Rapport Final
Projet Données Réparties
HDFS/HIDOOP

BOUDILI Younes
MORATA Jules
ROUX Thibault
SADURNI Thomas

Département Sciences du Numérique - Filière Image et Multimédia
2020-2021

Table des matières

Table des figures

1 Introduction

1.1 Présentation

Le but de ce projet est de mettre en oeuvre l'architecture *Map/Reduce*. Ce schéma consiste à effectuer en parallèle (sur un ensemble de machines) des traitements sur un grand volume de données. Les données sont découpées en fragments qui sont répartis sur les différentes machines de traitement. Ces fragments sont traités en parallèle sur les différentes machines où ils sont stockés, et les résultats partiels issus des traitements des fragments sont alors fusionnés pour donner le résultat final.

1.2 Architecture

Le schéma ci-dessous représente l'architecture implémentée pour un *cluster* de 2 machines (nodes) :

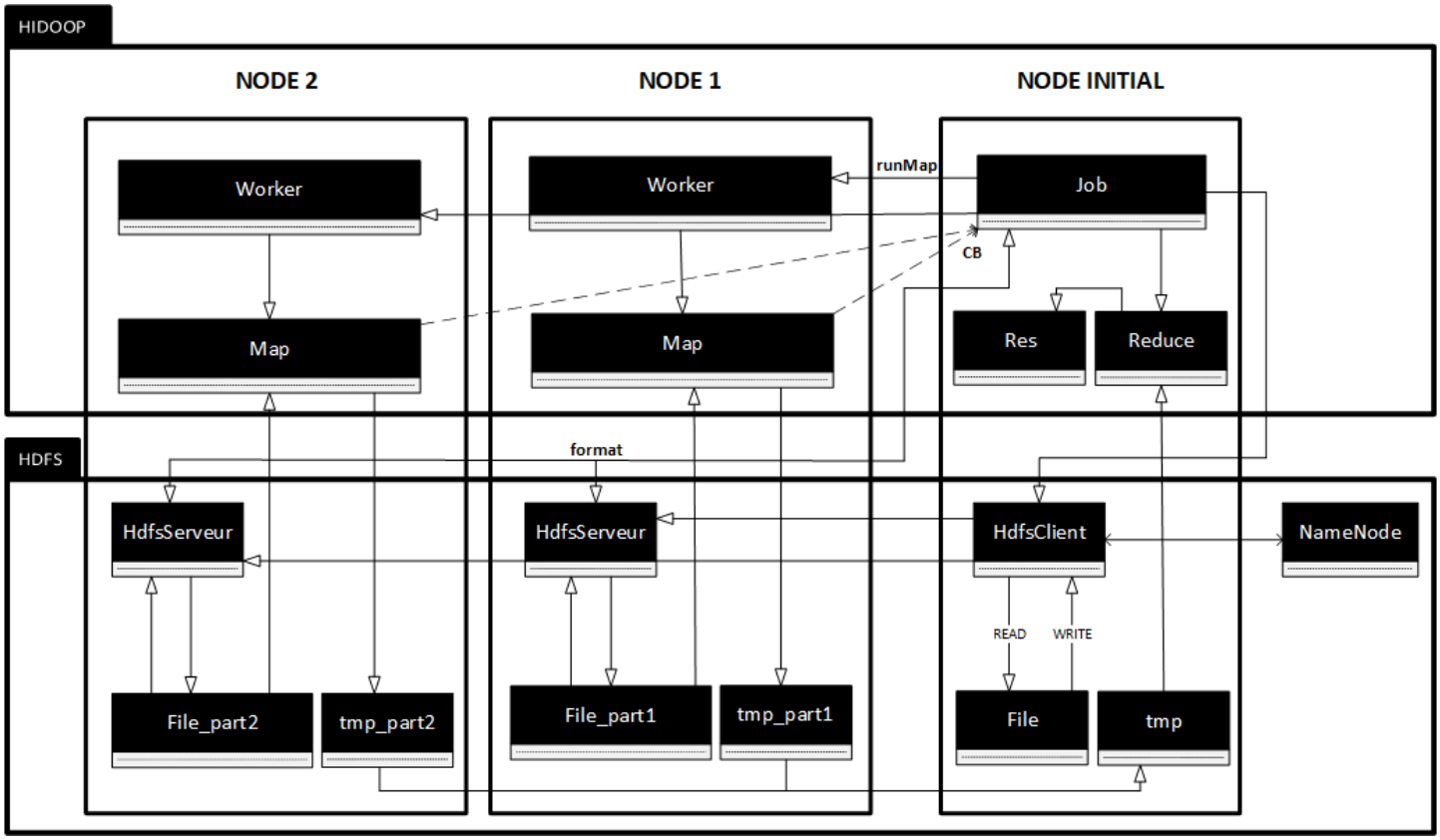

FIGURE 1 – Architecture générale du projet

1.3 Organisation

Pour s'organiser, nous avons utilisé principalement *Discord*, où nous avons créé un groupe de discussion et d'échange d'idées. Ensuite, nous avons aussi créé un tableau *Trello* pour organiser les grandes tâches à faire, et suivre l'avancement du travail.
Le lien de notre Trello est le suivant : : https ://trello.com/b/qcKtPvJU/projet-dr

2 Hidoop/ordo

HIDOOP contrôle l'exécution répartie et parallèle des traitements Map, la récupération des résultats et l'exécution du reduce. Un Worker doit être lancé sur chaque machine. Nous utiliserons RMI pour la communication entre ce *démon* et ses clients.

2.1 Côté client : Job

La classe Job implémente l'interface JobInterface, elle comprend la fonction $startJob$ qui s'occupe de recupérer la liste des $Workers$ (cf section appropriée) de lancer les $runMap$ en leur donnant le nom du fichier à lire et sur lequel écrire, puis elle utilise la fonction reduce pour concaténer les résultats des différents fichiers traités par $HDFS$.
