

Rapport Projet Données Réparties

Partie Hadoop

Axel GRUNIG
Thomas GUILLAUD

Département Sciences du Numérique - ASR
2020-2021

Contents

1	Architecture	3
2	Présentation des Classes et Interfaces	4
2.1	Interface JobInterface et Classe Job	4
2.2	Interface Worker et Classe WorkerImpl	6
2.3	Interface CallBack et Classe CallBackImpl	7
3	Difficultés Rencontrées	8
4	Test	8
5	Annexe	8

List of Figures

1	Architecture de la Partie Hidoop	3
2	Interface JobInterface	4
3	Classe Job	4
4	Classe Employe	5
5	Interface Worker	6
6	Classe WorkerImpl	6
7	Interface CallBack	7
8	Classe CallBackImpl	7
9	Architecture avec Commentaires	8

1 Architecture

Voici l'architecture de la partie Hadoop du Projet de Données Répartie:

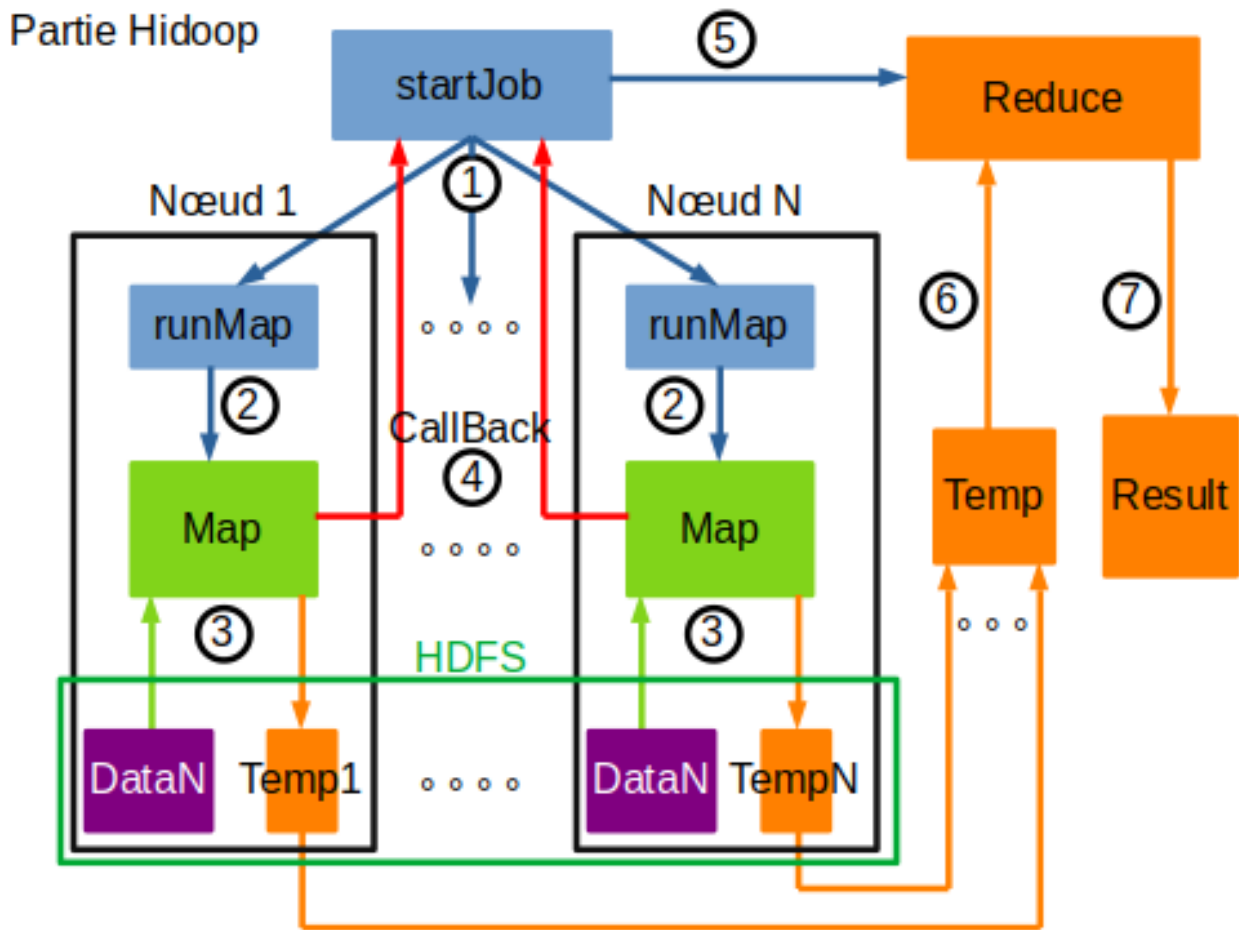


Figure 1: Architecture de la Partie Hadoop

Dans un premier temps, la classe qui implémente `startJob` va activer les méthodes `runMap` de chaque Nœud via RMI.

Chaque Nœud va donc lancer le `Map` localement, en prenant chaque fragment `Data` qui sont données par la partie HDFS, et retourne des fragments `Temp`.

à chaque fois qu'un Nœud fini son traitement, il lance la méthode `réveiller()` de `callback` afin de signaler au Thread principal (lancé par la classe `Job`) qu'il a fini.

Une fois que la classe principale `Job` reçoit tous les `callback`, il effectue le `Reduce` sur le fichier `Temp`, qui est le regroupement de chaque fragment, et renvoie un résultat `Result`.

2 Présentation des Classes et Interfaces

2.1 Interface JobInterface et Classe Job

```
public interface JobInterface {  
    // Méthodes requises pour la classe Job  
    public void setInputFormat(Format.Type ft);  
    public void setInputFname(String fname);  
  
    public void startJob (MapReduce mr);  
}
```

Figure 2: Interface JobInterface

L'Interface JobInterface ne fait que présenter les différentes méthodes que la classe Job devra implémenter.

```
public class Job implements JobInterface{  
    Format.Type fType;  
    String fName;  
    static String server[] = {"Noeud1", "Noeud2", "Noeud3"};  
    static int port[] = {2001, 2002, 2003};  
  
    public Job(){  
    }  
  
    @Override  
    public void setInputFormat(Format.Type ft) {  
        this.fType = ft;  
    }  
  
    @Override  
    public void setInputFname(String fname) {  
        this.fName = fname;  
    }  
  
    @Override  
    public void startJob(MapReduce mr) {  
