

Rapport Projet Données Réparties



Younes SAOUDI
Reda EL JAI
Faical TOUBALI HADAoui
Mehdi SENSALI

January 2021

1 Introduction

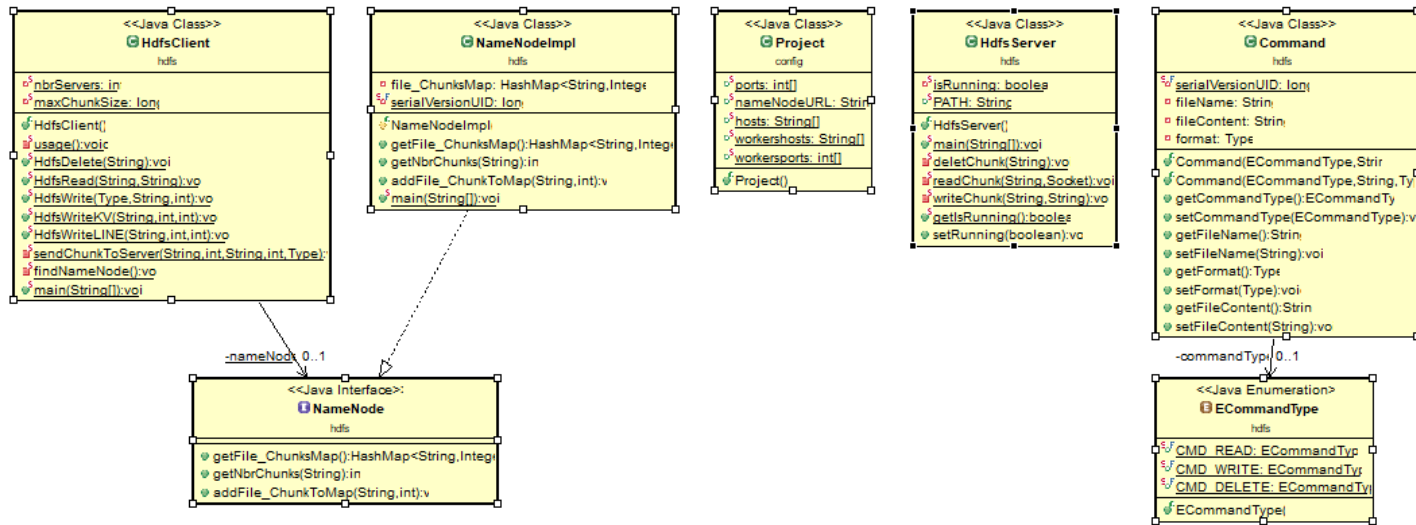
Le projet Hidoop constitue une initiation aux applications concurrents appliquées au calcul intensif et traitement des données massives. Il implante la stratégie MapReduce qui consiste à découper les données en fragments répartis sur des machines de traitement et les traiter en parallèle (map), pour pouvoir ensuite agréger ces calculs intermédiaires et donner le résultat final(reduce). Cette implantation nécessite donc la réalisation des deux services

- **HDFS** : le système de gestion de répartition des fichiers sur les différentes machines
- **Hidoop** : service garantissant l'exécution parallèle des map sur les différents cluster et l'aggrégation des résultats à l'aide de reduce.

Dans ce qui suit, nous allons proposer une architecture possible de Hidoop et tester son implémentation sur différentes machines.

2 HDFS

2.1 Architecture



2.2 Choix de conception

- la classe **HdfsClient** : Communique aux serveurs, par le biais des sockets HDfs, les commandes à exécuter, à savoir write, read et delete. Lorsque nous lançons le client, la première chose à faire est d'établir une connexion avec le serveur RMI, le **NameNodeImpl**, afin de récupérer les fichiers existants ou les modifier suivant l'appel des méthodes suivantes :
 - : **HdfsWrite** : Divise le fichier en un certain nombre de chunks de même taille, selon la taille du fichier en entrée, qu'on affecte aux différents serveurs selon le résultat de id des chunks modulo nombre de serveurs. On envoie au serveur des kv ou des lignes selon le format du fichier, puis ajoute le fichier et son nombre de chunks au hashmap du namenode.