Pour l'instant nous initialisons le nombre de machines à trois, mais il sera préférable par la suite d'en utiliser plusieurs.
Chaque worker (cf section suivante) lit le fragment du fichier puis écrit les résultats du map en local. Pour avoir une exécution en simultanée, nous avons créé la classe MapThread qui implémente la classe Thread et qui lance les $runMap$. Une fois l'exécution terminée, les workers le signalent au client via le CallBack.
Enfin, chaque fragment de fichier analysés se trouvent sous le nom "res_i_$nomdufichier$". Avec la version ne disposant pas le $HDFS$, il faut recomposer soit même le fichier dans "recompose_$nomdufichier$". Plus tard, la jonction entre Hdfs et Hidoop le fera automatiquement.

2.2 Côté serveur : Workers

La classe WorkerImpl implémente l'interface Worker. C'est du coté serveur, donc dans la classe WorkerImpl que nous retrouvons notre procédure $runMap$. Celle-ci est lancé par le client (Job), elle ouvre le fichier des $readers$ et $writers$ passés en paramètre de la fonction, exécute la fonction map en enregistrant en local sur le $writer$, pour enfin fermer les $writers$ et $readers$. Pour confirmer la fin de l'exécution, on utilise la fonction callBack (voir ci-après).
La fonction $main$ créé un $registry$ dans le $localhost$. L'utilisateur passe en argument l'identifiant du Worker. Ainsi les Workers se connecterons à l'adresse : "//localhost/Worker$nomduworker$" avec $nomduworker$ le nom du $worker$ donné par l'utilisateur.

2.3 CallBackImpl

Le CallBack nous permet d'informer Job que les $maps/Workers$ sont finis. On utilise un Semaphore $nbMapFini$ pour avoir accès à la ressource et pour éviter l'attente active.
Nous avons $finishedMaps$ qui confirme la fin des Maps et $waitMaps$ qui attend que les maps se terminent.

