RenPar'20 / SympA'14 / CFSE 8
Saint-Malo, France, du 10 au 13 mai 2011

Étude d'une architecture MapReduce tolérant les fautes byzantines

(1) Alysson N. Bessani, Vinicius V. Cogo, Miguel Correia, Pedro Costa et Marcelo Pasin
(2) Fabricio Silva
(3) Luciana Arantes, Jonathan Lejeune, Madeleine Piffaretti, Olivier Marin, Pierre Sens et Julien Sopena

(1) LASIGE - Universidade de Lisboa, Faculdade de Ciencias
Campo Grande 1749-016 - Lisboa, Portugal.

(2) Centro Tecnológico do Exército
Av. das Américas, 28705 - Rio de Janeiro, Brasil.

(3) LIP6 - Université de Pierre et Marie Curie- CNRS - INRIA
4, Place Jussieu 75252 Paris Cedex 05, France.

Résumé
Les pannes arbitraires sont inhérentes aux calculs massivement parallèles tels que ceux visés par le modèle MapReduce ; or les implémentations courantes du MapReduce ne fournissent pas les outils pour tolérer les fautes byzantines. Il est donc impossible de certifier l'exactitude de résultats obtenus au terme de traitements longs et coûteux sur ces implémentations. Nous présentons une architecture permettant de répliquer les tâches afin de garantir l'intégrité des traitements et d'isoler les processus défaillants. Notre travail permet ainsi non seulement de tolérer un large spectre de fautes lors de l'exécution de calculs sur le modèle MapReduce, mais aussi d'envisager le déploiement des implémentations sur des architectures matérielles ouvertes telles que les "Clouds".

Mots-clés : MapReduce, fautes byzantines, Hadoop, HDFS

---

1. Introduction

Cet article propose une architecture tolérant les fautes byzantines dans les calculs utilisant le modèle MapReduce. En effet, la gestion des pannes arbitraires est un élément essentiel dans les applications réparties, et à plus forte raison dans les calculs massivement parallèles [17] puisque la probabilité de comportements byzantins augmente proportionnellement avec le nombre de processus impliqués. Or, l'implémentation du MapReduce proposée avec la plate-forme Hadoop [1] se contente de reprendre un calcul lorsque le résultat associé tarde à émerger. En effet, le modèle MapReduce est généralement associé aux centres de traitements de données (*data centers*) mis en place par de grandes entreprises informatiques telles que Google ou Amazon. Ces datacenters offrent déjà des garanties vis-à-vis des fautes potentielles. Toutefois, la redondance intégrée dans leur architecture a essentiellement pour but de garantir la disponibilité des ressources et des données [3]. Dans ce contexte, les erreurs de calcul produites soit par des processus malveillants soit par des fautes matérielles temporaires ne seront pas prises en compte. Par ailleurs, il est tout à fait concevable de déployer des calculs massivement parallèles sur des architectures plus ouvertes et moins coûteuses en termes de maintenance : le "Cloud Computing" [2] et les architectures orientées service (SOA) [19] visent explicitement à répartir les charges de calcul sur un grand nombre de machines de manière décentralisée. Cette approche ôte toute maîtrise de la plate-forme physique, et donc toute garantie quant aux résultats retournés une fois le code déployé.
Le travail présenté dans cet article s'applique à présenter une solution pour tolérer les fautes byzantines dans le modèle MapReduce. Notre approche vient s'intégrer à l'implémentation Hadoop, qui a le double avantage d'être open-source et d'être la plus utilisée actuellement. Nos contributions principales sont les suivantes :

– Nous proposons une architecture permettant de répliquer des calculs dans le modèle MapReduce pour à la fois garantir l'intégrité des traitements et isoler les processus défaillants.

– Les mesures de performances effectuées avec notre prototype ont permis d'identifier plusieurs bonnes pratiques liées à la réplication dans le modèle MapReduce, et nous les présentons ci-après.

Cet article s'articule comme suit. La section 2 présente le modèle de programmation MapReduce ainsi que l'architecture Hadoop. Ensuite, la section 3 détaille notre solution de réplication de calculs pour la gestion des fautes byzantines et justifie, performances à l'appui, les choix exprimés lors de la conception de notre architecture. Enfin, nous situons notre travail par rapport à l'état de l'art dans le domaine en section 4 avant de conclure et de tracer des perspectives.

