École Polytechnique de l'Université de Tours
64, Avenue Jean Portalis
37200 TOURS, FRANCE
Tél. +33 (0)2 47 36 14 14
www.polytech.univ-tours.fr

## Département Informatique
## 5e année
## 2013 - 2014

# Hadoop: Optimisation et Ordonnancement

**Encadrants**
JLASSI Aymen
aymen.jlassi@univ-tours.fr
MARTINEAU Patrick
patrick.martineau@univ-tours.fr

**Étudiant**
CLOAREC Erwann
erwann.cloarec@etu.univ-tours.fr

DI5 2013 - 2014

Université François-Rabelais, Tours

Version du 11 mai 2014

# Table des matières

# Introduction

## 1.1 Présentation du sujet

### 1.1.1 Contexte

Ce projet est le projet de fin d'études d'Erwann Cloarec, étudiant en 5ème année à l'école d'ingénieur Polytech Tours, en spécialité informatique. Il s'inscrit dans le cadre de la thèse d'Aymen Jlassi, qui a encadré ce projet, et en partenariat avec l'entreprise Cyrès et son pôle Ingensis. Ingensis travaille sur des problématiques Big Data et plus particulièrement sur le framework Hadoop, et c'est dans ce cadre qu'un partenariat a été fait pour la thèse avec Polytech. Aymen a proposé deux sujets de projets, un sur le monitoring d'Hadoop avec Starfish, l'autre sur l'optimisation et l'ordonnancement. Le premier sujet a été effectué par Lucas Delcroix, et le second par moi-même. Les deux PFE sont donc proches, et ont eu une partie en commun au début sur l'apprentissage d'Hadoop.

### 1.1.2 Sujet

Le sujet de ce projet est "Hadoop : Optimisation et Ordonnancement". Ingensis recherche à faire gagner des performances à Hadoop. Dans ce cadre, l'amélioration de la politique d'ordonnancement s'avère être une solution intéressante. L'objectif principal de ce projet est donc d'étudier l'ordonnancement dans Hadoop, et de savoir comment il est possible d'implémenter une politique d'ordonnancement dans Hadoop.

Une des premières choses à faire sera de comprendre et maîtriser le fonctionnement d'Hadoop, afin de cerner les problématiques d'ordonnancement qu'il rencontre. Il faudra ensuite maîtriser les différents types d'ordonnancement présents dans Hadoop, et étudier des propositions d'amélioration ou de nouveaux ordonnanceurs. Au niveau du développement de l'ordonnancement dans Hadoop, il faudra savoir comment implémenter une politique d'ordonnancement, c'est-à-dire avoir bien observé et compris le code. Pour finir, il faudra également développer un ordonnanceur afin de mettre en pratique les connaissances acquises aux étapes précédentes.

## 1.2 Environnement

### 1.2.1 Environnement logiciel

Au niveau de l'environnement logiciel, ce projet a nécessité de nombreuses machines virtuelles. En effet, l'école n'ayant pas à disposition un clusters pour apprendre à installer Hadoop et faire tourner du code personnel de ce framework, il fallait travailler sur ce type de solution. De plus, la mise en place d'un environnement de développement pour Hadoop a nécessité également une machine virtuelle.

Les systèmes utilisés sont CentOS 6 pour les machines du cluster, et Debian 7 pour la machine de développement. Le choix d'une machine de développement différente d'une machine du cluster a été fait car il fallait que la machine soit assez légère pour pouvoir la copier plus facilement, et que l'environnement de développement était plus facilement préparable avec Debian.

Du côté logiciels utilisés, j'ai pris Eclipse pour développer sur les sources d'Hadoop, avec plusieurs bibliothèques ou outils comme par exemple :

— Java 6
— Maven
— Google Protocol Buffers

La version d'Hadoop que j'ai utilisé et sur laquelle j'ai développée est la version 2.2.0. Cette version était la dernière version stable du framework, qui introduisait la version 2 d'Hadoop dans une version stable, au début de mon projet. A la fin de mon projet, les versions 2.3.0 et 2.4.0 ont été publiées, mais la version stable de référence reste la version 2.2.0.

### 1.2.2 Environnement matériel

Comme dit précédemment, je n'avais pas accès à un cluster mais seulement à une machine. J'ai donc travaillé principalement sur une machine à l'école, et occasionnellement sur mes machines personnelles.

### 1.2.3 La gestion de projet

Ce projet de fin d'étude a permis d'utiliser toutes les connaissances en gestion de projet que j'ai pu accumuler dans mes études. Premièrement, la préparation et l'apprentissage du sujet était important afin de pouvoir découper le travail en différentes tâches. Il fallait également cerner le système et spécifier le travail afin d'avoir un cahier de spécification complet.

Pour gérer les tâches, j'ai utilisé Redmine. Cet outil m'a permis de gérer mes tâches et l'avancement du projet. De plus, il met à disposition un dépôt SVN, qui m'a permis de gérer les versions de code de mon développement.

Pour ce qui est du suivi de projet, nous avions régulièrement des réunions avec mon encadrant et avec Lucas afin de suivre l'avancement des deux PFE. Ces réunions permettaient de faire le point, de revoir les objectifs, de revoir certaines parties ou d'avancer dans le projet.

# Le déroulement du projet

## 2.1 Initialisation du projet

### 2.1.1 Compréhension du sujet

La compréhension du sujet passait par de la documentation sur Hadoop. C'est sur ce framework que ce porte l'intégralité du sujet de mon projet.

### 2.1.2 Premiers objectifs

Les premiers objectifs du projet étaient dans l'apprentissage et la documentation sur Hadoop. J'avais obligation de bien connaître le framework afin de continuer dans mon travail. Les premiers objectifs étaient donc de :

— Étudier Hadoop, ce qu'il permet de faire, quand est-il utilisé et comment.
— Étudier l'administration d'un écosystème Hadoop, de son installation à sa configuration.
— Rechercher la ou les politiques d'ordonnancement dans Hadoop.

Ces étapes étaient obligatoire afin d'arriver à une maîtrise totale du sujet. Elles représentent une grande partie du temps que j'ai passé sur ce projet, car Hadoop est un grand projet Open Source qui est difficile a appréhender totalement.

## 2.2 La mise en place des objectifs et d'un planning

### 2.2.1 Les objectifs prévus pour le projet

En connaissant les grandes étapes du projet, j'avais besoin de détailler et de séparer tout cela en tâches afin de bien spécifier mon projet. Dans le cahier de spécification de mon projet est détaillé les tâches prévues pour le projet, et le temps estimé. Voici la liste des tâches prévues :

1. Le pilotage du projet
2. La compréhension du sujet (documentation sur Hadoop)
3. Rédaction du cahier de spécification
4. L'installation d'Hadoop
5. L'étude d'ordonnanceurs dans Hadoop et d'améliorations possibles
6. Le choix d'un ordonnanceur à implémenter
7. L'implémentation de l'ordonnanceur
8. Tests d'intégration de l'ordonnanceur
9. Rédaction du rapport et préparation de la soutenance

### 2.2.2 La construction du planning

A partir de ces tâches, j'ai construit un planning. Ce planning prévoyait que la phase d'étude et de documentation se terminerait fin Janvier 2014, et que l'implémentation commencerait à ce moment là. Le début des tests d'intégration étaient prévu pour le 11 Mars 2014.

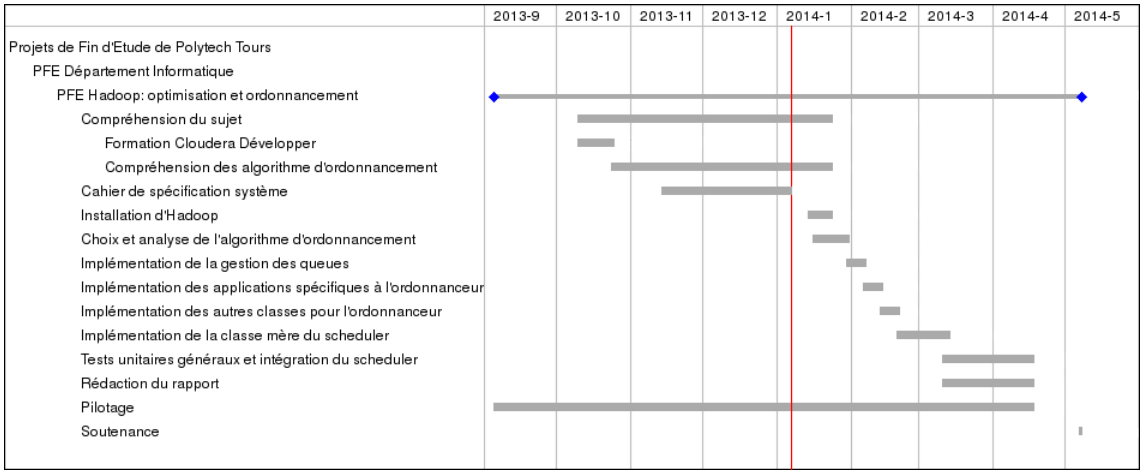


FIGURE 2.1 – Le diagramme de Gantt initial du cahier de spécification

## 2.3 Les résultats

### 2.3.1 Les objectifs

Les objectifs du projet ont été remplis, mais le temps passé sur chaque tâche n'était pas assez bien prévu. Notamment, la phase de recherche et documentation a été plus longue que prévu :

— L'installation d'Hadoop a pris plus de temps que prévu compte tenu de la difficulté et des connaissances nécessaires.

— La recherche sur l'ordonnancement dans Hadoop a pris également plus de temps que prévu car il fallait absolument que je connaisse précisément comment cela fonctionne sans quoi la phase de développement aurait été très difficile voir impossible. Cette phase était la plus importante et constituait le coeur de coeur de mon travail, alors que l'implémentation n'en était qu'un support. Nous avons donc préféré revoir avec mon encadrant ce qui était prévu pour l'ordonnanceur à implémenter. En effet, il valait mieux avoir un ordonnanceur moins ambitieux, mais plus de connaissances sur l'ordonnancement dans Hadoop, qu'un ordonnanceur plus ambitieux et finalement beaucoup plus difficile à implémenter sans ces connaissances. Cela a eu quelques conséquences au niveau des tests d'intégration, puisqu'ils prévoyaient une éventuelle comparaison de mon implémentation et de ce qui existe dans Hadoop, ce qui n'a pas été le cas. Les tests d'intégrations se sont donc limités à tester le Scheduler sur un cluster Hadoop.

### 2.3.2 Le planning

Concernant le planning, certaines durées prévues ont bougées. Comme nous l'avons vu précédemment, les temps prévus pour l'installation d'Hadoop, et les recherches sur l'ordonnancement ont été revues à la hausse, ce qui fait que le temps passé à développer et à tester l'intégration ont été réduis. J'ai donc commencé le développement à la fin Mars.

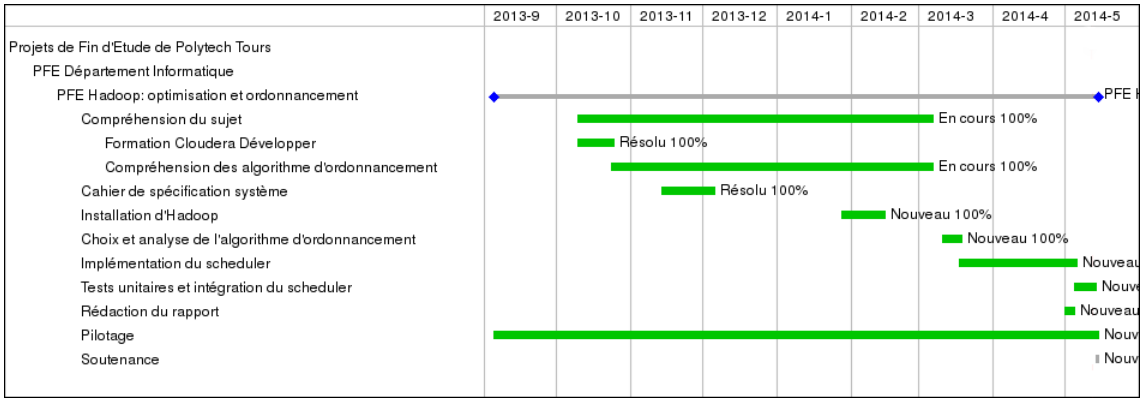


FIGURE 2.2 – Le diagramme de Gantt effectif

### 2.3.3 Les difficultés rencontrées

J'ai rencontré quelques difficultés pendant ce projet. Ces difficultés sont principalement ce qui n'étaient pas prévues et qui ont retardé certaines tâches.

L'installation d'Hadoop a duré plus de temps que prévu pour plusieurs raisons. Premièrement le matériel qui m'était fournis était une machine doté d'un processeur à 4 coeurs et 8 Go de RAM. Cela était largement suffisant pour faire tourner une seule machine virtuelle, mais lorsqu'il a fallut créer deux autres machines pour simuler un cluster, la machine hôte montrait rapidement ses limites. On pouvait voir cela au temps d'exécution d'un simple Map/Reduce. De plus la communication entre les machines virtuelles était assez compliqué, à cause de certains réglages de la machine hôte. Une deuxième difficulté était d'administrer les systèmes sous CentOS. La configuration d'un système étant différent d'une distribution Linux à une autre, j'ai dû me familiariser avec ce système afin de le configurer correctement pour Hadoop. Pour finir, une autre difficulté était de trouver des renseignements pour installer la dernière version d'Hadoop. En effet, cette version était relativement récente, et les aides à l'installation étaient plutôt pour de plus vieilles versions. Ainsi, les noms des configurations et des fichiers associés n'étaient pas forcément les mêmes. Ces difficultés ont fait que j'ai passé plus de temps que prévu à cette tâche.

Les recherches sur l'ordonnancement dans Hadoop m'ont également posé problème. La principale raison à cela était le manque d'articles ou de livres sur le sujet. En effet, la plupart des articles que je trouvaient expliquaient généralement l'ordonnancement, sans pour autant détailler. Je devais donc comprendre le code afin de mieux appréhender le sujet, ce qui était relativement difficile compte tenu du nombre de classes présentes pour l'ordonnancement dans Hadoop. Je devais donc extraire les données importantes de l'ordonnancement, c'est-à-dire trouver l'objectif d'un ordonnanceur dans Hadoop, les ressources qu'il gère, et les contraintes auxquelles il a affaire, en m'aidant seulement du code source d'Hadoop.

Ces difficultés rencontrées ne démontrent pas un problème de prévision de temps, mais plutôt un soucis à ne pas passer à une étape suivante sans être certain d'avoir terminé et bien assimilé l'étape en cours. Cela est particulièrement vrai pour l'étude de l'ordonnancement dans Hadoop, puisque c'est le coeur de mon projet, et que bien le comprendre est l'objectif principal, l'implémentation ne servant que de support à ce travail afin de prouver que mes recherches et mon apprentissage ont été bons. Il fallait donc privilégier ces recherches et que mon projet soit beaucoup plus axé sur cela que sur l'implémentation.

