|  |  |
| --- | --- |
|  | Version 0.0.1 ● |
| Documentation du projet | |
|  | |

Table Des matières

1 Description du logiciel 5

1.1 Diagramme de classe 5

1.2 Diagramme de séquence 6

1.3 Fonctionnement du logiciel 6

1.3.1 Démarrage 7

1.3.2 Splitting 8

1.3.3 Mapping 8

1.3.4 Shuffling & Reducing 9

1.3.5 Assembling 10

1.3.6 Résultats 10

2 Lancement du Logiciel 11

3 Etude des performances 12

3.1 Impact du paramètre *‘size\_split’* 12

3.2 Impact du nombre de machines (slaves) disponible 13

4 Problèmes rencontrés solutions apportées 14

5 Optimisation 15

6 Annexe 15

Préface

1. Version du document

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Auteurs (s)** | **Objectif** | **Date** |
| 0.0.1 | Eric FOKOU | Documentation du projet | 22/11/16 |

1. Documentation associée

|  |  |  |  |
| --- | --- | --- | --- |
| Titre du document | Auteurs | Description du document | Date de dernière |
| MASTER\_SHAVADOOP\_DOC | Eric FOKOU | Documentation code source sous forme HTML | 22/11/2016 |
| SLAVE\_SHAVADOOP\_DOC | Eric FOKOU | Documentation code source sous forme HTML | 22/11/2016 |

1. Plateforme Technique

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **OS** | **Logiciel** | **Version** |
| Langage de programmation | Linux | Java | 1.7 |
| IDE | Linux | Eclipse | Luna 4.4.2 |

Introduction

Le but de ce document est de donner une description du projet Shavadoop. Ce projet consiste à implémenter une version du ‘**Wordcount’,** programme qui permet de compter les mots dans un document dans le système de fichier distribué **Hadoop**. Ce type de programme distribué est très important lorsque la taille des données est importante et notamment dans ce cas précis lorsque le fichier en entrée est assez volumineux.

Afin d’effectuer des traitements distribués, notre programme utilisera des ressources de calcul (mémoire, processeur, etc…) de plusieurs machines pour lancer des traitements en parallèle.

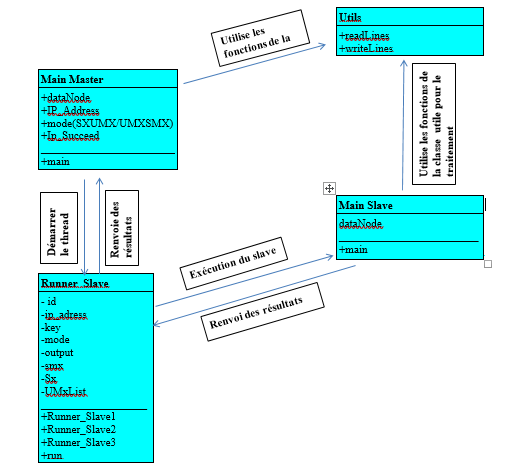
Pour l’implémentation de notre solution, nous avons créé deux projets :

* MASTER\_SHAVADOOP : programme maitre chargé d’effectuer des traitements mineurs et principalement de coordonner l’ensemble des traitements. Il peut être vu comme le **Namenode** dans le système de gestion de fichiers distribués dans Hadoop.
* SLAVE\_SHAVADOOP : programme chargé de réalisé des traitements majeurs (écriture et lecture des données, etc…). Il peut être vu comme le **Datanode** dans le système de gestion de fichiers distribués dans Hadoop.

# Description du logiciel

Notre projet est organisé en deux sous projets principaux : **MASTER\_SHAVADOOP**  et **SLAVE\_SHAVADOOP**. Nous commençons par donner un diagramme de classe simplifié illustrant les interactions entre les classes des deux sous projets, ensuite nous donnons une description simple des différentes interactions.