2. MapReduce

Le MapReduce est un modèle de programmation associé à une implémentation proposée initialement par Google [9]. Il permet le traitement et la génération de données volumineuses.

Le format des données en entrée est spécifié par l'utilisateur et dépend des applications. La sortie est un ensemble de couples $< \texttt{clé}, \texttt{valeur} >$. L'utilisateur décrit son algorithme en utilisant les deux fonctions *Map* et *Reduce*. La fonction *Map* prend un ensemble de données et produit une liste intermédiaire de couples $< \texttt{clé}, \texttt{valeur} >$. La fonction *Reduce* est appliquée à toutes les données intermédiaires et fusionne les résultats ayant la même clé. Un pseudo-code classique de ces fonctions est donné ci-dessous. Il s'agit d'un exemple qui compte le nombre d'occurrences de chaque mot dans un ensemble de documents.

```
void Map(key, string value) {
    // key document name
    // value : document contents
    for each word w in value {
        EmitIntermediate(w, "1");
    }
}
```

```
void Reduce(string key, list <string> values) {
    // key: a word
    // values : a list of contents
    int count = 0;
    for each v in values {
        count += StringToInt(v);
    }
    Emit(key, IntToString(count));
}
```

Un des principaux avantages de ce modèle est sa simplicité. Chaque *Job* MapReduce soumis par un utilisateur est une unité de travail composée de données en entrée, des deux fonctions *Map* et *Reduce* et d'informations de configuration. Le *Job* est alors automatiquement parallélisé et exécuté sur une grappe ou sur un ensemble de datacenters. La figure 1 présente l'architecture générale extraite de [1]. La donnée en entrée est découpée en M unités notées *splits*. Chaque *split* est traité en parallèle par une tâche *Map* invoquant la fonction *Map*. Les clés intermédiaires produites par les tâches *Map* sont divisées en R fragments, appelés *partitions*, qui sont distribués aux tâches *Reduce*. Les données des fragments reçus par une tâche *Reduce* sont alors triées par clé et traitées par la fonction *Reduce*. Comme les tâches *Map*, les tâches *Reduce* sont exécutées en parallèle.

Les données initiales et celles produites à la fin d'un MapReduce sont stockées dans un système de stockage distribué tolérant aux fautes tel que HDFS[22] ou Google File System [9]. Les résultats intermédiaires produits par les tâches *Map* sont en revanche uniquement gardés sur les disques des nœuds exécutant les tâches sans mécanisme particulier pour tolérer les fautes.

Les implémentations courantes reposent sur une architecture maître-esclave. Un MapReduce est soumis par l'utilisateur à un nœud maître qui choisit des nœuds esclaves libres et leur attribue des tâches *Map* ou *Reduce* à exécuter. Les fautes franches de nœud sont traitées automatiquement. Si un nœud est suspecté d'être défaillant, le maître relance ses tâches sur une nouvelle machine. En revanche, les défaillances du maître ne sont pas prises en compte. D'autre part, si un nœud exécutant des tâches est peu performant, le maître peut lancer de façon spéculative une copie de ses tâches sur un autre machine pour obtenir un résultat plus rapidement. Cela dit, le mécanisme de spéculation consiste à ordonnancer une réplique supplémentaire si le calcul de la tâche n'a pas été effectué dans un délai pré-défini.

Hadoop [22] est une implémentation open-source de MapReduce largement utilisée. Elle a été proposée par Apache et utilise le système de stockage HDFS (Hadoop Distributed File System). Dans HDFS, chaque fichier est découpé dans un ensemble de blocs de taille fixe (e.g., 64Mo) stockés dans des unités

