ComPAS'2013 (RenPar / SympA / CFSE)
Grenoble, France, du 16 au 18 janvier 2013

Évaluer la tolérance aux fautes de systèmes MapReduce

Laurent Lemke[†], Amit Sangroya[*], Damián Serrano[*], Sara Bouchenak[*]

[*]Université de Grenoble I – Laboratoire LIG
{Amit.Sangroya, Damian.Serrano, Sara.Bouchenak}@imag.fr

[†] Université de Grenoble I, Grenoble
lemkel@e.ujf-grenoble.fr

Résumé
MapReduce est un modèle de programmation et un environnement d'exécution populaires, permettant le traitement de grands volumes de données dans des environnements répartis à plus ou moins grande échelle. De nombreux travaux de recherche ont été menés afin d'améliorer la tolérance aux fautes de MapReduce, travaux allant de la prise en compte de nouveaux modèles de fautes à des approches adaptatives de tolérance aux fautes. Cependant, ces travaux ont été évalués pour la plupart avec des micro-benchmark simples, ne permettant de capturer la diversité et le dynamisme d'applications réelles. Cet article présente MRBS, un environnement complet d'évaluation et d'analyse de la tolérance aux fautes de systèmes MapReduce. MRBS considère plusieurs domaines d'applications MapReduce et illustre divers scénarios d'exécution : des applications orientées données vs. des applications orientées calcul, des applications interactives vs. des applications par lots. Dans ce contexte, MRBS permet d'injecter dans des systèmes MapReduce en ligne différents types de fautes à différentes fréquences, et de produire des statistiques détaillées en termes de fiabilité, disponibilité et performance. Cet article illustre l'utilisation de MRBS à travers des évaluations et une étude de cas menées dans des grappes Hadoop MapReduce hébergées dans le cloud public Amazon EC2 ou dans une grappe privée.

Mots-clés : Benchmark ; Tolérance aux fautes ; Performance ; MapReduce ; Hadoop

---

1. Introduction

MapReduce est un modèle de programmation et un environnement d'exécution populaires et largement utilisés par des applications réparties [21]. Il offre une solution élégante pour le traitement de grands volumes de données et l'exécution de grands volumes de calcul, dans des environnements distribués à plus ou moins grande échelle. Les systèmes MapReduce offrent ainsi des solutions gérant automatiquement le partitionnement et la duplication des données pour une meilleure tolérance aux pannes, des techniques d'ordonnancement des tâches de calcul visant à améliorer les performances. Hadoop, un des environnements MapReduce les plus populaires, fournit des solutions de tolérance aux fautes en cas de pannes matérielles de machines ou en cas de fautes logicielles lors du calcul [4].
Dans ce contexte, de nombreux travaux ont été menés pour améliorer la tolérance aux fautes de systèmes MapReduce. Certains travaux se sont intéressés à la tolérance aux fautes à la demande [26], d'autres ont investigué les techniques de duplication et de partitionnement [15, 24], des approches de tolérance aux fautes adaptative [29, 32], ou l'extension de MapReduce avec de nouveaux modèles de fautes [16, 31]. Cependant, il y a eu très peu de travaux sur l'évaluation empirique de la tolérance aux fautes de MapReduce. Les travaux cités précédemment ont, pour la plupart, été évalués de manière ad-hoc, en tuant 'manuellement' un processus en cours d'exécution ou en éteignant une machine. Une telle approche permet, en effet, d'effectuer des tests préliminaires mais ne permet malheureusement pas une évaluation complète dans des scénarios de pannes plus complexes. Par ailleurs, des outils récents sont apparus tels que l'environnement d'injection de fautes de Hadoop HDFS, qui permet d'émuler des exceptions non déterministes dans le système distribué Hadoop HDFS sous-jacent à MapReduce [6]. Un tel outil

