BEN AÏM, CLAVERIE, DOMINGUEZ, PETIT

Rapport Final

Les améliorations de Hidoop

Partie Hidoop

Gestion de la redondance Fragment & FragmentHandler

Nous avons implémenté un ***FragmentHandler*** qui est une classe séparée qui va décider de quel nœud va traiter quel fragment.
Contrairement aux versions précédentes ou chaque nœud traitait l'ensemble de ses fragments d'un coup chacun dans un thread séparé, nous lançons initialement pour chaque nœud autant de traitement de fragment que le processeur de nœud possède de cœurs.
Lorsque un nœud a terminé de traiter un de ses fragments, il va demander au *FragmentHandler* si il reste de travail à effectuer dont il possède le fragment nécessaire. Si c'est le cas, il s'occupera de ce travail.

Ainsi, si un nœud est particulièrement performant (+ de coeurs, meilleur processeur…) il pourra s'occuper de davantage de fragments et ainsi faire gagner du temps sur l'exécution.

Gestion de panne de noeuds pendant un job

Afin de surveiller que le traitement des fragments avance bien, nous avons fait une classe ***FragmentWatcher***. Plusieurs fois par seconde, durant un Job, *FragmentWatcher* va interroger le *FragmentHandler* pour :

- Récupérer le temps de traitement moyen d'un fragment
- Regarder les fragments en cours de traitement par un node :
    - Si le temps d'exécution de ce fragment à dépassé 3 (seuil à fixer) fois le temps moyen, alors on considère que l'exécution a échoué. On va alors demander à un autre noeud d'également traiter ce fragment. Le premier de ces deux noeuds (celui très lent, ou le nouveau) qui aura fini marquera ce fragment comme traité

Système de Tâches

Lorsque nous avons voulu développer de nouvelles applications (telles que QuasiMonteCarlo & PageRanking), nous nous sommes aperçus qu'elles ne nécessitent pas de fichier en entrée, et donc que notre répartition de travail par fragment de fichier n'était pas possible.
Pour cela, nous avons créé un ensemble de fonctions dans le Job et des classes (désignées par ***FileLess***) pour gérer ces applications qui fonctionnent avec des tâches.
Le fonctionnement de la répartition et des pannes est similaire aux fragments, si ce n'est que nous n'avons pas besoin de vérifier si un nœud est capable de faire une tâche contrairement aux fragments où il fallait vérifier que le fragment était disponible sur le nœud.

Gestion panne de Job

Il est possible de reprendre un Job qui a crashé sans recalculer tous les fragments.
Pour cela, nous demandons avant de démarrer un Job à HDFS si il possède des fragments de résultats complets (On ne prend pas en compte les fragments qui étaient en cours de calculs et donc non valides!).
Puis on marque tous ces fragments comme déjà traités dans le FragmentHandler, puis on lance le Job.

Partie HDFS

Lors de ce semestre nous avons beaucoup renforcé la fiabilité du système de fichier grâce à de nombreuses fonctionnalités. En plus de la possibilité de lire, d'écrire et de supprimer, nous avons ajouté :

