RAPPORT

Projet Données Réparties
HDFS v0

Groupe L1

ALAUZY Estelle
DAVIN Marie
RONGIER Chloé

Sommaire

Introduction

L'objectif de ce projet est de gérer des traitements d'un grand volume de données. Pour ce faire, on souhaite découper les données en fragments stockés sur des machines de traitement. Le traitement des données peut ainsi se faire en parallèle sur plusieurs machines. Les résultats du traitement sont renvoyés et concaténés pour produire un résultat final.

Nous avons pour l'instant réaliser une version v0 du service HiDFS (Hidoop Distributed File System). Elle fournit les services de base en termes de gestion des accès à des fichiers fragmentés, et d'ordonnancement pour les tâches map-reduce.

Architecture générale

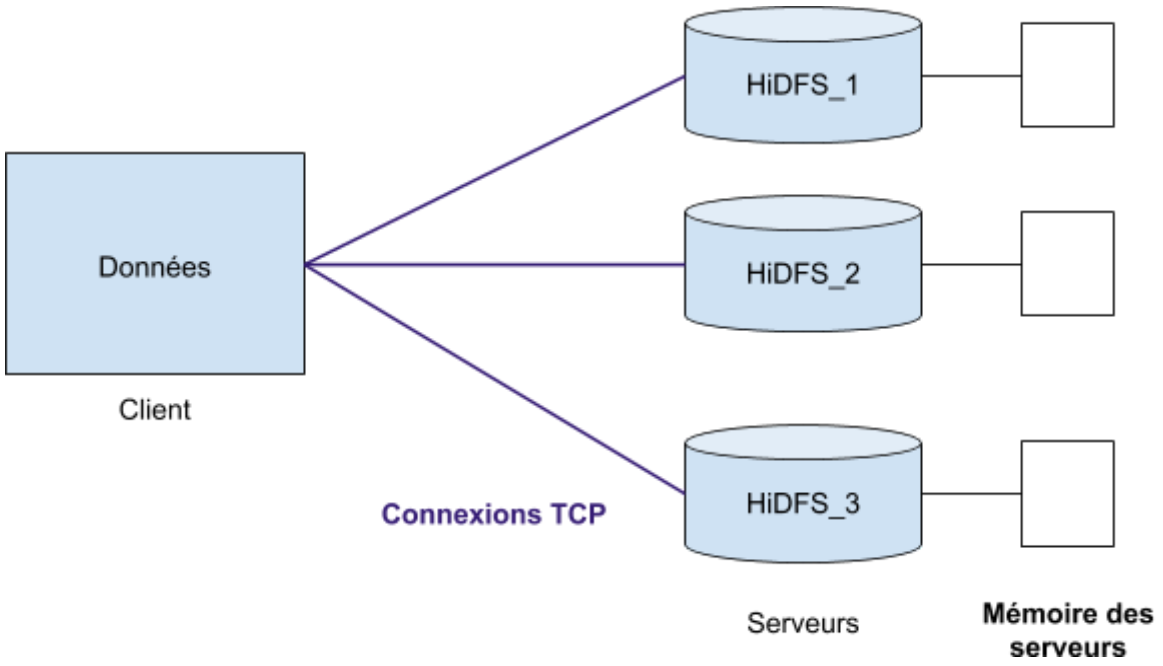

Schéma 1 : Architecture du système

I - HdfsClient

La classe HdfsClient est utilisée pour représenter un client souhaitant faire exécuter des calculs sur des fichiers volumineux. Le but de ce client est d'envoyer ces données aux différents serveurs auxquels il a accès, afin que ces derniers effectuent les calculs voulus. Il a aussi besoin de pouvoir récupérer les données traitées. Le Client va donc disposer de 3 commandes :

- Write : écrire un fichier dans HDFS. Le fichier est lu sur le système de fichiers local, découpé en fragments et les fragments sont envoyés pour stockage sur les différentes machines.
- Read : lire un fichier à partir de HDFS. Les fragments du fichier sont lus à partir des différentes machines, concaténés et stockés localement.
- Delete : supprimer les fragments d'un fichier stocké dans HDFS.

