Rapport final
Map Reduce v.0

Martin Dubois / Yannis Mehidi

Architecture

Ordo.Daemon_dataNode : Serveur RMI tournant sur toutes les machines, il peut lancer les méthodes Map(), Reduce(), envoyerVers() et recevoir().

Ordo.Job : Le client RMI qui appelle les différentes méthodes des Daemon_dataNode à distance.

Ordo.SlaveMap, Ordo.SlaveEnvoyerVers, Ordo.SlaveRecevoir : Threads utilisés par Job permettant l'envoi de requêtes RMI en parallèle.

Formats.KVFormat : Permet la lecture et l'écriture de fichiers de type KV, c'est à dire un fichier où chaque ligne contient une clé et sa valeur associée.

Formats.LineFormat : Permet la lecture et l'écriture de fichiers de type ligne, c'est à dire un texte.

Config.Project : Contient le PATH du projet, utilisé notamment par les formats pour savoir où aller chercher les fichiers.

Ordo.InterfaceMapReduce : Permet de lancer un interpréteur de commande qui se charge de lancer les daemons à la place de l'utilisateur. Cet interpréteur permet l'utilisation de MapReduce.

Ordo.Hidoop_lancement : Permet de lancer un interpréteur de commande qui se charge de lancer les daemons, clients et serveurs à la place de l'utilisateur. Cet interpréteur combine MapReduce et Hdfs et permet leur utilisateur conjointe.

Ordo.Callback : Permet de savoir quand on a fini d'exécuter une méthode remote mais il n'est pas utilisé car on utilise la méthode `join()` pour le remplacer.

Opérations essentielles

Daemon_dataNode : (Serveur RMI)

- runMap(MapReduce m, Format reader, Format writer, CallBack cb) ()

Permet de lancer map en local en suivant les instructions du 'MapReduce' fourni en argument. Le fichier est lu en suivant les instructions du 'reader' et le résultat est écrit en suivant celles du 'writer'.

- runReduce(MapReduce m, Format reader, Format writer, CallBack cb) ()

Permet de lancer reduce en local en suivant les instructions du ‘MapReduce’ fourni en argument. Le fichier est lu en suivant les instructions du ‘reader’ et le résultat est écrit en suivant celles du ‘writer’.

- envoyerVers(String addr,int port,String name) ()

Permet d’envoyer un fichier de nom ‘name’ en utilisant des sockets à l’adresse ‘addr:port’. Cette méthode est utilisée pour envoyer le résultat du RunMap() vers le reducer.

- recevoir(int nbData,int port,String fname) ()

Permet de recevoir un fichier fragmenté en ‘nbData’ morceaux et de le reconstituer en un fichier ‘fname’ en utilisant des sockets sur le port ‘port’.

Job : (Client RMI)

- setInputFname(String fname)

Permet de saisir le nom du fichier que l’on a fragmenté et que l’on souhaite traiter (attention comme mapReduce traite plusieurs fichiers les fichiers réels doivent s’appeler ‘fname.i’ où i est le numéro du fragment).

- setDataNode(Hashtable<Integer, Inet4Address> t)

Permet de donner la liste des adresses des dataNodes et leurs fragments associés (le i vu dans la fonction setInputFname().

- setReducer(String hostname)

Permet de donner l’adresse du Reducer.

- startJob (MapReduce mr)

Permet de lancer un nouveau Job en utilisant les méthodes de MapReduce du MapReducer fourni en arguments. Attention ! Avant de lancer un job il faut donner le reducer, les dataNodes et un inputFname.

Format : (Permet l'écriture/lecture de fichiers)

- `setFname(String fname)`

Permet de donner le nom du fichier à lire/écrire, on peut d'ailleurs mettre le PATH du fichier avant son nom : ../data/test.txt par exemple. Attention ! Il faut utiliser cette méthode avant toute utilisation des formats sinon on ne sait pas où aller chercher le fichier et quel est son nom.

- `open(OpenMode mode)`

Permet d'ouvrir un fichier en lecture : `open(OpenMode.R)` ou en lecture `open(OpenMode.W)`.

- `close()`

Permet de fermer un fichier précédemment ouvert, à utiliser après avoir fini de traiter un fichier dans son ensemble .

- `read` ()

Permet de lire une ligne du fichier dont le nom est précisé avec la fonction `setFname()` et de renvoyer le KV lu.

- `write(KV record)`

Permet d'écrire 'record' dans une ligne du fichier dont le nom est précisé avec la fonction `setFname()`. Tant que le fichier n'est pas fermé par `close()` cette écriture ajoute 'record' à la fin du fichier.

Points délicats

- Gestion d’exceptions pour savoir si l’exécution de map reduce se passe correctement :

Des exceptions peuvent être levées un peu partout dans Job et donc bloquer l’éxécution en cas d’erreur mais dans les cas où l’on invoque des Threads avec la méthode start() ou run() on ne peut pas retourner d’exceptions pour notifier Job qu’une erreur s’est produite.

Nous n’avons pas trouvé de solution à ce problème donc si une erreur se produit dans un Thread un message d’erreur s’affiche pour que l’utilisateur se rende compte qu’une erreur s’est produite, cependant l’exécution continue.

- Le lancement des daemons à distance : (ssh user@machine commande)

Ce point a été difficile car il faut lancer les commandes bash depuis java et la syntaxe n'était pas très claire.

Ce problème a été résolu entièrement en utilisant RunTime.exec() et en faisant un ssh user@machine 'cd PATH && java Daemon' au lieu d'un simple ssh user@machine java Daemon.

- La combinaison de Hdfs et MapReduce :

La combinaison est délicate car les deux systèmes ont leurs spécificités,par exemple Hdfs renvoie des Inet4Adress et MapReduce prend des Hostname sous forme de String.

Ce problème a été résolu en faisant les conversions nécessaires dans Hidoop_lancement ainsi qu’en changeant quelques parties de code de Hdfs et MapReduce quand une simple conversion n’est pas possible.

- Le lancement de plusieurs commandes à la suite :

Il arrive qu'il ne soit pas possible de lancer un nouveau MapReduce à la suite d'un autre, le programme plante au niveau de la reception. Nous avons essayé de nombreuses méthodes pour résoudre ce bug.

Ce bug était causé par la socket serveur du reducer qui n'était jamais fermé, ce bug a été corrigé.

Performances

Pour un fichier test.txt0 fragmenté en 81 parties contenant 728.597 mots (4.345.246 caractères) il faut :

- 4944 ms avec 1 DataNode (2608/4178/4604/8533/4799)
- 5224 ms avec 2 DataNodes (5707/8672/2540 /4585/4618)
- 1678 ms avec 4 DataNodes (3450/1394/1373/777/1396)

Pour un fichier filesample.txt fragmenté en 4 parties contenant 79.784 mots (431.256 caractères) il faut :

- 3733 ms avec 1 DataNode (3248/4219)
- 1798 ms avec 2 DataNodes (2323/1273)
- 375 ms avec 4 DataNodes (427/323)

Pour vérifier que les résultats sont corrects nous avons utilisé le site https://www.browserling.com/tools/word-frequency qui compte le nombre d'occurences des mots d'un texte. Le résultat n'est pas exactement le même car le MapReduce fourni compte (,and), (and) et ("and) comme trois mots différents. Cependant les résultats ont un ordre de grandeur similaire, on peut donc dire que le résultat est correct, pour le fichier de petite taille (filesample.txt) et celui de grande taille (test.txt0).