1. **La réplication des fragments.** Lorsqu'on uploade un fichier avec le client, il est possible de définir le nombre de fois où un fragment est répliqué. Par exemple, si on choisit un facteur de réplication 2, alors le fichier est stocké en double sur le réseau. Par conséquent, si un nœud se déconnecte, le fichier est toujours complet.
2. **La détection des nœuds déconnectés.** Le serveur de noms envoie régulièrement un ping à destination de tous les nœuds pour savoir s'ils sont toujours disponibles. Si un nœud ne répond pas au ping, alors il est considéré comme déconnecté, et si tous les fragments d'un fichier ne sont pas accessibles, alors le fichier est déclaré incomplet et ne peut pas être téléchargé. Un nœud déconnecté ne reçoit plus de ping du serveur de nom.
3. **L'auto-reconnexion des nœuds.** Les nœuds envoient régulièrement un ping à destination du serveur de noms, même lorsque le nœud est déconnecté du serveur de nom. Cela permet au nœud de se reconnecter automatiquement au serveur de noms si la connexion est rétablie après que le nœud a été déclaré déconnecté.
4. **La gestion de la déconnexion pendant un upload.** Si un nœud ou le client se déconnecte pendant l'upload, le réseau récupère proprement la situation. Si c'est un nœud qui se déconnecte alors qu'il est en train de recevoir un fragment, son fragment sera distribué à un autre nœud disponible. Si c'est le client qui se déconnecte, le fichier est supprimé et doit être uploadé à nouveau.
5. **La gestion de la déconnexion d'un nœud pendant un download.** Le serveur de noms temporise tous les fragments qu'il reçoit des nœuds pendant un téléchargement. Ainsi, si un nœud se déconnecte alors qu'il envoie un fragment, alors le serveur de noms va chercher un autre nœud qui dispose de ce fragment pour reconstituer le fichier. Si aucun autre nœud n'est disponible, le client reçoit un message d'erreur.
6. **Détection des fragments corrompus au démarrage d'un nœud.** Lorsqu'un nœud démarre, il scanne ses fragments pour chercher les fragments corrompus (taille non réglementaire, dernier caractère qui n'est pas un \n) et les supprimer. Cette fonctionnalité a été désactivée car elle casse le fonctionnement d'une des applications du projet Hidoop.
7. **Le téléchargement a une barre de progression.** Cette fonctionnalité est purement cosmétique mais rend l'utilisation plus agréable.

D'autres fonctionnalités ont été rajoutées pour les besoins des applications Hidoop, comme la possibilité d'obtenir la liste des fragments ou des nœuds. Ces fonctionnalités ne sont pas directement utilisables, mais ont demandé un certain temps de développement. **Cela permet au projet Hidoop de bénéficier des fonctionnalités de fiabilité du projet HDFS**, comme la reconnexion automatique, sans développement supplémentaire.

Par ailleurs, le système de fichier est **beaucoup plus rapide lors des transferts**, d'un facteur 20, grâce à l'utilisation de flux binaires temporisés, plutôt que de flux d'objets. Bien que ça ne soit pas le cœur de notre travail, nous tenons à mentionner que le débit d'upload est de 700 Mbps pour l'upload sur les machines de l'école, et 500 Mbps pour le download.

Les outils développés pour faciliter le déploiement, le test et l'évaluation des performances de votre plateforme

Script de génération de données

Le script de génération de donnée - *biglorem_generator.sh*, permet de générer un fichier texte de la taille que l'on souhaite en dupliquant un texte lorem ipsum.

Pour l'utiliser, il faut placer le script ainsi que lorem_original.txt à l'endroit où l'on souhaite générer le fichier.

Puis executer bash biglorem_generator.sh et il est ensuite demandé d'entrer le nombre de duplications à réaliser. 18 duplications pour un fichier de 8 GB, 17 pour 4 GB, 16 pour 2 GB...

Scripts de déploiement

Nous avons développé deux scripts permettant de déployer notre application. Les deux scripts sont situés dans le répertoire src/application du dossier Hidoop.

Le premier script, *deploy.sh,* permet de compiler et de placer le dossier Hidoop sur les disques durs (dans le répertoire /work) des machines qui vont être utilisées pour HDFS. Il met également à disposition des noeuds certaines dépendances (comme jsoup).

Le deuxième script, *exec.sh*, doit être lancé sur la machine principale. En effet, il va permettre de lancer, sur la machine d'exécution du script, le NameServer ainsi que le RmiRegistry. Le script va ensuite lancer les Binodes (HDFSnode et Worker) sur chaque machine du HDFS. Enfin, le script permet aussi de stopper le RmiRegistry et les Binodes si l'utilisateur appuie sur la touche `!`.

Le dernier script, *force_stop.sh*, est présent en cas de soucis si exec.sh n'arrête pas Hidoop (en cas de soucis).

Comment utiliser les scripts

Les scripts se trouvent dans le dossier Hidoop qui contient les fichiers compilés.