II - Communication entre le Client et les Serveurs

Nous avons utilisé les sockets en mode TCP pour implanter la communication entre HdfsClient et HdfsServer. En effet, une connexion RMI n'aurait pas été optimale dans l'optique qu'une grande quantité de données soit à transmettre. Chaque interaction entre HdfsClient et HdfsServer est initiée par l'envoi d'un message spécifiant la commande à exécuter par HdfsServer.

III - HdfsServer

La classe HdfsServer implémente le comportement du serveur Hdfs. Ce dernier reçoit une commande du Client et applique la méthode correspondant à la commande envoyée. Il peut recevoir 3 commandes :

- Write : le Serveur reçoit alors un fragment du fichier et doit l'enregistrer en mémoire (de façon à pouvoir le retrouver).
- Read : les fragments du fichier sont renvoyés au Client à partir de la mémoire.
- Delete : supprimer les fragments d'un fichier de la mémoire du serveur.

Implémentation

I - Formats

Nous avons implémenté les 2 formats demandés. Nous avons généralisé les traitements des deux formats en considérant que les fichiers texte (LINE) étaient en fait constitués de lignes de KV en attribuant pour chaque ligne le numéro de la ligne comme clef et le contenu de la ligne comme valeur.

Nous avons donc naturellement choisi de traiter les fichiers ligne par ligne. Pour cette raison, la taille des fragments est un nombre de lignes, l'index auquel on est dans le fichier en mode lecture correspond à la ligne, et les fragments sont des listes de lignes.

Nous avons donc créé une classe abstraite "AbstractFormat" afin de traiter un certain nombre de méthodes de la même façon pour les deux formats (open, close, getIndex, getFname et setFname). Nous avons cependant dû implémenté 2 méthodes indépendantes pour chaque format : "read" et "write".

1. AbstractFormat

AbstractFormat a pour attributs le String fname, le long index, le BufferedReader lecture, le FileWriter ecriture ainsi que l'énumération "etat" correspondant à l'état du fichier (fermé, ouvert en lecture, ou ouvert en écriture).

- **getIndex** renvoie l'attribut "index" (la ligne du fichier où le lecteur se trouve).
- **getFname** renvoie l'attribut fname.
- **setFname** modifie l'attribut fname.
- **open** ouvre le fichier si cela est possible, en initialisant le BufferedReader lecture et le FileWriter ecriture à nul.
- **close** ferme le fichier si l'état est ouvert; en fonction du mode d'ouverture, le BufferedReader ou le FileWriter est alors fermé et l'état est mis à jour.

2. Read

Pour chaque format, cette méthode lit une ligne et renvoie un KV contenant cette ligne sans oublier d'incrémenter l'index après chaque lecture.

- Pour le format KV, on "split" la ligne à l'aide du SEPARATEUR, "<->", et on stocke la clef et la valeur.
- Pour le format LINE, on stocke le numéro de ligne (index) comme clef et la ligne comme valeur.

3. Write

Pour chaque format, cette méthode écrit une ligne (donc un KV) dans le fichier.

- Pour le format KV, on crée un String à partir de la clef, du SEPARATEUR et de la valeur puis on l'écrit.
- Pour le format LINE, on récupère la valeur du KV et on l'écrit.

4. GetType

Chaque format renvoie le Format.Type associé à son Format. Pour écrire cette méthode sans modifier l'interface Format, nous avons créé la classe Format1 qui "extend" Format, et AbstractFormat implemente donc Format1 au final.

II - Client

Le Client dispose d'un attribut "ports" ainsi qu'un attribut "hosts" tous deux des listes. "ports" contient la liste des ports sur lesquels le client communique avec les serveurs. "hosts" contient la liste des noms des serveurs, rangés dans le même ordre que les ports à utiliser pour communiquer. Il dispose aussi d'un attribut "nb_serveur" correspondant au nombre de serveurs avec lesquels il peut communiquer (c'est-à-dire la longueur des listes "ports" et "hosts").