- : **HdfsRead** : On récupère d’abord le nombre de chunks à partir du namenode puis on envoie les noms des chunks aux serveurs, en utilisant la même fonction d’attribution, et on se met en attente qu’ils nous renvoient les fragments qu’on regroupe dans le fichier portant le même nom que le fichier suivi de -res. -
- : **HdfsDelete** Même logique que HdfsRead sauf qu’il suffit de recomposer les nom des chunks se trouvant dans chaque serveur, pour leur envoyer le commande de suppression. On supprime ensuite le fichier du namenode
- la classe **HdfsServer** : Reçoit les commandes du client et agit en conséquence, soit en supprimant les fichiers qu’il lui est demandé de supprimer, soit de créer les fichiers contenant le chunk reçu dans le cas de write, soit en renvoyant le contenu d’un chunk au client pour la méthode read. A noter que l’ensemble des méthodes s’appliquent sur les fichiers se trouvant dans le dossier **tmp**
- la classe **NameNode** : hérite de la classe Remote et constitue l’interface du serveur rmi qui nous permet d’ajouter, modifier, supprimer les fichiers de HDFS, notamment en récupérant le nombre de chunks auquel il a été fragmenté.
- la classe **NameNodeImpl** : L’implémentation du namenode
- la classe **Command** : Permet d’encapsuler la création des commandes à envoyer au serveur.
- la classe **ECommandType** : Enumère les différents commandes communiqués entre le client et le serveur.
- la classe **Project** : C’est une classe du fichier **config** et contient la configuration des serveurs Hdfs (ports et hosts), du namenode(URL) ainsi que des workers de hidoop.
-