        try {  
            CallbackImpl cb = new CallbackImpl(server.length);  
  
            for (int i=0 ; i<server.length ; i++) {  
                Thread t = new Thread(new Employe(i, mr, this.fType, this.fName, cb));  
                t.start();  
            }  
  
            Format frReduce;  
            Format fwReduce;  
            if (fType.equals(Format.Type.LINE)) { // détection du type de format d'input,  
                                                // l'output est obligatoirement l'autre  
                frReduce = new KVFormat(fName + "-tot");  
                fwReduce = new KVFormat(fName + "-res");  
            } else {  
                frReduce = new KVFormat(fName + "-tot");  
                fwReduce = new KVFormat(fName + "-res");  
            }  
  
            cb.attente();  
            frReduce.open(Format.OpenMode.R);  
            fwReduce.open(Format.OpenMode.W);  
            mr.reduce(frReduce, fwReduce);  
            frReduce.close();  
            fwReduce.close();  
            System.out.println("Fini Reduce");  
        } catch (InterruptedException | RemoteException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Figure 3: Classe Job

Les méthodes `setInputFormat()` et `setInputFname()` permettent respectivement d'initialiser le format de départ du fichier et le nom de base du fichier.

La méthode `startJob()` permet de lancer les différents worker puis de faire le Reduce.

Dans un premier temps, on va initialiser le callback en indiquant le nombre de serveurs, puis on va lancer plusieurs Thread (autant que de Noeuds) afin de paralléliser l'action des Worker. Ces thread vont lancer la classe `Employe` qui est `Runnable`, et qui prends en paramètre le numéro du thread "i", le `mapReduce` "mr", l'`InputFormat` "ftype", le nom du fichier "fname" et le callback "cb". L'`Employe` va donc se chargé de lancer `runMap()` sur chaque Worker, et donc d'initialiser les formats de lecture et d'écriture du Map.

Ensuite, on va lancer la méthode `attente()` de callback afin d'attendre la fin des différents Worker. Son fonctionnement sera détaillé plus tard (partie `CallBack`)

Enfin, on va initialiser les format de lecture et d'écriture du Reduce, en indiquant le fichier qu'on doit lire et le fichier dans lequel on doit écrire. le reduce va prendre comme format de lecture, et d'écriture, le même format que celui d'écriture du Map.