# Étude d'Hadoop

## 3.1 Présentation générale d'Hadoop

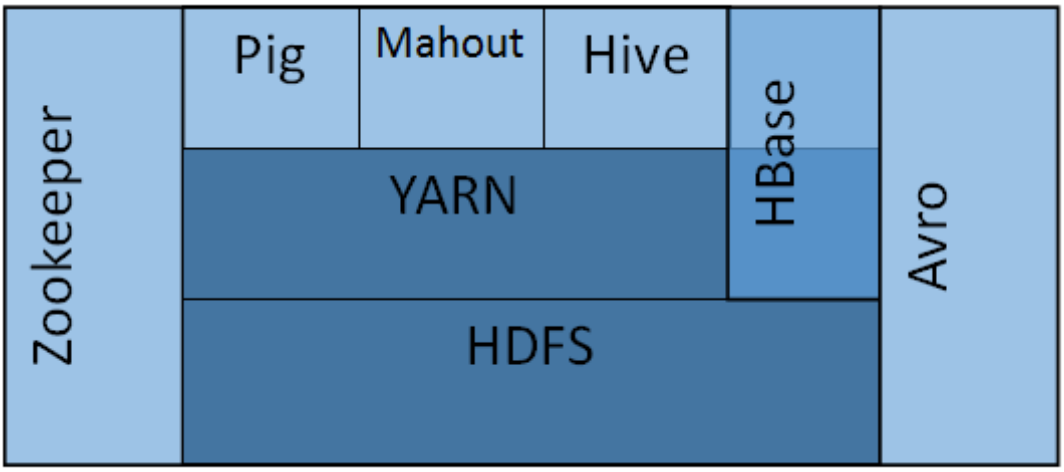


FIGURE 3.1 – Présentation de l'écosystème Hadoop : les principaux composants

Hadoop est un framework open-source fournissant des outils pour exploiter un cluster dans le but de stocker et manipuler des grands volumes de données rapidement et optimalement, et d'exploiter la puissance de calcul des machines présentes dans ce cluster. Hadoop est géré par la fondation Apache et est sous licence Apache Licence 2.0.

Hadoop a été créé en 2005, basé sur des travaux de Google publié en 2004 sur le Map/Reduce et sur GoogleFS, un système de fichier distribué. C'est Doug Cutting qui l'a créé et a choisi le nom et le logo grâce à la peluche de son fils, un éléphant jaune qu'il appellait Hadoop.

Le framework utilise l'environnement Java. Il représente en fait deux composants principaux et de plusieurs composants secondaires. Les deux composants principaux sont le système de fichier HDFS, et le Map/Reduce. Le système de fichier viens directement du système de fichier GoogleFS, et permet de stocker de gros volumes de données de façon sécurisé sur tout le cluster. Le Map/Reduce viens également des travaux de Google publiés en 2004, qui permet de lancer des algorithmes distribués.

Hadoop est utilisé par de nombreuses entreprises ayant de grands besoin en données, comme Yahoo !, Facebook ou Ebay parmis les plus connus. On compte par exemple 532 noeuds pour Ebay, ou 1400 pour Facebook en 2013 [1].

Depuis Septembre 2013, la version 2 d'Hadoop est sortie, avec quelques changements au niveau du composant Map/Reduce. En effet, il est passé à la version 2, et s'appelle désormais YARN (Yet Another Resource Negotiator). YARN gère divers types d'application pour utiliser le cluster et ses données. Le Map/Reduce de la première version d'Hadoop est donc devenue un type d'application de YARN. Dans les faits, cela ne change pas beaucoup de choses, puisque les Map/Reduce classiques sont toujours gérés par Hadoop.

## 3.2 Le système de fichier HDFS

Comme nous l'avons vu précédemment, HDFS est un système de fichier basé sur GoogleFS. Il permet de stocker des données très importantes sur un cluster en faisant abstraction des capacités des machines, ou de leur état. Les données sont sécurisées, et le système est tolérant à la panne. Sur un cluster de taille importante, la panne est assez commune, c'est pour cela que HDFS doit savoir gérer ce genre d'évènement. Son architecture est donc basée et pensée en fonction de cela.

Le système utilise des blocs de données d'une plus grande taille que les systèmes de fichiers que l'on connait. Cette valeur est de 64Mo, mais peut être modifiée. Chaque fichier est donc décomposé en bloc, et ces blocs sont distribués sur le cluster. Il existe un taux de réplication des blocs par défaut mis à 3, mais là encore modifiable. Chaque bloc est donc présent sur 3 noeuds à la fois. C'est grâce à cela que la perte d'un noeud n'est pas grave, puisque les blocs perdus sont répliqués dans un autre noeud.

L'architecture est faite de la façon suivante. HDFS se compose d'un noeud principal, appelé le NameNode. Ce noeud est très important car c'est lui qui va gérer l'emplacement de l'ensemble des données. Il fait la correspondance entre un fichier et ses blocs associés (les metadata d'un fichier), et il sait également sur quels noeuds chaque bloc se situe. Sur les autres noeuds se trouvent les DataNode. Un DataNode va gérer les blocs de données présent sur son noeud. Le DataNode tiens très souvent le NameNode au courant des blocs qu'il gère, et c'est avec ce principe qu'il est possible au NameNode de détecter des problèmes et de demander la réplication des blocs. Les DataNodes ne gèrent pas de fichiers, mais seulement des blocs. La notion de fichier est géré par le NameNode. Il va pouvoir ouvrir, fermer, supprimer des fichiers, et répercuter ces changements aux DataNodes concernés. Il va donc demander aux DataNodes de créer des blocs, de les supprimer, de les lire ou écrire dedans. Le NameNode peut poser problème en cas de défaillance de son noeud, c'est pour cela qu'il existe un Secondary NameNode, qui va recevoir de temps en temps les données du NameNode, et qui va pouvoir, en cas de défaillance du NameNode prendre sa place. Les données sont donc bien sécurisées dans la plupart des cas, et leur accès est donc garantie.

## 3.3 La gestion des applications : YARN

### 3.3.1 Architecture

YARN est le deuxième composant principal d'Hadoop. Il va gérer diverses applications comme le montre la figure 3.2.

Le fonctionnement de son architecture se fait avec un système maîtres/esclaves. Le RessourceManager représente l'autorité suprême du cluster. Il est composé de deux rôles : la gestion des ressources du cluster et la gestion des applications. Il va donc gérer soumission des applications sur le cluster, et va donc assigner à chaque application des ressources d'un noeud (ce qu'on appelle un conteneur ou Container) qui pourra gérer l'exécution de cette application. L'exécution des applications n'est donc pas centralisé sur un seul noeud. Chaque application aura donc son ApplicationMaster tournant sur un noeud du cluster.

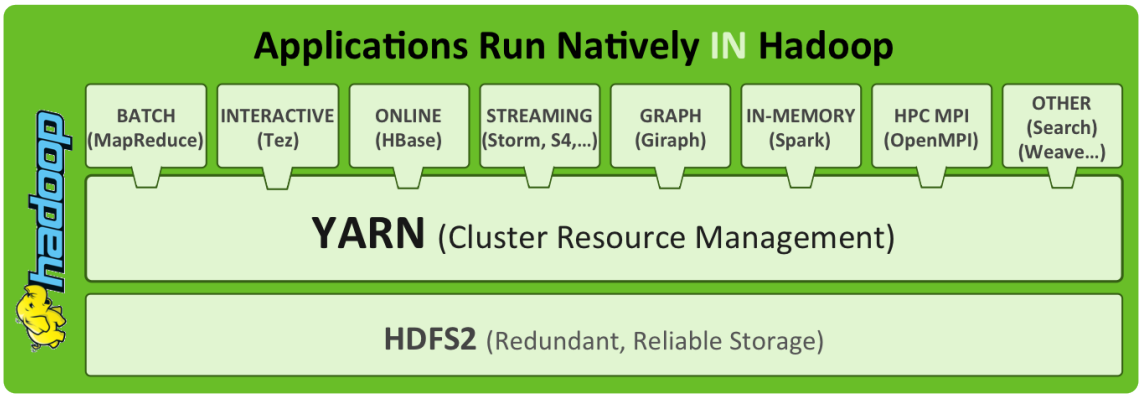


FIGURE 3.2 – Les différentes applications sous YARN (http://hortonworks.com/hadoop/yarn/)

La gestion des ressources du cluster se fait avec le Scheduler, qui fait parti du ResourceManager. Il va devoir assigner aux ApplicationMaster des ressources venant de noeuds suivant la demande de ces-derniers, et suivant le type d'ordonnancement. Les applications peuvent être différentes, mais elles ne sont traitées selon leurs types par le Scheduler, elles sont traitées par leurs demande en ressources sur le cluster. Le figure 3.3 schématise un exemple de distribution des ressources d'un cluster pour deux applications.

Chaque noeud du cluster est composé d'un NodeManager, qui va gérer les demandes de ressources sur ce noeud. Il va tenir le ResourceManager au courant grâce au heartbeat. Le heartbeat est envoyé par tous les noeuds au ResourceManager pour donner ses informations. Les ressources demandées, regroupées en conteneurs, sont des ressources d'une machine :

- La mémoire
- Le CPU
- Le disque
- Le réseau
- ...

Le rôle du Scheduler du RessourceManager n'est pas de gérer tous les détails d'exécution des applications, mais bien de gérer les ressources demandées par chaque application par rapports aux noeuds disponibles. Il y a également un Scheduler pour chaque ApplicationMaster, qui va gérer les conteneurs alloués à cette application et distribuer ses propres tâches sur ces conteneurs.

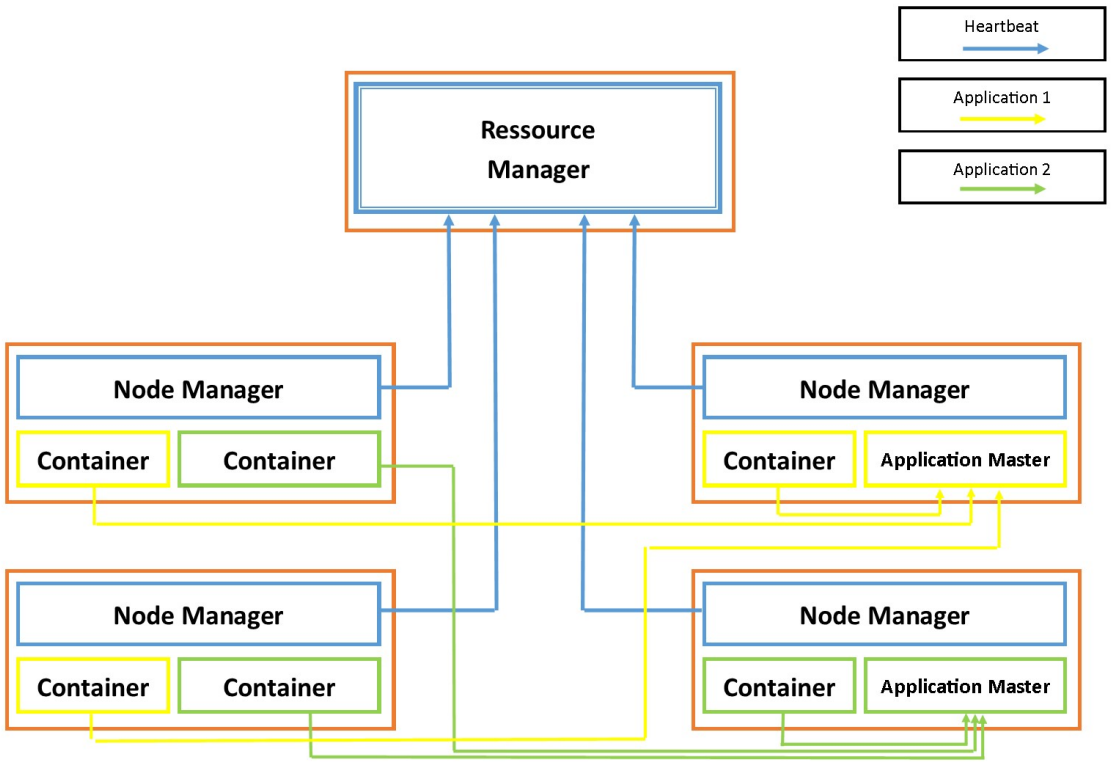


FIGURE 3.3 – Présentation de l'architecture de YARN avec 2 applications

### 3.3.2 Le Map/Reduce

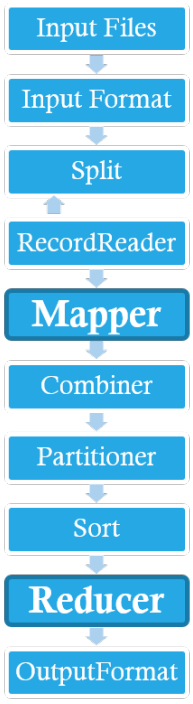


FIGURE 3.4 – Les étapes d'un Map/Reduce

Le MapReduce est un type d'application lancé sur Hadoop, c'est le plus célèbre. Il gère des paires clé-valeur. Il est séparé en deux étapes importantes : le Map et le Reduce. L'objectif est de partager le travail sur le cluster (Map) et de réunir les résultats (Reduce). Voici le détail des opérations d'un MapReduce, donné dans la figure 3.4 :

1. InputFile : c'est le(s) fichier(s) d'entrée, il est donné par l'ApplicationMaster. La plupart du temps stocké dans HDFS.
2. InputFormat : cette étape permet de sélectionner les fichiers à traiter. Elle définie les InputSplits, ce qui va séparer l'application en tâches suivant le fichier. Chaque tâche va donc être un Map. Enfin, elle donne un lecteur générique pour le(s) fichier(s).
3. InputSplit : cette étape découpe le fichier en plusieurs tâches à effectuer. Elle sait comment le fichier est découpé. Voir la figure 3.5.
4. RecordReader : cette étape va transformer un InputSplit en paire de clé-valeur. Elle est rappellé jusqu'à ce que l'InputSplit soit terminé. C'est elle qui garantie la non-duplication de couples. Voir la figure 3.6.
5. Mapping : c'est la tâche de base de MapReduce. Elle est définie par l'utilisateur. Elle transforme les paires clé-valeurs en une nouvelle liste de paires clé-valeur. Voir figure 3.7.
6. Combiner : il sert à regrouper les résultats des mapping effectués sur un noeud. Cela permet de regrouper les données à envoyer sur le réseau.
7. Partitioner et shuffle. Cette étape vise à décider à qui envoyer les valeurs stockées pour le moment sur le noeud ayant effectué des map. Cette décision dépend des clés. Le shuffle est l'étape d'envoi des données
8. Sort : il sert à réorganiser les données d'entrée d'un Reducer. Il regroupe les valeurs par clés. Par exemple : si le Reducer reçoit («chat», 4), («chat», 3), («chat», 1), le tri permet de donner : («chat», (4,3,1)).

