Rapport Hidoop HDFS V0 :
Systèmes Concurrents et Intergiciel

Amélie Canin Marie Pelissier Combescure

Sciences du Numérique - Image et Multimédia

Décembre 2018

Sommaire

1 HDFS Server

Nous avons écrit cette classe pour modéliser le comportement des différentes machines vis à vis des fragments de fichier qu'elles vont recevoir.

Cette classe HdfsServer ne contient qu'une seule méthode principale.

1.1 Socket et SocketServer

Toutes les machines du réseau vont avoir le même comportement. En effet, elles recoivent chaque un port sur lequel elles ouvrent un `ServerSocket`.

Ensuite, après avoir accepté la communication, dans une boucle `while(true)`,un Socket est ouvert, sur lequel sera envoyé le fichier et la commande à effectuer. On ouvre également un canal dit de lecture :
`ObjectInputStream ois = new ObjectInputStream(sock.getInputStream())` et un canal dit d'écriture :
`ObjectOutputStream oos = new ObjectOutputStream(sock.getOutputStream())`.

1.2 Traitement des commandes

Il y a trois commandes aux choix que nous avons traité sous la forme d'un `switch ... case` :

- Commande `CMD_READ` : on écoute le ficheir sur `ois.readObject()` et on le transforme sous la forme d'un fichier du type `FileReader` nommé `fr`. On utilise un `BufferedReader(fr)` nommé `buff` pour mettre dans un variable un le message à envoyer. On utilise alors la méthode `buff.readLine()`. On envoie le message sur `oos.writeObject(message)`.

- Commande `CMD_WRITE` : de même, on écoute le fichier sur `ois.readObject()`. Mais ici, on le transforme en un fichier de type `FileWriter` nommé fw. Puis on receptionne le message sur `ois.readObject()`, que l'on doit écrire dans notre fichier `fw` sur . Ensuite on utlise la méthode :
`fw.write(strW,0,strW.length()-1)` qui existe déjà dans la classe `FileWriter`. La variable `strW` correspoind au message à écrire.

- Commande `CMD_DELETE` : de même, on écoute le fichier que l'on transforme ici en `File` nommé `f`. Et on utilise la méthode `delete()` qui existe déjà dans la classe `File`.

1.3 Choix

Après chaque ouverture d'un `Socket` ou `ServerSocket`, on ferme les `ObjectOutputStream(sock.getOutputStream())` et `ObjectInputStream(sock.getInputStream())` associés.

En plus de cela, nous avons rajouté des afficheurs de texte en début et à la fin de chaque "demandes", ainsi nous pouvons voir le bon déroulement de notre code lors de son éxécution.

Bien sûr, certaines méthodes utilisées peuvent soulever des exceptions, notamment celle ci : `ClassNotFoundException`. Nous avons aors écrite notre méthode principale sous la forme d'un unique bloc du type `try { ... } catch(ClassNotFoundException e){ ... }`, au lieu de faire un `try { ... } catch(Exception e){ ... }` à chaque fois que l'on appelle une méthode.

2 HDFS Client

Le HDFS Client correpond à l'ordinateur de l'utilisateur qui veut traiter sa demande sur plusieurs serveurs. Ainsi pour un fichier donné, nous allons le fractionner et l'envoyer aux differents serveurs.

Nous avons 3 fonctions différentes qui correspondent aux 3 requetes possibles (lecture, écriture, suppression).

2.1 Sockets

Au début de chaque procédure (`HdfsRead`, `HdfsWrite`, `HdfsDelete`) nous ouvrons des sockets pour pouvoir comuniquer avec les différents serveurs. Nous choisissons donc des numéros de ports différents pour chaque serveur.
On ouvre ensuite un canal de lecture (`ObjectInputStream`) et un canal d'écriture (`ObjectOutputStream`) pour chaque serveur.
Une fois les demandes traitées, nous pensons bien à fermer ces différents canaux (par exemple `oos.close();`)

2.2 Trois fonctions principales

2.2.1 Hdfs Delete

La fonction `HdfsDelete` est appelée lorsque l'utilisateur veut supprimer un fichier.

