Rapport Projet Données Réparties
Partie Hidoop

Axel GRUNIG
Thomas GUILLAUD

Département Sciences du Numérique - ASR
2020-2021

Contents

List of Figures

1 Architecture

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

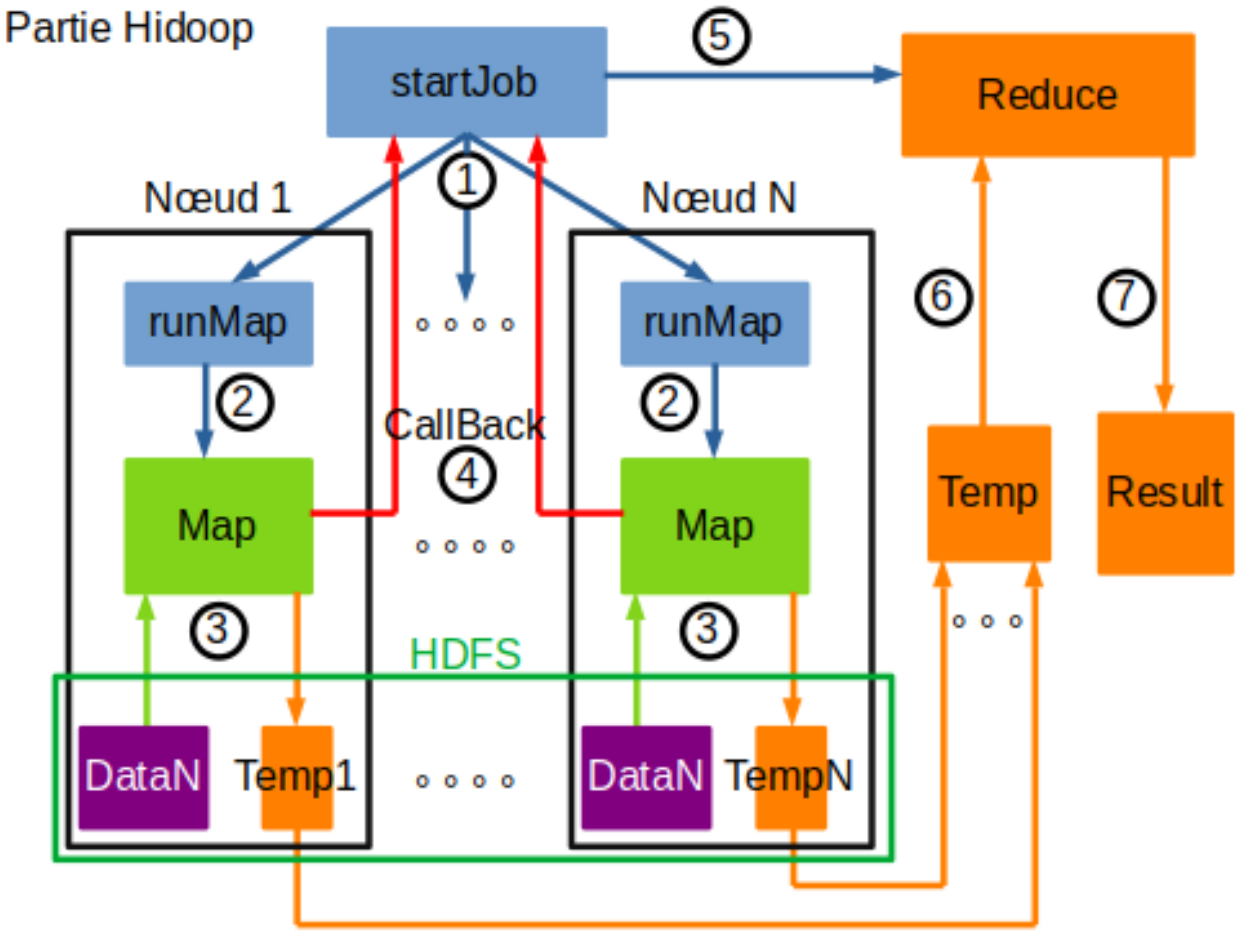

Figure 1: Architecture de la Partie Hidoop

Dans un premier temps, la classe qui implémente startJob va activer les méthodes rumMap de chaque Noeud via RMI.
Chaque Noeud 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 Noeud 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 recoit tous les callback, il effectue le Reduce sur le fichier Temp, qui est le regroupement de chaque fragment, et renvoie un résultart Result.