```
class Employe implements Runnable{
    private final CallbackImpl cb;
    private final int numServ;
    private final MapReduce mr;
    private Format.Type fType;
    private String fName;

    public Employe(int nb, MapReduce mr, Format.Type fType, String fName, CallbackImpl cb){
        this.numServ = nb;
        this.mr = mr;
        this.fType = fType;
        this.fName = fName;
        this.cb = cb;
    }

    @Override
    public void run() {
        try {
            // Création des formats d'input (fr) et d'output (fw)
            Format frMap;
            Format fwMap;
            if (fType.equals(Format.Type.LINE)){ // détection du type de format d'input,
                                                // l'output est obligatoirement l'autre
                frMap = new LineFormat(fName + "-" + (numServ+1));
                fwMap = new KVFormat(fName + "-" + (numServ+1) + "res");
            } else {
                frMap = new KVFormat(fName + "-" + (numServ+1));
                fwMap = new LineFormat(fName + "-" + (numServ+1) + "res");
            }

            System.out.println("//localhost:" + Job.port[numServ] + "/" + Job.server[numServ]);
            Worker worker = (Worker) Naming.lookup("//localhost:" + Job.port[numServ] + "/" + Job.server[numServ]);
            worker.runMap(mr, frMap, fwMap, cb);
        } catch (NotBoundException exception) {
            exception.printStackTrace();
        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}
```

Figure 4: Classe `Employe`

La classe `Employe` permet d'appeler la méthode `runMap()` des Worker tout en initialisant les formats d'écriture et de lecture, et en indiquant les fichiers de lecture et d'écriture.

Le `fType` est le format de lecture du fichier dont le nom est en paramètre, et le format d'écriture correspond à l'autre format (ici, `Line` est le format de lecture et `KV` le format d'écriture si `fType` est `Line`) Ensuite, `Employe` fait un `Naming.lookup` afin de récupérer le stub du Worker dont il a la charge (indiqué par `numServ`) et lance le `runMap()` du Worker.

2.2 Interface Worker et Classe WorkerImpl

```
public interface Worker extends Remote {  
    public void runMap (Mapper m, Format reader, Format writer, CallBack cb) throws RemoteException;  
}
```

Figure 5: Interface Worker

L'interface Worker contient une unique méthode que devra obligatoirement implémenter la classe WorkerImpl. On notera aussi le fait que cette interface étend Remote. En effet nous voulons lancer la méthode runMap sur des machines distantes à partir de notre machine principale.