2.4 Tests

En local, la séparation du fichier donnée dans le dossier *data* se fait correctement, les fichiers res et $recompose$ sont bien créés. Il faut le recomposer à la main pour l'instant car nous n'avons pas encore lié les parties $Hidoop$ et $HDFS$.

3 HDFS

3.1 Client

Le côté Client offre 3 possibilités à l'utilisateur : lire un fichier, écrire un fichier et supprimer un fichier.
La lecture consiste en la création d'un fichier sur la machine du client dans lequel on va concaténer les différents fragments présents sur les différents *nodes*.
L'écriture correspond à la division d'un fichier fournit par l'utilisateur sur les différents *nodes*. Pour mener à bien ces opérations on utilise un fichier *metadata.txt* qui recense tous les noms des fichiers créés ainsi que leur format (pour l'instant KV ou $LINE$).
La suppression correspond justement au fait de retirer le fichier fournit par l'utilisateur du fichier *metadata.txt*.
Pour faire tout cela on a ajouté une méthode *is_meta* qui permet de savoir si un fichier est référencé *metadata.txt*.

3.2 Serveur

Chaque serveur reste à l'écoute permanente d'une nouvelle demande de connexion, que de la part du client à ce stade là. Ensuite, il crée le *socket* correspondant et il attend à nouveau la commande du client. Selon la demande, il exécute les traitements nécessaires et communique avec le client.
Nous avons adopté une convention de nommage pour les fragments manipulés : *nomDuFichierSurHdfs_fmg.txt*. Et dans notre cas d'exécution sur une machine locale pour les tests, chaque serveur dispose d'un dossier *node*.

3.3 Améliorations envisagées

Pour l'écriture des fragments, nous choisissons pour le moment d'écrire le maximum de lignes sur les $n-1$ premiers noeuds puis le reste sur le dernier (dans le cas où le nombre de lignes à écrire n'est pas un multiple de du nombre de noeuds). Cette approche est légèrement sous-optimale dans le cas de données très volumineuses et où il n'y aurait que 1 ou 2 lignes "de surplus". Il faudrait donc changer l'implémentation pour que le nombre nécessaire de noeuds ait une ligne de moins et ainsi limité l'écart de travail à réaliser entre les noeuds.

4 Combinaison HDFS/Hidoop

Après avoir travaillé séparément sur les deux parties $HIDOOP$ et $HDFS$, nous devons les lier dans cette deuxième partie du projet pour avoir une application fonctionnelle. Il faut alors assurer la communication entre la classe Job et $Hdfs$, à travers les *sockets*.

4.1 NameNode

Nous avons remplacé le fichier *metadata.txt* par un $NameNode$. Celui ci attend les commandes de $HDFS$ qui peuvent être ADD, GET, $DELETE$, $CONTAINS$, respectivement pour ajouter un fichier dans la *table* (on utilise une $HashTable$ pour stocker), demander le format d'un fichier, supprimer un fichier, et demander l'existence d'un fichier.

4.2 HDFS

Du côté client, nous avons ajouté la communication entre $HdfsClient$ et le $NameNode$ pour les différentes interrogations de la base de données.

Du côté serveur, nous avons ajouté le traitement de la nouvelle commande CMD_MAP. Elle est envoyé par le Job aux serveurs, pour récupérer les formats *Reader* (pour lire le fragment) et $Writer$ (pour écrire le résultat local du traitement), et les passer en paramètre aux $Workers$.

A ce stade là, les serveurs pouvaient traiter qu'une seule demande client en même temps. Du coup, nous avons amélioré le code en passant à des serveurs $multi-threads$.

4.3 Job

Nous avons dû faire des modifications dans la classe Job. En effet, pour pouvoir lier les deux parties, nous avons rajouté des *sockets* dans cette classe au niveau des maps. Chaque map se fera sur un serveur séparé en accédant au fichier stocké sur le serveur.

De plus, les *nodes* ne sont plus "*localhost*" mais le nom des machines de l'enseeiht. Dans notre cas, on utilise 3 machines : *rattata*, *roucool* et *rondoudou*. Pour en ajouter, il faut compléter l'attribut *nodes*, *ports* et $portsWorkers$ dans $Job.java$ et $HdfsClient$.