2 Présentation des Classes et Interfaces

2.1 Interface JobInterface et Classe Job

```java
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.

```java
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 (parte 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 implémentter la classe WorkerImpl. On notera aussi le fait que cette interface étends 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 étends UnicastRemoteObject pour les mêmes raisons (et implémente Worker bien évidemment). On y trouve la méthode runMap(...) qui prend en paramètre le traitememnt 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 fragement 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 prévénir que le Worker a terminé son exécution via l'objet CallBack ("cb.reveiller()").

2.3 Interface CallBack et Classe CallBackImpl

```java
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 distances.

```java
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 moniter 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 diffculté 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 reveiller() 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 signal 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 serveurs, 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 créé, il faut lancer MyMapReduce avec comme argument 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 Améliorations

Suite aux retours, nous avons ajouter du multithreading au niveau des Worker dans la classe WorkerImpl, afin qu'ils puissent continuer d'accepter des requêtes du client, malgré le fait qu'il soit en train d'en traiter une.

```
public class WorkerImpl extends UnicastRemoteObject implements Worker {

    static public int PORT = 2000;


    protected WorkerImpl() throws RemoteException { }

    @Override
    public void runMap(Mapper m, Format reader, Format writer, CallBack cb) throws RemoteException {
        reader.setFname(Project.getDataPath()+reader.getFname());
        writer.setFname(Project.getDataPath()+writer.getFname());
        Thread t = new Thread(new RunMap(m, reader, writer, cb));
        t.start();
    }

    public static void main(String args[]){
        try {
            WorkerImpl worker = new WorkerImpl();
            LocateRegistry.createRegistry(PORT);
            Naming.rebind( name: "//localhost:" + PORT + "/worker", worker);
        } catch (RemoteException | MalformedURLException e) {
            e.printStackTrace();
            System.exit( status: 1);
        }
    }
}
```

Figure 9: Classe WorkerImpl améliorée

C'est donc la classe RunMap qui est lancé dans un autre thread qui va effectuer les différentes opérations pour exécuter le mapper.

```java
class RunMap implements Runnable {
    private final Mapper map;
    private final Format reader;
    private final Format writer;
    private final CallBack cb;

    public RunMap(Mapper m, Format reader, Format writer, CallBack cb){
        this.map = m;
        this.reader = reader;
        this.writer = writer;
        this.cb = cb;
    }

    @Override
    public void run() {
        reader.open(Format.OpenMode.R);
        writer.open(Format.OpenMode.W);
        map.map(reader, writer);
        reader.close();
        writer.close();
        System.out.println("Fini Map :"+writer.getFname().substring(Project.getDataPath().length()));
        try {
            cb.reveiller();
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}
```

Figure 10: Classe RunMap

De plus, nous avons décider de laisser le multithreading au niveau du client hidoop (classe Job) afin de paralléliser l'appel des Worker. Cela leur permet donc d'être lancées en même temps, et donc d'effectuer leur tache en même temps.

Concernant la piste d'amélioration fournie par l'autre groupe, nous l'avons suivie puisqu'elle nous a parue très pertinente. Nous avons donc rajoutée un timer d'attente pour la terminaison des Workers qui permet de ne pas attendre indéfiniment un Worker qui aurait un problème de connexion ou un autre dans problème du même type. Nous avons arbitrairement choisi la valeur de départ de ce timer que nous avons fixé à 30s. Ce temps peut totalement être modifié si nécessaire pour les tests.

