Projet Systèmes concurrents HIDOOP

Rendu HIDOOP v1

Hamza Boukraichi Mohammed Amine Tourari

Table des matières

1 Choix de conception et architecture

Pour cette version nous avons opté pour l'implémentation du lancement de plusieurs map et reduce sur le même nœud ainsi qu'implémenter la fonctionnalité shuffle afin de réduire le temps de calcul et le transit réseau.

Nous avons d'abord commencé par créer 2 nouvelles classes JobHelper et HidoopHelper contenant des méthodes statiques afin d'augmenter la lisibilité du code et de réduire la charge de code sur la classe Job.

Nous avons aussi modifier l'interface Daemon pour y ajouter les methodes runReduce et rubShuffle, afin de pouvoir lancer des threads pour appliquer les Reduce et les Shuffle sur chaque nœud.

Nous avons aussi supprimer les appels à Hdfs afin d'avoir une application indépendante.

Nous avons aussi mis en place un dispositif de détection de pannes de serveurs, en implémentant un émetteur au niveau des démons, et des *listeners* au niveau du registre. Cela donne donc l'architecture suivante :

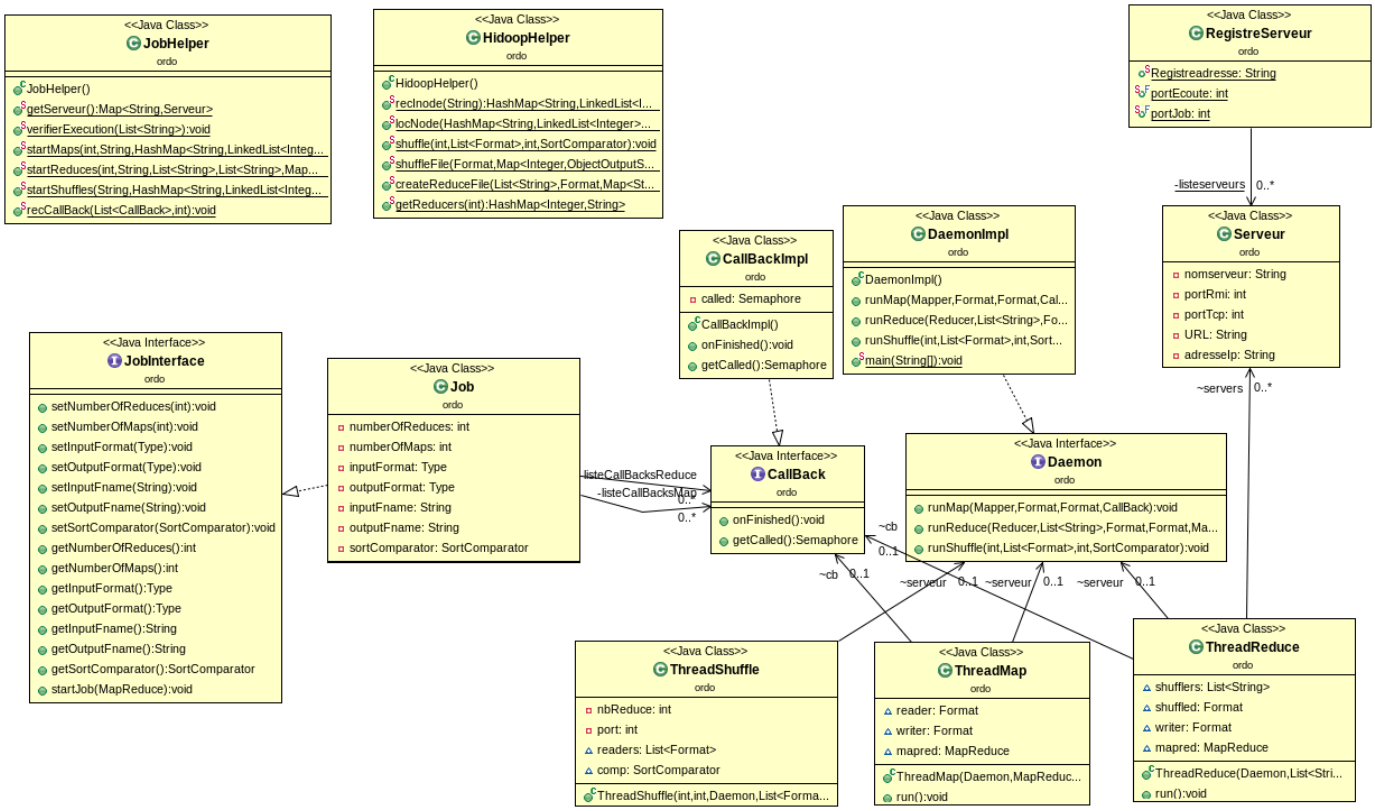

Figure 1 – Diagramme de Classe

2 Les fonctionnalités

2.1 Traitement Map-Reduce & Ajout du Shuffle

Comme indiqué précédemment, nous avons implémenté le lancement de plusieurs map sur le même nœud, pour cela : Job récupère le positionnement des différents blocs sur les différents serveurs par l'appel à la méthode *HidoopHelper.recInode*, puis à partir de son résultat il détermine les nœuds qui vont lancer les différents maps en répartissant les taches de façons équitables en faisant appel à la méthode *HidoopHelper.locNode*. Puis pour chaque nœud, Job crée et lance des threads pour traiter les différents blocs. Après la réception des `callBacks`, chaque noeud qui a lancé au moins un thread Map doit lancer un Shuffle pour gérer ce fichier. Ainsi le Shuffle va traiter tous les résultats des Map lancés et envoyer les données au Reduce correspondant. Dès que les shuffles ont été lancés, les Reduce sont également lancés pour recevoir les données envoyées par les Shuffle, chaque Shuffle indique par un message au Reduce correspondant sa fin, afin de pouvoir appliquer le Reduce au fichier généré.