Pour chaque Workers, on regarde l'adresse de chaque machines lancée avec le nom du *worker* correspondant.

Pour recomposer les fichiers "res_", nous faisons appel à la procédure $HdfsRead$ sur les fichiers temporaires *tmp* placés dans les dossiers *node* et créés au préalable par $HdfsClient()$. Ces fichiers sont temporaires et non visible par l'utilisateur car ils sont supprimés grâce à la procédure $HdfsDelete$. Enfin, le fichier recomposé se trouve dans le dossier *job* créé sous le nom "recompose_$nomdufichier$" et le résultat du $MyMapReduce$ sous le nom "res_$nomdufichier$".

4.4 WorkerImpl

Dans cette partie du projet, nous ne sommes plus en *localhost*, c'est pour celà qu'on créé un registre pour chaque $Worker$ avec la procédure *createRegistry* sur le port entré en argument.

5 Lancement de l'application

Lancer l'application revient à lancer le script *lancement.sh* qui se trouve à la racine du projet. Ce script se connecte en ssh à différentes machines de l'ENSEEIHT et y lance le NameNode, les Serveurs et les Workers. Par contre, il ne prend pas en compte le mot de passe de l'utilisateur, nous avons décider de le laisser dans les livrables.
Etapes à suivre pour 3 Workers et 3 Serveurs :

1. Se connecter en SSH sur 3 machines de l'école (*rattata*, *roucool*, *rondoudou*)
2. Lancer un NameNode sur le port 8080 sur l'host.
3. Sur chaque machine lancer un Serveur et un Worker de la façon suivante (i pour 1,2,3 respectivement sur *rattata*, *roucool* et *rondoudou*)
   (a) java ordo.WorkerImpl machinesi 808i
   (b) java hdfs.HdfsServeur i 8080
4. Lancer les commandes suivantes sur l'host
   (a) java hdfs.HdfsClient write line test ../data/filesample.txt
   (b) java hdfs.HdfsClient read test ./test
5. Lancer ensuite l'application avec la commande : java application.MyMapReduce test

Ainsi, un dossier job est créé avec le fichier recomposé *recompose_* et le fichier résultat *res_* à l'intérieur.

Remarque

Il y a une limitation à cause de la méthode $File.createFile()$ utilisée dans Job et HdfsClient.read(). Pour refaire une manipulation sur un même fichier stocké sur HDFS, il faut supprimer le dossier *job* créé par l'ancien traitement, après avoir déplacé le résultat afin de ne pas le perdre.

Tests

Les tests en répartie marchent bien en général. Pour des fichiers volumineux, les Write/Read du côté HDFS prennent du temps, mais les traitements Map/Reduce ne sont pas beaucoup affectés vu la parallélisation des calculs.

Améliorations

Pour plus de faciliter dans la manipulation, nous aurions pu créer des fichiers de configuration qui renseignent les ports et les machines utilisées. Cela aurait été plus simple et plus flexible que de modifier le code de $Job.java$, et $HdfsClient$...

6 Conclusion

Pour conclure, ce fut un projet intéressant qui nous a permis de créer une application pour la gestion de fichier volumineux. Le plus de ce projet a été de le faire à partir de machines différentes. Ainsi nous avons pu, à notre échelle, faire de la gestion de fichiers $BigData$.
Bien sûr, ce fut assez difficile d'implanter cette application, mais surtout de mettre les deux parties en commun, car nous nous sommes rendu compte que chaque binôme avait une vision différente de la fin du projet.
Finalement, notre application fonctionne correctement à distance sur les machines de l'ENSEEIHT mais pour qu'elle soit optimal il faudrait implanter les améliorations que nous avons proposées ci-dessus.

7 Références

[1] : https ://www.lebigdata.fr/hadoop

[2] : http ://sd-127206.dedibox.fr/hagimont/resources-N7/hidoop/hagidoop.pdf