Rapport-sc-v1

EGELE Romain, PRIOU Cyrille, DIOCHOT David

Janvier 2018

Contents

1 Introduction

Hidoop est une plateforme dont le principe est de traiter une grande quantité de données en même temps en utilisant pleinement toutes les ressources possibles. Le noyau d'Hidoop est constitué d'une partie de stockage: HDFS (Hadoop Distributed File System), et d'une partie de traitement appelée MapReduce.
Dans ce cadre, le premier groupe avait réalisé une première version autour d'une architecture simpliste. Celle-ci était composée de démons qui réalisait des tâches Map, puis le reduce était effectué en une seule fois lors du retour des résultats de ces démons. Dans cette nouvelle version nous développerons cette architecture afin de pouvoir gérer une connexion/déconnexion dynamique des démons, d'effectuer un nombre variable de maps et de reduces, d'être plus à l'épreuve des pannes et d'optimiser l'encombrement du réseau. Cette architecture étant plus conséquente nous attendons des temps de calculs moins bons pour des petites tailles de fichiers.

2 Architecture

L'architecture réalisée est représentée par le diagramme de classe 1. En ce qui concerne le fonctionnement, pour que le service puisse s'exécuter il faut tout d'abord lancer le Job en tant que RessourceManager, alors seront lancés le Launcher qui a pour rôle de charger les démons lors de leur première connexion et le HeartBeatReceiver qui reçoit le signal émis par le HeatBeatEmitter que possède le démon et qui est lancé dès que celui-ci démarre. Puis nous devons lancer les différents démons que nous allons utiliser (un seulement par machine). Enfin depuis la même machine que celle où a été lancé le RessourceManager nous pouvons exécuter un Job au sens du sujet (une application MapReduce).

2.1 Le Job

La classe Job a deux aspects. D'un part une vision en tant que RessourceManager et d'autre part en tant "qu'ordonnanceur" des tâches lors de l'exécution d'une application MapReduce.

2.2 Le Launcher

Le Launcher est un objet distant instancié par Job en tant que RessourceManager. Cette classe sert à ajouter les démons lors de leur connexion à la liste que possède Job.

2.3 Le HeartBeat

Le Job en tant que RessourceManager utilise une liste des démons dynamique qui gère les connexions et déconnexions de ceux-ci. Périodiquement le HeartBeatReceiver possédé par le Job vérifie la réception d'un signal envoyé par les

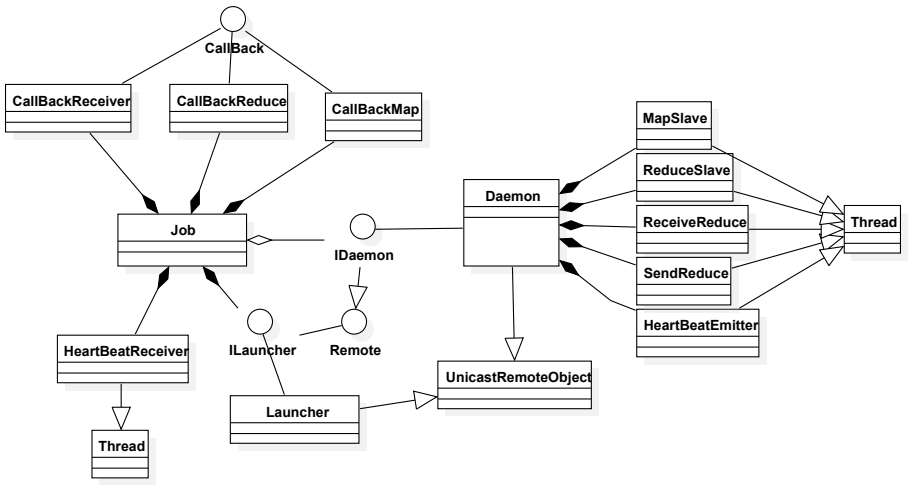

Figure 1: Architecture globale

différentes démons pour s'assurer de leur état de marche. La liste des démons est ainsi mise à jour constamment.

2.4 La gestion des mappers et reducers

Pour la gestion des mappers nous utilisons un format particulier, SocketAndKvFormat, ce format est utilisé en tant que "writer" par un mapper. Il nous permet lors de la fermeture d'envoyer l'ensemble (HashSet) des clefs connues par le mapper au Job. Ce format produit un fichier au format KV résultat du mapper (tel que le KvFormat). Une amélioration possible serait de ne plus passer par des fichiers mais d'utiliser une HashMap résultat du mapper, cela nous permettrait de réaliser des écritures mémoires moins coûteuses. Nous n'avons pas fait cette amélioration car nous aurions aimé pouvoir changer les paramètres "Format" des méthodes "runMap" et "runReduce" pour ajouter par exemple une méthode "getResult()" qui nous retournerait le résultat de l'écriture sous la forme d'une hashmap ou d'une liste de KV mais nous n'aurions alors plus respecté les interfaces fournies par le sujet.
Le Job génère une HashMap dans laquelle est associée une clé et l'adresse ip du démon qui est chargé de faire le reducer pour cette clé. Puis le Job envoie cette HashMap à tous les démons. Ainsi chaque démon sait à qui envoyer ses résultats de mapper et envoie donc au démon responsable les key-values nécessaires au reducer.
Avant d'attribuer et de lancer les reducers pour chaque serveurs, il faut être sur que tous les maps soient finis sur tous les démons mais également que chaque démon possède les clefs dont il est responsable : pour respecter cela, nous avons utilisé le principe de barrière (une barrière pour vérifier la fin des mappers et une barrière pour vérifier la fin des ReceiveReduce).