```
public class WorkerImpl extends UnicastRemoteObject implements Worker {  
    static String server[] = {"Noeud1", "Noeud2", "Noeud3"};  
    static int port[] = {2001, 2002, 2003};  
    static int choix;  
  
    protected WorkerImpl() throws RemoteException {  
    }  
  
    @Override  
    public void runMap(Mapper m, Format reader, Format writer, CallBack cb) throws RemoteException {  
        reader.open(Format.OpenMode.R);  
        writer.open(Format.OpenMode.W);  
        m.map(reader, writer);  
        reader.close();  
        writer.close();  
        System.out.println("Fini Map " + choix);  
        cb.reveiller();  
    }  
  
    public static void main(String args[]){  
        try {  
            choix = Integer.parseInt(args[0]) - 1;  
            WorkerImpl worker = new WorkerImpl();  
            LocateRegistry.createRegistry(port[choix]);  
            Naming.rebind("//localhost:" + port[choix] + "/" + server[choix], worker);  
        } catch (RemoteException | MalformedURLException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Figure 6: Classe WorkerImpl

La classe WorkerImpl étend UnicastRemoteObject pour les mêmes raisons (et implémente Worker bien évidemment). On y trouve la méthode runMap(...) qui prend en paramètre le traitement à réaliser sur les différents fragments du fichier d'origine, le format du fragment (in), le format du fichier temporaire produit par le traitement (out) ainsi qu'un objet CallBack pour pouvoir prévenir le thread principal de Job que le Worker a fini d'exécuter son traitement. Cette méthode va simplement ouvrir le fichier contenant le fragment et le fichier temporaire dans lequel elle va écrire respectivement dans les modes lecture et écriture. Elle va ensuite exécuter le traitement (m.map(...)) puis fermer les deux fichiers. Elle va enfin prévenir que le Worker a terminé son exécution via l'objet CallBack ("cb.reveiller(")).

2.3 Interface Callback et Classe CallbackImpl

```
public interface Callback extends Remote {
    public void attente() throws InterruptedException, RemoteException;
    public void reveiller() throws RemoteException;
}
```

Figure 7: Interface Callback

L'Interface Callback ne fait que présenter les différentes méthodes que la classe Job devra implémenter. On notera aussi le fait que cette interface étends Remote, car voulons qu'elle soit utilisé sur la machine principale par les différents Worker, qui sont sur les machines distantes.

```
public class CallbackImpl extends UnicastRemoteObject implements Callback {

    private final ReentrantLock moniteur;
    private final Condition signal;
    private final int nbNoeuds;
    private final Condition noeudAttente;

    public CallbackImpl(int nbNoeuds) throws RemoteException {
        this.nbNoeuds = nbNoeuds;
        this.moniteur = new ReentrantLock();
        this.signal = this.moniteur.newCondition();
        this.noeudAttente = this.moniteur.newCondition();
    }

    public void attente() throws InterruptedException {
        moniteur.lock();
        int i = nbNoeuds;
        while (i > 0) {
            noeudAttente.signal();
            signal.await();
            i--;
            System.out.println(i + " Signaux en attente");
        }
        System.out.println("Fin Attente");
        moniteur.unlock();
    }

    public void reveiller() throws RemoteException {
        moniteur.lock();
        while (!moniteur.hasWaiters(signal)) {
            try {
                noeudAttente.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        signal.signal();
        moniteur.unlock();
    }
}
```

Figure 8: Classe CallbackImpl

La classe CallbackImpl étends UnicastRemoteObject pour les mêmes raisons (et implémente Callback bien évidemment). On y trouve, en plus du constructeur, les méthodes attente() et reveiller(). On a fait le choix d'utiliser un moniteur afin de synchroniser les Worker et le Job.

La méthode attente() permet au Thread principale de Job d'attendre que tous les Worker ont fini leur traitements.

La méthode reveiller() permet au Worker de réveiller le Thread principal de Job.

Le principe est que dans une boucle while, le Job va attendre X signaux "signal.signal()" (X correspond au nombre de Noeuds, initialisé dans le constructeur) avant de reprendre la suite du startJob(). Le Worker, lui, va juste signaler le Thread principal de Job, mais doit le faire uniquement si le Job est en attente d'un signal, afin d'éviter que le Job attende toujours alors que les Worker ont tous fini (il y a un risque que des Worker fassent "signal.signal()" alors que Job n'est pas encore en attente).