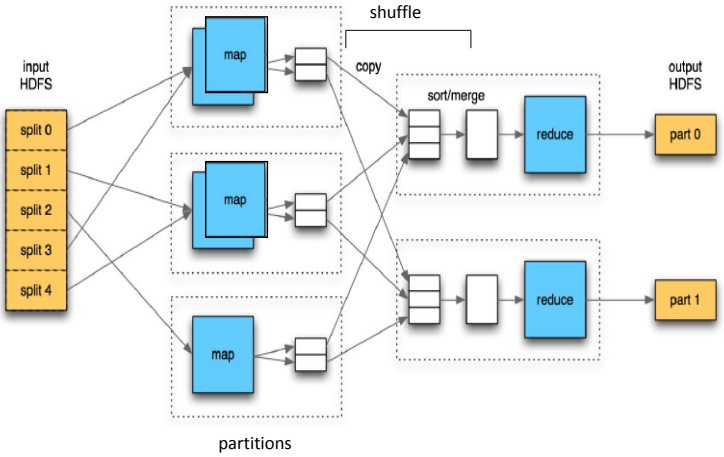


FIG. 1 – Architecture du MapReduce

indépendantes sur les disques de la plate-forme. HDFS fournit une haute disponibilité et fiabilité en répliquant chaque donnée avec un degré de réplication fixé par défaut à 3.
Pour contrôler l'exécution des tâches, Hadoop définit un maître unique, le *JobTracker*, qui gère l'état des *Job* soumis et ordonnance l'exécution des tâches. Sur les machines esclaves, un processus, le *TaskTracker*, contrôle la disponibilité des créneaux d'exécution de cette machine. Notez que pour améliorer les performances, les phases de calcul et les entrées/sorties peuvent être recouverts en exécutant plusieurs tâches *Map* ou *Reduce* en parallèle sur la même machine esclave. Chaque fois qu'un *TaskTracker* détecte un créneau disponible sur sa machine, il contacte le JobTracker pour réclamer une nouvelle tâche. S'il s'agit d'une tâche *Map*, le *JobTracker* prend en compte la localisation de l'esclave et choisit une tâche dont la donnée en entrée (son "split") est la plus proche possible de la machine du TaskTracker. En revanche pour les tâches *Reduce*, il n'est pas possible de prendre en compte la localité des données.

3. MapReduce tolérant les fautes byzantines

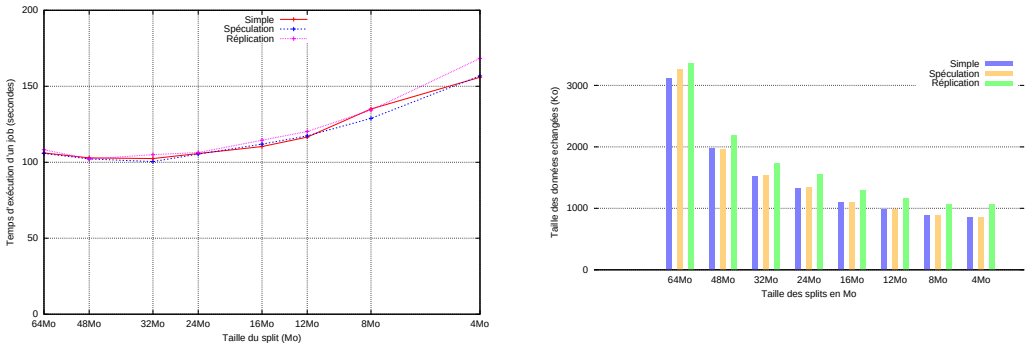
Nous proposons de traiter des pannes byzantines sur les *TaskTrackers*. Nous supposons que le *JobTracker* est fiable (comme proposé par *Hadoop* [22]), ainsi que le *HDFS*. En effet, pour ce dernier il existe dans la littérature des solutions qui permettent d'y assurer l'intégrité des données [7]. L'utilisation de *HDFS* dans notre architecture sera discutée dans la section 3.2. Enfin, on considère que les clients sont toujours corrects.
Le principe de notre approche repose sur l'utilisation de votes majoritaires sur le résultat de chaque tâche *Map* et *Reduce*. Si l'on considère que f est le nombre de fautes maximum pour une tâche, au moins $f+1$ résultats sur $2f+1$ répliques d'une tâche seront identiques. Autrement dit, il suffit d'avoir $f+1$ fois le même résultat pour le considérer comme correct. En pratique, on peut envisager deux approches : (1) ordonnancer directement $2f+1$ répliques en même temps pour chaque tâche et arrêter (supprimer des listes d'ordonnancement) les répliques toujours en cours lorsque l'on a obtenu $f+1$ résultats identiques ou (2) ne commencer que par $f+1$ répliques puis, le cas échéant, lancer des répliques supplémentaires. Le choix entre ces deux méthodes sera discuté en section 3.1.
La méthode du vote majoritaire implique aussi un autre choix pour l'architecture, celui du processus qui fera les comparaisons des résultats. Nous proposons ici de valider les résultats directement sur le *JobTracker*. Ce choix présente deux avantages : d'une part il simplifie l'arrêt ou l'ajout de nouvelles répliques si nécessaire, d'autre part il évite de multiplier les validations puisque le *JobTracker* est considéré comme fiable. Ce choix pose néanmoins un problème car dans le framework original de Hadoop MapReduce les résultats intermédiaires des différentes tâches de *Map* sont stockés en local sur les *TaskTrackers*. Une stricte comparaison sur le *JobTracker* impliquerait donc une forte charge réseau, la taille des résultats des *Maps* pouvant, suivant les applications, être relativement volumineux. Nous proposons donc de mettre