permet de mettre en place des tests unitaires pour HDFS, il s'adresse plutôt à des développeurs qui ont une connaissance des détails techniques internes à HDFS, et non pas aux utilisateurs finaux de MapReduce.
À ce jour, la problématique d'injection automatique de fautes dans MapReduce, avec usage simple et de haut niveau, reste ouverte. De plus, il est nécessaire de pouvoir caractériser une *charge de fautes* dans MapReduce pour décrire *quelle type* de faute (p. ex. une panne de machine), *où* injecter la faute (p. ex. quelle machine dans une grappe MapReduce), et *quand* injecter la faute (p. ex. une heure après le démarrage du système). Par ailleurs, la plupart des solutions de tolérance aux fautes de MapReduce ont été évalués avec des micro-benchmarks basés sur des programmes MapReduce simples tels que *grep*, *sort* ou *word count*. De tels micro-benchmarks peuvent être utiles pour effectuer des tests spécifiques et ponctuels, mais ils ne sont malheureusement pas représentatifs d'applications distribuées complètes et ne fournissent pas de charge multi-utilisateurs réaliste.
Dans cet article, nous présentons MRBS (MapReduce Benchmark Suite), le premier ensemble de bancs d'essai permettant d'évaluer la tolérance aux fautes de systèmes MapReduce. MRBS permet la génération et l'injection automatique de fautes dans MapReduce, en considérant différents types de fautes, injectées à différentes fréquences. Ceci permet d'analyser et évaluer la pertinence d'algorithmes et techniques de tolérance aux fautes dans différents scénarios, ou de comparer différents algorithmes entre eux. MRBS permet ainsi de quantifier le niveau de tolérance aux pannes des solutions proposées, via une évaluation empirique de la disponibilité et de la fiabilité du système MapReduce. Ces métriques de tolérance aux pannes sont complétées par des métriques de performance et de coût, pour évaluer l'impact des fautes dans MapReduce sur ces métriques.
MRBS considère cinq domaines d'applications : les systèmes de recommandation, le domaine de business intelligence, la bioinformatique, l'analyse de données textuelles, ou l'extraction de connaissances. MRBS propose des charges de calcul et des volumes de données à traiter aux caractéristiques diverses, allant des applications orientées calcul aux applications orientées données, ou des applications interactives aux application par lots (i.e. exécutées en mode *batch*). MRBS utilise des ensembles de données réelles, telles que les données issues d'un site de recommandation en ligne [9], des données Wikipedia [14], ou des génomes réels [7].
Cet article illustre l'utilisation de MRBS pour évaluer la tolérance aux pannes de Hadoop MapReduce dans le cloud public Amazon EC2 et dans un cluster privé. Les résultats des expériences menées avec MRBS montrent que dans le cas d'applications de bioinformatique, Hadoop MapReduce est capable de faire face à une centaine de fautes logicielles et à trois pannes de machines avec un haut niveau de fiabilité (94% de requêtes réussies) et un haut niveau de disponibilité de service (96% du temps). Par ailleurs, l'article présente une étude de cas illustrant l'utilisation de MRBS.
La suite de l'article est organisée comme suit. La Section 2 présente des rappels et définitions des systèmes MapReduce. La Section 3 décrit la solution proposée permettant l'évaluation de la tolérance aux pannes dans MapReduce. La Section 4 présente les résultats de l'évaluation expérimentale, et la Section 5 décrit une étude de cas de MRBS. Les Sections 6 et 7 présentent, respectivement, les travaux apparentés et nos conclusions.

2. Modèle du système sous-jacent

MapReduce est un modèle de programmation basé sur deux fonctions principales : la fonction *map* qui traite les données en entrée et génère des résultats intermédiaires, puis la fonction *reduce* qui se base sur les données intermédiaires pour produire le résultat final. MapReduce est également un environnement d'exécution parallèle sur plusieurs machines (i.e. nœuds) d'une grappe. L'exécution d'un programme MapReduce est représentée par un *job*. Un *job* est constitué de plusieurs *tâches* exécutées en parallèle, ces tâches étant responsables de l'exécution de fonctions *map*, *reduce*, etc.