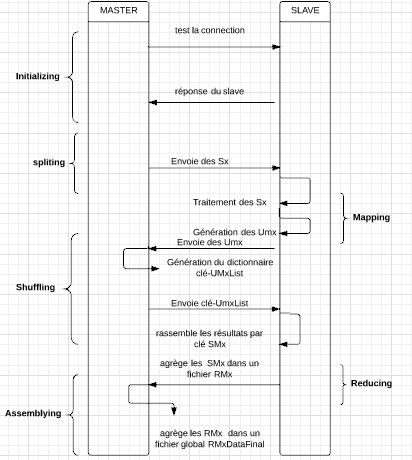
## Diagramme de classe



**Figure 1 :** Diagramme de classe

Le MASTER\_SHAVADOOP  contient deux classes principales à savoir : **Main** et **Runner\_slave** et une troisième classe **Utils** qui contient des fonctions utiles pour le traitement. La classe Main lance les différents traitements selon deux mode (**modeSXUMX** et **modeUMXSMX**), sur SLAVE\_SHAVADOOP à travers la classe Runner\_Slave qui hérite de la classe Thread. Le SLAVE\_SHAVADOOP contient une classe **Main** qui exécute les différentes tâches et une classe **Utils**. Pour une description détaillée de l’implémentation, référez-vous aux documents : **MASTER\_SHAVADOOP\_DOC** et **SLAVE\_SHAVADOOP\_DOC**

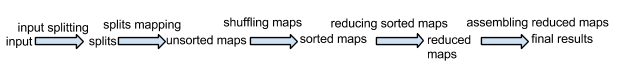
## Diagramme de séquence



**Figure 2 :** Diagramme de séquence

## Fonctionnement du logiciel

Le Projet se divise en cinq grandes phases en plus d’une phase de démarrage.



**Figure 3** : Etapes de traitement

Nous allons donner une description de chacune des étapes relatives à notre implémentation. Avant de donner une description des étapes, commençons par donner un exemple de commande pour lancer le logiciel car ceci est important pour la description des phases qui va suivre.

Commande de lancement :

**java – jar MASTER\_SHAVADOOP.jar file pathTemp/ size\_split list\_ip\_adress**

Avec

**file** : Le fichier en entrée à traiter.

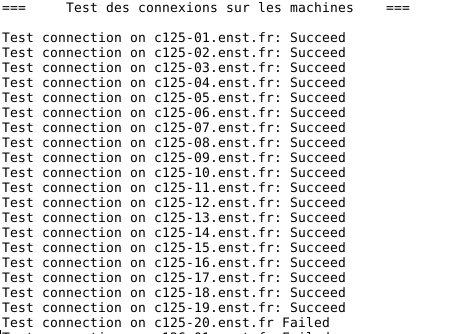
**pathTemp/** : Le chemin ou dossier ou sera créé les fichiers temporaires des traitements.

**size\_split**: Nombre de lignes par split (morceaux) qui seront créés

**list\_ip\_adress**: Liste des Adresses à laquelle le master va essayer de se connecter

### Démarrage

Dans cette phase le Master teste la connexion sur les machines renseignées dans le fichier passé en paramètre (list\_ip\_adress**)**. Ce test est réalisé en effectuant une petite addition sur la machine hôte. A la fin de cette phase la liste des machines sur lesquelles la connexion a réussi est enregistré dans un fichier créé (**Ip\_Succeed**). Ce fichier est créé dans le répertoire contenant le MASTER\_SHAVADOOP.jar. Les machines contenues dans Ip\_Succeed seront utilisées dans la suite pour lancer des traitements en parallèle.



**Figure 3** : démarrage du logiciel avec le test des connexions sur les machines

### Splitting

Dans cette phase le Master commence par faire un Splitting (division) ligne par ligne du fichier en entrée. Un ensemble de fichier **Sx** sont créés à ce niveau. Il faut noter ici que le nombre de **Sx** crée dépend du paramètre **size\_split** en entrée ; ceci constitue une optimisation que nous présenterons à la section **1.3 optimisation**.