en place un mécanisme reposant sur l'utilisation de *sommes de contrôles* (*digests*) de façon à minimiser le surcoût en transferts ainsi que le temps de comparaison.
Le principe de notre approche est le suivant : à la fin de l'exécution d'une tâche de *Map* m, le *TaskTracker* calcule le *digest* du résultat de la tâche m (*digest global*) ainsi qu'un digest pour chacune des R partitions de ce résultat. L'ensemble de ces *digests* est ensuite envoyé par le *TaskTracker* au *JobTracker*. À chaque réception, ce dernier compare le *digest global* à ceux déjà reçu pour la tâche m et, dès lors qu'il obtient $f + 1$ *digest* résultats identiques, il considère le résultat comme valide et enregistre les *digests* de chaque partition de la tâche m.
La phase de *Reduce* commence lorsque le *JobTracker* a validé l'ensemble M résultats. Comme dans le framework initial, il va alors assigner chaque tâche *Reduce* r (et ses répliques) aux *TaskTrackers*. Nous proposons d'ajouter au message d'assignation l'ensemble des *digests* des partitions. Pour chacun de ces *digests* la tâche *Reduce* r va récupérer une copie valide de la partition correspondante : elle charge une copie depuis l'un des $f + 1$ *TaskTracker* qui ont calculé la tâche *Map* validée par le *JobTracker*, en calcule le *digest*, puis le compare avec l'un des *digests* reçus du *JobTracker*. On peut ainsi détecter une corruption de partition après validation (les deux *digests* ne correspondent pas). Dans ce cas, le *TaskTracker* exécutant le *Reduce* charge d'autres copies jusqu'à obtenir une copie valide. Notons que par hypothèse il ne peut y avoir plus de f corruptions ; au moins une des $f + 1$ copies correspondra donc au *digest* reçu.
Le calcul du *Reduce* r commence lorsque le *TaskTracker* en charge de cette tâche a récupéré et validé toutes les partitions nécessaires. Les résultats des *Reduces* sont ensuite validés sur le *JobTracker* par le mécanisme de *digest*, le *Job* ne se terminant que lorsque le *JobTracker* a validé tous les résultats des *Reduces* ($f + 1$ fois le même *digest* pour chaque tâche *Reduce*). Le mécanisme permettant de garantir la validité des résultats des *Reduces* sera discuté à la fin de la section 3.2.

3.1. Gestion des répliques

Comme nous l'avons vu précédemment, deux méthodes sont envisageables pour obtenir $f + 1$ résultats identiques par tâche : commencer par lancer $f + 1$ répliques puis en ajouter dynamiquement lorsque les réponses diffèrent ou lancer directement $2f + 1$ répliques puis stopper les répliques devenues inutiles lorsque l'on a obtenu les $f + 1$ réponses nécessaires. Dans cette section nous allons étudier ces deux approches.
Pour commencer détaillons le mécanisme de lancement d'une tâche dans la plate-forme *Hadoop*. Lorsqu'il décide de lancer une tâche, le *JobTracker* l'ajoute dans une liste de tâches à ordonnancer. Plusieurs implémentations d'ordonnanceur sont disponibles mais leur principe reste le même : il parcourt la liste et affecte si possible les tâches à des *TaskTrackers* dont la charge (le nombre de tâches à exécuter) est faible. Chaque ordonnanceur possède alors ses propres critères de choix des *TaskTrackers* : niveau de charge, utilisation de la localité des données, etc.
Chaque tâche est donc ajoutée une fois dans l'un des créneaux disponibles des *TaskTrackers*, mais *Hadoop* propose aussi d'ajouter un mécanisme dit de *spéculation* pour tolérer les pannes franches, comme décrit dans la section 2. Ce mécanisme de *spéculation* est en fait relativement proche de la première méthode proposée, puisqu'il s'agit d'ajouter dynamiquement des répliques supplémentaires. Mais cette approche, développée dans *Hadoop* pour tolérer un nombre non borné de pannes franches, sera-t-elle efficace pour tolérer au plus f fautes byzantines ? Ne vaut-il pas mieux pour gagner en réactivité ordonnancer directement $2f + 1$ répliques ? Pour répondre à ces questions, nous avons réalisé une expérience permettant de comparer le surcoût de l'ordonnancement des f répliques inutiles et de leur suppression par rapport à l'approche de réplication dynamique.
Nous avons donc implémenté une version d'*Hadoop* (notée *Hadoop* avec réplication) permettant de tolérer f pannes franches en ordonnançant $f + 1$ répliques de chaque tâche et supprimant les répliques inutiles dès qu'un résultat a été calculé. Nous avons ensuite comparé ses performances avec une version d'*Hadoop* utilisant la spéculation et sa version simple (sans spéculation, ni réplication). Pour cette étude nous avons lancé 15 *Job WorldCount* sur un fichier d'1Go dans une plate-forme composée d'un *JobTracker* et de 10 *TaskTrackers* (1 par machine d'un cluster dédié). Le nombre de *Reduces* a été fixé à 10 et les résultats présentés sont une moyenne excluant le premier et le dernier *Job* de façon à observer les phénomènes en régime stationnaire. f est fixée à 2.
Lors de ces expériences, nous avons fait varier la taille des *splits* de 64Mo à 4Mo de façon à mesurer tout effet de bord dû à la quantité de données utilisées et à la localité. Ainsi, les figures 2(a) et 2(b) présentent