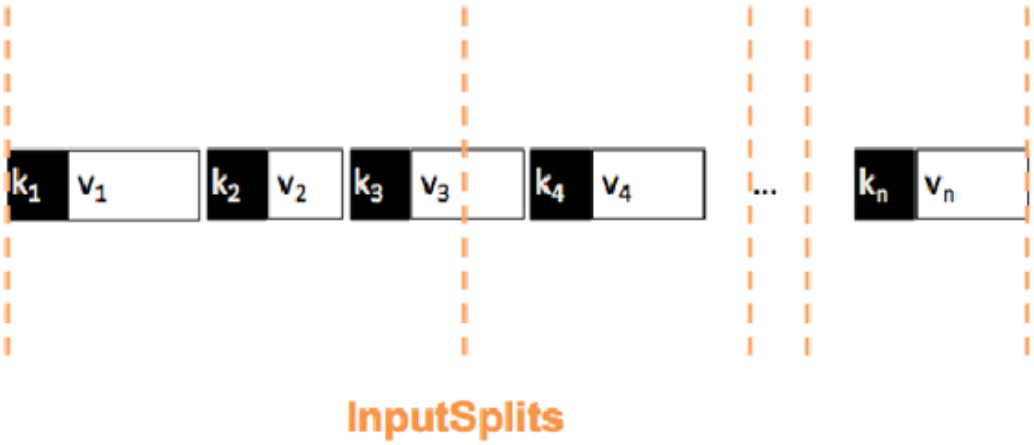


FIGURE 3.5 – L'étape du InputSplit

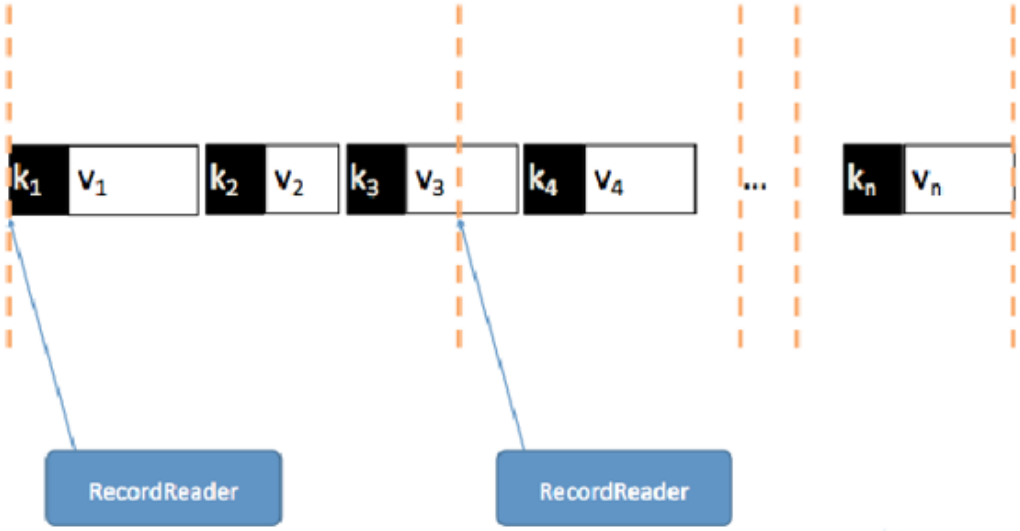


FIGURE 3.6 – L'étape du RecordReader

9. Reducer : c'est la deuxième tâche importante du Map/Reduce. Il va donner une valeur de sortie (output) pour chaque clé qu'il traite. Voir figure 3.8.
10. OutputFormat : c'est l'écriture des résultats sur le système HDFS.

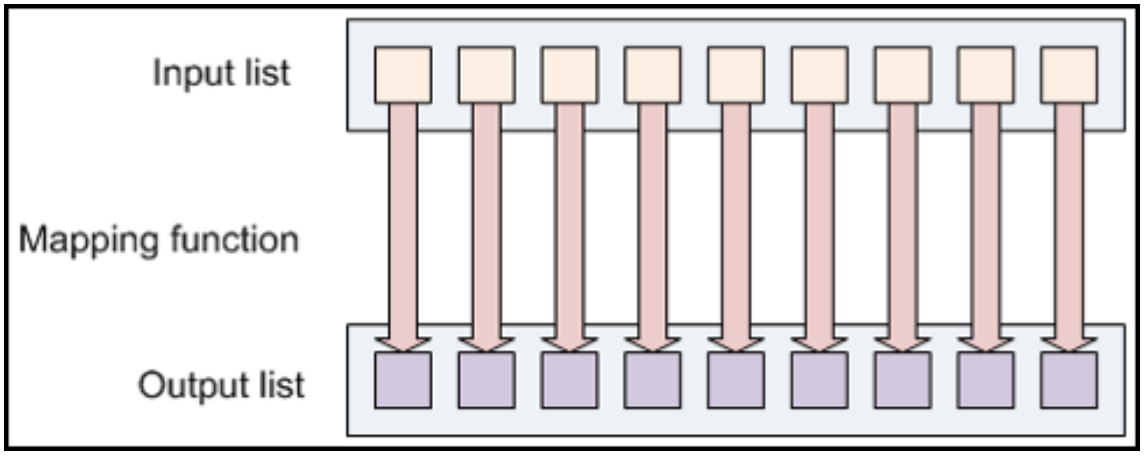


FIGURE 3.7 – L'étape du Mapping (source : [7])

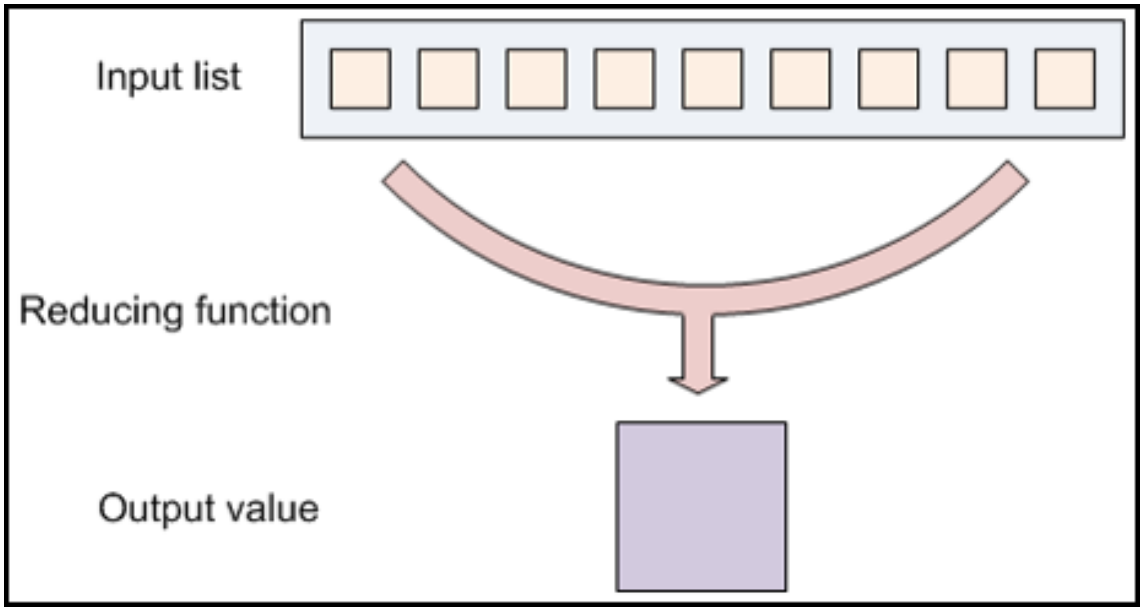


FIGURE 3.8 – L'étape du Reducer (source : [7])

## 3.4 Les autres composants

Il existe également d'autres composants d'Hadoop, comme HBase. Hbase est une base de données de type NoSQL distribuée tournant sur HDFS. Cette base est structurée pour de très grandes tables. On peut également noter Hive, qui permet d'effectuer des applications Map/Reduce avec du SQL, ou Pig, qui fait la même chose avec un langage proche du SQL. ZooKeeper sert a centraliser et faciliter les services d'Hadoop.

## 3.5 L'installation d'Hadoop : l'exemple avec des machines virtuelles sous CentOS

Cette partie détaille l'installation d'Hadoop 2.2.0 sur un cluster avec un noeud dit master qui contiendra le ResourceManager et le NameNode, et deux noeuds dits esclaves, qui vont chacun avoir un DataNode et un NodeManager. Pour commencer, j'ai téléchargé Hadoop : http://apache.lehtivihrea.org/hadoop/common/hadoop-2.2.0. L'objectif étant de faire 3 machines virtuelles, j'ai tout d'abord créé une seule machine que j'ai dupliqué par la suite, une fois hadoop installé et configuré. J'ai donc installé CentOS 6 sur une nouvelle machine virtuelle. J'ai créé un utilisateur $hadoop$, qui aura les droits sur Hadoop. J'ai ensuite fixé l'adresse IP de la machine à $192.168.1.1$, ce qui est très important, puisqu'avec une adresse IP changeante, les communications seraient impossible. Dans un cadre optimal, il faudrait supprimer tout serveur DHCP. J'ai également fixé le nom de la machine à $master$. Pour que les machines se reconnaissent entre elles, il faut configurer le fichier $/etc/hosts$ avec :

```
192.168.1.1  master
192.168.1.2  slave1
192.168.1.3  slave2
```

J'ai ensuite extrait les fichiers téléchargés d'Hadoop dans $/opt/hadoop$, et j'ai donné les droits à l'utilisateur $hadoop$ avec la commande :

```
#chmod -R hadoop /opt/hadoop
```

Il fallait ensuite éditer les fichiers de configuration. Dans les fichier core-site.xml, il fallait éditer :

```
<property>
    <name>fs.default.name</name>
    <value>hdfs://master:9000/</value>
</property>
<property>
    <name>dfs.permissions</name>
    <value>false</value>
</property>
```

Pour le fichier hdfs-site.xml :

```
<property>
        <name>dfs.data.dir</name>
        <value>/opt/hadoop/hadoop/dfs/name/data</value>
        <final>true</final>
</property>
<property>
        <name>dfs.name.dir</name>
```

```
        <value>/opt/hadoop/hadoop/dfs/name</value>
        <final>true</final>
</property>
<property>
        <name>dfs.replication</name>
        <value>1</value>
</property>
```

On a ici configuré le taux de réplication des blocs à 1, pour ne pas surcharger les machines, et configuré les chemins vers les dossier qui contiendront les fichiers de HDFS. On peut ensuite dupliquer la machine, et configurer les adresses IP et les noms des machines slave1 et slave2. Pour finir, il faut absolument configurer les clés ssh des machines, pour qu'elles puissent communiquer sans que l'on ait besoin de taper le mot de passe à chaque fois. Sur chaque machine, il faut donc :

```
$ ssh-keygen -t rsa
```

Il faut ensuite copier toutes les clés publiques des autres machines dans $/.ssh/authorized_keys$. Les machines pourront alors communiquer sans mot de passe.

Pour finir, il faut encore configurer hadoop pour qu'il connaisse les machines présentes dans le cluster, pour cela il faut éditer les fichiers de configuration $masters$ et $slaves$ avec les noms de machines. Avant de lancer Hadoop pour la première fois, il faut formater le NameNode et avec

```
$ /opt/hadoop/hadoop/bin/hadoop namenode -format
```

On peut ensuite lancer Hadoop :

```
$ bin/start-all.sh
```

Normalement, le NameNode et le ResourceManager sont lancés sur le noeud master, et sur chaque esclave sont lancés un NodeManager et un DataNode. Cela peut être vérifié avec la commande $jps$.

# Etudes de types d'ordonnancement

Dans ce chapitre, nous allons voir ce l'ordonnancement dans Hadoop d'un point de vue théorique.

## 4.1 L'ordonnancement dans Hadoop : définition

### 4.1.1 Ce qui doit être ordonnancé

Dans Hadoop, le Scheduler du ResourceManager a pour but d'attribuer des ressources aux applications. Dans la version 2.2.0 d'Hadoop, le seul type de ressource géré sont la mémoire des noeuds. Une évolution est possible et déjà prévue. Le scheduler doit donc gérer les applications qui lui sont soumises de la façon qu'il le souhaite, et leur attribuer des conteneurs (ie des ressources de noeuds), suivant les demandes de ces applications.

Une application doit aussi gérer les ressources qui leur sont attribuées en les partageant à des tâches qu'elle doit exécuter. Ceci peut être fait par l'utilisateur codant son application, puisque les demandes de ressources peuvent être faites à la main.

### 4.1.2 Les contraintes

Les contraintes auxquelles doit se plier le Scheduler sont les limites du cluster. En effet, il ne peut attribuer plus de ressources que le cluster possède. Il ne peut également pas attribuer plus de ressources qu'un noeud ne peut donner. Le Scheduler doit donc toujours être au courant des ressources utilisées, des ressources totales et des ressources disponibles, que ce soit pour le cluster en entier ou pour chaque noeud.

Du point de vue de l'ApplicationMaster, les contraintes sont différentes, puisque les ressources qui lui ont été attribuées sont légitimes. Il doit donc seulement répondre aux besoins de ses tâches.

### 4.1.3 L'objectif

L'objectif du Scheduler du ResourceManager est de gérer toutes les demandes d'applications en utilisant le maximum de ressource du cluster. En effet, il ne serait pas tolérable d'avoir un Scheduler n'attribuant pas les ressources du cluster alors qu'un certain nombre d'applications en attendent. Le rôle du Scheduler est donc de satisfaire au maximum les applications.

Au niveau de l'ApplicationMaster, le but de l'ordonnancement est d'exécuter toutes les tâches en un minimum de temps.

## 4.2 Les ordonnanceurs dans Hadoop

Nous allons maintenant voir les différents Scheduler présents dans Hadoop

### 4.2.1 FIFO Scheduler

Le FIFO Scheduler est l'ordonnanceur de base d'Hadoop. Il gère les applications dans une file First In First Out. L'application la plus prioritaire sera la plus vieille. Ce scheduler va tout de même répartir toutes les ressources du scheduler, une assignation n'étant pas bloquante. C'est le premier scheduler qui a été implémenté dans Hadoop. L'inconvénient d'un tel scheduler est le temps d'attente qui peut être long, voir très long dans certains cas où il y beaucoup d'applications gourmandes en tête de file. C'est pour cela que deux autres scheduler ont été implémentés dans Hadoop.

### 4.2.2 Capacity Scheduler

Le Capacity Scheduler est un ordonnanceur qui va gérer les applications selon les utilisateurs qui les ont soumises. Il va répartir les applications dans des queues selon un certain critère, par défaut l'utilisateur. L'allocation des ressources se fait de façon équitable avec les queues. Les ressources inutilisées sont réparties aux autres queues jusqu'à ce que la queue demande de nouvelles ressources. Les queues peuvent avoir une priorité supérieure aux autres queues. L'intérêt de ce scheduler est de partager équitablement un cluster entre plusieurs entités.

### 4.2.3 Fair Scheduler

Le Fair Scheduler va répartir les applications dans des pools, et va s'efforcer à partager les ressources de façon équitable. Par défaut, un pool va être créé pour chaque utilisateur, mais on peut également spécifier le pool dans lequel on veut que l'application se situe. L'assignation des ressources se fait de façon équitable. Un système de priorité est mis en place pour les pools qui n'ont pas assez de ressources, on dit que ces pools sont en dessous de leur minimum share (part minimum). Ils sont alors prioritaires pour recevoir de nouvelles ressources disponibles. La part équitable d'un pool peut être calculée avec une priorité configurée au préalable. Cela permet également de prioriser certaines personnes ou entités soumettant des applications.

Dans les pools, le partage des ressource se fait en FIFO, mais cela peut être configurable pour être également fait de façon équitable. En effet, un pool peut également contenir des pools, avec qui il pourra également partager les ressources, comme pour les pools principaux. On a donc un système d'arbre avec ces pools, et les ressources sont distribuées de façon équitable ou en FIFO selon l'endroit dans l'arbre. Les feuilles de cet arbre sont les applications.

Ce système permet de partager les ressources équitablement et d'éviter la famine de certaines applications. Cela diminue également le temps d'attente des petites applications, qui vont se voir allouer plus facilement ce qu'ils demandent.

## 4.3 Les améliorations des ordonnanceurs d'Hadoop

J'ai étudié des améliorations possibles de la politique d'ordonnancement dans Hadoop. Il était intéressant de voir ce qui pouvait être amélioré, cela faisait parti de ma formation sur l'ordonnancement dans Hadoop. J'ai donc étudié en détail un article fourni par mon encadrant.