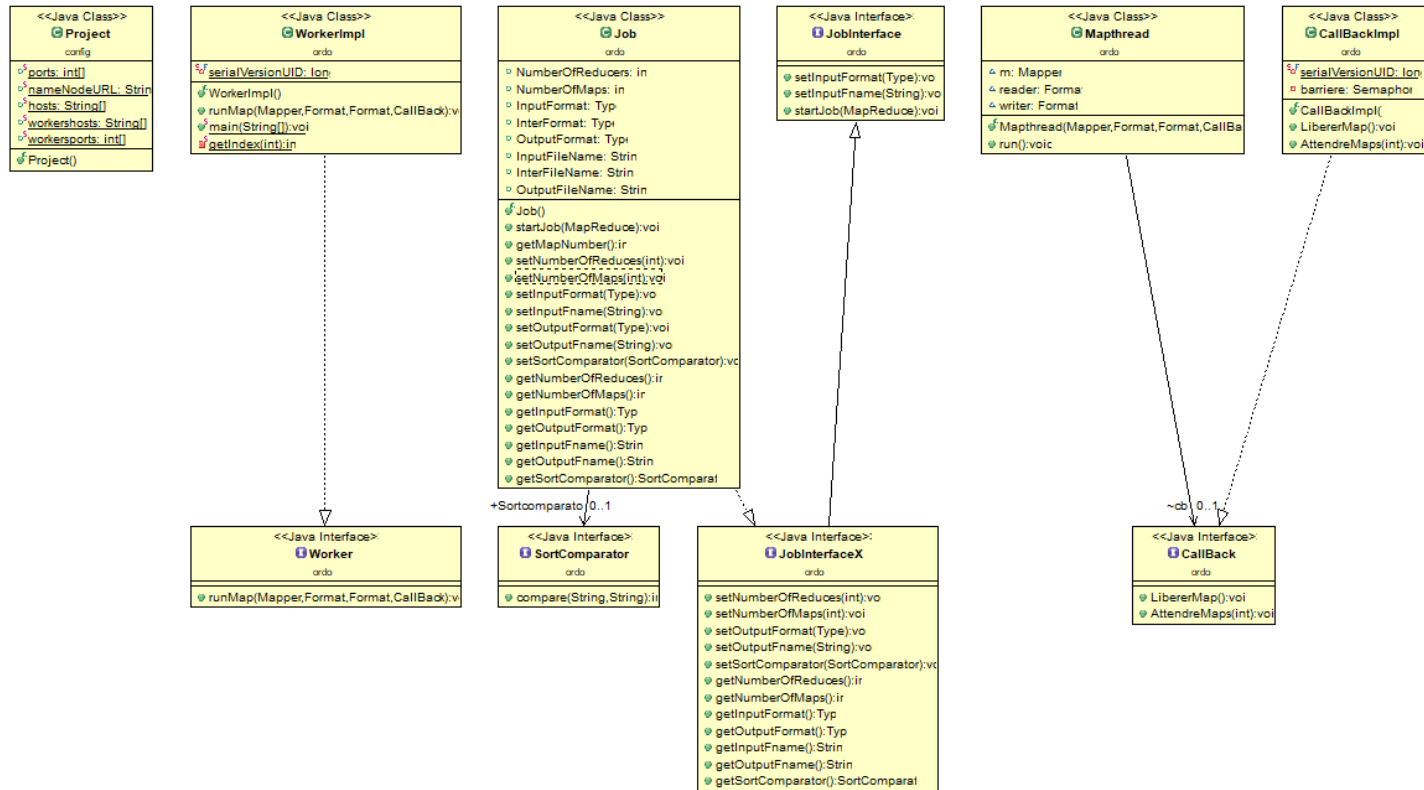
2.3 Exécution

- Lancer la namenode `java ordo.NameNodeImpl`.
- Lancer les Hdfs serveurs avec les ports 8001,8002,8003 avec la commande `java hdfs.HdfsServer <port Number>`.
- Lancer le client avec la commande voulue par exemple : `java hdfs.HdfsClient write line test.txt`.

3 Hidoop

3.1 Architecture

Le diagramme UML décrivant l’architecture choisie de la partie hidoop est le suivant :



3.2 Choix de conception

Les principaux éléments de conception :

- Classe **WorkerImpl** : Implémente le démon qui s'exécute sur chaque noeud afin de réaliser un map sur tous les fragments envoyé à ce dernier.
- Classe **Maphread** : s'occupe de gérer l'exécution en parallèle des runmap bar le biais des **sémaphore**. Ceux-ci bloquent l'exécution de **StartJob** tant que les démons n'ont pas tous terminé leurs calculs sur les fragments qui leur sont alloués.
- Classe **CallBackImpl** : Implémente les sémaphores ainsi que les procédures **Acquire** et **release** afin de gérer le parallélisme de traitement.
- Classe **Job** : La classe qui permet de gérer toute l'opération du **MapReduce** ainsi que la création des fichiers tampon sur lesquels seront stockés les résultats de l'exécution de chaque map sur les fragments pour ensuite être rassemblés dans un fichier temporaire par l'opération **HdfsClient.HdfsRead()**, à partir de ce dernier sera généré le fichier final sur lequel l'opération du **Reduce** aura effectué son traitement.

3.3 Exécution

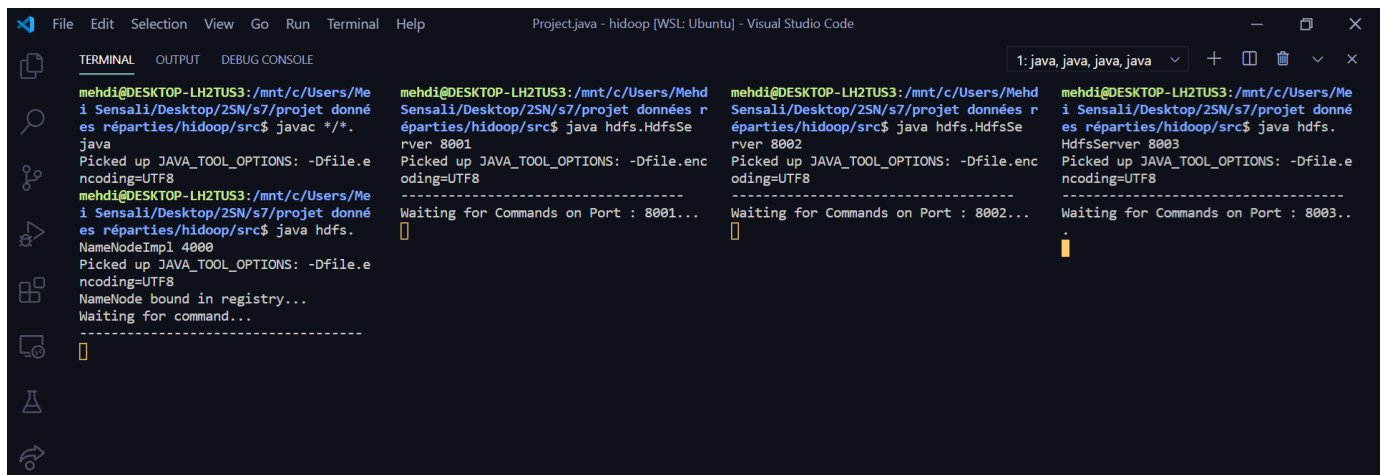
Il suffit de :

- Lancer chaque démon dans un terminal en se plaçant dans le dossier `src` avec la commande `java ordo.WorkerImpl <port number>`.
- Lancer une classe de test qui implémente les méthodes `Map` et `Reduce` et qui exécute `StartJob` dans sa classe `Main` comme par exemple pour le word-count : `java application.MyMapReduce`.