pour chacun de ces *splits* respectivement le temps moyen pour calculer un *Job* et la quantité de messages de contrôle (assignation, localisation des données, etc) nécessaires.



(a) Temps d'exécution du MapReduce

(b) Volume des données échangées

FIG. 2 – Comparaison des stratégies de réplication

L'étude de ces figures montre deux choses : premièrement que l'ajout de la spéculation n'augmente pas en l'absence de panne, ni le temps d'exécution, ni la charge réseau et ce quelle que soit la taille du *split* ; deuxièmement que la réplication n'influence que peu le temps de calcul pour des tailles de *split* supérieures ou égales à 24Mo et l'augmente progressivement lorsque la taille du split tend à se rétrécir. Le surcoût réseau dû à la réplication est lui constant en volume mais son importance relative tend à diminuer lorsque le split augmente. Il est intéressant de remarquer que ce surcoût devient négligeable (par rapport à la spéculation) si l'on considère le réglage d'*Hadoop* par défaut de 64Mo et qu'il reste peu important pour un split de 32Mo qui correspond au réglage optimum, en terme de temps d'exécution, de ce benchmark classique pour cette plate-forme.

Ces résultats montrent donc que l'utilisation de la réplication impacte peu les performances du système. Ceci s'explique pour deux raisons :

– d'une part, l'assignation des tâches répliquées au *TaskTracker* par le *JobTracker* ne constitue pas un mécanisme à part entière. Il ne génère pas de message supplémentaire, mais se superpose (*piggybacking*) à certains messages de contrôle que s'échangent périodiquement le *JobTracker* et le *TaskTracker*. L'augmentation du nombre de répliques ne génère donc pas de message d'assignation supplémentaire.
– d'autre part, les tâches ne sont pas assignées sur des *TaskTracker* oisifs mais sont allouées à l'un des créneaux disponibles (*slot*) d'un *TaskTracker*. Par conséquent, il est possible qu'une réplique ne soit jamais assignée ou soit assignée puis annulée avant même d'avoir commencé. Nous avons donc modifié l'ordonnanceur de façon à allouer en priorité les répliques des tâches qui ont le moins de répliques déjà assignées. Dès que les tâches sont suffisamment longues, seule une réplique est réellement exécutée. On comprend alors pourquoi le surcoût augmente quand la taille du *split* devient trop petite.

Cette étude de performance nous a montré qu'il était possible d'utiliser la réplication initiale totale ($2f+1$ répliques) sans craindre pour les performances du système en l'absence de panne. Cette approche permettra entre autres d'optimiser le temps de réaction en cas de conflit dans les résultats, puisque les répliques supplémentaires seront déjà ordonnancées.

3.2. Utilisation d'un *HDFS* tolérant aux fautes byzantines

L'architecture d'*Hadoop* repose sur l'utilisation d'un système de fichier sous-jacent : *HDFS* (voir section 2). La conception d'un mécanisme de tolérance aux fautes byzantines peut donc passer par l'utilisation d'un *HDFS* garantissant la pérennité des données en présence d'éléments byzantins (*BFT-HDFS*) [7].