- Placer ce dossier sur une machine de l'ENSEEIHT.
- Dans **deploy.sh**, mettre dans la variable HOST la liste des machines sur lequelles vous voulez déployer. Puis executer le script deploy (./deploy.sh)
- De la même manière, lister les HOST dans le script **exec.sh**, puis executer le script(./exec.sh)
- Vous pouvez maintenant utiliser une application telle que MyMapReduce en executant : "java application.MyMapReduce <nomDuFichier>" (depuis la machine qui a lancé le deploy) ou uploader un fichier sur le HDFS avec "java hdfs.HdfsClient write line <cheminDuFichier>"
- Lorsque vous avez fini, appuyez sur une touche pour arrêter le script exec, puis lancer le script **force_stop.sh** (en renseignant la liste dans HOST des machines sur lesquelles les binodes se trouvent)

Les applications supplémentaires

Nous avons mis en place deux applications supplémentaires afin de compléter l'analyse des performances de notre système.
Dans un premier temps, nous avons choisi d'implémenter la méthode de Monte-Carlo quasi aléatoire, permettant d'estimer pi à l'aide des séquences (quasi aléatoires) de Halton. Cet algorithme devrait être a priori CPU bound car le map-reduce effectue principalement des calculs.
De plus, nous avons aussi réalisé un algorithme de Page Ranking, celui-ci est a priori I/O bound, cela nous permettra de compléter l'analyse faite sur WordCount. En effet, le WordCount étant une application simple, nous avons eu envie de mettre en place une application plus complexe dans le but d'observer le comportement de notre système face à l'apparition de nouveaux éléments tels la connexion à un serveur (pour la construction du graphe) par exemple.
Nous pouvons aussi noter que, comme expliqué dans la partie du rapport sur les tâches, ces applications n'ont pas besoin de fichier pour fonctionner.
L'application QuasiMonteCarlo prend en entrée le nombre d'échantillons par tâche que l'on souhaite ainsi que le nombre de tâche. Cela permet d'observer plus facilement l'efficacité du système quand on fait évoluer la quantité de calculs mais aussi le degré de parallélisme.
L'application EvaluationPageRank prend en entrée seulement l'URL du site web que l'on veut consulter. Le nombre de tâches générées correspond alors simplement au nombre de pages que contient le site.

Etude de performances

Etude sur WordCount

- Faire varier le nombre de coeurs total
- Faire varier la taille du fichier
- Faire varier le type de fichier(lorem avec des vrais mots, ou charabia)

Challenge 8GB (performances)

Génération du fichier : Nous avons utilisé le script biglorem_generator.sh avec 18 duplications (bash biglorem_generator.sh puis entrer 18 lorsqu'il demande le nombre de duplications). Cela donne un fichier de 7.9 GB.

Temps séquentiel :
1 minute 50 secondes (110 secondes)

```
gclaveri@stark:/work/gclaveri/Hidoop$ time java application.Count /work/fat_lorem.txt
time in ms =109996

real    1m50,081s
user    1m49,603s
sys     0m1,534s
```

Temps d'upload sur hdfs :
Avec **5 noeuds (Truite, Anguille, Brochet, Carpe, Goujon),** le temps d'upload a pris 1 minute 33 secondes (93 secondes).

Temps d'exécution :
1873 Fragments.
Executions de MyMapReduce :

- 12 Secondes pour la première exécution (à froid, fichier pas en cache)

```
Pas de fragment supplementaire trouve pour 147.127.143.115:46161
> Le node 147.127.143.115:46161 a fini l'element 1813 en 181ms. Il reste 9
Pas de fragment supplementaire trouve pour 147.127.143.115:46161
> Le node 147.127.143.115:46161 a fini l'element 1828 en 170ms. Il reste 8
Pas de fragment supplementaire trouve pour 147.127.143.115:46161
> Le node 147.127.143.115:46161 a fini l'element 1838 en 121ms. Il reste 7
Pas de fragment supplementaire trouve pour 147.127.143.115:46161
> Le node 147.127.143.115:46161 a fini l'element 1853 en 107ms. Il reste 6
Pas de fragment supplementaire trouve pour 147.127.143.115:46161
> Le node 147.127.143.115:46161 a fini l'element 1843 en 131ms. Il reste 5
Pas de fragment supplementaire trouve pour 147.127.143.115:46161
> Le node 147.127.143.115:46161 a fini l'element 1868 en 94ms. Il reste 4
Pas de fragment supplementaire trouve pour 147.127.143.115:46161
> Le node 147.127.143.115:46161 a fini l'element 1848 en 131ms. Il reste 3
Pas de fragment supplementaire trouve pour 147.127.143.115:46161
> Le node 147.127.143.115:46161 a fini l'element 1863 en 122ms. Il reste 2
Pas de fragment supplementaire trouve pour 147.127.143.115:46161
> Le node 147.127.143.115:46161 a fini l'element 1858 en 126ms. Il reste 1
> All done ! Let's request a file refresh...
> Let's reduce /work/gclaveri/Hidoop/tmp/fat_lorem.txt_result_result_temp
> On telecharge les résultats des machines
1873/1873 fragments
> Telechargement termine
> On reduit les resulats en un seul...
> Done ! Resultat dans /work/gclaveri/Hidoop/res/fat_lorem.txt_result
> Let's delete temporary files
time in ms =6926

real    0m7,337s
user    0m7,733s
```