### Mapping

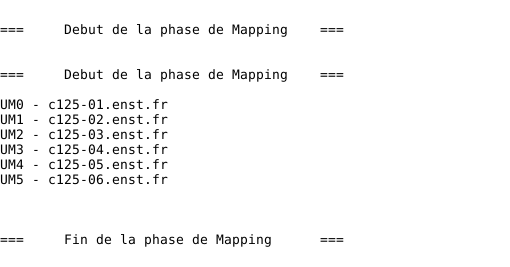
Cette étape marque le début de l’interaction entre les deux sous projet : MASTER\_SHAVADOOP et SLAVE\_SHAVADOOP. Le programme du Master commence par créer un ensemble d'objet de la classe Runner\_Slave (classe qui hérite de la classe **Thread**) et crée le dictionnaire **UMx - machines**. Le Master au moment de la création des Runner\_Slave passe trois paramètres importants au constructeur de la classe Runner\_Slave :

* **mode** : le mode d'exécution qui vaut **modeSXUMX** dans ce cas.
* **ip\_address**: l'adresse sur laquelle le Runner\_Slave sera créer
* **Sx** : le fichier à traiter

Ces Runner\_Slave sont ensuite lancés par le Master avec la méthode **start** de la classe Thread. Les Runner\_Slave à leur tour grâce à la classe **ProcessBuilder** exécutent le fichier SLAVE\_SHAVADOOP.jar en passantles paramètres transmis par le Master.

Le programme SLAVE\_SHAVADOOP.jar dans cette phase génère un fichier **UMx** associé au fichier **Sx** en paramètre. Le fichier UMx contient pour chaque occurrence d’un mot dans le fichier Sx une association **clé – valeur** ou clé est le mot et la valeur est égale à **1**.

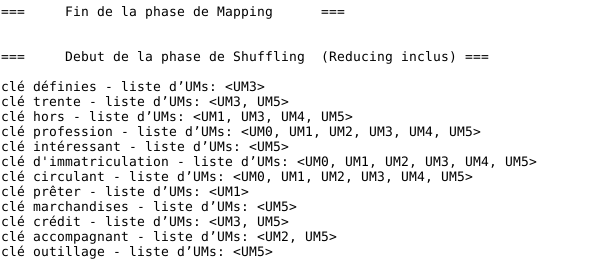
Pour une meilleure compréhension des interactions (paramètres des fonctions, …), se référer à la documentation associée MASTER\_SHAVADOOP\_DOC et SLAVE\_SHAVADOOP\_DOC.



**Figure 4** : Phase de Mapping : Affichage des UMx crées sur chaque machine

### Shuffling & Reducing

Cette phase débute par la réception des clés issus du traitement Mapping des slaves. Le Master peut alors construire le dictionnaire **clés -UMx**



**Figure 5** : Phase de Shuffling : création du dictionnaire clé – liste UMx

Le Master crée ensuite un ensemble d'objet de la classe Runner\_Slave en passant trois paramètres importants au constructeur de la classe Runner\_Slave:

* **mode** : le mode d'exécution qui vaut **modeUMXSMX** dans ce cas.
* **ip\_address**: l'adresse sur laquelle le Runner\_Slave sera créer
* **key** : la clé (mot) à traiter
* **SMx :** le fichier à créer par le programme SLAVE\_SHAVADOOP\_DOC
* **UMxList :** la liste des fichiers UMx contenant la clé. Pour notre implémentation nous avons créé une chaine de caractères contenant cette liste, chaque UMx séparé par un espace (variable de type **String**).

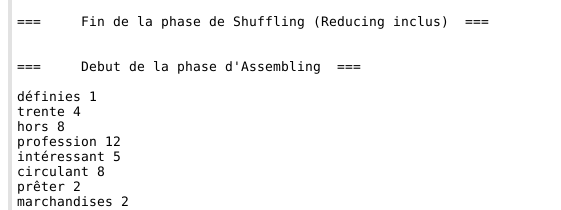
Les Runner\_Slave sont ensuite lancés par le Master avec la méthode **start** de la classe Thread. Les Runner\_Slave à leur tour grâce à la classe **ProcessBuilder** exécutent le fichier SLAVE\_SHAVADOOP.jar en passant les paramètres transmis par le Master.