Dans la plate-forme originale, *HDFS* n'est utilisé que pour stocker les données d'entrée et le résultat final du *Job* ; toutes les autres données sont stockées en local sur les disques des différents *TaskTrackers*. Une solution serait donc d'étendre l'utilisation d'un *BFT-HDFS* pour fiabiliser tout ou une partie de ces

résultats intermédiaires.
Pour mesurer le surcoût d'une telle approche, nous avons instrumenté le code original de la plate-forme et mené une étude quantitative sur l'importance des transferts des données locales entre les *Maps* et les *Reduces*. Ainsi, nous avons ajouté des sondes dans les tâches *TaskTracker* pour mesurer la durée des différentes phases lors du calcul d'une tâche de type *Reduce*. La figure 3 indique le durée du temps passé à télécharger les différentes partitions nécessaires au calcul de la fonction *Reduce* (noté *download*), le temps passé à trier les données (noté *sort*) et le temps de calcul à proprement parler (noté *reduce*).

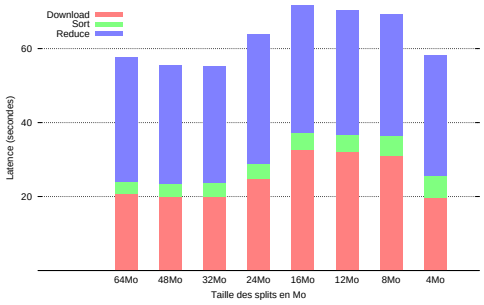


FIG. 3 – Durées des phases d'un *Reduce*

Les résultats de cette étude montrent que la phase de *download* constitue une part importante de la tâche. Par conséquent, une simple utilisation d'un *BFT-HDFS* pour fiabiliser les données locales entraînerait un surcoût relativement important. Une solution permettant de l'amortir serait d'exploiter la réplication sous-jacente pour augmenter la localité des données. Cependant, cette solution demanderait à réimplémenter tous les ordonnanceurs, car ceux-ci n'exploitent pas la localité pour l'assignation des *Reduces*. Il faudrait aussi modifier *BFT-HDFS* de façon à synchroniser tous les placements des *Maps* en regroupant les partitions utilisées par un même *Reduce*. En effet, les reduces n'utilisent pas le résultat de tel ou tel *Map*, mais une combinaison de partitions de plusieurs *Maps*.
L'utilisation d'un *BFT-HDFS* reste cependant utile pour fiabiliser l'*input* et l'*output* du *Job*. La validation du résultat final, enregistré sur le *BFT-HDFS*, sera faite par le *JobTracker* à partir du *digest* validé. Ce système à toutefois quelques limites, notamment si l'on considère des applications utilisant un chaînage de plusieurs MapReduce [16]. En effet, dans ce cas les résultats des Reduces, servant d'input aux *Map* de la phase suivante, sont laissées en local sur les disques des *TaskTrackers*. Une solution peut être de réutiliser notre approche basée sur les *digests*, en conservant $f + 1$ copies des résultats *Reduces* ainsi que les *digests* associés.
Nous avons donc, dans un premier temps, choisi de ne pas étendre l'utilisation du *BFT-HDFS* aux données intermédiaires. En contre-partie, nous proposons de conserver localement $f+1$ copies (validées par le *JobTracker*) de tous les *Maps*, et ce jusqu'à validation de la terminaison du *Job*. Ainsi, on se prémunit contre toute corruption, après validation (voir mécanisme des *digests*), des résultats intermédiaires.

4. Travaux connexes

Il y a peu de travaux sur la sécurité et le traitement des fautes Byzantines dans les architectures MapReduce [21]. Dans [4], nous avons proposé une version préliminaire de notre solution. À notre connaissance, seuls [21] et [15] traitent explicitement de la sécurité dans les architectures MapReduce. Les auteurs de [21] proposent un protocole distribué pour vérifier la validité des résultats produits par les tâches *Map* et *Reduce*. Le principe est proche de notre solution : de manière similaire aux digests, un "commitment" est envoyé au maître qui le valide et le retransmet aux exclaves. Ces derniers peuvent alors réclamer les données et vérifier leur contenu en regénérant les commitments. En revanche, les auteurs abordent peu la réplication et n'adressent pas les problèmes liés au nombre de copies. [15] propose un mécanisme distribué pour valider les résultats produits pour un MapReduce dans le cadre de