- 7 Secondes pour les exécutions successives

On active la réplication des fragments avec un facteur de 3 sur ces mêmes noeuds
Temps d'exécution :
1873 Fragments.
Executions de MyMapReduce :
- 10 Secondes pour la première exécution (à froid, fichier pas en cache)
- 7 Secondes pour les exécutions successives

Légère amélioration du temps lorsque les fichiers fragments n'ont jamais été chargés en RAM.

Etude de l'influence de réplication des fragments sur des machines aux performances différentes (Une à 2 coeurs, une à 16 coeurs).

Prenons **Pikachu** et **Truite.**
Nous allons mettre en ligne le fichier de 8GB lorem généré ci dessus, d'abord sans réplication, puis avec réplication.

Temps d'exécution sans réplication:
1873 Fragments.
Executions de MyMapReduce :
- 21.5 Secondes pour la première exécution (à froid, fichier pas en cache)
- 20.5 Secondes pour les exécutions successives

On remarque à l'exécution que Truite a terminé le traitement de ses fragments bien avant Pikachu, qui retarde l'issue du programme.

Temps d'exécution AVEC réplication:
1873 Fragments.
Executions de MyMapReduce :
- 17.5 Secondes pour la première exécution (à froid, fichier pas en cache)
- 15 Secondes pour les exécutions successives

Remarque : en changeant des paramètres, le temps d'exécution pourrait être meilleur. En effet, le FragmentWatcher remarque que les fragments sont trop longs à s'exécuter sur Pikachu et il les relance souvent sur truite pour essayer d'aller plus vite. Il faudrait augmenter le seuil de relance ou faire un seul différent par machine.

Variation du nombre de thread

On reprend le fichier utilisé sur le challenge. Pour le challenge, on a utilisé les ordinateurs **Truite, Anguille, Brochet, Carpe, Goujon,** avec 16 coeurs / 32 threads chacun, soit 80 coeurs / 160 threads de noeuds. Essayons le même fichier avec **pikachu, carapuce et salameche (2 coeurs / 4 threads chacun, soit 6 coeurs / 12 threads de noeud).**
Temps d'exécution :
1873 Fragments.
Executions de MyMapReduce :

- 16 Secondes pour la première exécution (à froid, fichier pas en cache)
- 15 Secondes pour les exécutions successives

```
> Let's reduce /work/gclaveri/Hidoop/tmp/fat_lorem.txt_result_result_temp
> On telecharge les résultats des machines
1873/1873 fragments
> Telechargement termine
> On reduit les resulats en un seul...
> Done ! Resultat dans /work/gclaveri/Hidoop/res/fat_lorem.txt_result
> Let's delete temporary files
time in ms =14984

real    0m15,406s
user    0m7,829s
sys     0m0,806s
gclaveri@stark:/work/gclaveri/Hidoop$ time java application.MyMapReduce fat_lorem.txt
```

Essayons le même fichier avec **truite, goujon (16 coeurs / 32 threads chacun, soit 32 coeurs / 64 threads de noeud).**
Temps d'exécution :
1873 Fragments.
Executions de MyMapReduce :

- 12 Secondes pour la première exécution (à froid, fichier pas en cache)
- 9.5 Secondes pour les exécutions successives

```
> Let's reduce /work/gclaveri/Hidoop/tmp/fat_lorem.txt_result_result_temp
> On telecharge les résultats des machines
1873/1873 fragments
> Telechargement termine
> On reduit les resulats en un seul...
> Done ! Resultat dans /work/gclaveri/Hidoop/res/fat_lorem.txt_result
> Let's delete temporary files
time in ms =9360

real    0m9,789s
user    0m8,153s
sys     0m0,775s
gclaveri@stark:/work/gclaveri/Hidoop$
```