Une fois les sockets ouverts, on envoie à chaque serveur la commande `CMD_DELETE` pour que le serveur sache quoi faire et on envoie également le nom du fichier à supprimer. Grâce au nom du fichier, le serveur saura quel fichier supprimer et il ne supprimera que le fragment qu'il a reçu.
Par *envoi de la commande*, on entend que nous allons écrire sur le canal d'écriture du socket : `oos1.writeObject(Commande.CMD_DELETE);`
De même pour le nom du fichier.

2.2.2 Hdfs Read

La fonction `HdfsRead` est appelé lorsque l'utilisateur veut lire les résultats d'un traitement sur un fichier. Il concatène alors les résultats des différents serveurs sur chaque fragment.

On commence à créer un nouveau fichier à partir de `hdfsFDestFname`. De ce fichier on crée un `FileOutputStream` :
`FileOutputStream fos = new FileOutputStream(fichier);`
C'est ce fichier `fos` qui va nous permettre d'écrire les différents résultats obtenus en les concaténant.

Un fois les sockets ouverts, nous envoyons aux serveurs la commande `CMD_READ` et le nom du fichier que l'on veut lire.
Ensuuite, pour chaque serveur, nous lisons sur son canal de lecture (`readbytes = ois1.read(buffer)`) et tant qu'il y a des lignes à lire, nous écrivons le résultat dans le fichier `fos`. Puis nous passons au serveur suivant.

2.2.3 Hdfs Write

La fonction `HdfsWrite` permet de traiter le fichier voulu sur différents serveurs. C'est à ce moment que nous fractionnons le fichier pour en envoyer un bout à chaque serveur.

Comme ci-dessus, nous commençons par créer un nouveau fichier : `File fichier = new File(localFSSourceFname);`
On crée ensuite une fichier `FileReader` (`fr`) et un BufferedReader.
Pour fractionner le fichier, nous avons choisi de compter le nombre de lignes et d'envoyer à chaque serveur un fichier contenant $\frac{nbLigne}{nbServeurs}$ lignes (le dernier serveur récupère également les lignes restantes).
Une fois le nombre de lignes compté, nous ajoutons dans un nouveau `string` les lignes du fichier correspondant à chaque serveur. Nous envoyons ensuite la commande `CMD_WRITE` ainsi de le nom du fichier, son format et le fragment du fichier correspondant au serveur.

2.3 Choix

- Pour l'instant nous avons choisi de n'envoyer le fichier qu'à seulement 2 serveurs. Dans les prochaines versions, nous augmenterons ce nombre.
- Pour fractionner, nous avons choisi de compter le nombre de lignes puis d'envoyer à chaque serveur le même nombre de lignes. Cependant cette solution n'est pas optimale car nous parcourons 2 fois le fichier et si le fichier est conséquent nous perdons beaucoup de temps.
- Dans nos différentes fonctions les exceptions sont attrapées mais pour l'instant il n'y a pas encore de message d'erreurs correpondant à la nature des différentes exceptions, seul est affiché un message permettant de savoir dans quelle fonction il y a eu une erreur.