DesktopGrid. Chaque tâche (map et reduce) est répliquée en r exemplaires. Pour éviter de surcharger le maître, les résultats intermédiaires des *Maps* sont directement envoyés aux *Reduces*. Chaque tâche *Reduce* reçoit donc r copies des partitions. La partition est supposée valide si $r/2 + 1$ copies sont identiques. En revanche, les répliques de résultats produits par les tâches *Reduce* sont envoyés directement au master qui effectue la validation finale. Un mécanisme de digest permet également de vérifier que les données produites par un *Map* correspondent bien à un *split* en entrée. Même, si ce protocole évite de surcharger les masters, la quantité de données échangées entre les *Maps* et *Reduces* est directement proportionnelle au degré de réplication, ce qui peut entraîner une charge de communication importante si les données produites sont volumineuses.

D'autres travaux se rapprochent également de notre problématique. [18, 20] montrent qu'il est nécessaire de fournir des services distribués s'exécutant de manière continue malgré toutes sortes de fautes, y compris celles byzantines. D'autre part, ils affirment que les protocoles classiques BFT [6, 13, 8] basés sur une cohérence forte et où le système de réplication apparaît aux clients comme un seul serveur séquentiel correct sont très coûteux dans les grands systèmes en termes de disponibilité et de performance. Pour surmonter ces contraintes, certains auteurs ont proposé d'assouplir la cohérence des protocoles BFT. Par exemple, dans [18], les auteurs proposent le protocole Zeno qui assure uniquement à terme une tolérance aux fautes byzantines cohérente. BFT2F [14] quant à lui prévoit, lorsqu'il y a entre f et $2f$ pannes, une cohérence faible appelée fork* qui limite le type de violations pouvant être fait par les serveurs malveillants. Une discussion intéressante à propos de l'importance de la tolérance aux pannes byzantines dans les datacenters peut être trouvée dans [20].

Bhatotia et al. [5] affirment que le modèle des fautes franches et byzantines n'est pas adapté aux datacenters. Ils présentent ainsi une nouvelle approche pour traiter des fautes non-franches et non-byzantines. Plus précisément, ils proposent un mécanisme de détection de panne pour Pig, un système de traitement de données volumineuses [16]. Pig offre un langage de haut niveau, Pig Latin, qui permet de faire des requêtes semblables à SQL. Le programme réalise ensuite une série d'étapes de compilation qui produisent un MapReduce pouvant être exécuté dans un cluster Hadoop. L'idée de base est que Pig Latin possède un nombre limité de constructions ayant une sémantique basée sur l'algèbre relationnelle. Des vérifications sur la sémantique des constructions permettent alors de détecter les fautes à l'exécution des programmes Pig.

Dans [10, 11], les auteurs considèrent un système distribué composé d'un processeur maître fiable et d'un ensemble de n processeurs esclaves exécutant les tâches. Les esclaves peuvent être byzantins. Chaque tâche renvoie une valeur binaire. L'objectif du maître est d'accepter avec une forte probabilité les valeurs correctes tout en minimisant la quantité de travail (le nombre de processeurs exécutant la tâche). Le nombre d'esclaves fautifs est borné soit par une valeur fixe $f < n/2$, soit en considérant une probabilité $p < 1/2$ de défaillance des processeurs. Les auteurs démontrent alors qu'il est possible d'obtenir avec une grande probabilité un niveau correct de succès avec une faible surcharge.

Enfin, [7] propose la bibliothèque UpRight dont l'objectif est, par réplication, d'aider à rendre le système tolérant aux fautes byzantines. Les auteurs ont utilisé UpRight pour rendre HDFS tolérant aux byzantins.

5. Conclusion

Dans cet article nous avons proposé une nouvelle architecture pour le *framework MapReduce* permettant de tolérer f fautes byzantines par tâche. Basée sur un mécanisme de réplication des tâches et sur l'envoi de somme de côntrole (*digest*), notre approche tend à minimiser le surcoût en message et en temps d'exécution. Certains choix architecturaux s'appuient sur des résultats expérimentaux en environnement réel. Nous avons montré qu'il était possible, en modifiant l'ordonnanceur, de lancer provisoirement $2f + 1$ répliques avec un faible surcoût aditionnel. De plus, nous avons étudié l'utilisation d'un HDFS tolérant aux fautes byzantines, notamment pour la fiabilisation des données intermédiaires.