Le programme SLAVE\_SHAVADOOP.jar dans cette phase génère Un fichier SMx. Le fichier SMx contient toutes les occurrences contenues dans les fichiers UMx de la clé passée en paramètre avec la valeur 1 associé. Directement à la suite de ce traitement le slave exécute une étape de Reducing (agrégation sur la clé en sommant la valeur associée pour toutes les occurrences). Le slave génère un fichier **RMx** contenant le résultat de cette agrégation et renvoie la paire **clé – valeur** au Master. Ici **clé** représente le mot et **valeur** le nombres d’occurrence du mot dans le fichier.

Pour une meilleure compréhension des interactions (paramètres des fonctions, …), se référer à la documentation associée MASTER\_SHAVADOOP\_DOC et SLAVE\_SHAVADOOP\_DOC.

### Assembling

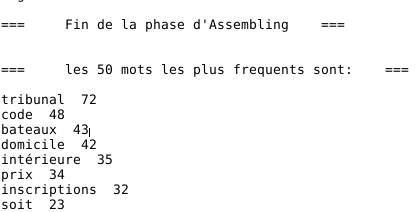
Le Master réceptionne les **RMx** et les fusionnent.



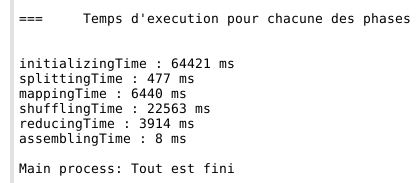
**Figure 6** : Phase d’Assembling

### Résultats

Pour finir le Master affiche les **50** mots les plus fréquents et un récapitulatif des temps d’exécution dans chacune des phases.



**Figure 7** : Affichage des mots les plus fréquents



**Figure 8** : Affichage des temps d’exécution

Un résultat de lancement complet est fourni avec le projet dans le dossier **ExempleResultat.**

* Fichier le compte de tous les mots. (**RMxDataFinal**)
* Fichier de la sortie standard du programme. (**OutPut**)

# Lancement du Logiciel

Le lancement du logiciel se fait à partir de des deux jar **MASTER\_SHAVADOOP.jar** et **SLAVE\_SHAVADOOP.jar**. Les deux jars doivent être placés dans le même répertoire et la commande de lancement est la suivante :

**java -jar MASTER\_SHAVADOOP.jar file pathTemp/ size\_split list\_ip\_adress**

Avec

**file** : Le fichier en entrée à traiter.

**pathTemp/** : Le chemin ou dossier ou sera créé les fichiers temporaires des traitements.

**size\_split**: Nombre de lignes par split (morceaux) qui seront créés

**list\_ip\_adress**: Liste des Adresses à laquelle le master va essayer de se connecter

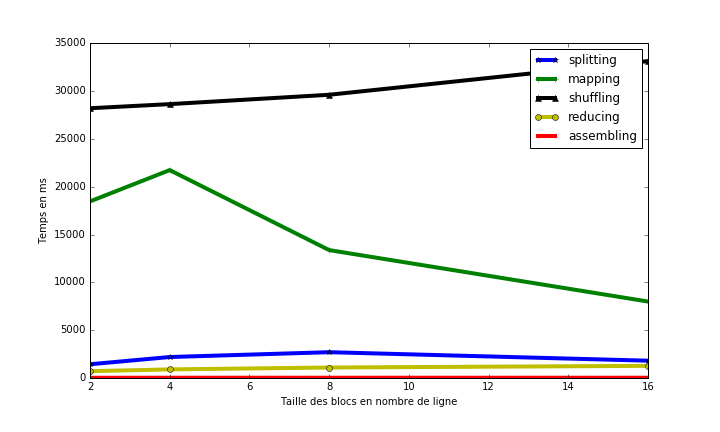
Le résultat final, le fichier **RMxDataFinal** se trouve au même niveau que les jars.