4 test de l'application

Afin de tester l'application du calcul de comptage des mots sur un fichier, on se place tout d'abord dans la dossier source `src` contenant les différents paquetages utilisés, on commence par le déploiement de l'architecture HDFS ensuite on fait appel aux `Workers` qui appliqueront les procédures Map, Reduce sur un fichier exemple appelé `filesample.txt`.

- Lancement du `NameNode` sur le port 4000
- Lancement de trois `HdfsServer` sur chacun des ports 8001, 8002, 8003. Ainsi, les `HdfsServer` sont en attente des requetes du client.



```
File Edit Selection View Go Run Terminal Help Project.java - hidoop [WSL: Ubuntu] - Visual Studio Code

TERMINAL OUTPUT DEBUG CONSOLE

mehdi@DESKTOP-LH2TUS3: /mnt/c/Users/Mehdi Sensali/Desktop/2SN/s7/projet données réparties/hidoop/src$ javac */*.java
Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF8

mehdi@DESKTOP-LH2TUS3: /mnt/c/Users/Mehdi Sensali/Desktop/2SN/s7/projet données réparties/hidoop/src$ java hdfs.NameNodeImpl 4000
Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF8
NameNode bound in registry...
Waiting for command...

mehdi@DESKTOP-LH2TUS3: /mnt/c/Users/Mehdi Sensali/Desktop/2SN/s7/projet données réparties/hidoop/src$ java hdfs.HdfsServer 8001
Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF8
Waiting for Commands on Port : 8001...

mehdi@DESKTOP-LH2TUS3: /mnt/c/Users/Mehdi Sensali/Desktop/2SN/s7/projet données réparties/hidoop/src$ java hdfs.HdfsServer 8002
Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF8
Waiting for Commands on Port : 8002...

mehdi@DESKTOP-LH2TUS3: /mnt/c/Users/Mehdi Sensali/Desktop/2SN/s7/projet données réparties/hidoop/src$ java hdfs.HdfsServer 8003
Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF8
Waiting for Commands on Port : 8003..
```

Figure 1: Lancement du NameNode et des HdfsServer

- Lancement des **Workers** sur chacun des ports 8887,8888,8889.

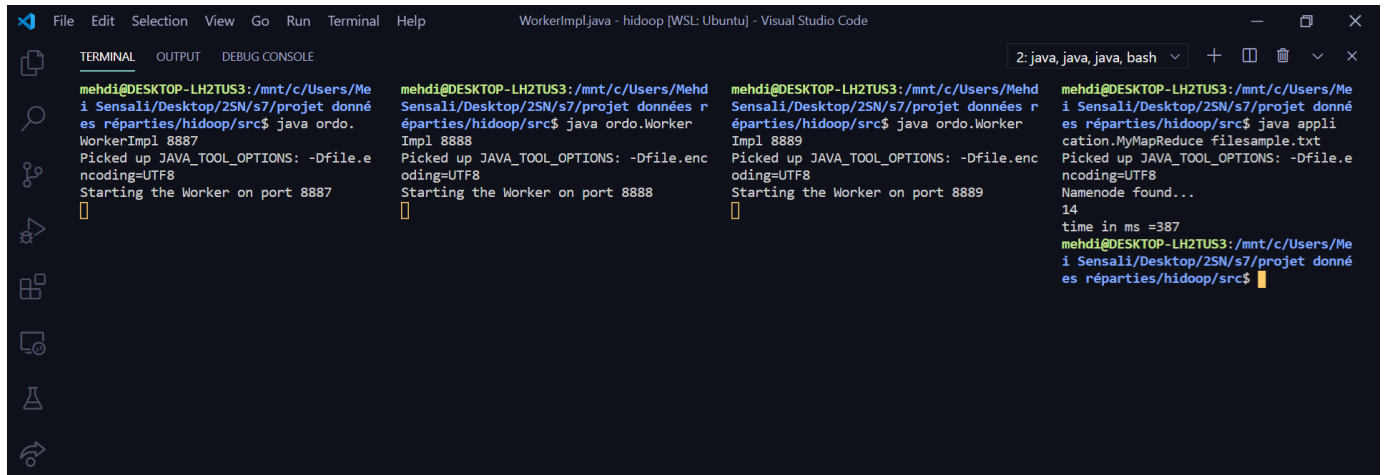


Figure 2: Lancement des Workers et execution de MyMapReduce sur le fichier filesample.txt