2.1. Hadoop MapReduce

Hadoop est un environnement MapReduce largement utilisé [4]. Un cluster Hadoop est constitué d'un *nœud maître* et de *nœuds esclaves*. Les utilisateurs de Hadoop soumettent l'exécution de leurs applications (i.e. *jobs*) au nœud maître de la grappe Hadoop. Le nœud maître héberge le démon *JobTracker* qui est responsable de l'ordonnancement des jobs. Hadoop inclut plusieurs algorithmes d'ordonnancement.

Par défaut, l'ordonnancement FIFO exécute les jobs des utilisateurs dans leur ordre d'arrivée, les uns à la suite des autres. Ici, un job et ses tâches sous-jacentes utiliseront exclusivement la grappe Hadoop jusqu'à la terminaison du job, puis ce sera au tour du job suivant. L'ordonnanceur *Fair Scheduler* permet, quant à lui, d'exécuter des jobs MapReduce de manière concurrente sur une même grappe de machines, en affectant à chaque job une part d'utilisation de la grappe dans le temps. Par ailleurs, chaque nœud esclave héberge un démon nommé *TaskTracker* qui communique périodiquement avec le nœud maître pour indiquer qu'il est prêt à exécuter de nouvelles tâches ordonnancées par le nœud maître. Sur un nœud esclave, chaque tâche est exécutée par un processus séparé.
Hadoop MapReduce est également basé sur HDFS, un système de gestion de fichiers distribués qui permet de stocker les données des applications MapReduce sur les nœuds de la grappe. HDFS se compose d'un démon *NameNode* et de démons *DataNodes*. Le *NameNode* est généralement hébergé par le nœud maître ; il est responsable de la gestion de l'espace de noms du système de fichiers et régule les accès aux fichiers. Un *DataNode* est hébergé par un nœud esclave et gère le stockage des données sur le nœud concerné. Par ailleurs, pour une meilleure tolérance aux pannes, HDFS duplique les données sur plusieurs nœuds.

2.2. Modèles de fautes dans Hadoop MapReduce

Hadoop MapReduce a la capacité de tolérer plusieurs types de fautes [37], que nous décrivons ci-dessous.
Panne de nœud. Un nœud d'une grappe MapReduce peut tomber en panne à n'importe quel moment. La version courante de Hadoop MapReduce tolère les pannes de nœuds esclaves. Le nœud maître détecte la panne d'un nœud esclave si le *JobTracker* sur le nœud maître ne reçoit plus de signaux *heartbeat* de la part du *TaskTracker* du nœud esclave. Dans ce cas, le *JobTracker* ôte ce nœud esclave de la liste des nœuds disponibles et re-soumet les tâches qui étaient en cours sur le nœud vers d'autres nœuds. La panne du nœud maître n'est pas tolérée dans la version actuelle de Hadoop.
Faute logicielle de tâche. Une tâche peut s'arrêter suite à une erreur ou une exception dans la fonction *map* ou la fonction *reduce* écrites par le programmeur de l'application MapReduce. Dans ce cas, le *TaskTracker* du nœud hébergeant la tâche notifie le *JobTracker* du nœud maître qui ordonne la re-exécution de la tâche défaillante jusqu'à un nombre limite de tentatives (quatre par défaut), au-delà duquel la tâche est considérée comme défaillante, et le job de cette tâche également.
Arrêt de tâche. Le processus exécutant une tâche peut s'arrêter inopinément, suite à un bogue transitoire dans la machine virtuelle sous-jacente par exemple. Dans ce cas, le *TaskTracker* du nœud hébergeant la tâche notifie le *JobTracker*, qui réordonnancera la tâche tel que décrit précédemment.
Blocage de tâche. Une tâche map ou reduce dont l'exécution reste bloquée sans progrès durant un certain temps est considérée comme défaillante. Lorsque le *TaskTracker* du nœud hébergeant la tâche ne reçoit plus de notification de progression de la tâche, le processus exécutant cette tâche est tué et le *JobTracker* est notifié pour une possible re-exécution de la tâche tel que décrit précédemment.