# Etude des performances

## Impact du paramètre *‘size\_split’*

Il faut noter ici que ce paramètre est très important par rapport à l’implémentation actuelle car plus il est petit, plus grand sera le nombre de fichiers Sx créés. La conséquence de ceci est donc que la totalité des machines disponibles (contenues dans IP\_Succeed) seront utilisées afin de paralléliser les traitements. Par contre si ce nombre est très grand, le nombre de fichier Sx sera petit et explicitement quelques machines disponibles seront utilisées laissant ainsi certaines machines inexploitées (**Nous précisons que ceci est le comportement de l’implémentation actuelle par rapport aux spécifications du projet et constitue une piste d’optimisation pour la suite)**.

Observons des résultats de performance de chacune des étapes par rapport à une modification de ce paramètre sur le fichier **domaine\_public\_fluvial.txt.** Notons que ce fichier contient 256 lignes non vides et que nous avons fixé pour ce test le nombre de machines à 8.

****

**Figure 8** : Performances en temps de calcul en fonction de la taille des blocs en entrée

Nous observons :

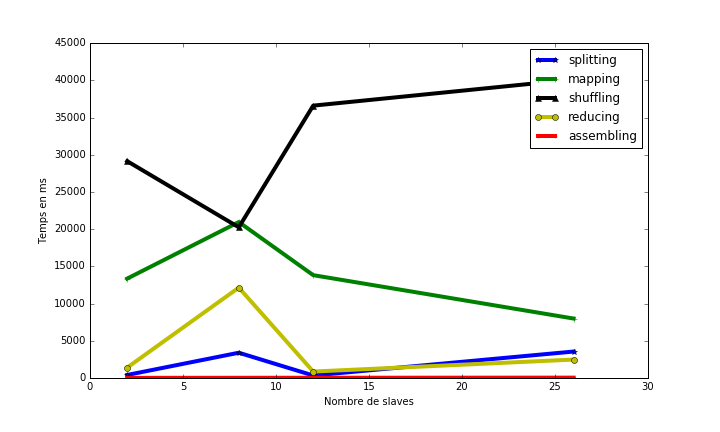
* L’augmentation du nombre de ligne dans les blocs impacte très peu les étapes de reducing et d’assembling
* Une légère augmentation du temps de calcul dans la phase de **shuffling** est à noter quand la taille des blocs augmente ; par contre notons une baisse assez importante du temps de **mapping** avec l’augmentation de la taille des blocs.

Asymptotiquement, par cette observation, nous pouvons dire que l’augmentation de la taille de **size\_split** devrait améliorer les résultats mais jusqu’à un certain seuil à déterminer.

## Impact du nombre de machines (slaves) disponible

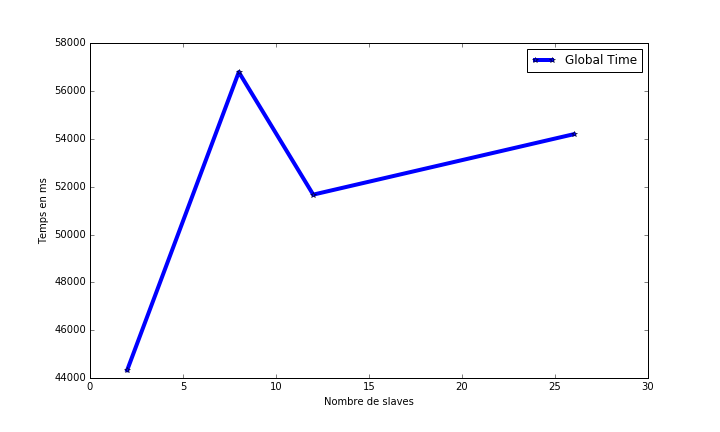
**Questions** : Nos performances (temps d’exécution) sont-elles toujours meilleures quand le nombre de machines augmentent ??