III - Serveur

Afin de mémoriser les fragments reçus et d'y accéder facilement pour les renvoyer, le Serveur dispose d'un attribut "files" qui est une HashMap ayant pour clef une chaîne de caractères (le nom du fichier) et pour valeur une HashMap ayant pour clef un entier (le numéro du fragment) et pour valeur un Format (qui correspond au fichier créé à partir du fragment reçu). On obtient ainsi pour chaque nom de fichier une liste des fragments associés à leurs numéros.

IV - Write

L'écriture du fichier dans HDFS comporte plusieurs étapes :

- Décomposition du fichier en fragments par le Client et l'envoi de ces fragments aux Serveurs.
- Réception des fragments par les serveurs. A la réception, le Serveur enregistre le fragment dans un fichier en mémoire.

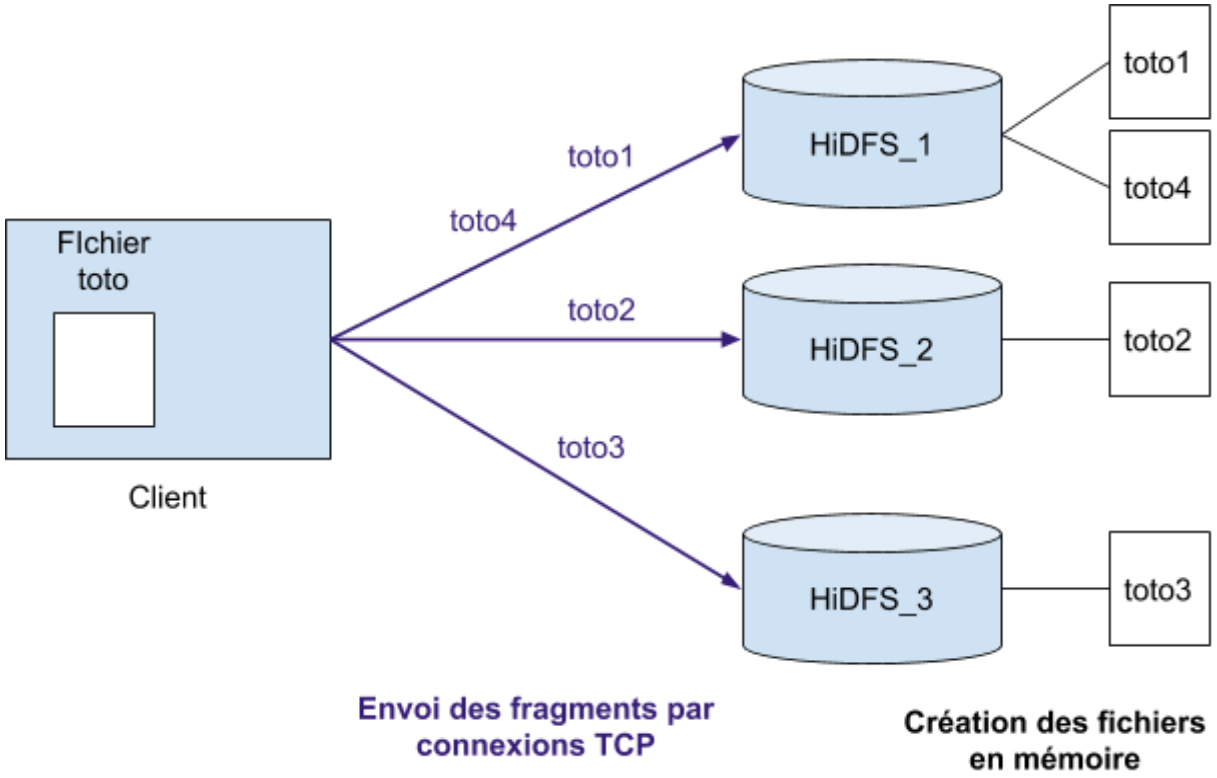

Schéma 2 : Procédure correspondant à la commande Write

1. Décomposition en fragments et envoi

Pour la découpe du fichier, nous voulions initialement découper le fichier pour envoyer à chaque serveur une quantité égale de données. Nous avons cependant réalisé que cela impliquait de parcourir tout le fichier une première fois pour connaître sa taille et cela augmenterait considérablement le temps de traitement pour une grand quantité de données. Nous avons donc choisi de découper le fichier en fragments de tailles égales (taille choisie dans une constante globale). Cela permet d'envoyer les fragments au fur et à mesure de la découpe du fichier.