De même, nous avons utilisé une barrière avant de récupérer les résultats des reducers pour faire leur concaténation et écrire le résultat final puisqu'il faut être sur que tous les reducers soient finis sur tous les serveurs (nous comptons utiliser le service Hdfs pour faire la concaténation des fichiers).

3 Encombrement du réseau

Cette architecture permet de soulager l'utilisation du réseau. En effet une autre stratégie aurait pu être d'envoyer l'ensemble des KVs au Job à la fin des mappers. Puis le Job envoie les KVs aux démons attribués pour effectuer le reducer. Si on a n démons, m couple clés valeur, (dans cet exemple on suppose que tous les démons ont toutes les clés, il y a m clés) alors :

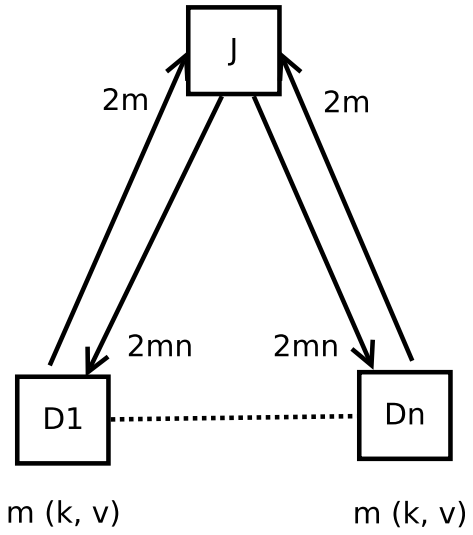

Figure 2: Schéma du réseau stratégie basique

Dans ce cas, dans le sens démon vers ressource manager chaque démon envoie $2 \times m$ éléments soit en tout $2 \times m \times n$. Dans le sens inverse le ressource manager envoie $2 \times m \times n$ éléments à chaque démon soit en tout $2 \times m \times n^2$

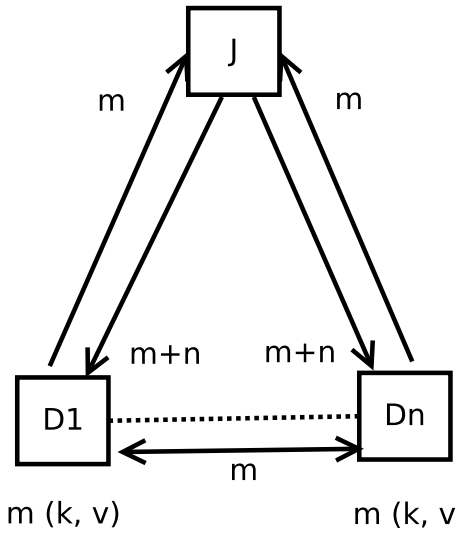

Figure 3: Schéma du réseau stratégie avancée

Dans ce cas, dans le sens démon vers ressource manager chaque démon envoie m éléments soit en tout $m \times n$. Dans le sens inverse le ressource manager envoie $m + n$ éléments à chaque démon soit en tout $(m + n) \times n$ mais en plus chaque démon envoie à tout les autres $m \times (n - 2)$ éléments en tout. Ils s'échangent donc entre les démons $m \times n - 2 \times n$

Au final le nombre d'élément échangé en tout est juste 2 fois inférieure à la première stratégie. Cependant sur une même ligne le nombre d'élément échangé est bien inférieur à celui de la première stratégie.

4 Tests

Pour évaluer le fonctionnement du service Hidoop, nous avons travaillé sur un fichier test.txt contenant (30 lignes):

```
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
```

```
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
```

Nous nous connectons sur trois machines différentes (tornade, luke et yoda), nous lançons le ressource manager sur la première machine (celle où nous allons lancer le job) :

```
regele@tornade:~/Cours2a/SysConc/proj-sc-v1/src\$ java ordo.Job
[] : Lancement du Ressource Manager...
[] : Host = 147.127.133.174
[] : RMI registry OK
[] : Launcher OK
[] : HeartBeat OK
[] : En attente de Daemons
```

L'ip affichée à la ligne "Host = 147.127.133.174" doit être la même que la variable inetAddress de la classe Job (on peut utiliser en ligne de commande : ip address show). Puis nous lançons un démon sur chaque machine.

```
regele@tornade:~/Cours2a/SysConc/proj-sc-v1/src: java ordo.Daemon
regele@yoda:~/Cours2a/SysConc/proj-sc-v1/src: java ordo.Daemon
regele@yoda:~/Cours2a/SysConc/proj-sc-v1/src: java ordo.Daemon
```