3 Formats,FormatReader et FormatWriter

3.1 Choix

Nous avons écrit la classe `FormatImpl` qui implémente l'interface `Formats`, qui elle même étend les interfaces `FormatReader et FormatWriter`.
Voici nos choix par rapport à notre code java :

- Au lieu de faire deux classes pour les deux types différents : `KV ou Line`, et qui réalisent la même interface, nous avons écrit une unique classe `FormatImpl` ;
- Des variables gloables :
  - `private FileReader fichierLecture` : le fichier sur lequel on travaille transformé en fichier du type `FileReader` ;
  - `private BufferedReader buffer` ;
  - `private FileWriter fichierEcriture` : le fichier sur lequel on travaille transformé en fichier du type `FileWriter` ;
  - `private String nameF` : nom du fichier ;
  - `private ArrayList<String> listeLigne = new ArrayList<>()` : chaque ligne du texte est mise dans une case du tableau ;
  - `private Type fmt` : format du fichier ;
  - `private int index = 0` : index de listeLigne ;
  - `boolean oLect = false` : booelan qui indique si la lecture est autorisée ;
  - `boolean oEcriture = false` : booelan qui indique si l'écriture est autorisée ;
- Deux nouvelles méthodes : `getFmt()` et `setFmt(Type fmt)`.

3.2 Réalisation des l'interfaces

3.2.1 Méthode de l'interface `Format`

Dans l'interface `Format` il y a cinq méthodes à réaliser :

- La méthode `open(OpenMode mode)` : le paramètre `mode` permet de savoir s'il on veut ouvrir le fichier en mode écriture ou lecture.
  Si c'est en mode écriture, on a alors `oEcriture = true`, on rend l'ouverture en mode écriture possible avec `setWritable(true)`. On crée alors le fichier `fichierEcriture = new FileWriter(fichier, true)` où l'écriture est possible.
  Sinon on est en mode lecture, on a alors `oLect = true`. On rend l'ouverture en mode lecture possible avec `setReadable(true)`. Ensuite on crée

le fichier `fichierLecture = new FileReader(fichier)` où la lecture est possible. On crée également un `BufferedReader` qui permet de remplir le tableau `listeligne` avec les ligne de fichierLecture. Certaines méthodes utilisées peuevnt soulever des exceptions, on utilise alors la structure du `try{ ... } catch (Exception e){ ...}` .

— La méthode `close()` : On ne peut fermer un fichier que s'il a été précédement autorisé à la lecture ou à l'écriture (`oLect = true` ou `oEcriture = true`). Dans ce cas, on utlise la méthode `close()` de la classe `FileReader` et `FileWriter`. La méthode `close()` peut soulever des exceptions, on utilise alors la structure du `try{ ... } catch (Exception e){ ...}` .

— La méthode `getIndex()` : renvoie index en cours, c'est à dire la ligne qui est en cours de lecture ;

— La méthode `getFname()` : renvoie le nom du fichier ;

— La méthode `setFname(String fname)` : change le nom du fichier.

3.2.2 Méthode de l'interface `FormatReader`

De cette interface, nous avons réalisé la méthode : `public KV read()`. Tout d'abord, on ne peut lire un texte que si le boolean `oLect` est à `True`, sinon l'opération demandée est interdite, et si l'index de lecture `index` n'est pas à la fin du texte.

Si le texte est du type `LINE`, on revoie alors un `KV` qui a pour clé : le numéro de la ligne en cours de lecture et en valeur : la ligne correspondante.

Sinon il est du type `KV`. On sait que la ligne est composée de deux parties séparées par le `KV.SEPARATOR`. Ainsi avec la méthode `split` de la classe `String` on peut récupérer ces deux parties. On retourne alors la `KV` associcée.

3.2.3 Méthode de l'interface `FormatWriter`

Enfin de cette interface, nous avons réalisé la méthode `public KV write(KV record)`. Comme la métode précédente, on ne peut écrire dans un texte que si le boolean `oEcriture` est à `True`, sinon l'opération demandée est interdite.

Si le texte est du type `LINE`, on écrit dans le fichier `fichierEcriture` la valeur du paramètre : `record.v`, qui est juste une ligne à ajouter au texte.

Sinon il est du type `KV`. Il faut alors écrit la ligne sous la forme : `record.k + KV.SEPARATOR + record.v`.

La méthode `write` peut soulever des exceptions, on utilise alors la structure du `try{ ... } catch (Exception e){ ...}` .