L'envoi des fragments se fait du Client aux Serveurs par des connexions TCP ouvertes et fermées au fur et à mesure. Le Client envoie le premier fragment au premier serveur de la liste "hosts", le deuxième au deuxième et ainsi de suite jusqu'au "nb_serveur"ème serveur. Il envoie alors le fragment suivant au premier serveur de nouveau et continue ainsi jusqu'à la fin du fichier (le dernier fragment peut être plus petit que les autres). Les données envoyées pour chaque fragment sont les suivantes :

- Nom du fichier.
- Numéro du fragment.
- Type de format du fragment.
- Fragment (liste de lignes - la variable est de type ArrayListe<KV> qui est Serializable - créé en lisant le fichier ligne par ligne grâce à la méthode "read" implémentée pour chaque Format).

2. Réception des fragments et mémorisation

Pour chaque fragment, un serveur réceptionne les données envoyées :

- Nom du fichier.
- Numéro du fragment.
- Type de format du fragment.
- Fragment (liste de lignes - la variable est de type ArrayListe<KV> qui est Serializable).

Il crée alors un fichier du Format correspondant et y "recopie" les lignes qu'il a reçues (grâce à la méthode "write" implémentée pour chaque Format). Il mémorise ce fichier dans son attribut "files" en vérifiant si une HashMap correspondant au nom du fichier existe déjà. Sinon, une nouvelle entrée sera créée dans "files" ayant pour clef le nom du fichier. Le nouveau fragment (valeur) ainsi que son numéro (clef) seront alors enregistrés dans la HashMap ayant pour clef le nom du fichier.

V - Read

La lecture du fichier à partir de HDFS comporte plusieurs étapes :

- Demande de lecture du fichier depuis le Client vers les Serveurs.
- Envoi des fragments demandés.
- Reconstitution du fichier.