Etude sur QuasiMonteCarlo

- Faire varier le nombre de tâches

L'objectif correspond ici à tester la gestion des calculs par notre système grâce à cette application CPU bound.
Ainsi, nous allons utiliser les cinq machines Truite, Anguille, Brochet, Carpe et Goujon. De plus, nous allons garder un nombre total d'échantillons constant tout en faisant évoluer le nombre de taches.

Nous réalisons donc l'expérience avec un nombre total constant de 1 000 000 000 d'échantillons. Ce nombre est choisi arbitrairement mais doit être assez élevé pour observer l'effet de l'évolution du nombre de tâches.

Résultat obtenu avec une unique tâche (donc presque équivalent au séquentiel) :

```
> All done ! Let's request a file refresh...
> Let's reduce /home/cdomingu/Hidoop/tmp/resultat_pi.txt_result_temp
> On telecharge les résultats des machines
1/1 fragments
> Telechargement termine
> On reduit les resulats en un seul...
> Done ! Resultat dans /home/cdomingu/Hidoop/res/resultat_pi.txt
> Let's delete temporary files
time in ms =8352
```

Résultat obtenu avec 10 tâches :

```
> All done ! Let's request a file refresh...
> Let's reduce /home/cdomingu/Hidoop/tmp/resultat_pi.txt_result_temp
> On telecharge les résultats des machines
10/10 fragments
> Telechargement termine
> On reduit les resulats en un seul...
> Done ! Resultat dans /home/cdomingu/Hidoop/res/resultat_pi.txt
> Let's delete temporary files
time in ms =1371
```

On peut déjà observer plusieurs choses. Premièrement, l'augmentation du nombre de tâches permet d'augmenter drastiquement l'efficacité du calcul. En effet, le calcul a été effectué environ 6.5 fois plus vite en passant de une à 10 tâches.
On peut tout de même dire que l'on a pas fait 10 fois en efficacité car le système de mise en place des tâches est coûteux en temps, ce qui est dû notamment au système d'attribution de nouvelles donc à la communication entre les machines.

Mais nous sommes loin d'avoir saturé le degré de parallélisme possible avec nos 5 machines, on peut augmenter le nombre de tâches à 100 pour s'en approcher et voir ce que cela produit.

Résultat obtenu avec 100 tâches :

```
> All done ! Let's request a file refresh...
> Let's reduce /home/cdomingu/Hidoop/tmp/resultat_pi.txt_result_temp
> On telecharge les résultats des machines
100/100 fragments
> Telechargement termine
> On reduit les resulats en un seul...
> Done ! Resultat dans /home/cdomingu/Hidoop/res/resultat_pi.txt
> Let's delete temporary files
time in ms =804
```

Ainsi, on peut observer que l'on a encore gagné un peu de temps, environ 1.8 fois moins de temps qu'avec 10 tâches mais cela est mineur comparé au taux de parallélisme que l'on a essayé de gagner. Cela vient du fait que le temps utilisé pour gérer les taĉhes commence à prendre le dessus sur le calcul à effectuer. De plus, on se rapproche du degré de parallélisme que l'on peut avoir avec ce système de cinq machines.

On peut alors essayer avec 1000 tâches pour voir si nos observations se confirment quand le nombre de tâches devient "trop" grand.

Résultat obtenu avec 1000 tâches :

```
> All done ! Let's request a file refresh...
> Let's reduce /home/cdomingu/Hidoop/tmp/resultat_pi.txt_result_temp
> On telecharge les résultats des machines
1000/1000 fragments
> Telechargement termine
> On reduit les resulats en un seul...
> Done ! Resultat dans /home/cdomingu/Hidoop/res/resultat_pi.txt
> Let's delete temporary files
time in ms =4489
```

On peut voir ici que l'on perd à présent de nombreuses secondes à gérer les tâches et même l'opération de reduce commence à occuper une part importante du temps dépensé.