Puis nous voyons alors dans le terminal faisant référence au ressource manager la connexion de ces deux démons.

```
[] : Daemons connectés:
 IP = 147.127.133.199
 IP = 147.127.133.174
 IP = 147.127.133.193
```

Nous pouvons essayer de déconnecter par exemple le démon sur "tornade". Alors nous obtenons l'affichage:

```
[] : Daemons connectés:
 IP = 147.127.133.199
 IP = 147.127.133.193
```

Pour pouvoir tester l'application, nous devons simuler le rendu de Hdfs (soit séparer un fichier sur différentes machines) que nous intégrerons lors de la dernière étape. Nous divisons alors notre fichier test.txt en trois parties, chacune contenant 10 lignes et nous plaçons une partie par démon.

Sur la machine tornade (où a été lancé la commande java ordo.Job) nous lançons alors l'application MapReduce de comptage de mots donné en exemple :

```
regele@tornade:~/Cours2a/SysConc/proj-sc-v1/src: java application.MyMapReduce test.txt
[] Job initialisé !
Nombre de Daemons connectés : 3
Lancement du Mapper : 1
Lancement du Mapper : 2
Lancement du Mapper : 3
En attente de la terminaison des mappers...
Clefs réceptionnées : [dix, trois, six, sept, cinq, quatre, un, huit, deux, neuf]
KeyToDaemon :
{
dix=147.127.133.199,
trois=147.127.133.199,
six=147.127.133.199,
sept=147.127.133.174,
cinq=147.127.133.174,
quatre=147.127.133.174,
un=147.127.133.193,
huit=147.127.133.193,
deux=147.127.133.193,
neuf=147.127.133.193
}
En attente des receivers...
En attente des reducers...
[MyMapReduce] time in ms =511
[Count] time in ms =607
```

Un mapper se fait sur chaque machine et produit "test.txt-mapper", puis le ressource manager décide qui fera quel reducer. Cela fait référence à l'ensemble des "mots=IP" de l'extrait précédent. Dans notre cas nous avons toujours un reducer par démon. Comme nous l'avons dit précédemment tous les démons savent à quel démon sont attribués les clés. Puis ils échangent les clés et valeurs en produisant chacun un fichier "test.txt-reducerIN", les reducers se font et

produisent un fichier résultat dans leur répertoire "test.txt-reducerOUT". Voici ce que contiennent les fichiers :

```
# IP = 147.127.133.199
# test.txt
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix

# test.txt-mapper
dix<->10
trois<->10
six<->10
sept<->10
cinq<->10
quatre<->10
un<->10
huit<->10
deux<->10
neuf<->10

# test.txt-reducerIN
dix<->10
trois<->10
six<->10
dix<->10
trois<->10
six<->10
dix<->10
trois<->10
six<->10

# test.txt-reducerOUT
dix<->30
trois<->30
six<->30

# IP = 147.127.133.174
# test.txt
```

```
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix

# test.txt-mapper
dix<->10
trois<->10
six<->10
sept<->10
cinq<->10
quatre<->10
un<->10
huit<->10
deux<->10
neuf<->10

# test.txt-reducerIN
sept<->10
cinq<->10
quatre<->10
sept<->10
cinq<->10
quatre<->10
sept<->10
cinq<->10
quatre<->10

# test.txt-reducerOUT
sept<->30
cinq<->30
quatre<->30

# IP = 147.127.133.193
# test.txt
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
```

```
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix
un deux trois quatre cinq six sept huit neuf dix

# test.txt-mapper
dix<->10
trois<->10
six<->10
sept<->10
cinq<->10
quatre<->10
un<->10
huit<->10
deux<->10
neuf<->10

# test.txt-reducerIN
un<->10
huit<->10
deux<->10
neuf<->10
un<->10
huit<->10
deux<->10
neuf<->10
un<->10
huit<->10
deux<->10
neuf<->10

# test.txt-reducerOUT
un<->30
huit<->30
deux<->30
neuf<->30
```

5 Amélioration possible

Pour l'instant, nous lançons un reducers par démon. À terme il vaudrait mieux adapter le nombre de reducers plutôt par rapport au nombre de clés.

6 Utilisation de l'application

Avant de pouvoir lancer l'application il faut avoir l'architecture que génère hdfs (les différents fragments de test.txt dans les dossier correspondant à la machine qui le contient)

Nous devons commencer par lancer sur une machine le ressource manager avec la commande :

```
java ordo.Job
```

Nous devons ensuite lancer des démons sur différentes machines (éventuellement la même machine que le ressource manager). Pour lancer les demons, on tape la commande :

```
java ordo.Daemon
```

Enfin sur la première machine nous lançons l'application grâce à la commande.

```
java application.MyMapReduce test.txt
```