INTERGICIEL & SYSTÈMES CONCURRENTS

RAPPORT FINAL
HIDOOP V0

November 24, 2017

Alexys Dussier & Marie Elisabelar
ENSEEIHT
Informatique et Mathématiques Apppliquées

Sommaire

Introduction

Le but de cette version de Hidoop est de permettre d'appliquer des fonctions de la forme map/reduce sur des fichiers stockés en réparti sur des serveurs distants HDFS. Pour cette version, on considère qu'une seule application sera exécutée à la fois.
On ne prend pas en compte la tolérance aux pannes.
Il n'y aura ni de fonction shuffle ni de combiner, on se contentera des gérer la fonction reduce.
On considérera également que chaque daemon lancé possède un fragment du fichier que l'on souhaite traîter.
Après une première version évaluée par l'autre binôme de notre groupe, nous avons maintenant une application fonctionnelle et s'appuyant sur leur implantation d'HDFS.

1 PAR OÙ COMMENCER ?

Le nombre de fichiers rendus étant relativement conséquent, nous précisons ici l'intéret de quelques-uns d'entre eux :

- toto.txt : un fichier test pour HDFS et Hidoop
- deploiementServeurs.sh (dans src) et lanceurServeur.sh (dans src/config) : des fichiers de déploiement pour lancer les serveurs HDFS en réparti
- ServiceHidoop (dans src/application) : programme permettant d'utiliser l'ensemble de l'application (très friendly)
- ServerRunner et DaemonRunner (dans src/application) : des classes pour lancer des Serveurs et des Daemons, utilisés dans ServiceHidoop
- mapreduceLanceur.sh (dans src): fichier de script pour lancer le MapReduce du client (ici MyMapReduce) à ne sourtout pas changer de place, utilisé dans ServiceHidoop

Nous vous conseillons de commencer par tester l'application avec ServiceHidoop. Au préalable, il est nécessaire de lancer "rmiregistry 1199 &" dans le dossier /bin pour que la partie Hidoop puisse fonctionner.

2 CÔTÉ CLIENT - JOB

Lorsque l'utilisateur souhaite executer un MapReduce, il fournit les fonctions map et reduce dans une classe MapReduce. Dans l'exemple, il s'agit de MyMapReduce. Le main crée un Job et lance Job.sartJob(), qui s'occupe de lancer les maps, récupérer et concaténer les résultats, faire le reduce et enfin écrire localement les résultats sur un fichier.

2.1 L'algorithme

Le coeur du programme se trouve dans StartJob(). On veut tout d'abord créer un Format input qui, pour chaque deamon, pourra lire le fragment correspondant au fichier d'entrée. Le type du format dépend de ce qui a été spécifié en entrée. Les Formats utilisés sont ceux implantés par le groupe HDFS.
On crée de même à l'avance les formats inter, output et resReduce qui permettront respectivement d'appliquer les maps sur chaque fragment, de récupérer la concaténation des maps, et de sauvegarder le résultat du reduce.

On veut ensuite récupérer la liste des daemons via un RMIRegistry devant être lancé au préalable sur le port 1199. Nous avons utilisé le port 1199 car le port 1099 était souvent utilisé par d'autres utilisateurs, pouvant créer des conflits.
On crée un callback (voir section 4).
Pour chaque Daemon, on envoie une copie des fonctions map et reduce, une copie du Format d'entrée et du format $inter$ (déjà suffixée par "-inter") suffixées par l'indice du Daemon et le CallBack cb. Les deamons vont lire leur fragment, correspondant à $input$, puis exécuter et écrire les résultats de Map en dur en local du Daemon via le format $inter$. Pour que ces opérations aient lieu en simultané, nous devons utiliser des Threads pour lancer ces différents maps. Ainsi nous utilisons une classe MapRunner héritant de Thread, dont nous créons une nouvelle instance pour chaque Daemon, qui sera lancée immédiatement.
Une fois cette opération terminée, les Daemons signifient au client que leur travail est terminé via le callback cb.
Le Job attend que tous les daemons aient fini leur travail grâce à une fonction du callback.
On utilise HDFS (la fonction read) pour concaténer tous les résultats intermédiaires locaux aux Daemons en un fichier intermédiaire local au client (suffixé par "-res").
Une fois cette opération terminée, on prend soin de fermer le format afin que les résultats soient écrits dans un fichier. Il nous faut également fermer le format car on s'en servait avant en écriture, on va maintenant avoir besoin de lire ce qu'il y a à l'intérieur, on l'ouvre donc en lecture.
À l'aide d'un ultime format, on peut enfin appliquer le reduce sur la concaténation puis écrire le résultat dans un fichier local au client suffixé par "-final".
On écrit le résultat dans un fichier de sortie. Cette dernière action a lieu quand on ferme le format.

2.2 Number Of Maps

On a considéré qu'il y avait un fragment de notre fichier sur chacun des Daemons lancés. Ainsi il y avait autant de maps à lancer que de Daemons. Nous avons pris par défaut 3 Daemons.