Ceci est une étude des améliorations proposées dans l'article "Survey on Improved Scheduling in Hadoop MapReduce in Cloud Environments" de B.Thirumala Rao et Dr. L.S.S.Reddy ([4]).

### 4.3.1 LATE Scheduling

Ce scheduler signigifie "Longest Approximate Time to End", à comprendre le temps approximatif le plus long avant la fin. Le principe de ce scheduler dans Hadoop est basé sur le fait que dans la première version d'Hadoop, l'ordonnancement des jobs MapReduce se faisait avec un système de slots. Il n'y avait pas de distinctions entre les puissances des noeuds, et le scheduler spéculait donc sur une homogénéité du cluster. Si ce type d'ordonnancement se faisait dans un environnement hétérogène, on pouvait se retrouver avec une grande perte de performance du scheduler.

Seulement depuis la version 2 d'Hadoop et YARN, il n'y a plus seulement du MapReduce, mais également d'autres types d'applications qui sont gérés par Hadoop. Le scheduler doit donc gérer toutes ces applications, et le système de slots est devenu obselète. La nouvelle version d'Hadoop considère désormais des Noeuds avec des containers, qui vont être gérés par le scheduler non plus en tant que slots, mais en tant que containers. Pour le moment, Hadoop ne gère que la mémoire comme ressources fournis par ces containers, mais dans un futur proche, il est prévu de gérer différentes ressources comme le CPU, la capacité du disque ou la disposition du noeud dans le réseau.

### 4.3.2 Delay Scheduling

Le Delay Scheduling est une amélioration qui peut être apporté au Fair Scheduler qui consiste à laisser une tâche de Mapping attendre qu'un noeud avec les données en local se libère. Cela permet de limiter le trafic réseau car la recopie des données est évitée.

Dans la version 2 d'Hadoop, le FairScheduler intègre déjà ce type de Scheduling. Lorsque l'on tente d'assigner un Container à une application (ie : le scheduler a donné au QueueManager un container à assigner, qui a donné à ses fils triés le container, et ainsi de suite jusqu'à un noeud feuille, qui va assigner le Container à l'application), un système de priorités se met en place pour voir si il est intéressant de donner ou non le container à l'application.

Ce système permet donc à une application de refuser les container qui lui sont proposés dans un soucis de localité des données. Au bout d'un certain temps, l'application changera de priorité, et choisira plutôt un container d'un noeud situé dans la même rack qu'un noeud contenant les données.

### 4.3.3 Dynamic Priority Scheduling

Ce scheduler vise à introduire la notion de partage du cluster de façon dynamique. Chaque queue peut dépenser à un moment donné son quota ou budget qui lui est accordé et devenir plus prioritaire que les autres.

Ce système est une amélioration du Scheduler de base d'Hadoop le FIFO. Mais il semble limité et beaucoup moins efficace que le Fair Scheduler.

### 4.3.4 Deadline Constraint Scheduler

Ce scheduler prend en compte une nouvelle contrainte que l'utilisateur peut donner : une deadline. On va ainsi ordonnancer d'une nouvelle façon : si l'application peut finir avant sa deadline, on pourra lui attribuer des ressources, sinon elle devra attendre.

### 4.3.5 Resource Aware Sheduling

Ce scheduler va être beaucoup plus attentif à ce que les applications ont besoin : que ce soit en mémoire, en CPU, en disque ou en réseau. Les effort fait à ce niveau dans Hadoop 2 font que cette généralisation des types de ressources demandées sera possible dans un futur plus ou moins proche. Ce type d'ordonnancement est un des challenge à relever pour la communauté d'Hadoop.

# L'ordonnancement dans Hadoop : étude approfondie

Je me suis familiarisé avec les problèmes d'ordonnancement des tâches dans Hadoop grâce à diverses documentations, et ainsi aller de plus en plus vers le détail, vers l'implémentation de ces ordonnanceurs. J'ai plus particulièrement travaillé sur l'implémentation du Fair Scheduler.

## 5.1 Le rôle précis du Scheduler dans le ResourceManager

Le Scheduler fait partie intégrante du ResourceManager. Il doit prévenir le ResourceManager lorsqu'il a besoin de créer un conteneur ou de le détruire. Il a besoin de ses propres instances en parallèle de celles du ResourceManager afin d'y enregistrer et de fournir des informations spécifiques au Scheduler et à l'assignation des ressource. Il y a par exemple une classe pour les noeuds dans le ResourceManager, et une classe pour le Scheduler. On a cela également pour les applications.

En revanche, le Scheduler ne va pas aller vérifier ce qui est fait des ressources. Il ne va pas vérifier l'exécution de tâches sur les container qu'il alloue. Il ne va pas non plus vérifier que les demandes des applications sont correctes.

## 5.2 L'ordonnancement au niveau de l'Application Master

L'ordonnancement au niveau de l'Application Master est très important, car sur de grosses applications, il peut jouer un rôle très important dans les performances de l'application, et donc à plus grande échelle sur les performances d'Hadoop. Cet ordonnancement vise à savoir l'ordre des différentes tâches de l'application à exécuter. Une application doit donc gérer en temps réel les tâches qu'elle a à lancer avec les ressources qui lui sont données.

L'ordonnancement au niveau de l'Application Master peut se faire par le créateur de l'application. Cet ordonnancement est choisi dès la découpe en tâches de l'application. La création des requêtes de ressource se fait par rapport à cet ordre dans les tâches qui a été décidé au préalable. En effet, les demandes de ressources peuvent être priorisées. Une application ne fait jamais une seule demande de ressources, mais en fait plusieurs, une pour chaque tâche qu'elle peut exécuter sur le moment. Cette correspondance entre demande et tâche permet lorsqu'une réponse à une demande est reçu, de directement savoir quelle tâche sera exécutée avec ces ressources.

Il existe cepandant dans Hadoop un comportement par défaut de l'Application Master. En effet, il n'est pas requis d'écrire un ApplicationMaster à chaque fois que l'on écrit une application. De ce fait, il existe également un comportement par défaut de l'ordonnancement des tâches. J'ai étudié en détail l'ordonnancement des tâches d'un Map/Reduce dans YARN. La classe gérant une application s'appelle $RMAppMaster$, elle se situe dans le package $org.apache.hadoop.mapreduce.v2.app$. Elle va avoir un service qui va s'occuper d'assigner les container disponibles pour l'application aux tâches. Ce service se nomme $RMContainerAllocator$. Pour le moment, son implémentation est fait de la façon suivante :

— Les tâches arrivent les unes après les autres et sont placées dans des files.
— Il existe une file par hôte demandé
— Chaque tâche demande un ou plusieurs hôtes
— Chaque tâche peut donc être placé plusieurs fois

Les files sont des FIFO. Lorsqu'un container est disponible, on prend la file associée, et on prend la première tâche de cette file.

## 5.3 Les requêtes de ressources

La formulation des requêtes de ressources est très important pour comprendre l'ordonnancement dans Hadoop. Les requêtes sont normalisées selon un modèle. Une requête est en fait divisée en plusieurs requêtes. Une requête représente globalement une certaine quantité de ressources sur une certaine machine, ou une certaine rack. Les ressources demandées sont formulées sous forme de quantité de RAM nécessaire, et du nombre de container que l'on souhaite. On peut par exemple représenter ce que demande un ApplicationMaster par :

| priorité | RAM | Nom Machine | Nombre de container |
|---|---|---|---|
| 2 | 2GB | node1<br>node2<br>rack1 | 1<br>1<br>2 |
| 1 | 2GB | node3<br>* | 1<br>3 |

TABLE 5.1 – Exemple d'un tableau illustrant une demande de ressources d'une application

Ce tableau est inspiré de celui présent sur le site [6]. On remarque qu'une application permet également à une application de ne pas spécifier de nom de noeud ou de rack. Une requête d'un ApplicationMaster vers le ResourceManager passe par la fonction *allocate*, que nous allons voir plus tard.

## 5.4 Les classes abstraites fournies par Hadoop

Hadoop fourni des classes abstraites afin de développer un scheduler. Ces classes doivent être implémentées pour avoir un scheduler fonctionnel. La classe à implémenter se situe dans le package :

```
org.apache.hadoop.yarn.server.resourcemanager.scheduler
```

### 5.4.1 Les classes principales de Scheduler

Il existe trois classes principales de Scheduler. La première se nomme ResourceScheduler. Cette classe abstraite étend une autre classe abstraite nommée YarnScheduler, la deuxième classe principale. Cette classe va donner toutes les méthodes qui seront appelées dans la vie du Scheduler. Parmi ces méthodes, il en existe deux très importantes. Les autres méthodes sont principalement des méthodes pour que le Scheduler fournisse des informations, comme des métriques par exemples.

Les méthodes fournissant des informations sont les suivantes :

```
QueueInfo getQueueInfo(String queueName, boolean includeChildQueues, boolean recursive)
List<QueueUserACLInfo> getQueueUserAclInfo();
Resource getMinimumResourceCapability();
Resource getMaximumResourceCapability();
```

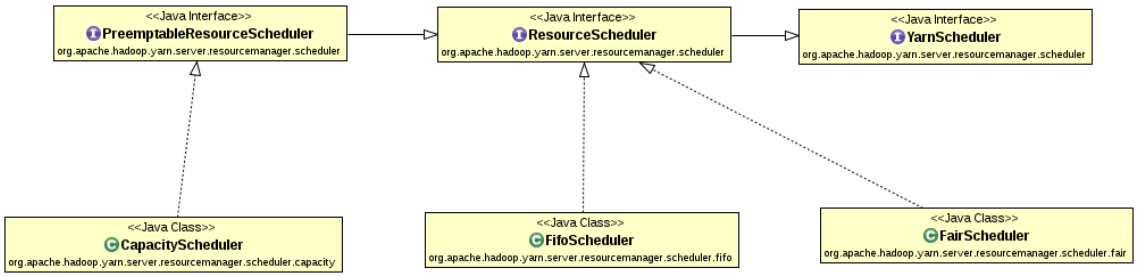


FIGURE 5.1 – Le diagra

```
int getNumClusterNodes();
SchedulerNodeReport getNodeReport(NodeId nodeId);
SchedulerAppReport getSchedulerAppInfo(ApplicationAttemptId appAttemptId);
QueueMetrics getRootQueueMetrics();
```

Ces méthodes ne sont pas importantes d'un point de vue ordonnanceur, mais permettent au reste du ResourceManager de connaître certaines informations qui seront utilisées pour monitorer le Scheduler entre autre.

Ce sont les deux méthodes suivantes qui vont être importantes. Pour commencer la fonction

```
Allocation allocate(ApplicationAttemptId appAttemptId, List<ResourceRequest> ask, List<
    ContainerId> release, List<String> blacklistAdditions, List<String>
    blacklistRemovals);
```

Cette méthode est la méthodes qui va gérer les demandes de ressources des applications. Une fois une application enregistrée auprès du Scheduler, ce-dernier pourra gérer les demandes de ressources via cette interface. Ces demandes de ressources sont stockées dans le paramètre $ask$. On va pouvoir également gérer ici les ressources allouées à une application mais qui n'en a plus l'utilité. Cela est stocké dans la variable $release$. Pour finir, on peut également gérer une blacklist lorsqu'une application rencontre un problème sur des noeuds particuliers par exemple. En plus de traiter ces demandes de l'application, elle doit également renvoyer des informations à l'application. Ces informations sont formulées via un objet $Allocation$. Cet objet va contenir les conteneurs qui sont alloués pour l'application. Il contient donc les ressources que le Scheduler a attribué à l'application. C'est dans cette méthode que se fait la communication des décisions du Scheduler. Cette méthode est en quelque sorte le heartbeat d'une application, car elle est souvent appellée et permet un échange d'informations importantes.

La seconde méthode importante est :

```
void handle(SchedulerEvent event);
```

Cette méthode n'appartiens pas réellement à la classe $YarnScheduler$ mais viens du fait que cette classe implémente un type de classe particulier pour la gestion d'évènement : $EventHandler < SchedulerEvent >$.
La méthode handle va donc gérer des évènements de type $SchedulerEvent$.
Il existe 6 types de $SchedulerEvent$ :

- $NODE\_ADDED$, l'ajout d'un noeud (une machine viens d'être ajoutée au cluster, on peut allouer ses ressources)
- $NODE\_REMOVED$, la suppression d'un noeud (une machine viens d'être enlevée du cluster)
- $NODE\_UPDATE$, la mise à jour d'un noeud (c'est le heartbeat d'un noeud)
- $APP\_ADDED$, l'ajout d'une application (soumission d'une application au ResourceManager)
- $APP\_REMOVED$, la suppression d'une application (l'application est terminée, ou obligée de s'arrêter par exemples)

— $CONTAINER\_EXPIRED$, un container viens doit se terminer (c'est un composant du ResourceManager qui peut en donner l'ordre)

Ces 6 types d'évènements rythment la vie du Scheduler, et lui permettent d'être informé de l'état du cluster et ainsi faire son travail avec les bonnes informations. Parmis ces évènements, il y en a un plus important que les autres : $NODE\_UPDATE$. C'est un noeud qui envois ses informations, comme par exemple les ressources qu'il a de disponible. Dans tous les exemples de Scheduler dans Hadoop que j'ai étudiés, c'est dans cette méthode qu'est fait l'assignation de ces ressources libres à des applications. Lors de cet évènement, on vérifie également que des container ont bien été lancés sur les noeuds.

Il reste deux méthodes que je n'ai pas encore parlé. Ce sont les méthodes :

```
void reinitialize(Configuration conf, RMContext rmContext);
public void recover(RMState state);
```

La première est la méthode d'initialisation du Scheduler. Elle fournit sa configuration via le paramètre $conf$, et un outil pour communiquer avec le reste du ResourceManager avec le paramètre $rmContext$. Cette méthode va principalement servir une fois : au démarrage du ResourceManager. Sans l'appel à cette méthode, le Scheduler n'est pas sensé pouvoir fonctionner. La seconde méthode doit servir lorsqu'il y a un problème au niveau du ResourceManager et que le Scheduler doit revenir à un état précédent. Cette méthode n'a implémentée dans aucun des Scheduler d'Hadoop. De plus, le manque de documentation sur cette fonctionnalité ne me permet pas d'être certain de son rôle.

### 5.4.2 La gestion des queues

Un Scheduler dans Hadoop doit absolument gérer au moins une queue. Une queue est un objet où les applications seront assignées. Son utilité est de gérer les utilisateurs et ainsi monitorer les ressources qu'ils utilisent afin de pouvoir éventuellement les limiter. Lorsqu'un Scheduler est implémenté dans Hadoop, il peut avoir besoin d'implémenter sa ou ses propres queues. Il faut donc implémenter l'interface $Queue$ fournie dans le package du Scheduler. Cette interface propose des méthodes d'informations sur les droits des utilisateurs dans cette queue. Elle propose également une méthode pour savoir le nom de la queue. Pour finir, elle propose une méthode afin d'avoir les métriques de cette queue. Ce sont les métriques qui vont donner les informations importantes de la queue, comme le nombre d'applications soumises à cette queue, le nombre d'utilisateurs, les limites en ressources des utilisateurs. Elle permet également de modifier ces valeurs (soumettre une application, retirer une application, limiter un utilisateur, etc...).

Il est nécessaire au Scheduler d'avoir au moins une queue car il faut absolument fournir des informations issues d'une queue. Cette queue est la queue principale, ou queue racine. Lorsqu'un Scheduler ne veut pas gérer de queue, il est tout de même obligé d'en créer une et d'y insérer toutes les applications entrantes.