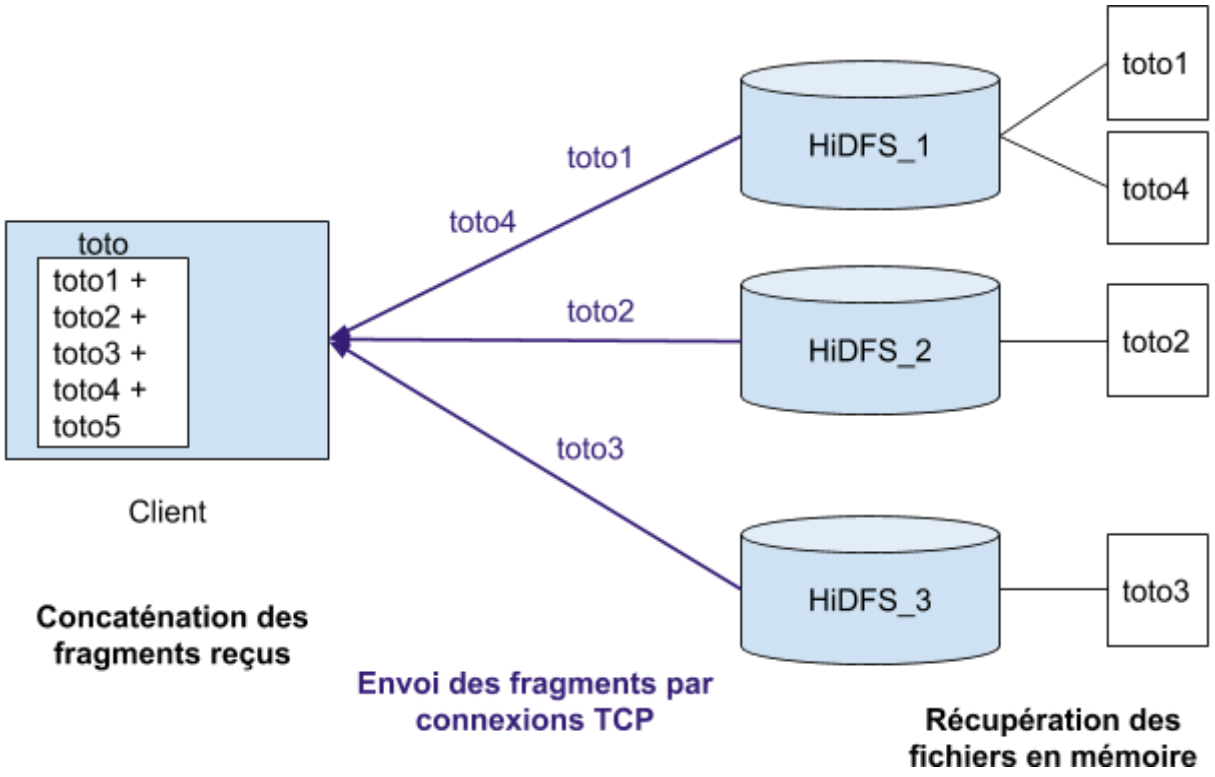

Schéma 3 : Procédure correspondant à la commande Write

1. Demande de lecture

Pour lire un fichier, le Client envoie à chaque Serveur une demande de lecture tour à tour en commençant par le Serveur 1. Une demande de lecture est constituée de la commande "READ", du nom du fichier et du numéro de fragment demandé.

Le Client demande d'abord le fragment 1 au Serveur 1, puis le fragment 2 au Serveur 2 et ainsi de suite jusqu'au "nb_serveur"ème Serveur. Il envoie alors la demande pour le fragment suivant au Serveur 1, et ainsi de suite. Chaque Serveur, à la réception de la demande, renvoie un message pour indiquer s'il a ou non le fragment demandé en mémoire. Si c'est le cas, il enverra ensuite le fragment demandé. Le Client cesse d'envoyer des demandes lorsqu'il reçoit un message négatif.

2. Envoi des fragments demandés

A la réception d'une demande de lecture, le Serveur réceptionne le nom du fichier et le numéro de fragment demandé. Il consulte "files" pour vérifier qu'il existe un fichier correspondant au nom et numéro demandé. Il envoie alors un message indiquant s'il possède ou non le fragment. S'il le possède, il envoie ensuite le format du fragment ainsi que le fragment en créant une liste de lignes à l'aide de la méthode "read" implémentée pour chaque Format.

3. Reconstitution du fichier

A la réception du premier fragment, le Client récupère le Format du fichier et crée un fichier du Format correspondant. Au fur et à mesure de la réception des fragments, il y "recopie" les lignes qu'il a reçues (grâce à la méthode "write" implémentée pour chaque Format). Le nouveau fichier ainsi créé porte le même nom que celui associé aux différents fragments auquel a été rajouté le suffixe "new" pour le différencier du fichier originel.

