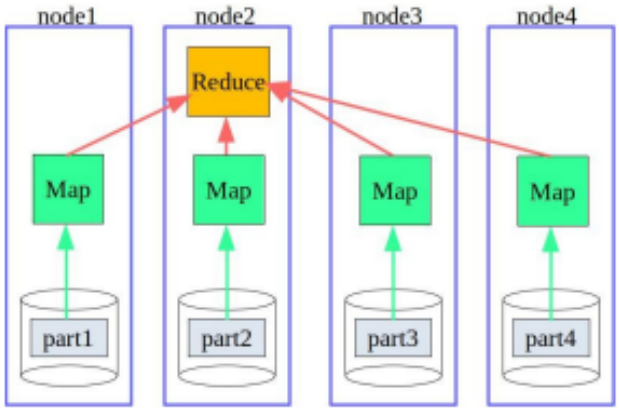
Projet Données Réparties - HAGIDOOP

Chekraoui Louiza et Safae Belahrach



Département Sciences du Numérique - Deuxième année
Systèmes Logiciels
2023-2024

1 Introduction

Le projet de cette UE consistait à réaliser une implémentation du modèle **Map-Reduce** en Java. Nous allons dans un premier temps commencé par expliquer les choix de conceptions que nous avons fait pour mener à bout ce projet et dans un second axe quels ont été les difficultés et limitations rencontrées durant ces semaines de travail, et les possibles résolutions et ce qu'on aurait pu faire. Enfin, nous conclurons sur la comparaison des temps d'exécution (évaluation des perfermances).

2 Choix de conception

Le service Hagidoop est composé de deux services : un service **HDFS** et un service **Hagidoop**. Le premier service est un système de gestion de fichier dans lequel nous allons connecter un HdfsClient qui fournit des commandes.

2.1 Implémentation du service HDFS

On a créé une connexion entre **HdfsClient** et **HdfsServer** via le mode TCP connexion assurée et fiable en utilisant la notion de socket vue en cours. Pour cela, nous avons créé un serverSocket pour chaque server qu'on va se mettre en mode accept() pour se mettre sur écoute des commandes envoyées par le client. Du côté client, on a créé des sockets pour se connecter à chaque server pour leur envoyer leurs requêtes sous forme de kv ("commande", filename). Et ensuite, on traite la requête selon la commande du côté server soit write, read ou delete.

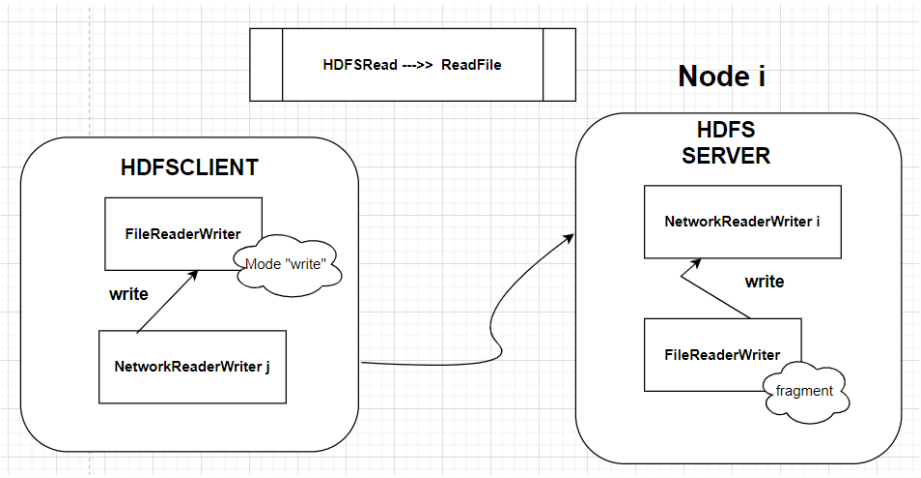


FIGURE 1 – HdfsRead

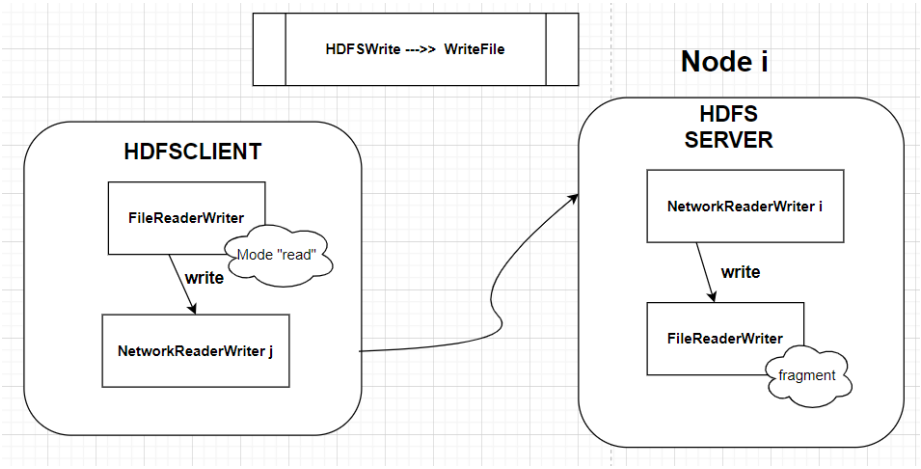


FIGURE 2 – HdfsWrite

2.2 Implémentation du service Hagidoop

La partie d'implémentation importante dans ce service consistait dans le multi-threading, que nous avons utilisé pour lancer les Runmap en parallèles sur les workers. Voir la capture ci-joint :

La sortie de chaque Map est écrite dans le **NetworkReaderWriter** du Worker Courant correspondant, ce **NetworkRW** est un socket connecté avec les **NetworkRW** correspondants qui se trouvant dans **JOBLauncher** et qui ont été générer par le point accept() du serverSocket, les données au niveau des NetworkRW sont mises dans un PipeReaderWriter(Queue) qui serait le reader du reduce.