### 5.4.3 La gestion des applications

Le Scheduler doit pouvoir gérer les applications qui lui sont soumises. L'interface que fournit Hadoop pour créer une application au niveau du Scheduler se nomme $SchedulerApplication$. Cette interface permet au Scheduler de stocker des informations importantes pour l'ordonnancement. En effet, elle oblige d'avoir une liste des container assignés à l'application, et une liste de container réservés à l'application. On doit également fournir le $ApplicationAttemptId$ et si l'application est en cours ou non. L'usage de cette classe peut évidemment s'étendre aux besoins du Scheduler.

Pour aider à la gestion des applications dans un Scheduler, Hadoop fournit également une classe $AppSchedulingInfo$, qui va permettre de stocker certaines données liées à l'application. Elle peut servir par exemple à garder les demandes de ressources d'une application lors d'un appel à $allocate$. Cela est très

pratique car grâce à cela, il est plus aisé de retrouver ces demandes. Il est tout de même possible de stocker les demandes manuellement.

### 5.4.4 La gestion des noeuds

Pour finir, Hadoop fournit également une classe à implémenter pour avoir sa propre gestion des noeuds. Cette classe se nomme $SchedulerNode$. Les méthodes qu'elle va demander à implémenter seront des méthodes qui fournissent des informations, comme le nom de l'hôte du noeud, le nom de sa rack, les ressources utilisées, les ressources disponibles, les ressources totales, et le nombre de containers alloués sur ce noeud. Ces informations pourront être utiles pour le Scheduler, afin de savoir si il reste des ressources à allouer sur un noeud par exemple.

## 5.5 Les objets fournis par Hadoop pour communiquer

### 5.5.1 Les liens avec le ResourceManager

Le Scheduler a certaines choses impératives à faire afin que le reste du ResourceManager soit au courant de l'activité du Scheduler. Plus particulièrement au niveau de la vie des container, qui est particulièrement régulée par le Scheduler. C'est aussi le Scheduler qui demande à créer les container. Lorsqu'il créé un container, le Scheduler demande la création de deux objet. L'un est de la classe $Container$ du package $org.apache.hadoop.yarn.api.records$, et un autre, le $RMContainer$, qui va représenter le Container au niveau du ResourceManager. C'est cet objet que l'on va devoir faire changer d'état au niveau du Scheduler. Le changement d'état va permettre à d'autres services du ResourceManager de savoir ce qu'il en est de ce container, et de réagir en conséquent. Si le Scheduler ne fait pas cela, il peut y avoir des problèmes comme par exemple ne pas lancer les container et bloquer toutes les applications. Les différents états d'un container sont définis dans la classe $RMContainerEventType$. Voici la liste quels sont les états du container que le Scheduler doit notifier :

- $START$ : le container a bien été créé. Il peut donc passer dans cet état.
- $ACQUIRED$ : l'application a été notifiée qu'elle peut utiliser ce container.
- $LAUNCHED$ : le container est bien lancé sur son noeud.
- $RELEASED$ : l'application qui utilisait ce container n'en a plus besoin.
- $FINISHED$ : le container est terminé, il n'est plus utile.
- $KILL$ : dans le cas où le container doit s'arrêter (quand l'application ou le noeud disparait), le Scheduler peut demander la fin du container.

Afin de modifier l'état d'un container, le Scheduler utilise la fonction $handle$ du $RMContainer$, qui va gérer un évènement de la classe $RMContainerEvent$, auquel on y donnera un des types donnés plus haut.

Un des objets utiles en lien avec le ResourceManager est la classe $BuilderUtils$. Cette classe commune à tout le ResourceManager permet de créer des objets utilisés dans tout le ResourceManager, comme par exemple un container. Elle contient des méthodes statiques qui permettent de faire cela.

Le Scheduler doit également communiquer d'autres informations au ResourceManager, pour cela on utiliser un objet de la classe $RMContext$. On va pouvoir envoyer différents évènements au reste du ResourceManager via un gestionnaire d'évènement qui transmettra ces évènements aux service intéressés. C'est par ce système que les évènements arrivent à la fonction $handle$ du Scheduler. Dans ce que j'ai pu rencontrer, le scheduler doit s'adresser au $RMContext$ lorsque :

- Un container a été créé, mais il n'est pas enregistré dans les données du Scheduler. A ce moment la, le Scheduler peur demander la destruction du container.

— Lorsqu'une application a été soumise au Scheduler via un évènement $APP\_ADDED$, le Scheduler doit obligatoirement signaler qu'il a bien géré l'application avec un évènement de la classe $RMAppAttemptEvent$ et de type $RMAppAttemptEventType.APP\_ACCEPTED$.
— Lorsqu'une application a été rejetée par le Scheduler, on envoir un évènement de la classe $RMAppAttemptRejectedEvent$.

Le $RMContext$ nous donne aussi divers méthodes pour des objets utiles, comme par exemple lors de la création d'un container, il est obligatoire de générer un Token, ce qui est possible grâce à un service accédé via le $RMContext$.

### 5.5.2 Les liens entre les objets du scheduler, les objets du Ressource Manager et les instances sur le cluster

Nous l'avons vu précédemment, le Scheduler a ses propres classes pour représenter un noeud et une application ($SchedulerApplication$ et $SchedulerNode$. Mais le ResourceManager a également des classes pour représenter un noeud et une application ($RMNode$ et $RMApp$). Bien qu'ils représentent les même entités, ces classes ont des objectifs et des utilisations différentes. Les classes du ResourceManager sont utilisées dans tous les services de celui-ci, et fournissent ainsi des services généraux, comme par exemple leur ID unique. Les classes du Scheduler quant à elles vont avoir un lien avec les classes du ResourceManager en ayant l'objet associé en attribut de la classe. Le fait d'avoir un ID unique permet d'être sûr d'avoir le bon objet et donc d'avoir les bons liens.

## 5.6 L'installation d'un environnement de développement Hadoop

J'ai installé un environnement de développement sur une distribution Debian, car mes connaissances pour cette distribution était supérieurs à celle de CentOS. Cela ne portait pas préjudice au résultat puisque la compilation donnait des fichiers exécutables sur tous les système UNIX. J'ai donc installé Java 6 et Eclipse sur la machine, et téléchargé les sources d'Hadoop. Afin de pouvoir utiliser les sources dans Eclipse, j'ai du suivre plusieurs étapes.

Premièrement, j'ai dû installer plusieurs requis pour compiler Hadoop :
— Maven (version 3+)
— Google ProtocolBuffers (version 2.5.0)
— CMake (version 2.6)

J'ai ensuite compilé une première fois Hadoop avec la commande :

```
$ mvn package -Pdist -DskipTests -Dtar
```

Une fois compilé, j'ai exécuté la commande suivante afin de créer les projets Eclipse de tous les projets d'Hadoop.

```
$ mvn eclipse:eclipse -DskipTests
```

Pour finir, il faut importer les projets dans Eclipse. Pour compiler, il faut tout de même utiliser la ligne de commande.

# Implémentation d'un ordonnanceur dans Hadoop : exemple du FIFO

La suite de mes recherche m'a donc mené à implémenter un Scheduler pour Hadoop. Le choix d'implémenter un Scheduler de type FIFO s'est car il ne me restait plus beaucoup de temps, et qu'il était préférable d'avoir un Scheduler fonctionnel plutôt qu'un qui ne soit pas terminé. En effet, cette implémentation s'est faite dans une optique de pédagogie pour répondre à la question

## 6.1 Environnement de développement et initialisation du projet

Nous avons vu comment installer l'environnement de développement d'Hadoop dans la partie 5.6. Le projet contenant toutes les classes du Scheduler dans le ResourceManager se nomme hadoop-yarn-server-resourcemanager. Dans ce projet se trouve des fichiers de tests et des fichiers sources. Les fichiers sources sont répartis dans des packages. Les classes dont nous avons parlé précédemment sont situées dans le package $org.apache.hadoop.yarn.server.resourcemanager.scheduler$. Les Scheduler qui ont été implémentés dans Hadoop ont pour packages :

— FIFO : $org.apache.hadoop.yarn.server.resourcemanager.scheduler.fifo$
— Capacity : $org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity$
— Fair : $org.apache.hadoop.yarn.server.resourcemanager.scheduler$

J'ai gardé le même principe pour mon Scheduler, que j'ai appelé $PFEFIFOScheduler$. J'ai donc créé le package $org.apache.hadoop.yarn.server.resourcemanager.scheduler.pfefifo$.

L'objectif de cet ordonnanceur est de n'utiliser que des classes fournies par Hadoop au niveau du Scheduler, et pas de classes implémentées pour les autres Scheduler.

J'ai donc codé quatre classes : $PFEFIFOScheduler$, $PFEFIFOApplication$, $PFEFIFONode$ et $PFEFIFOQueue$.

## 6.2 Le diagramme de classe

Le diagramme 6.1 montre ce qui a été développé, avec le détail des méthodes et des attributs. Le diagramme 6.2 montre les liens entre les classes de PFEFIFOScheduler et les classes du package Scheduler. On voit donc quelles classes sont étendues.

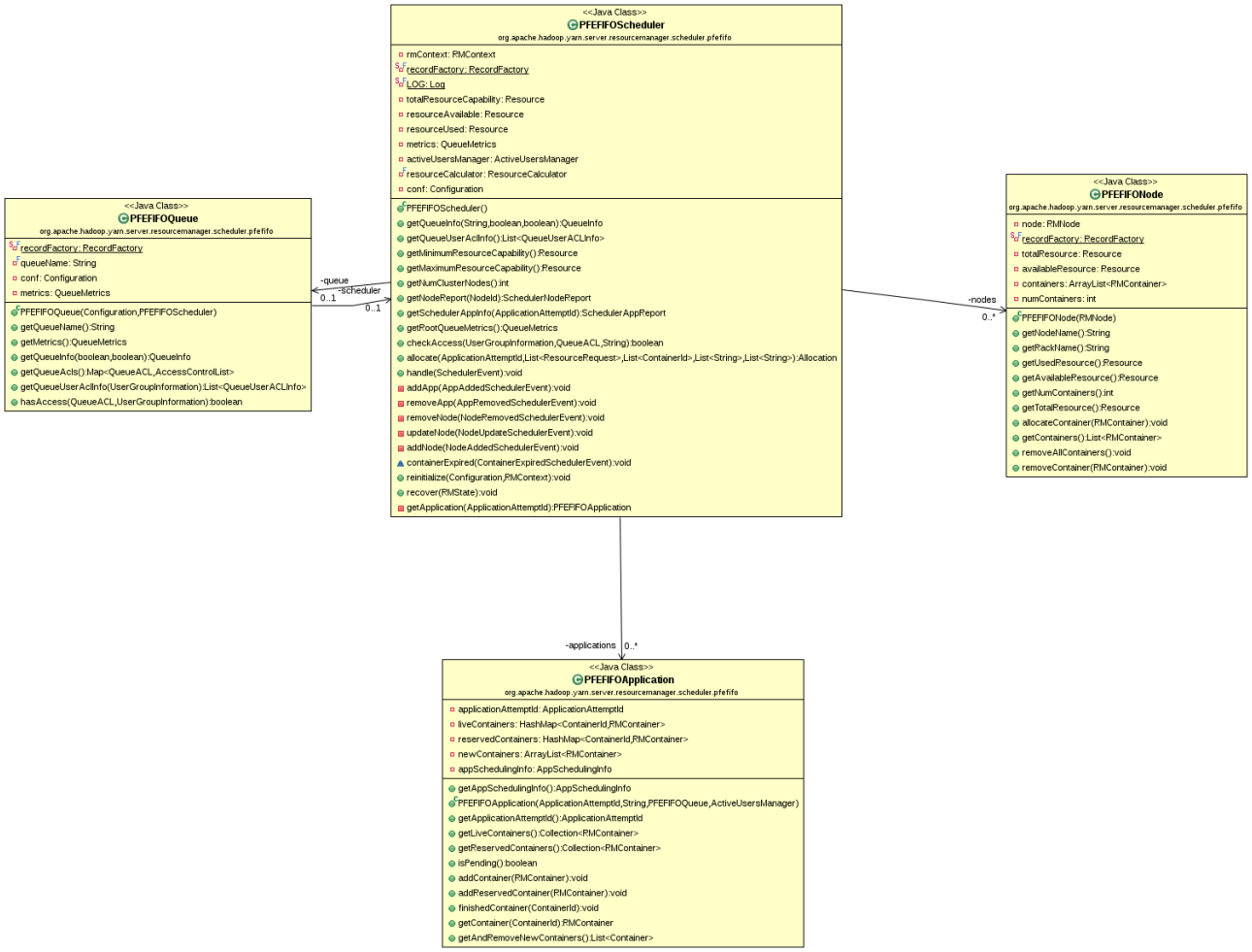


FIGURE 6.1 – Diagramme de classe du PFEFIFOScheduler

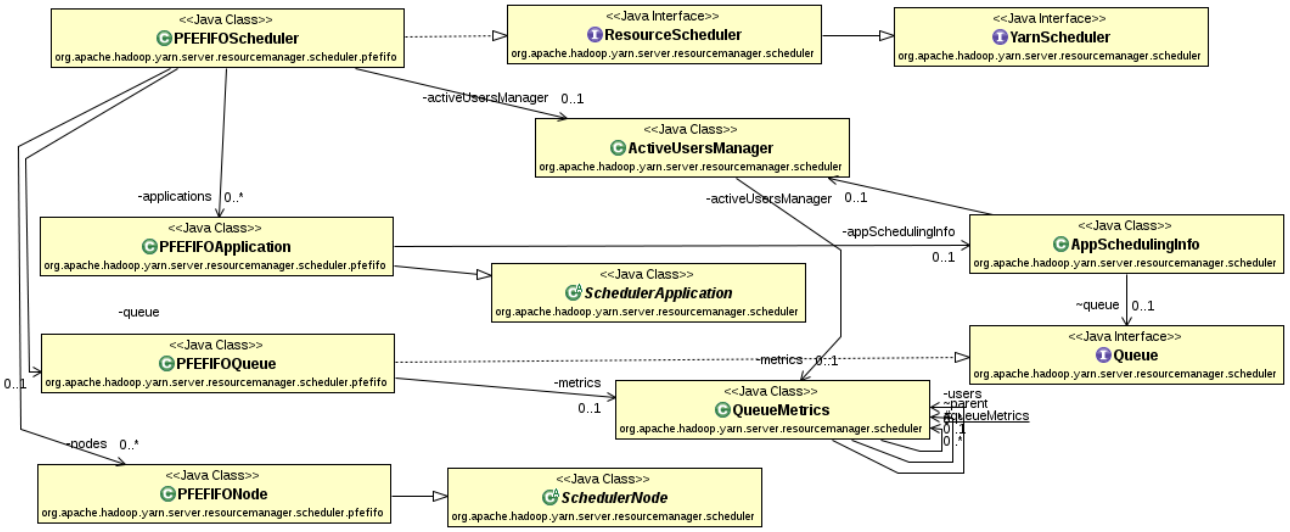


FIGURE 6.2 – Diagramme de classe du PFEFIFOScheduler et des classes du package scheduler d'Hadoop

## 6.3 La classe principale

La classe principale s'appelle $PFEFIFOScheduler$. Elle implémente l'interface $ResourceScheduler$.

### 6.3.1 Les attributs de la classes

Pour que la classe puisse avoir un bon fonctionnement, il était nécessaire d'avoir certains attributs.