Réalisé dans le cadre du projet *FTH-Grid* [12], ce travail ouvre de nombreuses perspectives en cours de réalisation : intégration complète de l'architecture sur *Hadoop*, extension des expérimentations à d'autres benchmarks ainsi qu'aux applications basées sur un chaînage de plusieurs MapReduce.

Bibliographie

1. Apache. Hadoop. `http://hadoop.apache.org/mapreduce/`.
2. Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, et Matei Zaharia. Above the clouds : A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.
3. Luiz André Barroso et Urs Hölzle. The datacenter as a computer : An introduction to the design of warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, 4(1) :1–108, 2009.
4. A. N. Bessani, V. V Cogo, M. Correia, P. Costa, M. Pasin, F. Silva, L. Arantes, O. Marin, P. Sens, et J. Sopena. Making Hadoop MapReduce Byzantine Fault-Tolerant. Fast abstract. In *40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2010)*, June 2010.
5. Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Flavio Junqueira, et Benjamin Reed. Reliable data-center scale computations. In *LADIS*, 2010.
6. Miguel Castro et Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4) :398–461, 2002.
7. Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Michael Dahlin, et Taylor Riche. Upright cluster services. In *SOSP*, pages 277–290, 2009.
8. James A. Cowling, Daniel S. Myers, Barbara Liskov, Rodrigo Rodrigues, et Liuba Shrira. Hq replication : A hybrid quorum protocol for byzantine fault tolerance. In *OSDI*, pages 177–190, 2006.
9. Jeffrey Dean et Sanjay Ghemawat. Mapreduce : Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.
10. Antonio Fernandez, Chryssis Georgiou, Luis López, et Agustín Santos. Reliably executing tasks in the presence of malicious processors. In *DISC*, pages 490–492, 2005.
11. Antonio Fernandez, Luis Lopez, Agustin Santos, et Chryssis Georgiou. Reliably executing tasks in the presence of untrusted entities. *Reliable Distributed Systems, IEEE Symposium on*, 0 :39–50, 2006.
12. Projet international EGIDE/PESSOA. Fth-grid. `http://fth-grid.di.fc.ul.pt/index.php?title=Public:Main_Page`.
13. Ramakrishna Kotla, Lorenzo Alvisi, Michael Dahlin, Allen Clement, et Edmund L. Wong. Zyzzyva : Speculative byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4), 2009.
14. Jinyuan Li et David Mazières. Beyond one-third faulty replicas in byzantine fault tolerant systems. In *NSDI*, 2007.
15. Mircea Moca, Gheorghe Cosmin Silaghi, et Gilles Fedak. Distributed results checking for mapreduce in volunteer computing. In *Fifth Workshop on Desktop Grids and Volunteer Computing Systems (PCGrid 2011) held in conjunction with the IEEE International Parallel and Distributed Processing Symposium*, 2011.
16. Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, et Andrew Tomkins. Pig latin : a not-so-foreign language for data processing. In *SIGMOD Conference*, pages 1099–1110, 2008.
17. Rodrigo Rodrigues, Petr Kouznetsov, et Bobby Bhattacharjee. Large-scale byzantine fault tolerance : safe but not always live. In *Proceedings of the 3rd workshop on on Hot Topics in System Dependability*, Berkeley, CA, USA, 2007. USENIX Association.
18. Atul Singh, Pedro Fonseca, Petr Kuznetsov, Rodrigo Rodrigues, et Petros Maniatis. Zeno : Eventually consistent byzantine-fault tolerance. In *NSDI*, pages 169–184, 2009.
19. M.H. Valipour, B. Amirzafari, K.N. Maleki, et N. Daneshpour. A brief survey of software architecture concepts and service oriented architecture. In *Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on*, pages 34 –38, Aug 2009.
20. Robbert van Renesse, Rodrigo Rodrigues, Mike Spreitzer, Christopher Stewart, Doug Terry, et Franco Travostino. Challenges facing tomorrow's datacenter : summary of the ladis workshop. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, pages 1–7, 2008.
21. Wei Wei, Juan Du, Ting Yu, et Xiaohui Gu. Securemr : A service integrity assurance framework for mapreduce. In *Twenty-Fifth Annual Computer Security Applications Conference, (ACSAC 2009), Honolulu, Hawaii*, pages 73–82, 2009.
22. Tom White. *Hadoop : The Definitive Guide*. Oreilly, first edition edition, 2009.