3 CÔTÉ SERVEURS - DEAMON

Ce sont les programmes qui tournent sur les différentes machines contenant les fragments des fichiers. Les Daemons doivent se connecter à l'annuaire de noms du RMI lorsqu'ils sont lancés. Chaque Daemon a un nom passé en paramètre. Nous fonctionnons actuellement en local et utilisant le port 1199. Ainsi les Daemons se connecterons à l'annuaire avec l'adresse "//localhost:1199/nomDaemon". Le nomDaemon signifiant sous quel nom est identifié le Daemon à l'intérieur de l'annuaire, et non pas son adresse dans le système de fichier. Le rôle des Daemons est de lancer runMap(). Pour cela ils vont ouvrir le format du reader et du writer passé en paramètre. Ils exécutent ensuite la fonction map passée en paramètre sur le reader, et enregistrent les résultats en local sur le writer. Ils ferment ensuite le reader et le writer. Et enfin confirment la fin de de leur éxécution via le callback.

4 CALLBACK

Le CallBack sert à informer Job que tous les Daemons ont bien fini leur travail. Il contient deux procédures:
confirmFinishedMap qui permet aux Daemons de dire qu'ils ont fini leur travail.
waitFinishedMap qui permet au Job d'attendre tous les Daemons.
CallBack utilise un Sémaphore pour éviter les accès simultanés à la ressource et éviter l'attente active.

5 CORRECTIONS APRÈS ÉVALUATION

Deux problèmes majeurs ont été soulevés par l'autre binôme :

- l'oubli de la parallélisation des maps en utilisant les Threads.
- le manque de verbosité de l'application.

Pour le premier nous avons ajouté la classe MapRunner qui permet de lancer les maps sur les Daemons en parallèle.
Pour le deuxième nous avons ajouté un message destiné à l'utilisateur au début et à la fin de chaque étape principale de l'application. Cela nous été très utile par la suite.

6 PROBLÈMES RENCONTRÉS

Le principal défaut de cette application est que les Daemons sont lancés sur une seule et même machine. Nous n'avons pas réussi à créer un script permettant de lancer les différentes applications sur différentes machines, nous sommes donc resté en local. En effet, l'élaboration d'un script bash permettant de tout lancer en réparti s'est avéré être très fastidieuse, et nous n'avons pas réussi à obtenir une version fonctionnelle. Un gros frein à cette avancée est qu'avant de pouvoir créer le script, nous avons passé beaucoup de temps avec l'autre équipe à réussir à faire fonctionner leur propre applicaiton en réparti, car MapReduce dépend fortement d'HDFS. Tous ces efforts n'ont pas été vains car nous disposons d'un script deploiementServeurs.sh qui permet de lancer HDFS en réparti, ce qui nous aidera sans doute pour HDFS V1.

Un autre problème directement lié au précédent a été qu'il a fallu s'adapter à l'implémentation de HDFS de l'autre binôme, très souvent changeante au fur et à mesure que les bugs étaient trouvés et corrigés. À force d'efforts communs nous avons réussi à surmonter ce problème.

Nous avons été également confrontés à un bug difficile à comprendre. Il nous fallait lancer l'annuaire RMIRegistry dans le dossier /bin, contenant les classes compilées, l'application ne fonctionnant pas sinon.

7 USAGE

Nous avons décidé de créer un programme java, nommé ServiceHidoop, pour lancer les différentes parties de l'application Hidoop. Ce programme, bien qu'il ne permette qu'un fonctionnement local de notre application, permet de lancer de manière conviviale et sure à la fois la partie HDFS et la partie Hidoop.
Pour la partie HDFS, absolument tout est pris en charge par le programme, que ce soit le lancement des Serveurs ou la commande du client. Pour la partie Hidoop, il faudra au préalable lancer manuellement le rmiregistry sur le port 1199, lancer le programme, puis exécuter le programme contenant le MapReduce manuellement.

Lors de la conception de cet exécutable, il nous est apparu qu'il était impossible de lancer directement MyMapReduce car nous n'étions pas dans le bon répertoire. Il est en effet nécessaire d'être dans /src. Nous aurions pu utitliser une commande bash, qui serait exécutée

en java. Malheureusement, il n'est pas possible d'utiliser la commande cd et changer de répertoire car dans l'environnement eclipse, le repertoire courant depuis lequel on exécute les programmes ne pouvant être modifié. Nous avons donc "contourné" les restrictions d'eclipse en appelant un fichier bash depuis le programme java, qui a son tour exécute MyMapReduce correctement après avoir changé de répertoire. Si cette technique est un peu compliquée, elle nous a néanmoins permis de centraliser toutes les étapes d'utilisation dans un unique fichier.

8 CONCLUSION

Ainsi nous avons créé une application permettant d'éxécuter une fonction de la forme map/reduce sur un fichier réparti sur des serveurs HDFS en utilisant RMI. Elle fonctionne pour le moment uniquement en local.