**rmContext**

```
RMContext rmContext;
```

C'est une variable de la classe $RMContext$, qui sert à faire le lien avec le reste du ResourceManager. Elle a été traitée dans les sections 5.4.1 et 5.5.1.

**recordFactory**

```
RecordFactory recordFactory;
```

Cet objet va permettre de construire des objet de la classe $Resource$.

**LOG**

```
Log LOG;
```

C'est l'objet qui va permettre d'utiliser le LOG afin de fournir des informations sur le fonctionnement du Scheduler.

**applications**

```
LinkedHashMap<ApplicationAttemptId, PFEFIFOApplication> applications;
```

C'est la liste des applications que le Scheduler traite. Le choix du type LinkedHashMap viens du fait que cela permet d'avoir une liste FIFO des applications.

**nodes**

```
HashMap<NodeId, PFEFIFONode> nodes;
```

C'est la liste des noeuds que le Scheduler traite.

**queue**

```
PFEFIFOQueue queue;
```

C'est la queue principale qui va contenir toutes les applications. La queue va principalement servir à enregistrer les informations sur toutes les applications et les utilisateurs, et va donner les métriques.

**totalResourceCapability**

```
Resource totalResourceCapability;
```

C'est la capacité totale du cluster en terme de ressources. Il correspond à la somme des capacités des noeuds.

**resourceAvailable**

```
Resource resourceAvailable;
```

Ce sont les ressources libres du cluster, ce qui peut encore être ordonnancé.

**resourceUsed**

```
Resource resourceUsed;
```

Ce sont les ressources utilisés du cluster.

**minimumResourceCapability**

```
Resource minimumResourceCapability;
```

C'est la taille minimum d'un container que l'on peut allouer sur un noeud. Cette variable viens de la configuration que l'administrateur a fournis, et est issue de la variable $conf$ fourni dans la méthode $reinitialize$.

**maximumResourceCapability**

```
Resource maximumResourceCapability;
```

C'est la taille maximum d'un container. C'est la même chose que $minimumResourceCapability$, il viens de la configuration d'Hadoop.

**activeUsersManager**

```
ActiveUsersManager activeUsersManager;
```

C'est un objet nécessaire dans les objets $PFEFIFOApplication$.

**resourceCalculator**

```
ResourceCalculator resourceCalculator;
```

C'est un objet qui va permettre de faire des opérations sur les Resources (additionner, soustraire, comparer).

**conf**

```
Configuration conf;
```

C'est la configuration du Scheduler, qui va donner les informations comme $minimumResourceCapability$ et $minimumResourceCapability$, il est donné comme paramètre dans la méthode $reinitialize$.

### 6.3.2 Implémenter les méthodes nécessaires

Implémenter la classe $ResourceScheduler$ donne obligation d'implémenter toutes les méthodes abstraites. J'ai donc codé toutes ces méthodes.

**getQueueInfo**

```
QueueInfo getQueueInfo(String queueName, boolean includeChildQueues, boolean recursive)
```

Cette méthode va renvoyer les informations contenues dans la variable $queue$.

**getQueueUserAclInfo**

```
List<QueueUserACLInfo> getQueueUserAclInfo();
```

Cette méthode est censée renvoyer des informations de type $QueueUserACLInfo$, seulement ce n'est pas géré dans mon Scheduler. Les ACL sont un système d'autorisations des utilisateurs, ce qui n'est pas utile pour ce Scheduler. La méthode renvoie donc $null$, ce qui ne pose pas de problèmes.

**getMinimumResourceCapability**

```
Resource getMinimumResourceCapability();
```

Cette méthode renvoie la variable $minimumResourceCapability$

**getMaximumResourceCapability**

```
Resource getMaximumResourceCapability();
```

Cette méthode renvoie la variable $maximumResourceCapability$

### getNumClusterNodes

```
int getNumClusterNodes();
```

Cette méthode renvoie le nombre de noeuds du cluster, c'est donc la taille de la liste $nodes$.

### getNodeReport

```
SchedulerNodeReport getNodeReport(NodeId nodeId);
```

Cette méthode demande des informations sur un noeud précis. On construit un objet $SchedulerNodeReport$ à partir du $PFEFIFONode$ associé au nodeId donné en paramètre.

### getSchedulerAppInfo

```
SchedulerAppReport getSchedulerAppInfo(ApplicationAttemptId appAttemptId);
```

Cette méthode demande des informations sur une application précise. On construit un objet $SchedulerAppReport$ à partir du $PFEFIFOApplication$ issu du appAttemptId passé en paramètre.

### getRootQueueMetrics

```
QueueMetrics getRootQueueMetrics();
```

Cette méthode demande les métriques de la queue principale.

### allocate

```
Allocation allocate(ApplicationAttemptId appAttemptId, List<ResourceRequest> ask, List<
    ContainerId> release, List<String> blacklistAdditions, List<String>
    blacklistRemovals);
```

C'est une méthode importante qui va prendre en compte les demandes de l'application dont l'ID est appAttemptId. Elle se déroule de la façon suivante. Premièrement on récupère le PFEFIFOApplication associé au appAttemptId. Puis, on libère les container de la liste du paramètre release (au niveau du PFEFIFONode et du PFEFIFOApplication). On stocke ensuite les demandes du paramètre ask dans le PFEFIFOApplication. Pour finir, on renvoie à l'application les container qui lui ont été alloués et dont elle n'a pas connaissance. Cette liste est contenue dans l'objet PFEFIFOApplication.

### handle

```
void handle(SchedulerEvent event);
```

Nous avons vu précédemment que cette fonction gère 6 types d'évènements. Pour chaque évènement, j'ai créé une fonction qui va gérer cet évènement. Les fonctions d'ajout de noeud ou d'application vont créer les objets associés (PFEFIFONode et PFEFIFOApplication respectivement), et les ajouter à leur liste (node et applications respectivement). Pour l'application, le Scheduler doit signaler qu'il a bien pris en compte l'application (voir 5.5.1. Les fonctions de suppression de noeud ou d'application vont eux supprimer les objets de leurs listes, en supprimant tous les container qui tournait dessus ou qui leur était attribué respectivement. La méthode gérant la fin d'un container va demander la suppression à son noeud associé et à son application associée. Pour finir, il reste la gestion de $NODE_UPDATE$, qui gère le heartbeat d'un noeud. Cette méthode doit effectuer trois traitements. Premièrement, cet évènement permet de connaître les container qui sont terminés sur le noeud. On va donc relâcher ses container au niveau du noeud et de l'application qui leur est associée. Ensuite, cet évènement permet de connaître les container qui viennent d'être lancés sur le noeud. On va donc devoir passer leur état à $LAUNCHED$. Pour finir, on

va pouvoir attribuer les ressources disponibles du noeud. C'est donc ici que se prennent les décisions. On va donc lire la liste $applications$, qui va être dans le bon ordre (FIFO). On va ensuite demander au $AppSchedulingInfo$ de l'application sélectionnée les demandes de ressource de l'applications. On va ensuite parcourir les demandes de ressources par priorités (rappels au tableau 5.1). On va ensuite traiter les demandes une par une, en lui accordant container par container jusqu'à ce qu'il n'y ait plus de ressources disponible, ou qu'il n'y ait plus de container à attribuer. On passe alors à la requête suivante, jusqu'à ce qu'on ait utilisé tout le noeud, ou que l'on ait plus de requêtes à traiter au niveau de cette application. On passe alors à l'application suivante, et ainsi de suite. Par ce système, si une application ne se voit pas attribuer de container (parcequ'elle a une demande en ressources trop élevé par exemple), le Scheduler va quand même attribuer les ressources restantes aux autres applications.

### 6.3.3 Établir les règles d'ordonnancement

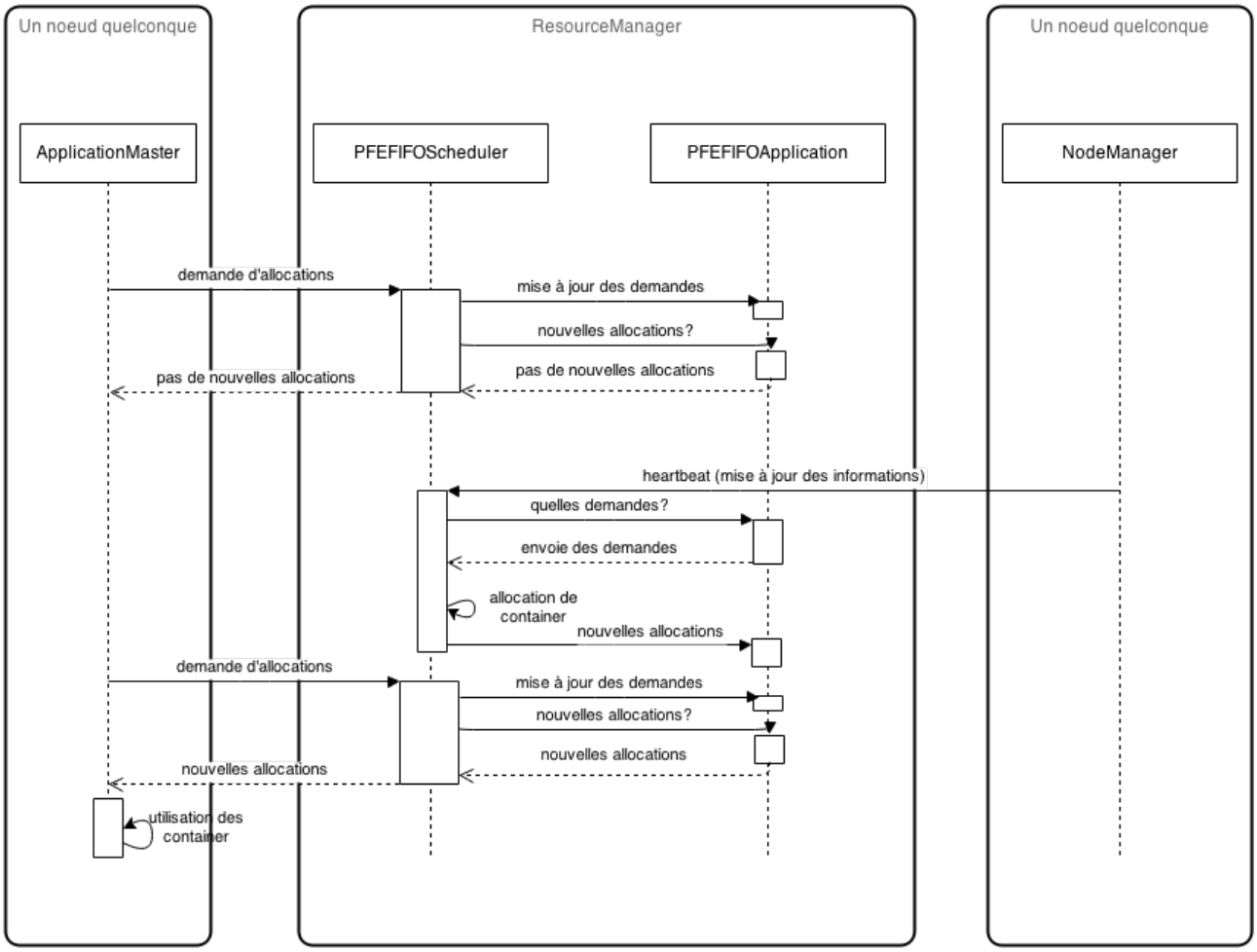


FIGURE 6.3 – Diagramme de séquence de l'assignation des ressources

Le diagramme 6.3 présente un exemple simple de l'allocation des ressource du cluster. Il présente une application, représentée par son ApplicationMaster sur un certain noeud, qui va faire ses demandes de ressources via un allocate, et en retour de cette fonction lui sera passé des container. Lors de son premier allocate, le Scheduler ne lui renvoie pas de container, car il ne lui en a pas encore attribué. L'attribution se fait lorsqu'un noeud, via son NodeManager, va faire un heartbeat, qui va déclencher l'évènement $NODE_UPDATE$ au niveau du Scheduler. En effet, les ressources libres dans ce noeud seront

attribuée aux applications. Dans l'exemple, des containers sont créés et attribués à l'application. C'est l'objet PFEFIFOApplication qui garde en mémoire ces attributions, l'ApplicationMaster n'est pas encore au courant de ces container. Il le sera lorsqu'il fera à nouveau appel à allocate avec ses nouvelles demandes. Il récupère en retour de cette fonction les container qui lui sont alloués.

### 6.3.4 Répondre à l'objectif d'ordonnancement

Le Scheduler gère donc les ressources lorsqu'elles deviennent disponibles, c'est-à-dire quand le noeud fait un heartbeat. Lorsqu'un noeud viens d'être ajouté, on attend tout de même qu'il fasse un heartbeat afin d'attribuer ses ressources à une ou des applications. Ensuite, une fois qu'un container est terminé, cette information est remontée via le heartbeat et le Scheduler peut de nouveau attribuer les ressources du cluster.

Mon implémentation de Scheduler utilise un ordonnancement de type FIFO, grâce à la liste $applications$ de type $LinkedHashMap$, qui joue le rôle de file FIFO. Les applications sont ajoutées dans la file lors de l'évènement $APP_ADDED$, et supprimée de la file lors de l'évènement $APP_REMOVED$. L'attribution se fait en prenant les applications les unes après les autres jusqu'à ce que toutes les ressources soient attribuées ou qu'on a parcouru toutes les applications.

## 6.4 Les classes secondaires

### 6.4.1 La classe pour les applications (PFEFIFOApplication)

Cette classe va être utile pour garder en mémoire les container déjà alloués à l'application. Elle va avoir la liste des container nouvellement alloués, et la liste des container. Lorsque le Scheduler créé un container pour l'application, il va appeler la méthode $AddContainer$ qui va ajouter le container dans les deux listes gérées par la classe. Ainsi, dans la fonction $allocate$, lorsque le Scheduler a besoin de la liste des nouveaux container alloués pour l'application, et appelle donc la fonction $getAndRemoveNewContainers$. Cette fonction va retourner une copie de la liste des nouveaux container de l'application, et va la vider afin de ne donner qu'une seule fois chaque container. Également, on signal à chaque container qu'il change d'état, il passe à $ACQUIRED$.

Elle va permettre également de stocker les demandes de ressources envoyés par l'ApplicationMaster par la méthode $allocate$. Ces demandes seront stockées dans un objet de la classe $AppSchedulingInfo$. Cet objet, initialisé lors du constructeur du PFEFIFOApplication, va permettre de stocker et gérer les demandes de l'ApplicationMaster. Ainsi, on pourra récupérer les besoins de l'application facilement lorsqu'il s'agira d'attribuer les ressources aux applications.

### 6.4.2 La classe pour la queue

Le scheduler a besoin d'un objet de la classe $Queue$ pour fournir des informations dans certaines méthodes obligatoire. Dans notre cas, cette objet sert seulement à y placer toutes les applications afin de générer des informations intéressantes sous forme de métriques. La classe $Queue$ est une interface qu'il faut implémenter, je l'ai donc fait avec la classe $PFEFIFOQueue$. Cette classe ne sera utilisé que sous une seule instance dans le Scheduler. Elle contient quelques méthodes obligatoire également informatives. J'ai instancié un objet de la classe $QueueMetrics$, qui va stocker de façon formelle les informations que l'on va lui transmettre. C'est le Scheduler qui va lui donner. Ces informations sont l'ajout d'une application dans la queue, l'assignation d'un container à une application, la fin d'un container et la fin d'une application.