2.2 Registre De Serveur

Étant donné que des communication entre Serveur Shuffle et Reduce doivent être effectuée en TCP, nous sommes retrouvés contraints de stocker plusieurs données relatives à chaque serveur (nom, adressIp, UrlRmi, différents ports ..). Nous avons alors géré cela en créant des structures de données "Serveur" contenant toutes les données à stocker, et un Registre pour gérer la liste des serveurs stockés.

2.3 Détection et Gestion des pannes

La détection de pannes consiste à crée des émetters au niveau des démons, et des listeners sur les registre.
Quand un démon a lancé, il envoie un signe de vie toute les dix secondes, et une fois que le serveur n'en envoie plus, le registre considère qu'il est en panne et l'affiche à l'utilisateur de l'application. La détection de panne conduit donc à l'interruption de l'application.

3 Implémentation

Afin d'augmenter le degré de parrallélisme de l'application plusieurs thread ont été implémentés. Dans la section suivante, nous allons en présenter quelques uns.

3.1 Fonctionnalités de bases

Avant de lancer les Shuffle et les Reduce, les Map doivent être achevés, aucune amélioration du parrallélisme n'est possible ici, mis à part le lancement parralèle de plusieurs Map sur le même serveur. Après réception des CallBacks des Map, les Shuffle et les

Reduce sont lancés sans plus attendre, chaque Shuffle indique sa fin au Reduce afin de pouvoir entamer la 2ieme étape du Reduce. Ainsi un Shuffle et un Reduce sont lancés de façons parallèle sur le même serveur. Les 2 étapes du Reduce néanmoins sont éxécutées séquentiellement.

3.2 Registre & Détection des pannes Serveurs

Le **RegistreServeur** gère les serveurs en marche et permet de savoir à tout moment les serveurs qui sont actifs et fonctionnels. Une fois qu'un démon est lancé, un objet de type **Serveur** est envoyé au registre, contenant les informations sur les ports et l'URL, et est stocké dans le registre. Le démon lance après un émetteur des *Heartbeat* vers le registre avec une connexion fixée par convention sur le port $Port.Tcp + 2000$