3 Difficultés Rencontrées

La principale difficulté rencontrée a été la compréhension de l'objet `CallBack`. Nous ne comprenions pas exactement son utilité ni comment il devait fonctionner. Après une séance de question avec un professeur à ce sujet, tout cela est devenu plus clair et nous avons pu mettre en place l'objet `CallBack`.

Par la suite une difficulté importante a été de mettre en place la synchronisation des Worker qui font appel à `veiller()` et du Job qui fait appel à `attente()`. En effet, un Worker ne peut signaler ("`signal.signal()`") le Job que si il est en attente ("`signal.await()`"), sinon, le Worker doit attendre ("`neoudAttente.await()`") que le Job soit en attente d'un réveil: le Job signale donc les Worker avant de se placer en attente ("`neoudAttente.signal()`").

Cela évite que plusieurs Worker ne passe dans la section critique de réveiller() alors que Job ne détecte qu'un seul "`signal.signal()`"

4 Test

Pour créer les seuveu, il suffit de faire : "`Java WorkerImpl`" et de rajouter en argument le numéro du serveur (de base, c'est 1, 2, ou 3) Puis, une fois créer, il faut lancer `MyMapReduce` avec comme arument le nom de base du fichier.

Les fragments ont pour nom "`fname-X`" avec X le numéro du fragment

Le résultat retourné par le map est "`fname-Xres`" avec X le numéro du fragment

Le fichier de lecture du reduce a pour nom "`fname-res`"

Le résultat du reduce est dans le fichier "`fname-tot`"

5 Annexe

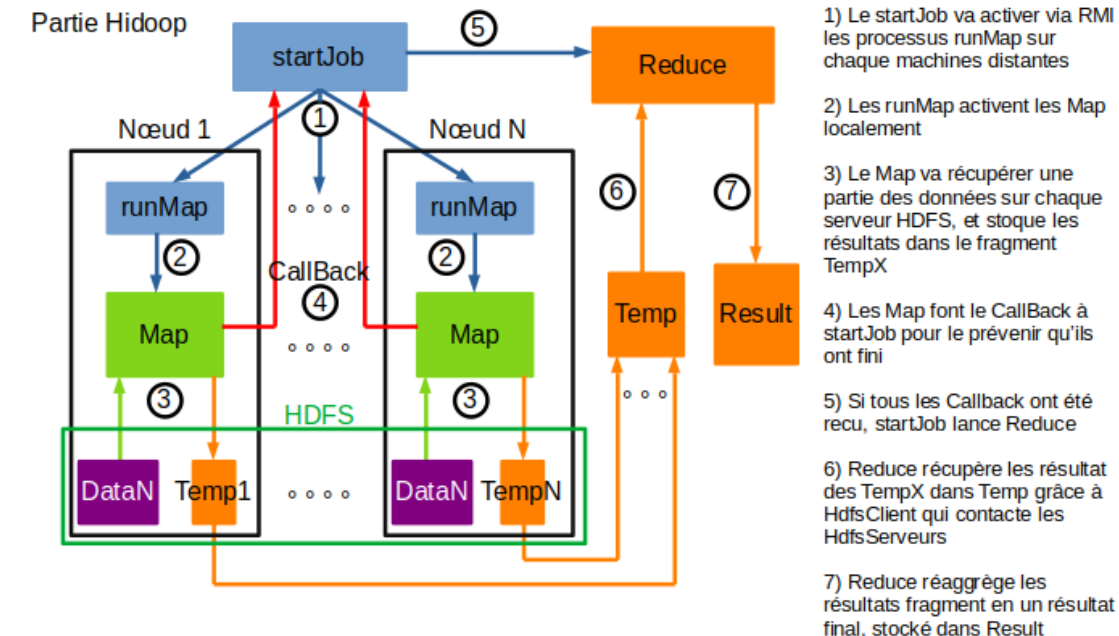


Figure 9: Architecture avec Commentaires