### 6.4.3 La classe pour les noeuds

Cette classe va permettre de gérer les ressources d'un noeud pour le Scheduler. Il va donc enregistrer les ressources totales, les ressources disponibles et les ressources utilisées. Il va également enregistrer les container qui ont été alloué sur ce noeud.

## 6.5 Les tests unitaires

Afin de garantir le fonctionnement de toutes mes fonctions, j'ai écrit des tests unitaires. Ces tests ont permis de vérifier que mes fonctions d'ajout, de suppression d'applications fonctionnent bien. Ils ont également permis de vérifier l'allocation des ressources à un noeud, puis la libération de ces ressources. A un autre niveau, un test m'a permis de vérifier que les applications sont bien traitées en FIFO.

## 6.6 Les tests d'intégration : le déploiement du Scheduler

### 6.6.1 Le déploiement

Une fois Hadoop compilé avec mon Scheduler, il fallait l'installer et le configurer pour qu'il utilise le bon Scheduler. L'installation d'Hadoop a été vu à la section 3.5. Il faut néanmoins ajouter une option dans le fichier yarn-site.xml dans le dossier de configuration :

```
<property>
  <name>yarn.resourcemanager.scheduler.class</name>
  <value>org.apache.hadoop.yarn.server.resourcemanager.scheduler.pfefifo.
      PFEFIFOScheduler</value>
</property>
```

Une fois lancé, j'ai pu vérifier le bon fonctionnement dans un cas réel de mon ordonnanceur en lançant plusieurs applications sur Hadoop. J'ai pu observer qu'on leur attribuait bien des ressources, et qu'elle se finissaient sans problème. De plus, j'ai pu lire les différents logs que j'ai mis dans mon implémentations. Les fichiers de logs se situent dans le répertoire d'installation d'Hadoop, dans le dossier logs.

## 6.7 Les difficultés rencontrées

Au cours de ce développement, je me suis buté à un problème qui est le manque de documentations sur le rôle précis du Scheduler dans Hadoop. Je devais donc à chaque étape de mon travail d'implémentation, vérifier si je n'oubliais pas certaines notions obligatoires, comme par exemple le changement d'état d'un container. Je pense que cette difficulté rencontré met d'autant en valeur mon travail d'analyse, car ces notions ne sont pas documentées, mais sont nécessaires à un Scheduler.

## 6.8 Les problèmes restants

Je n'ai pas eu le temps de tester plus profondément mon Scheduler. Je n'ai notemment pas vérifier si l'accès au données de façon concurrente ne posait pas de problème. En effet, au cours du développement, j'ai fait attention à ce que les données importantes de ma classe principale ne puisse pas être accédées par deux threads en même temps grâce au mot-clé synchronized de Java. Je n'ai par contre pas prêté assez d'attention aux accès concurrents et je donc il est possible que dans certains cas, des données lues dans une méthodes ne soient pas viable. Il faut donc penser, ce que je n'ai pas fait, dès le début de l'implémentation, à ce genre de problématiques.

# Conclusion

## 7.1 Bilan personnel

Ce projet a été très inspirant pour moi. Il m'a permis d'agrandir mes connaissances au niveau du Big Data et du monde de l'Open Source. J'ai également été sensibilisé aux problématiques liées au Big Data, que ce soit la nécessité d'un tel framework comme Hadoop, la difficulté à appréhender un tel framework, ou les soucis d'optimisation de ce dernier, comme par exemple au niveau de l'ordonnancement. Ces problématiques sont fortement liées au monde de l'entreprise, particulièrement l'entreprise Cyrès.

Ce projet m'a aussi permis de mettre en pratique mes connaissances en gestion de projet et m'a permis d'avoir de nouvelles compétences dans ce domaine, notamment avec le cahier de spécification. Il m'a également permis d'avoir de plus de compétences en Java, et dans l'utilisation de l'IDE Eclipse, grâce à la lecture du code pour comprendre le fonctionnement des ordonnanceurs d'Hadoop, ou de l'implémentation de mon propre scheduler.

## 7.2 Remerciements

Je tiens à remercier mon encadrant, pour avoir proposé ce sujet de projet et pour m'avoir guidé et conseillé durant ce dernier.

J'aimerais également remercier mes camarades à l'école, plus particulièrement Lucas Delcroix et Simon Kesteloot pour m'avoir beaucoup aidé lorsque j'en avais le besoin

Pour terminer, j'aimerais remercier l'entreprise Cyrès de m'avoir accueilli dans le cadre du projet, et de m'avoir accepté en stage de fin d'études.

## 7.3 Suite du projet

Ce projet a permis de poser les bases afin d'implémeter un nouvel ordonnanceur dans Hadoop. Un projet futur pourra donc s'axer sur des problèmes plus précis liés à l'ordonnancement. Ce rapport permettre donc de mieux comprendre comment fonctionne Hadoop, et mes sources permettront de poser une base pour un autre Scheduler.

# Bibliographie


[1] Jimmy Wong "Which Big Data Company has the World's Biggest Hadoop Cluster ?" http://www.hadoopwizard.com/which-big-data-company-has-the-worlds-biggest-hadoop-cluster/ 20 Janvier 2013

[2] Tom White "Hadoop The Definitive Guide" O'Reilly

[3] http://tecadmin.net/set-up-hadoop-multi-node-cluster-on-centos-6

[4] "Survey on Improved Scheduling in Hadoop MapReduce in Cloud Environments" B.Thirumala Rao Dr. L.S.S.Reddy

[5] Vinod Kumar Vavilapalli http://hortonworks.com/blog/apache-hadoop-yarn-resourcemanager/ 31 Août 2012

[6] https://developer.yahoo.com/blogs/hadoop/next-generation-apache-hadoop-mapreduce-scheduler-4141.html

[7] https://developer.yahoo.com/hadoop/tutorial/module4.html

# La javadoc

## A.1 La classe PFEFIFOScheduler

org.apache.hadoop.yarn.server.resourcemanager.scheduler.pfefifo

## Class PFEFIFOScheduler

java.lang.Object
  org.apache.hadoop.yarn.server.resourcemanager.scheduler.pfefifo.PFEFIFOScheduler

**All Implemented Interfaces:**

org.apache.hadoop.yarn.event.EventHandler<SchedulerEvent>, Recoverable, ResourceScheduler, YarnScheduler

---

```
@InterfaceAudience.LimitedPrivate(value="yarn")
@InterfaceStability.Evolving
public class PFEFIFOScheduler
extends Object
implements ResourceScheduler
```

Cette classe est la classe principale du PFEFIFOScheduler. Elle va gérer la vie du Scheduler au travers des divers évènement qu'il a à traiter. Elle s'occupe de l'ordonnancement des ressources.

**Author:**

Erwann Cloarec

### Constructor Summary

Constructors

| Constructor and Description |
|---|
| PFEFIFOScheduler() |

### Method Summary

Methods

| Modifier and Type | Method and Description |
|---|---|
| Allocation | allocate(org.apache.hadoop.yarn.api.records.ApplicationAttemptId appAttemptId, List<org.apache.hadoop.yarn.api.records.ResourceRequest> ask, List<org.apache.hadoop.yarn.api.records.ContainerId> release, List<String> blacklistAdditions, List<String> blacklistRemovals)<br>The main api between the ApplicationMaster and the Scheduler. |
| boolean | checkAccess(org.apache.hadoop.security.UserGroupInformation callerUGI, org.apache.hadoop.yarn.api.records.QueueACL acl, String queueName)<br>Check if the user has permission to perform the operation. |
| org.apache.hadoop.yarn.api.records.Resource | getMaximumResourceCapability()<br>Get maximum allocatable Resource. |
| org.apache.hadoop.yarn.api.records.Resource | getMinimumResourceCapability()<br>Get minimum allocatable Resource. |
| SchedulerNodeReport | getNodeReport(org.apache.hadoop.yarn.api.records.NodeId nodeId)<br>Get node resource usage report. |
| int | getNumClusterNodes()<br>Get the number of nodes available in the cluster. |
| org.apache.hadoop.yarn.api.records.QueueInfo | getQueueInfo(String queueName, boolean includeChildQueues, boolean recursive)<br>Get queue information |
| List<org.apache.hadoop.yarn.api.records.QueueUserACLInfo> | getQueueUserAclInfo()<br>Get acls for queues for current user. |
| QueueMetrics | getRootQueueMetrics()<br>Get the root queue for the scheduler. |
| SchedulerAppReport | getSchedulerAppInfo(org.apache.hadoop.yarn.api.records.ApplicationAttemptId appAttemptId)<br>Get the Scheduler app for a given app attempt Id. |
| void | handle(SchedulerEvent event)<br>Cette fonction est le point d'entrée de différents évènements. |
| void | recover(RMStateStore.RMState state)<br>Cette méthode permet probablement de récupérer d'un certain état lorsque le ResourceManager a eu un problème. |
| void | reinitialize(org.apache.hadoop.conf.Configuration conf, RMContext rmContext)<br>Re-initialize the ResourceScheduler. |

### Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

### Constructor Detail

**PFEFIFOScheduler**

```
public PFEFIFOScheduler()
```

## Method Detail

### getQueueInfo

```
public org.apache.hadoop.yarn.api.records.QueueInfo getQueueInfo(String queueName,
                                                                 boolean includeChildQueues,
                                                                 boolean recursive)
```

**Description copied from interface: YarnScheduler**
Get queue information

**Specified by:**

getQueueInfo in interface YarnScheduler

**Parameters:**

queueName - queue name

includeChildQueues - include child queues?

recursive - get children queues?

**Returns:**

queue information

### getQueueUserAclInfo

```
public List<org.apache.hadoop.yarn.api.records.QueueUserACLInfo> getQueueUserAclInfo()
```

**Description copied from interface: YarnScheduler**
Get acls for queues for current user.

**Specified by:**

getQueueUserAclInfo in interface YarnScheduler

**Returns:**

acls for queues for current user

### getMinimumResourceCapability

```
public org.apache.hadoop.yarn.api.records.Resource getMinimumResourceCapability()
```

**Description copied from interface: YarnScheduler**
Get minimum allocatable Resource.

**Specified by:**

getMinimumResourceCapability in interface YarnScheduler

**Returns:**

minimum allocatable resource

### getMaximumResourceCapability

```
public org.apache.hadoop.yarn.api.records.Resource getMaximumResourceCapability()
```

**Description copied from interface: YarnScheduler**
Get maximum allocatable Resource.

**Specified by:**

getMaximumResourceCapability in interface YarnScheduler

**Returns:**

maximum allocatable resource

### getNumClusterNodes

```
public int getNumClusterNodes()
```

**Description copied from interface: YarnScheduler**
Get the number of nodes available in the cluster.

**Specified by:**

getNumClusterNodes in interface YarnScheduler

**Returns:**

the number of available nodes.

### getNodeReport

```
public SchedulerNodeReport getNodeReport(org.apache.hadoop.yarn.api.records.NodeId nodeId)
```

**Description copied from interface: YarnScheduler**
Get node resource usage report.

**Specified by:**

getNodeReport in interface YarnScheduler

**Returns:**

the SchedulerNodeReport for the node or null if nodeId does not point to a defined node.

### getSchedulerAppInfo

```
public SchedulerAppReport getSchedulerAppInfo(org.apache.hadoop.yarn.api.records.ApplicationAttemptId appAttemptId)
```

Get the Scheduler app for a given app attempt Id. On va construire un SchedulerAppReport à partir de l'ApplicationAttemptId donné, si on l'a d'enregistré dans notre liste.

**Specified by:**

getSchedulerAppInfo in interface YarnScheduler

**Parameters:**

appAttemptId - the id of the application attempt

**Returns:**

SchedulerApp for this given attempt.

### getRootQueueMetrics

```
public QueueMetrics getRootQueueMetrics()
```

**Description copied from interface: YarnScheduler**
Get the root queue for the scheduler.

**Specified by:**

getRootQueueMetrics in interface YarnScheduler

**Returns:**

the root queue for the scheduler.

### checkAccess

```
public boolean checkAccess(org.apache.hadoop.security.UserGroupInformation callerUGI,
                  org.apache.hadoop.yarn.api.records.QueueACL acl,
                  String queueName)
```

**Description copied from interface: YarnScheduler**
Check if the user has permission to perform the operation. If the user has QueueACL.ADMINISTER_QUEUE permission, this user can view/modify the applications in this queue

**Specified by:**

checkAccess in interface YarnScheduler

**Returns:**

true if the user has the permission, false otherwise

### allocate

```
public Allocation allocate(org.apache.hadoop.yarn.api.records.ApplicationAttemptId appAttemptId,
                  List<org.apache.hadoop.yarn.api.records.ResourceRequest> ask,
                  List<org.apache.hadoop.yarn.api.records.ContainerId> release,
                  List<String> blacklistAdditions,
                  List<String> blacklistRemovals)
```

The main api between the ApplicationMaster and the Scheduler. The ApplicationMaster is updating his future resource requirements and may release containers he doens't need.

Cette méthode est une des plus importante du Scheduler.

Elle est appellée lorsqu'un AppplicationMaster fait ses demandes de ressources.

Les demandes de ressource sont donc intégrées au PFEFIFOApplication associé à l'ApplicationAttemptId.

Les container terminés sont terminés et enlevés de leur PFEFIFONode.

On retourne les ressources allouées récemment par le Scheduler.

**Specified by:**

allocate in interface YarnScheduler

**Returns:**

the Allocation for the application

### handle

```
public void handle(SchedulerEvent event)
```

Cette fonction est le point d'entrée de différents évènements. Il va gérer:

- Ajout d'une application
- Suppression d'un application
- Ajout d'un noeud
- Suppression d'un noeud
- Mise à jour d'un noeud
- L'expiration d'un container

**Specified by:**

handle in interface org.apache.hadoop.yarn.event.EventHandler<SchedulerEvent>

**Parameters:**

event -

### reinitialize

```
public void reinitialize(org.apache.hadoop.conf.Configuration conf,
                RMContext rmContext)
```

Re-initialize the ResourceScheduler. Cette méthode est appelée pour initialiser le scheduler. Il faut donc initialiser les composants du Scheduler ayant besoin de la configuration du Scheduler.

**Specified by:**

reinitialize in interface ResourceScheduler

**Parameters:**

conf - La configuration du Scheduler

rmContext - Le contexte du Scheduler, qui permet de communiquer avec le reste du ResourceManager.

### recover

```
public void recover(RMStateStore.RMState state)
             throws Exception
```

Cette méthode permet probablement de récupérer d'un certain état lorsque le ResourceManager a eu un problème.