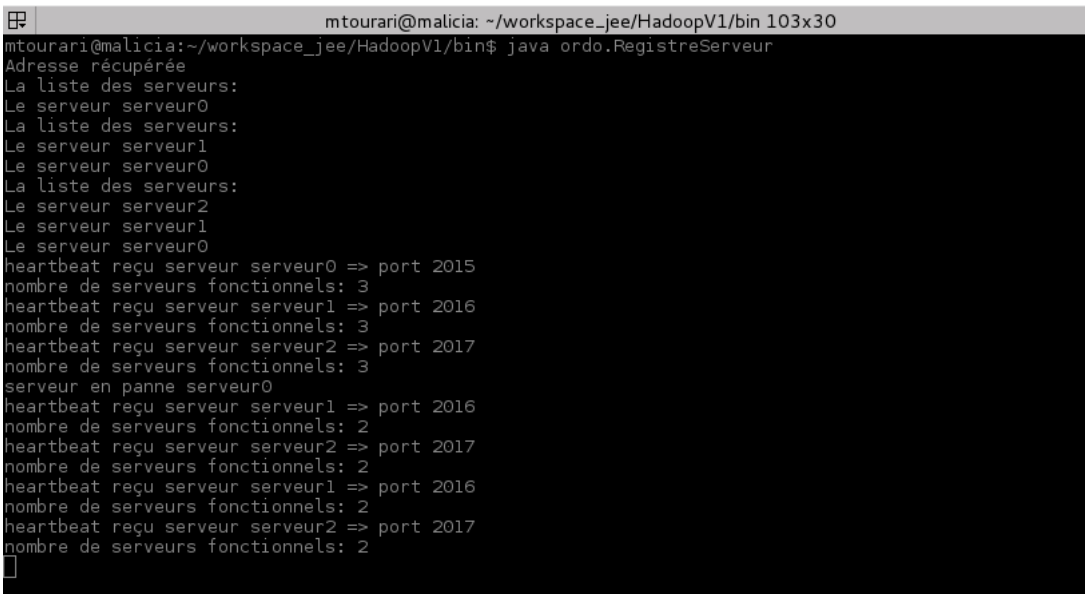

Figure 2 – Registre Serveur

Puisque le temps de vie d'un démon est celui de l'exécution d'une application (map, shuffle & reduce), nous nous contentons d'alerter l'utilisateur qu'un serveur est en panne et donc que son résultat peut être en défaut.

4 Difficultés rencontrées

— La première difficulté rencontrée se situe au niveau du lancement des shuffles et des reduces, étant donné que chaque shuffle devait communiquer avec tous les reduces et par conséquent chaque reduce recevait des données de tous les shuffles. La gestion de cette communication a été très délicate.
— Ensuite, au niveau de la gestion de la liste de serveurs disponible, communiquer cette liste à la classe Job et veiller à ce qu'elle soit à jour ont soulevé beaucoup de problèmes, qu'on a résolu en établissant une communication par socket TCP entre la classe Job et le Registre.

— Au niveau de la détection et la gestion des pannes, nous avons réussi à implémenter la détection, la gestion par contre ne nous sommes limité à l'abandon des traitements faute de temps. La difficulté majeure ici est bien la gestion de notre temps qui s'est avérée bien difficile.

5 Test des fonctionnalités

5.1 définition de l'environnement de test

Le fichier `filesample.txt` a été divisé en 4 blocs (manuellement) et réparti avec duplication sur 3 dossiers relatifs à chaque serveur (pour les tests le nombre de serveurs a été fixé à 3).

— Le serveur 0 contient les blocs 1,2 et 3.
— Le serveur 1 contient les blocs 3 et 4.
— Le serveur 2 contient les blocs 2, 3 et 4.