Pour répondre à cette question, observons la figure suivante où nous avons tracé pour différents nombres de machines utilisés le temps mis par le programme pour chaque étape. Nous avons fixé le paramètre **size\_split** à8 pour le test avec le fichier **domaine\_public\_fluvial.txt**:



**Figure 9.1** : Performances en temps de calcul en fonction du nombre de slave

Ce graphique est difficile à interpréter ; additionnons les temps obtenus pour chaque nombre de slave pour une meilleure visualisation. On obtient le graphique suivant :



**Figure 9.2** : Performances globale en temps de calcul en fonction du nombre de slave

Nous observons donc ici une variation non uniforme de la courbe. Cependant notons un temps assez réduit quand le nombre de machine est fixé à 2. Ce comportement doit provenir certainement des échangent entre les machines slaves et le master qui dégrade considérablement les performances.

# Problèmes rencontrés solutions apportées

Comme problème rencontré nous pouvons citer principalement :

* La difficulté d’identifier les bugs qui se produisaient dans le jar SLAVE\_SHAVADOOP.jar au moment de l’appel par le Master (notamment si les paramètres étaient bien passés). Cette difficulté était réelle d’autant plus qu’il y’avait pas moyen d’utiliser le debugger intégré de l’éditeur eclipse. Afin de surmonter cela nous avons mis des messages d’évolution dans le slave avec un formatage spécifique et nous avons récupéré ces messages sur le Master en testant si l’affichage progressif des messages et vérifiant le formatage défini dans le slave.
* La récupération de la sortie du SLAVE\_SHAVADOOP.jar depuis notre classe Runner\_Slave en utilisant la classe **AfficheurFlux** fournit. En effet il y’avait un problème de synchronisation car il fallait que le Runner\_Slave attende explicitement la fin de l’exécution du **process** crée par la méthode **ProcessBuilder**. Pour surmonter cette difficulté, nous avons ramené les traitements le plus importants de AfficheurFlux dans Runner\_Slave et nous avons synchronisés le **thread** (Runner\_Slave) avec le **process** simplement avec la méthode **process.waitFor().** Les sorties du slave ont ensuite été remises proprement au Master.
* D’autres difficultés rencontrées ont été résolues en allant sur Internet et récupérant des solutions.

# Optimisation

Comme optimisation que nous avons réalisé dans ce programme, nous citons :

* La possibilité de lancer le programme en spécifiant des paramètres en entrées (répertoire temporaire, taille du splitting, fichier des adresses IP). Ceci est très important car ça rend le programme plus flexible dans l’utilisation.
* Création des classes (Runner\_Slaves, Utils) afin d’améliorer l’évolutivité et la lisibilité du code.

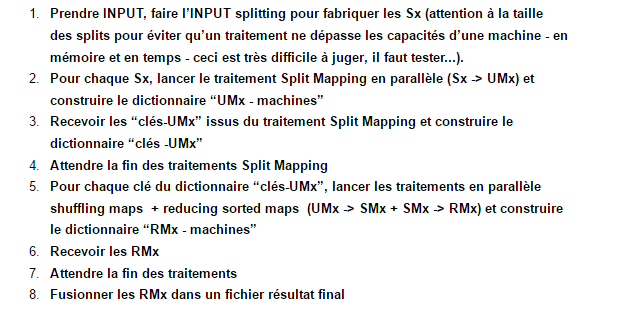
Comme optimisation futur :

* Utilisation complète des machines quelques soit le nombre de fichiers Sx crées (nous rappelons que dans notre programme cette valeur dépend du paramètre **size\_split** en entrée). On pourra par exemple trouver une règle de division adéquate en fonction de la taille du fichier en entrée et du nombre de machine disponible.
* Gestion des TIMEOUT : relancer les traitements sur les slaves qui ne répondent pas après un délai (ou encore lancer les traitements sur des machines qui ont terminées leurs traitements).
* Le Master devra vérifier le statut des slaves après un moment pour remettre à jour sa liste de machines actives.

# Annexe



**Figure 10** : Word count pipeline



**Algorithme 1** : Algorithme Word count pipeline