Conclusion de l'étude sur QuasiMonteCarlo :

On peut noter ici que le système de Map-Reduce permet grandement de réduire le temps de calcul pour une application CPU bound grâce à la mise en place du parallélisme. En revanche, il faut y trouver un juste milieu car c'est un système lourd qui demande un certain temps de gestion. De plus, dépasser un certain degré de parallélisme (au-delà du nombre de cœurs du système) pourrait être pénalisant car la re-répartition des tâches est aussi coûteuse en temps d'exécution.

Etude sur PageRanking

- Faire varier le nombre de coeurs
- Faire varier le nombre de tâches

L'objectif ici est d'étudier le comportement du système sur une application I/O plus complexe que celle du WordCount, notamment avec la nécessité d'effectuer des connexions à des serveurs.

Ici, nous allons principalement étudier l'influence de paralléliser ou non la construction du graphe (CG), c'est-à-dire les différentes connexions. En effet, la deuxième partie du travail correspondant à l'évaluation du page ranking une fois le graphe construit, ce qui n'apporte rien de plus que ce qui a déjà été fait dans les autres applications.

Ainsi, nous pouvons d'abord observer le résultat obtenu avec une CG réalisée en séquentiel. Nous utilisons pour cela un système HDFS composé des trois machines Salamèche, Carapuce et Bulbizarre.

De plus, nous utilisons le site web https://ben-aim.fr qui est de taille modérée.

La commande utilisée pour lancer l'application est la suivante :
java -cp /work/user/Hidoop/ application.EvaluationPageRank https://ben-aim.fr

Résultat pour CG en séquentiel :

```
> Let's reduce /home/cdomingu/Hidoop/tmp/resultat_pageRank.txt_result_temp
No more tasks available!
> On telecharge les résultats des machines
8/8 fragments
> Telechargement termine
> On reduit les resulats en un seul...
https://ben-aim.fr/about/<->0.1869887853908406;

https://ben-aim.fr<->0.03778763333166408;

https://ben-aim.fr/fr/<->0.16279787130390275;

https://ben-aim.fr/<->0.17090280224009047;

https://ben-aim.fr/fr/about/resume/<->0.06608041131245618;

https://ben-aim.fr/es/about/<->0.11383959570323694;

https://ben-aim.fr/about/resume/<->0.0907000984777186;

https://ben-aim.fr/fr/about/<->0.17090280224009047;

> Done ! Resultat dans /home/cdomingu/Hidoop/res/resultat_pageRank.txt
> Let's delete temporary files
time in ms =12578
```

On peut comparer ce résultat avec la parallélisation de la CG en utilisant un map pour les différentes connexions :

```
> Let's reduce /home/cdomingu/Hidoop/tmp/resultat_pageRank.txt_result_temp
No more tasks available!
> On telecharge les résultats des machines
No more tasks available!
No more tasks available!
8/8 fragments
> Telechargement termine
> On reduit les resulats en un seul...
https://ben-aim.fr/about/<->0.1869887853908406;

https://ben-aim.fr<->0.03778763333166408;

https://ben-aim.fr/fr/<->0.16279787130390275;

https://ben-aim.fr/<->0.17090280224009047;

https://ben-aim.fr/fr/about/resume/<->0.066080411312456l8;

https://ben-aim.fr/es/about/<->0.11383959570323694;

https://ben-aim.fr/about/resume/<->0.0907000984777186;

https://ben-aim.fr/fr/about/<->0.17090280224009047;

> Done ! Resultat dans /home/cdomingu/Hidoop/res/resultat_pageRank.txt
> Let's delete temporary files
time in ms =4518
```

On observe alors que le calcul prend trois fois moins de temps avec la parallélisation de la CG. Ainsi, on en déduit que les connexions aux serveurs occupent la majeure partie du temps d'exécution et il est très important de paralléliser cela.

On peut augmenter le nombre de machines dans le système pour voir l'effet de l'augmentation du nombre de tâches et du nombre de cœurs.
Ainsi, avec les cinq machines Truite, Anguille, Brochet, Carpe et Goujon, on obtient un temps total d'exécution de **7139 ms.**
Ce temps a quasi doublé et cela provient du fait que la construction du graphe se fait en itérant plusieurs Map-Reduce jusqu'à avoir parcouru tout le site. Donc si l'on augmente le nombre de machines, on va augmenter le nombre de taches générées et comme nous l'avons déjà remarqué précédemment, le temps de gestion des tâches n'est pas à négliger.