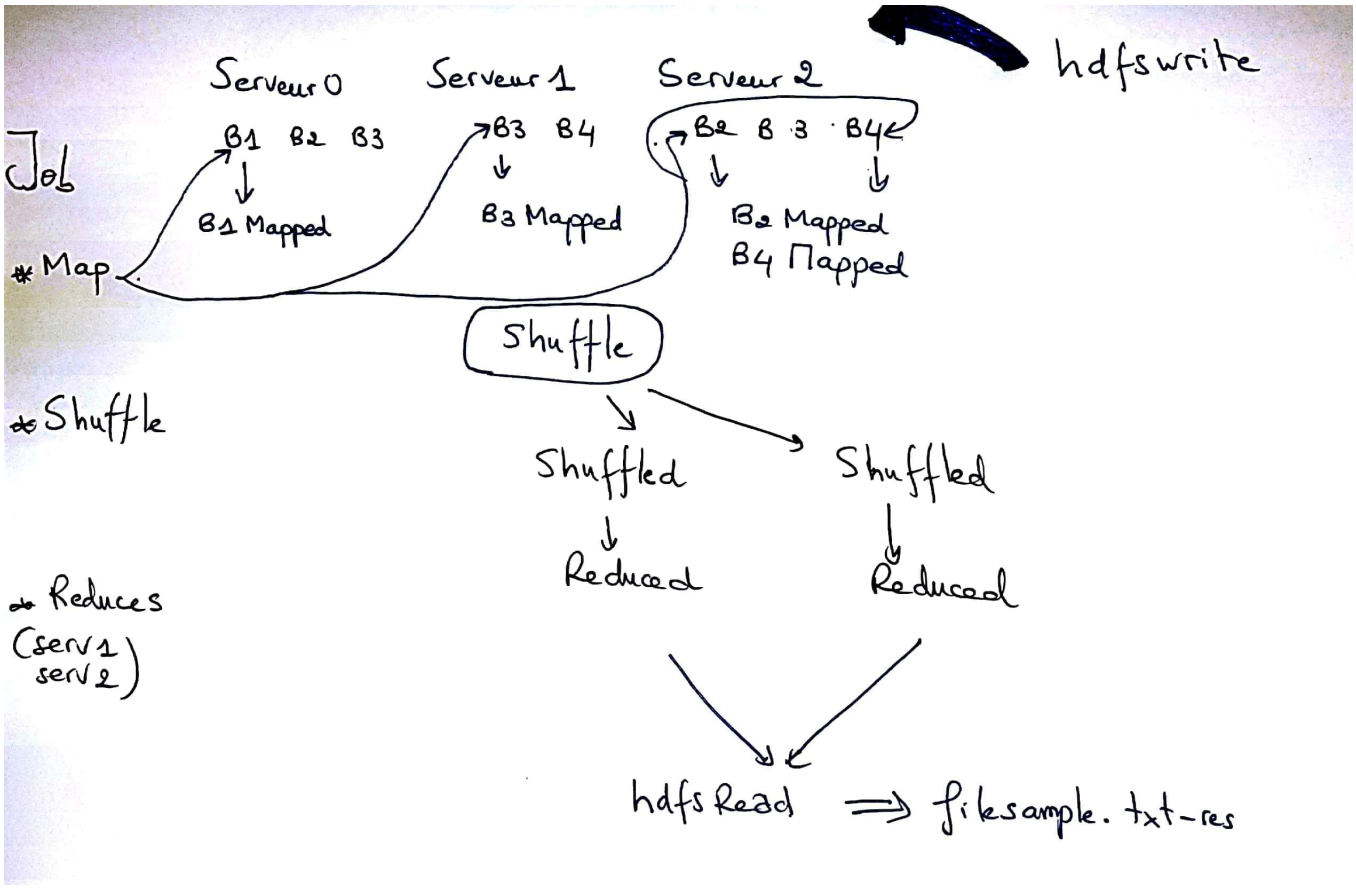

FIGURE 3 – Schéma des tests

5.2 Les captures des tests

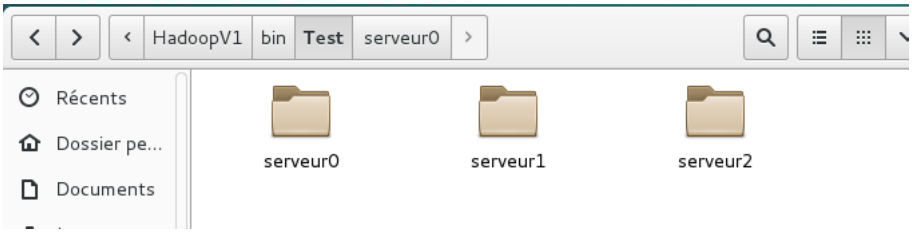

Figure 4 – Environnement de test

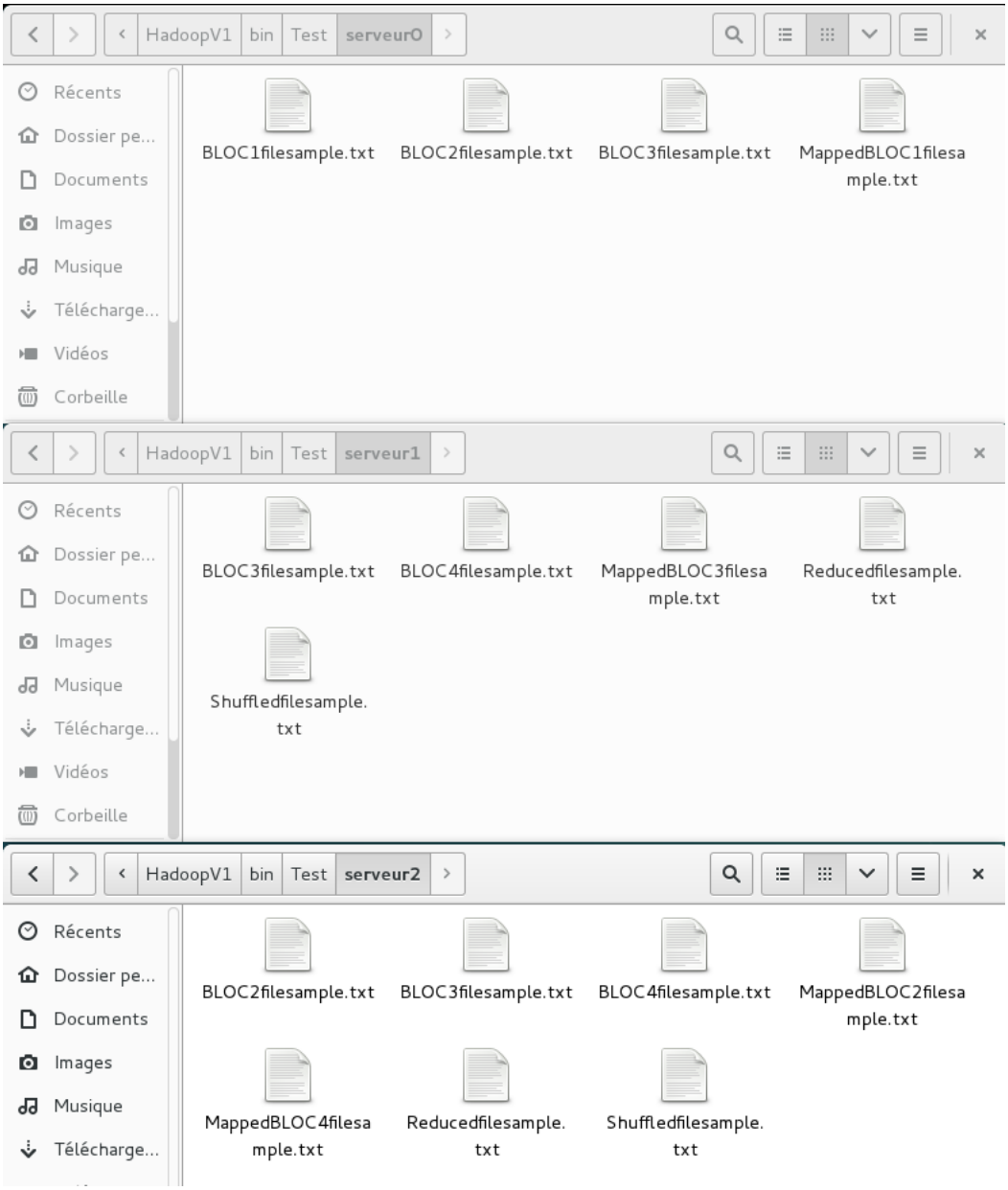

Figure 5 – Test de l'application & fichiers générés

6 Conclusion

Lors de l'élaboration de cette partie du projet , nous avons rencontré plusieurs difficultés, comme cité plus haut dans le rapport, mais la bonne lecture du sujet et le temps passé pour bien maîtriser le projet nous a permis de surmonter les difficultés et d'avoir une vision sur la totalité des tâches à réaliser et de la manière dont il faut s'y prendre.

A la fin de cette partie du projet, nous avons été capable de réaliser un traitement `Map-Reduce` avec un *Shuffle* et plusieurs *Reduce*, et on a implémenté un mechanisme de transmission entre les serveurs et un gestionnaire centralisé pour donner plus de sens au résultat de l'application pour savoir à tout moment l'état des machines et des serveurs. Pour la dernière partie, nous souhaitons fusionner la gestion des DataNodes et des serveurs pour pouvoir une application complète.

La réalisation de cette partie nous a permis aussi de nous familiariser avec ce système de fichiers en simulant son fonctionnement.