**Specified by:**

recover in interface Recoverable

**Throws:**

Exception

## A.2 La classe PFEFIFOApplication

org.apache.hadoop.yarn.server.resourcemanager.scheduler.pfefifo

## Class PFEFIFOApplication

java.lang.Object
org.apache.hadoop.yarn.server.resourcemanager.scheduler.SchedulerApplication
org.apache.hadoop.yarn.server.resourcemanager.scheduler.pfefifo.PFEFIFOApplication

---

```
public class PFEFIFOApplication
extends SchedulerApplication
```

Cette classe va représenter une application au niveau du Scheduler. Elle va enregistrer les informations nécessaires à l'ordonnancement des ressources pour celle-ci. Elle va donc garder:

- les requêtes de ressources que son ApplicationMaster a fait
- les containers que le scheduler lui alloue
- les containers en cours
- les containers réservés (pas utilisé par le PFEFIFOScheduler)

**Author:**

Erwann Cloarec

### Constructor Summary

Constructors

| Constructor and Description |
|---|
| **PFEFIFOApplication**(org.apache.hadoop.yarn.api.records.ApplicationAttemptId applicationAttemptId, **String** user, **PFEFIFOQueue** queue, **ActiveUsersManager** activeUserManager)<br>C'est le constructeur de la classe. |

### Method Summary

Methods

| Modifier and Type | Method and Description |
|---|---|
| void | **addContainer**(**RMContainer** container)<br>Un container vient d'être alloué pour l'application. |
| void | **addReservedContainer**(**RMContainer** container)<br>Cette méthode est utile lorsque le Scheduler gère la réservation d'un container. |
| void | **finishedContainer**(org.apache.hadoop.yarn.api.records.ContainerId containerId)<br>L'application n'a plus besoin du container, on va donc l'enlever des container en cours. |
| **List**<org.apache.hadoop.yarn.api.records.Container> | **getAndRemoveNewContainers**()<br>Cette méthode est appellée lorsqu'on va renvoyer à l'ApplicationMaster les container nouvellement alloués pour l'application. |
| org.apache.hadoop.yarn.api.records.ApplicationAttemptId | **getApplicationAttemptId**()<br>Get ApplicationAttemptId of the application master. |
| **AppSchedulingInfo** | **getAppSchedulingInfo**()<br>On va souvent avoir besoin dans le Scheduler des informations contenues dans l'AppSchedulingInfo de l'application. |
| **RMContainer** | **getContainer**(org.apache.hadoop.yarn.api.records.ContainerId containerId)<br>Le Scheduler peut avoir besoin d'un container précis de l'application. |
| **Collection**<**RMContainer**> | **getLiveContainers**()<br>Get the live containers of the application. |
| **Collection**<**RMContainer**> | **getReservedContainers**()<br>Get the reserved containers of the application. |
| boolean | **isPending**()<br>Is this application pending? |

### Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

### Constructor Detail

#### PFEFIFOApplication

```
public PFEFIFOApplication(org.apache.hadoop.yarn.api.records.ApplicationAttemptId applicationAttemptId,
                          String user,
                          PFEFIFOQueue queue,
```

```
                    ActiveUsersManager activeUserManager)
```

C'est le constructeur de la classe. Il contient toutes les informations obligatoires pour que l'on puisse créer l'objet AppSchedulingInfo.

**Parameters:**

`applicationAttemptId` - C'est l'ID de l'ApplicationAttempt

`user` - Le nom de l'utilisateur

queue - La queue dans laquelle se situe l'application

`activeUserManager` - Le gestionnaire des utilisateurs

## Method Detail

### getAppSchedulingInfo

```
public AppSchedulingInfo getAppSchedulingInfo()
```

On va souvent avoir besoin dans le Scheduler des informations contenues dans l'AppSchedulingInfo de l'application.

Cet objet sert pour stocker les demandes de ressources de l'application.

Il sera demandé par le Scheduler lorsque l'application aura de nouvelles requêtes, et lorsqu'il y aura des ressources de disponible.

**Returns:**

l'AppSchedulingInfo de l'application

### getApplicationAttemptId

```
public org.apache.hadoop.yarn.api.records.ApplicationAttemptId getApplicationAttemptId()
```

**Description copied from class: SchedulerApplication**
Get `ApplicationAttemptId` of the application master.

**Specified by:**

`getApplicationAttemptId` in class `SchedulerApplication`

**Returns:**

`ApplicationAttemptId` of the application master

### getLiveContainers

```
public Collection<RMContainer> getLiveContainers()
```

**Description copied from class: SchedulerApplication**
Get the live containers of the application.

**Specified by:**

`getLiveContainers` in class `SchedulerApplication`

**Returns:**

live containers of the application

### getReservedContainers

```
public Collection<RMContainer> getReservedContainers()
```

**Description copied from class: SchedulerApplication**
Get the reserved containers of the application.

**Specified by:**

`getReservedContainers` in class `SchedulerApplication`

**Returns:**

the reserved containers of the application

### isPending

```
public boolean isPending()
```

**Description copied from class: SchedulerApplication**
Is this application pending?

**Specified by:**

`isPending` in class `SchedulerApplication`

**Returns:**

true if it is else false.

### addContainer

```
public void addContainer(RMContainer container)
```

Un container vient d'être alloué pour l'application. On va le placer dans la liste des nouveaux container, afin de prévenir l'ApplicationMaster lors d'un nouvel appel à allocate. On va également le placer dans la liste des container en cours.

**Parameters:**

`container` - Le container alloué pour l'application

### addReservedContainer

```
public void addReservedContainer(RMContainer container)
```

Cette méthode est utile lorsque le Scheduler gère la réservation d'un container. Ce n'est pas utilisé dans le PFEFIFOScheduler, mais son implémentation est obligatoire.

**Parameters:**

`container` - Le container à réserver

### finishedContainer

```
public void finishedContainer(org.apache.hadoop.yarn.api.records.ContainerId containerId)
```

L'application n'a plus besoin du container, on va donc l'enlever des container en cours.

**Parameters:**

`containerId` - Le container terminé

### getContainer

```
public RMContainer getContainer(org.apache.hadoop.yarn.api.records.ContainerId containerId)
```

Le Scheduler peut avoir besoin d'un container précis de l'application.

**Parameters:**

`containerId` - L'ID du container souhaité

**Returns:**

Le RMContainer demandé, ou null si il n'existe pas

### getAndRemoveNewContainers

```
public List<org.apache.hadoop.yarn.api.records.Container> getAndRemoveNewContainers()
```

Cette méthode est appellée lorsqu'on va renvoyer à l'ApplicationMaster les container nouvellement alloués pour l'application. On va donc renvoyer le contenue de la liste newContainers dans une liste nouvellement allouée.

**Returns:**

Les containers nouvellement alloués pour l'application

## A.3 La classe PFEFIFONode

org.apache.hadoop.yarn.server.resourcemanager.scheduler.pfefifo

# Class PFEFIFONode

java.lang.Object
org.apache.hadoop.yarn.server.resourcemanager.scheduler.SchedulerNode
org.apache.hadoop.yarn.server.resourcemanager.scheduler.pfefifo.PFEFIFONode

---

```
public class PFEFIFONode
extends SchedulerNode
```

Cette classe va représenter un noeud au niveau du Scheduler. Elle va gérer les ressources totales et les ressources disponibles du noeud. Elle va également gérer une liste de container qui ont été alloués sur ce noeud.

**Author:**

Erwann Cloarec

## Constructor Summary

Constructors

| Constructor and Description |
|---|
| **PFEFIFONode**(**RMNode** node)<br>Le constructeur prenant en paramètre le RMNode associé au PFEFIFONode que l'on veut créer. |

## Method Summary

Methods

| Modifier and Type | Method and Description |
|---|---|
| void | **allocateContainer**(**RMContainer** rmContainer)<br>Permet d'ajouter un RMContainer à la liste des container alloués sur ce noeud. |
| org.apache.hadoop.yarn.api.records.Resource | **getAvailableResource**()<br>Get available resources on the node. |
| **List**<**RMContainer**> | **getContainers**()<br>Donne la liste des container de ce noeud |
| **String** | **getNodeName**()<br>Get the name of the node for scheduling matching decisions. |
| int | **getNumContainers**()<br>Get number of active containers on the node. |
| **String** | **getRackName**()<br>Get rackname. |
| org.apache.hadoop.yarn.api.records.Resource | **getTotalResource**()<br>Get total resources on the node. |

| | |
|---|---|
| org.apache.hadoop.yarn.api.records.Resource | **getUsedResource**()<br>Get used resources on the node. |
| void | **removeAllContainers**()<br>Permet d'enlever tous les container de la liste. |
| void | **removeContainer**(**RMContainer** container)<br>Permet d'enlever un container de la liste. |

### Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

## Constructor Detail

### PFEFIFONode

```
public PFEFIFONode(RMNode node)
```

Le constructeur prenant en paramètre le RMNode associé au PFEFIFONode que l'on veut créer. On va ici initialiser les attributs de la classe.

**Parameters:**

node - Le RMNode

## Method Detail

### getNodeName

```
public String getNodeName()
```

**Description copied from class: SchedulerNode**
Get the name of the node for scheduling matching decisions.

Typically this is the 'hostname' reported by the node, but it could be configured to be 'hostname:port' reported by the node via the YarnConfiguration.RM_SCHEDULER_INCLUDE_PORT_IN_NODE_NAME constant. The main usecase of this is Yarn minicluster to be able to differentiate node manager instances by their port number.

**Specified by:**

getNodeName in class SchedulerNode

**Returns:**

name of the node for scheduling matching decisions.

### getRackName

```
public String getRackName()
```

**Description copied from class: SchedulerNode**

Get rackname.

**Specified by:**

`getRackName` in class `SchedulerNode`

**Returns:**

rackname

### getUsedResource

```
public org.apache.hadoop.yarn.api.records.Resource getUsedResource()
```

**Description copied from class: SchedulerNode**
Get used resources on the node.

**Specified by:**

`getUsedResource` in class `SchedulerNode`

**Returns:**

used resources on the node

### getAvailableResource

```
public org.apache.hadoop.yarn.api.records.Resource getAvailableResource()
```

**Description copied from class: SchedulerNode**
Get available resources on the node.

**Specified by:**

`getAvailableResource` in class `SchedulerNode`

**Returns:**

available resources on the node

### getNumContainers

```
public int getNumContainers()
```

**Description copied from class: SchedulerNode**
Get number of active containers on the node.

**Specified by:**

`getNumContainers` in class `SchedulerNode`

**Returns:**

number of active containers on the node

### getTotalResource

```
public org.apache.hadoop.yarn.api.records.Resource getTotalResource()
```

**Description copied from class: SchedulerNode**

Get total resources on the node.

**Specified by:**

`getTotalResource` in class `SchedulerNode`

**Returns:**

total resources on the node.

### allocateContainer

```
public void allocateContainer(RMContainer rmContainer)
```

Permet d'ajouter un RMContainer à la liste des container alloués sur ce noeud.

**Parameters:**

`rmContainer` - Le RMContainer à ajouter

### getContainers

```
public List<RMContainer> getContainers()
```

Donne la liste des container de ce noeud

**Returns:**

La liste des container du noeud

### removeAllContainers

```
public void removeAllContainers()
```

Permet d'enlever tous les container de la liste. On a donc toutes les ressources disponibles.

### removeContainer

```
public void removeContainer(RMContainer container)
```

Permet d'enlever un container de la liste.

**Parameters:**

`container` - Le RMContainer à supprimer

## A.4 La classe PFEFIFOQueue

org.apache.hadoop.yarn.server.resourcemanager.scheduler.pfefifo

## Class PFEFIFOQueue

java.lang.Object
  org.apache.hadoop.yarn.server.resourcemanager.scheduler.pfefifo.PFEFIFOQueue

**All Implemented Interfaces:**

Queue

---

```
public class PFEFIFOQueue
extends Object
implements Queue
```

Cette classe représente une queue dans un Scheduler. Elle va enregistrer des informations concernant les applications et leurs utilisateurs qui sont dans celle-ci. C'est principalement l'objet QueueMetrics qui va enregistrer ces informations.

**Author:**

Erwann Cloarec

### Constructor Summary

Constructors

| Constructor and Description |
|---|
| **PFEFIFOQueue**(org.apache.hadoop.conf.Configuration conf, **PFEFIFOScheduler** scheduler)<br>Le constructeur de la classe qui va initialiser les attributs. |

### Method Summary

Methods

| Modifier and Type | Method and Description |
|---|---|
| **QueueMetrics** | **getMetrics**()<br>Get the queue metrics |
| **Map**<org.apache.hadoop.yarn.api.records.QueueACL,org.apache.hadoop.security.authorize.AccessControlList> | **getQueueAcls**()<br>Get ACLs for the queue. |
| org.apache.hadoop.yarn.api.records.QueueInfo | **getQueueInfo**(boolean includeChildQueues, boo<br>Get queue information |
| **String** | **getQueueName**()<br>Get the queue name |
| **List**<org.apache.hadoop.yarn.api.records.QueueUserACLInfo> | **getQueueUserAclInfo**(org.apache.hadoop.securi<br>Get queue ACLs for given user. |
| boolean | **hasAccess**(org.apache.hadoop.yarn.api.records<br>org.apache.hadoop.security.UserGroupInformat |

### Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

### Constructor Detail

#### PFEFIFOQueue

```
public PFEFIFOQueue(org.apache.hadoop.conf.Configuration conf,
            PFEFIFOScheduler scheduler)
```

Le constructeur de la classe qui va initialiser les attributs.

**Parameters:**

conf - La configuration du Scheduler

scheduler - Le Scheduler

### Method Detail

#### getQueueName

```
public String getQueueName()
```

**Description copied from interface: Queue**
Get the queue name

**Specified by:**

getQueueName in interface Queue

**Returns:**

queue name

### getMetrics

```
public QueueMetrics getMetrics()
```

**Description copied from interface: Queue**
Get the queue metrics

**Specified by:**

getMetrics in interface Queue

**Returns:**

the queue metrics

### getQueueInfo

```
public org.apache.hadoop.yarn.api.records.QueueInfo getQueueInfo(boolean includeChildQueues,
                                                                 boolean recursive)
```

**Description copied from interface: Queue**
Get queue information

**Specified by:**

getQueueInfo in interface Queue

**Parameters:**

includeChildQueues - include child queues?

recursive - recursively get child queue information?

**Returns:**

queue information

### getQueueAcls

```
public Map<org.apache.hadoop.yarn.api.records.QueueACL,org.apache.hadoop.security.authorize.AccessControlList> getQueueAcls()
```

**Description copied from interface: Queue**
Get ACLs for the queue.

**Specified by:**

getQueueAcls in interface Queue

**Returns:**

ACLs for the queue

### getQueueUserAclInfo

```
public List<org.apache.hadoop.yarn.api.records.QueueUserACLInfo> getQueueUserAclInfo(org.apache.hadoop.security.UserGroupInformati
```

**Description copied from interface: Queue**
Get queue ACLs for given user.

**Specified by:**

getQueueUserAclInfo in interface Queue

**Parameters:**

user - username

**Returns:**

queue ACLs for user

### hasAccess

```
public boolean hasAccess(org.apache.hadoop.yarn.api.records.QueueACL acl,
                         org.apache.hadoop.security.UserGroupInformation user)
```

**Specified by:**

hasAccess in interface Queue

Overview Package Class Use Tree Deprecated Index Help

Prev Class Next Class Frames No Frames All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

Département Informatique
5e année
2013 - 2014

Hadoop: Optimisation et Ordonnancement

**Résumé :** Rapport de Projet de Fin d'Etudes, Hadoop: Optimisation et Ordonnancement. Ce projet vise à étudier le framework Hadoop en profondeur. L'objectif est de comprendre le fonctionnement de l'ordonnancement des tâches dans Hadoop.

**Mots clefs :** Rapport, Projet, Hadoop, Ordonnancement, Optimisation, étudier, tâches

**Abstract:** End of Studies Project resume, Hadoop: Scheduling and Optimization. This project's objectif is to deeply study Hadoop and understanding its tasks scheduling.

**Keywords:** Project, resume, Hadoop, Scheduling, Optimization, study, tasks

---

**Encadrants**
JLASSI Aymen
aymen.jlassi@univ-tours.fr
MARTINEAU Patrick
patrick.martineau@univ-tours.fr

**Étudiant**
CLOAREC Erwann
erwann.cloarec@etu.univ-tours.fr

DI5 2013 - 2014

Université François-Rabelais, Tours