6 Performance

Les lignes commençant par "hidoop>" sont les commandes exécutées pour faire ce test de performance.
Taille des fichiers utilisés : big.txt $\rightarrow 120Mo$ $\quad$ $medium.txt \rightarrow 4Mo$

hidoop> hdfs -w medium.txt −−chunks-size = 1000000
Splitting file in 5 chunks...
100 %
medium.txt successfully saved.
−− time (ms) : 1167
hidoop> mmr -ip 172.22.232.226 medium.txt
Launching workers...
0 signaux en attente
Fin Attente
Reading file...
100 %
medium.txt-res successfully read to medium.txt-res.
Launching reduce task...
Reduce done : medium.txt-tot
−− time (ms) : 1306
Deleting file...
100 %
medium.txt-res successfully deleted.
hidoop> cmp_ref medium.txt
−− time (ms) : 375
hidoop> hdfs -w big.txt −−chunks-size=10000000
Splitting file in 13 chunks...
100 %
big.txt successfully saved.
−− time (ms) : 25292
hidoop> mmr -ip 172.22.232.226 big.txt
Launching workers...
0 signaux en attente
Fin Attente
Reading file...
100 %
big.txt-res successfully read to big.txt-res.
Launching reduce task...
Reduce done : big.txt-tot
−− time (ms) : 2533
Deleting file...
100 %
big.txt-res successfully deleted.
hidoop> cmp_ref big.txt
−− time (ms) : 3037

On constate donc qu'il devient intéressant d'utiliser la version MapReduce de l'application WordCount plutôt que sa version séquentielle à partir d'un fichier d'approximativement 150 Mo.

7 Annexe

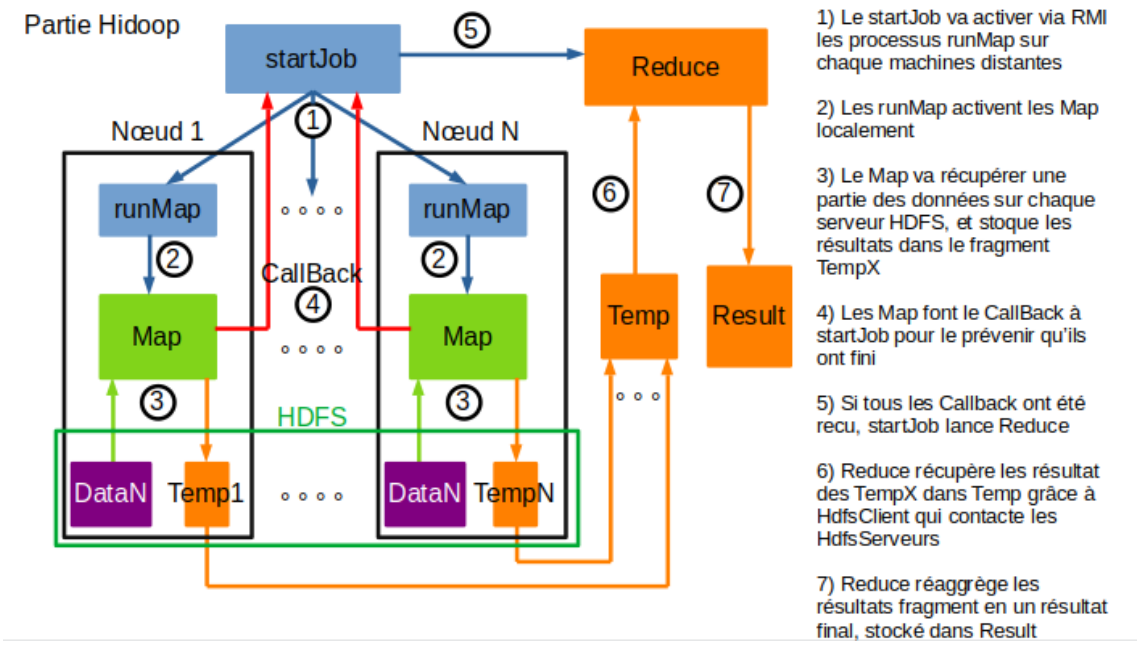

Figure 11: Architecture avec Commentaires