- Lancement de HdfsWrite sur le fichier `filesample.txt` qui est de format `Line` qui le divisera en 14 chunks de format `Line` (ce nombre est en fonction de la taille du fichier) et ajoutera ensuite ce fichier et ses chunks au hashmap du namenode.

```
mehdi@DESKTOP-LH2TUS3:/mnt/c/Users/Mehdi Sensali/Desktop/2SN/s7/projet données réparties/hadoop/src$ java hdfs.
HdfsClient write line filesample.txt
Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF8
Namenode found...
```

(a) Lancement de HdfsWrite sur le fichier filesample.txt

```
filesample.txt-chunk0
filesample.txt-chunk1
filesample.txt-chunk2
filesample.txt-chunk3
filesample.txt-chunk4
filesample.txt-chunk5
filesample.txt-chunk6
filesample.txt-chunk7
filesample.txt-chunk8
filesample.txt-chunk9
filesample.txt-chunk10
filesample.txt-chunk11
filesample.txt-chunk12
filesample.txt-chunk13
```

(b) Les chunks du fichier filesample.txt apres le HdfsWrite

Figure 3: Production des chunks du fichier filesample.txt par HdfsWrite

- On affecte à chaque HdfsServer la tâche de décomposer le fichier en chunk i en format Line modélisé par le reste de la division du nombre total des chunks qui est 14 par le nombre des HdfsServer qui est de 3.

Par exemple, Le HdfsServer n°1 reçoit les chunks : 0,3,6,9,12.

- à la fin, si le découpage du fichier est réussi par les HdfsServer, le fichier filesample.txt et ses chunks seront ajoutés au HashMap du NameNode.

Figure 4: Réussite du découpage du fichier filesample.txt en chunks de format Line

- Execution de MyMapReduce sur le fichier filesample.txt qui lancera Job qui fait appel à son tour à la procédure qui calculera le nombre d'occurrences d'un mot sur chaque fragment, puis, il fait appel à HdfsRead qui groupera le résultat de l'application du map sur les chunks en un seul fichier auquel la procédure Reduce sera appliqué en fin de compte.
- Le fichier résultant par application du map sur le chunk filesample.txt-chunk i est : filesample.txt-res i .
- Le fichier résultant après l'appel du HdfsRead est filesample.txt-res
- Le fichier résultant après l'appel à reduce sur filesample.txt-res est : filesample.txt-final

```

mehdi@DESKTOP-LH2TUS3: /mnt/c/Users/Mehdi Sensali/Desktop/2SN/s7/projet données réparties/hadoop/src$ java application.MyMapReduce filesample.txt
Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF8
Namenode found...
14
time in ms =392
mehdi@DESKTOP-LH2TUS3: /mnt/c/Users/Mehdi Sensali/Desktop/2SN/s7/projet données réparties/hadoop/src$

```

(a) Lancement de MyMapReduce sur le fichier filesample.txt

```

filesample.txt-final
filesample.txt-res
filesample.txt-res0
filesample.txt-res1
filesample.txt-res2
filesample.txt-res3
filesample.txt-res4
filesample.txt-res5
filesample.txt-res6
filesample.txt-res7
filesample.txt-res8
filesample.txt-res9
filesample.txt-res10
filesample.txt-res11
filesample.txt-res12
filesample.txt-res13

```

(b) Les résultats de l'application de map sur les chunk et le résultat final après la procédure Reduce

Figure 5: Lancement du map sur les chunks et lancement de Reduce sur les résultats du map.

5 Conclusion :

A partir de nos observations, il s'avère que le principe diviser pour mieux régner permet une manipulation plus efficace des données massives : En effet la version concurrente offre des résultats plus satisfaisant que la version itérative implémentant le même service. Ce projet nous a donc permis de comprendre toute l'importance du calcul réparti, en nous appuyant notamment sur les connaissances acquises en itergiciels et systèmes concurrents.