'un nom de fichier et plus particulièrement de son format. L'interface `Fragmenteur` déclare une méthode qui fragmente un tableau d'octet. Cela permet de fragmenter les fichiers de façon appropriée à leur format. La classe `FragmenteurTXT` fragmente les fichiers au format `.txt` de telle façon qu'aucun mot de soit coupé.

1.2.3 Enregistrement des fragments

Les daemons enregistrent tous les fragments d'un même fichier dans un seul fichier. Cela évite des difficultés non seulement lors de la lecture des fragments pour le download, mais évite aussi d'avoir à lancer plusieurs `runMap` en parallèle sur un même daemon pour la partie Hidoop.

1.2.4 Tests

Pour tester la partie HDFS, on teste les trois fonctionnalités proposer par cette partie, le upload, le download et le delete. En utilisant différent fichiers et en appelant les fonctionnalités dans différents ordres, on vérifie que le résultat correspond bien à nos attente. En faisant ainsi varier les paramètres de tests, on prend en compte de plus en plus de cas particuliers. Nous avons donc tester plusieurs scénario et ils ont fonctionnés.

1.3 Synthèse

Le programme fonctionne donc : un fichier peut être envoyer sur le serveur, télécharger du serveur et supprimer du serveur. Le code permet aussi une certaine extensibilité grâce à l'abstraction de la fragmentation des fichiers, mais le programme pourrait encore être amélioré. Pour l'instant, les daemons ne peuvent lire et écrire qu'avec la classe `KVFormat`. Afin de pouvoir changer cela, il faut que le serveur envoie un identifiant en plus de la commande et du nom de fichier permettant d'identifier la classe de type `Format` à utiliser.

2 Partie Hidoop

2.1 Petit manuel d'utilisation

En l'état, le hidoop est fait pour s'exécuter sur une machine de l'ENSEEIH, avec :

- Le MapReduce de votre choix sur la machine courante , dans le répertoire src
- Un daemon tournant sur la machine Griffon (commande depuis le dossier .../src : java DaemonImpl 1)
- Un daemon tournant sur la machine Pixie (commande depuis le dossier .../src : java DaemonImpl 2)

Si on veut l'exécuter en local, il faut décommenter la ligne 6 et commenter la ligne 9 de config/Project.java. Puis il faut faire exactement comme indiqué ci-dessus, mais en lançant les Daemons sur la même machine que le MapReduce.

2.2 Structure

Remarque : Cette section est en grande partie une redite de l'évaluation croisée.

Explication des classes créées :

- config :
 - Project.java : contient les noms des machines et les numéros de ports utilisés dans les communications RMI.
- map :
 - MapReduceImpl_est1.java : le map/reduce qui nous a permis de tester la partie hidoop
- ordo :
 - CallBack : interface du callback
 - CallBackImpl : implémentation du callback
 - CallBackImpl_test : implémentation du callback utilisée pour les tests
 - Daemon : interface du daemon
 - DaemonImpl : implémentation du Daemon
 - DaemonImpl_test : implémentation du Daemon utilisée pour les tests
 - JobInterface : interface du Job
 - Job : implémentation du Daemon
 - Job_test : implémentation du job utilisée pour les tests

2.3 Choix de conception

2.3.1 paramètres de la communication RMI

Dans Project, nous avons inscrit en dur les noms des machines (tableau nomMachine) et les numéros de ports (tableau numPortHidoop) qui seront utilisés par hidoop. Nous avons choisi d'utiliser des numéros de ports différents pour chaque daemon afin que cela fonctionne quand tout se passe en localhost.

Nous avons choisi d'attribuer à chaque daemon un identifiant unique (entier strictement positif). Cet identifiant est l'argument à donner quand on lance un daemon. Cet identifiant correspond à l'indice du nom de la machine sur lequel est lancé le daemon dans nomMachine, et permet de la même manière d'attribuer un numéro de port unique. Il faut donc veiller à lancer les daemons avec les bons ID sur les bonnes machines (daemon 1 sur Griffon et daemon 2 sur Pixie).

2.3.2 parallélisation des map

Nous avons choisi de gérer la parallélisation des maps dans l'objet Daemon : quand Job appelle la méthode runMap sur un Daemon, celle-ci crée un thread secondaire qui effectue le map sur son fragment de fichier et redonne la main du thread principal immédiatement au Job, qui peut alors relancer en parallèle un deuxième runMap sur un autre daemon, etc...

2.3.3 construction du callback

Nous avons d'implémenter le callback comme suit :

- Le job crée le callback et lui passe un objet témoin et le nombre de runMap lancés . Le job lance les runmaps en leur passant le callback. Quand il a lancé tout les runMap, il attend sur le témoin.
- Quand un daemon à fini son runMap, il appelle la méthode MapFinished sur le callback
- Quand il est créé, le callback initialise un entier nbServeurs avec le nombre de runMap lancés. A chaque fois que la méthode MapFinished est appelé, cet entier est décrémenté (accès exclusif sur la décrémentation). Quand tout les runMap sont fini (donc quand nbServeurs est nul), le callback notifie le job.

2.3.4 ajout d'un nouveau format de fichier

Pour que toute l'application supporte un nouveau type de fichier en entier, il suffit de rajouter un case dans le switch ligne79 du job.

2.3.5 conventions de nommage

nous avons choisi de nommer nos fichiers et nos fragments de fichiers comme suit :

- en supposant que le fichier initial s'appelle "nom"
- les fragments de ce fichier stockés sur le daemon i sont stockés dans un fichier nom+i
- les résultats du map sur le daemon i sont stockés dans un fichier nom-resTemp+i
- le résultat du reduce est le fichier nom-res

2.4 Tests

Nous avons testé notre programme grace aux classes CallbackImpl_test, Job_test, MapReduceImpl_test et DaemonImpl_test.

Idans la classe MapReduceImpl_test, la méthode map fait juste afficher la console "je suis dans le map", attendre quelques secondes, afficher "j'ai fini le map". la méthode reduce fait juste afficher la console "je suis dans le reduce", attendre quelques secondes, afficher "j'ai fini le reduce".

Les autres classes font la même chose que les classes de même nom sans le suffixe "_test", à la différence près qu'elles indiquent dans la console chaque étape clé qu'elles passent. Typiquement, lors d'une exécution où tout se passe bien, la console d'un daemon va afficher :

```
daemon1
daemon2
daemon3
daemon4
daemon5
daemon6
je suis dans le map
j'ai fini le map
daemon7
daemon8
daemon9
daemon9
```

Ce test permet 2 choses : d'une part, il permet de visualiser immédiatement qu'est ce qui s'exécute quand, de voir que la parallélisation fonctionne par exemple. D'autre part, pendant toute la phase de débuggage, il permet de situer exactement sur quelle ligne s'est produit l'erreur.

2.5 Synthèse

La partie hidoop fonctionne avec au moins un ou deux daemons (nous n'avons pas testé avec plus) sur la machine local ou sur des machine distantes (au sein de réseau de l'école). Les maps se font bien en parallèles, le job attend bien un signal du callback pour se réveiller.

3 Conclusion

Après quelques ajustement, la fusion des 2 parties fonctionne. Nous arrivons, à partir d'un fichier texte et avec le MyMapReduce fourni, à obtenir un fichier dans lequel est compté l'occurrence de chaque mot du texte.