On peut donc conclure qu'il faut adapter le nombre de tâches, donc le nombre de machines du HDFS, au site web que l'on veut sonder. En effet, il faut encore une fois trouver un juste milieu entre parallélisation du travail effectué et gestion des tâches générées.

Etude expérimentale

Tests de panne HDFS

Afin de vérifier que le système HDFS est bien résistant aux pannes partielles (déconnexion d'un ou plusieurs noeud pendant une opération), le protocole suivant a été développé :

Pour vérifier la fiabilité à l'upload :

- Allumage du réseau avec N nœuds.
- Upload d'un fichier avec un facteur de réplication R ≤ N.
- Lors de l'upload, éteindre et parfois rallumer les nœuds, en s'assurant qu'au moins 1 nœud soit toujours connecté.
- Télécharger le fichier.
- Calculer la somme de contrôle md5 du fichier d'origine et du fichier d'arrivée.

Pour vérifier la fiabilité au download :

- Allumage du réseau avec N nœuds.
- Upload d'un fichier avec un facteur de réplication R ≤ N.
- Télécharger le fichier.
- Éteindre et rallumer aléatoirement des nœuds, en s'assurant qu'au moins N-R+1 nœuds soient toujours connectés.
- Calculer la somme de contrôle md5 du fichier d'origine et du fichier d'arrivée.

Les pannes totales (déconnexion de tous les nœuds ou du serveur de noms) ne sont pas récupérables (le système ne peut pas trouver façon de remplir la tâche demandée) et par conséquent ne font pas l'objet de tests particuliers.

Par ailleurs, comme la connexion entre les nœuds Hidoop repose sur le système HDFS, les tests ci-dessous permettent aussi de tester la robustesse du système HDFS.

Tests de panne Hidoop

Critère de succès : pouvoir récupérer n'importe quel travail effectué en redémarrant le point d'échec, ou si la redondance est disponible, pouvoir prendre le relai sur d'autre node.

Nous avons tenté d'arrêter un noeud ou un job à des moments variés de l'exécution (commande pkill -9 -f Job / pkill -9 -f BiNode)

Arrêt d'un noeud (scénario de panne : un noeud devient indisponible / java plante sur une machine) :

- S'il est fait en dehors de l'exécution d'un Job, le programme détecte que le nœud est indisponible et il ne sera plus utilisé pour les applications. S' il contenait des fichiers non répliqués, il sera indiqué dans le terminal que ces fichiers sont maintenant incomplets.
  Si le nœud revient en ligne, le fichier sera de nouveau marqué complet.

- S'il est fait durant l'exécution d'un Job, et que le nœud ne se reconnecte pas avant la fin de l'exécution, les fragments qui étaient en cours de traitement sur ce nœud seront pris en charge par un autre nœud. Une fois que tous les nœuds possibles

seront exécutés, le job va s'arrêter en indiquant que certains des fragments traités sur le nœud ne sont plus disponibles. Il suffit alors de relancer le job, et il ne traitera que les fichiers manquants en raison du nœud qui a lâché. Si le nœud se reconnecte avant la fin de l'exécution de Job, aucun progrès n'est perdu et le Job se finit correctement.

Arrêt d'un job (scénario de panne : la machine exécutant le Job rencontre un soucis) :

- Les fragments traités entièrement ne seront pas impactés : leur exécution est sauvegardée. Les fragments en cours de traitement seront supprimés au lancement suivant du Job, et il traitera uniquement les fragments manquants.

README

Le projet s'utilise à travers trois scripts en plus de l'application distribuée. Voici les commandes à exécuter pour :

- **bash deploy.sh** : compile le projet et l'installe sur trois machines distantes
- **bash exec.sh** : démarre les noeuds et le serveur de nom
- **java votre.ApplicationDistribuee** : uploadez des fichiers et lancez une ou plusieurs applications
- **bash force_stop.sh** : arrête les serveurs

Pour uploader un fichier sur les noeuds, il faut utiliser
java -cp /work/$USER/Hidoop hdfs.HdfsClient write <fichier>