VI - Delete

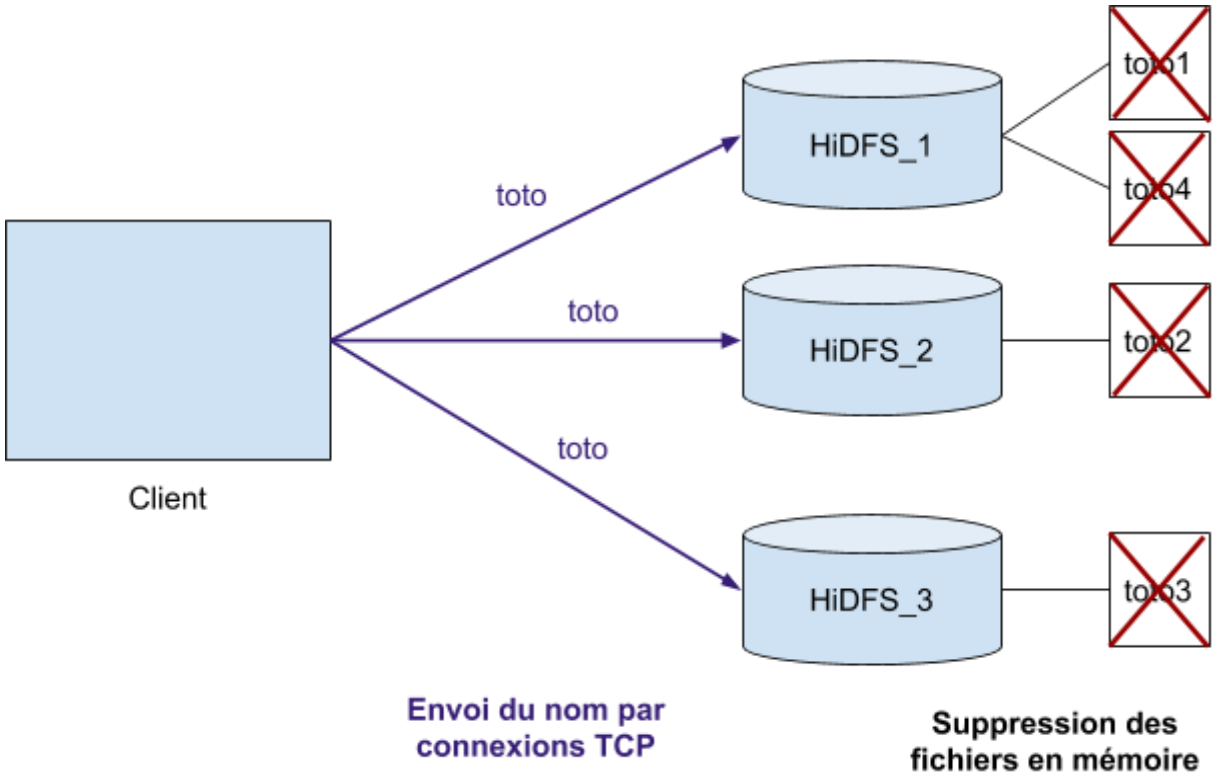

Schéma 4 : Procédure correspondant à la commande Delete

Pour supprimer les fichiers, le Client envoie une commande "Delete" à chaque Serveur, suivie du nom du fichier dont on veut supprimer les fragments. Chaque Serveur consulte son "files" et supprime toutes les entrées correspondant au nom du fichier envoyé ainsi que les données en mémoire associées.

Tests

Nous avons commencé par réaliser des tests sur des fichiers simples de quelques lignes seulement afin de débugger, puis nous avons progressivement augmenté en complexité jusqu'à tester les fichiers tests fournis dans le dossier data.

Conclusion

Nous avons pu implémenter un Client utilisant des Serveurs pour stocker des données. Les commandes Read et Write fonctionnent. Nous avons passé beaucoup de temps à réfléchir à une implantation intelligente pour les formats ainsi que pour les méthodes d'envoi et de mémorisation des fragments afin d'avoir une implémentation claire et efficace. Il nous reste à concevoir notamment un interpréteur de commandes mais également à réaliser de nombreuses améliorations sur notre structure actuelle.

Parmi les améliorations possibles, nous en avons relevé quelques unes notamment concernant la taille des BufferedReader et Filewriter utilisés pour la lecture et l'écriture, qui pourraient être fixée en fonction du fichier lu ou écrit pour optimiser ces deux méthodes. Nous n'avons pas pour l'instant créé et géré toutes les exceptions rencontrées, seuls des affichages signalent les erreurs. Nous n'avons pas non plus réussi à rendre serializables les formats, et il est donc impossible de les envoyer par les sockets. On est obligé de recréer un format à partir de son type (qui est lui serializable et donc envoyable). Idéalement, nous souhaiterions créer une nouvelle classe serializable qui contiendra toutes les informations sur un format, son nom, son type, etc pour n'avoir qu'un elément à envoyer et recevoir. Nous pourrions enfin envisager la création de nouveaux formats en plus de KV et de Line.