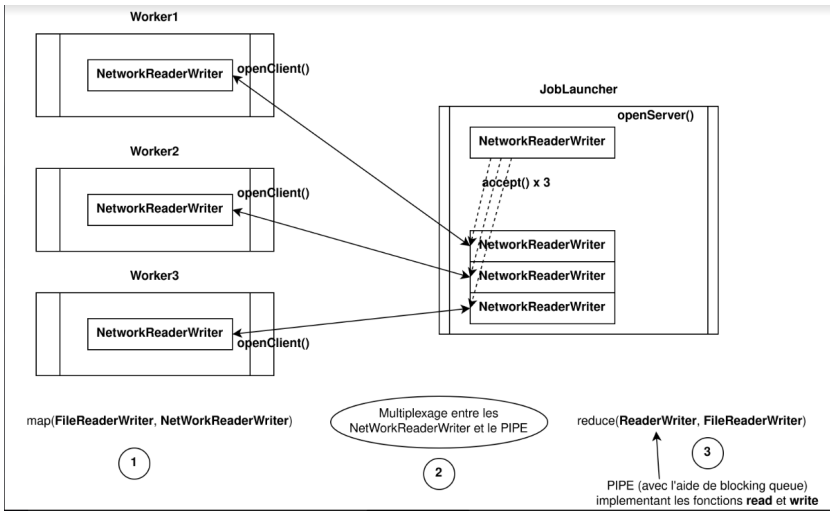


FIGURE 3 – JobLauncher et Workers

3 Difficultés rencontrées et Solutions proposées

Première limitation : Nous n'avons pas pu créer un script de déploiement à partir duquel on peut lancer toutes les commandes sur les différents démons à partir d'un seul script, nous avons plutôt implémenté qui consiste à ouvrir différents terminaux dans lesquels on lance nos commandes.
Nous aurions pu créer un fichier script qui contient les informations des machines sur lesquelles on souhaite lancer les commandes, en créer une boucle sur ces machines. Peut-être un script de type shell.
Deuxième limitation, nous avons passé plus de temps à comprendre et à faire marcher le code sur les fichiers de type .txt et par manque de temps, nous n'avons pas implémenté le fait que le fichier d'entrée en local pourrait être de type KV.
La méthode ne change pas, nous aurions pu faire une conditionnelle sur le format fmt sur fichier FMT_TXT ou FMT_KV.

4 Évaluation des performances réalisées

Suite à la démonstration, nous avons ajouté les tests avec 1 et 2 nodes

Nous avons lancé le MapReduce en local avec un, deux, trois et quatres nœuds pour une taille de fichier **"dataGega.txt"** égale à 1,3 Go qui résulte de la duplication de **"filesample.txt"** grâce au generate.sh via la commande suivante : **" - ./generate.sh filesample.txt numéro "**.
Nous avons obtenu les résultats suivants qui comparent le temps d'exécution

avec la méthode itérative de Count. **Voir courbes ci-dessous.**

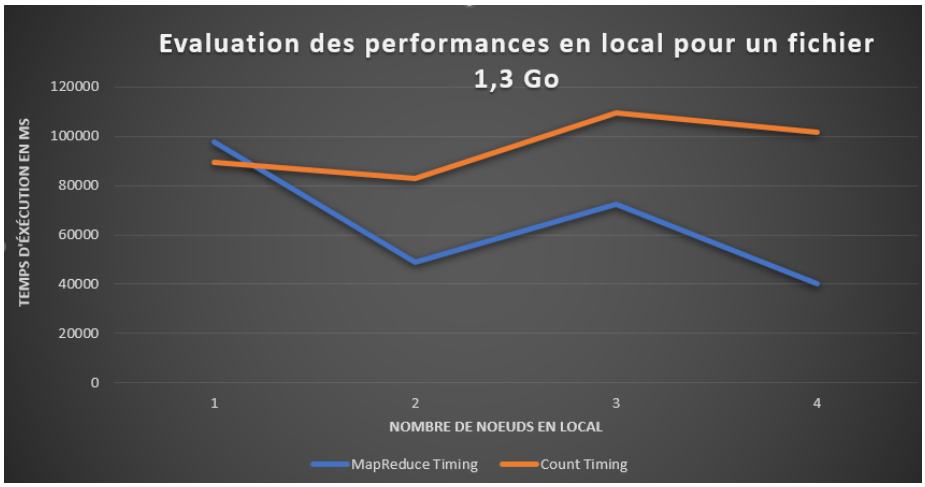


FIGURE 4 – Évaluation des performances réalisées en local

Conclusion : Nous observons des résultats assez cohérents qui nous montrent que pour des fichiers d'assez grande taille, plus on lance de workers (cœurs en local) plus le temps d'exécution est rapide. On voit donc l'avantage du muli-threading. PS : Comme nous pouvons le remarquer pour les fichiers d'assez petite taille, le temps demandé pour l'exécution du MapReduce est largement supérieur à celui de Count, on peut supposer que la fragmentation des fichiers de petites tailles et la répartition sur différents nœuds annule les avantages du parallélisme. Pour les fichiers de grande taille, on remarque les avantages de la méthode MapReduce, car le temps d'éxecution est inférieur à celui de la méthode itérative.Ce qui argumente et prouve que la méthode Map-Reduce fait bien analogie à "Diviser pour mieux régner".