3. Évaluer la tolérance aux fautes de MapReduce

Nous proposons, dans cet article, la première solution permettant d'évaluer la tolérance aux fautes de systèmes MapReduce. Nous présentons également une mise en œuvre de la solution proposée pour le système Hadoop MapReduce, à travers le logiciel MRBS que nous avons développé et qui est disponible ici : *http ://sardes.inrialpes.fr/research/mrbs/*
L'évaluation de la tolérance aux fautes de systèmes MapReduce requiert plusieurs éléments tels que définir les fautes à étudier, injecter ces fautes dans un système MapReduce en cours d'exécution et analyser l'impact de ces fautes sur la fiabilité, la disponibilité et la performance du système. Ceci est brièvement illustré par la Figure 1 et décrit plus en détails dans la suite.

3.1. Quelle charge de fautes

Une charge de fautes décrit les types de fautes qui surviennent et à quels moments ces fautes surviennent. Dans MRBS, il est possible de décrire via un fichier une charge de fautes soit par *extension*, soit par *intention*. Un fichier de charge de fautes par extension contient plusieurs lignes, chaque ligne étant constituée des éléments suivants : l'instant d'occurrence de la faute, le type de faute et, éventuellement, le lieu (i.e. nœud) d'occurrence ; ce dernier peut également être choisi de manière aléatoire parmi

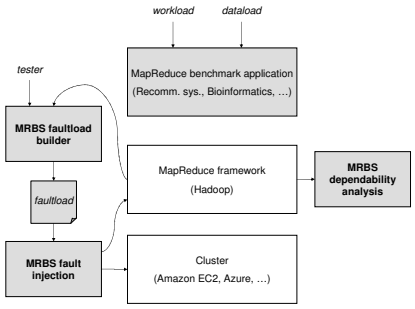


FIGURE 1 – Évaluer la tolérance aux fautes avec MRBS

l'ensemble des nœuds de la grappe du système MapReduce. Un fichier de charge de fautes par intention permet une description plus succincte où chaque ligne du fichier indique un type de faute et l'intervalle de temps moyen entre deux occurrences de ce type de fautes (i.e. *MTBF : Mean Time Between Failures*).
Un fichier de charge de fautes peut être construit des trois manières suivantes. L'utilisateur peut, par exemple, explicitement définir ses charges de fautes dans le but de tester ses propres scénarios. Le fichier de charge de fautes peut également être généré automatiquement et de manière aléatoire, pour tester des scénarios synthétiques. Enfin, MRBS peut automatiquement analyser des traces d'exécutions précédentes d'un système MapReduce pour en extraire la charge de fautes.

3.2. Comment injecter les fautes

L'injection d'une faute dans un système MapReduce avec MRBS est traitée différemment selon le type de faute considéré, comme décrit ci-dessous.
Panne de nœud. Ce type de faute est mis en œuvre par simple arrêt du nœud. Cet injecteur de faute se base sur l'interface de l'infrastructure sous-jacente pour commander l'arrêt du nœud. Dans le cas d'un cloud public tel que Amazon EC2, l'injection d'une faute de ce type s'effectue en terminant l'instance Amazon EC2 correspondante. Une mise en œuvre plus efficace de cet injecteur de faute consiste à tuer l'ensemble des processus du système MapReduce hébergés sur le nœud. Dans le cas de Hadoop, ceci revient à tuer les démons *TaskTracker* et *DataNode* du nœud.
Arrêt de tâche. L'injection de ce type de faute consiste à tuer le processus exécutant une tâche MapReduce hébergée par un nœud de la grappe MapReduce.
Faute logicielle de tâche. Ce type de faute est implanté par une exception logicielle qui est levée lors de l'exécution d'une tâche map ou d'une tâche reduce. Cet injecteur de faute est mis en œuvre par un intercepteur ajouté à la bibliothèque d'appel de fonctions map et reduce dans le logiciel Hadoop MapReduce, ce qui permet de traiter de manière générique l'injection de ce type de faute quelle que soit l'application MapReduce considérée et sans modification du code de cette application. Il est à noter que pour plus de portabilité de MRBS, aucune modification manuelle du logiciel Hadoop MapReduce n'a été effectuée pour la mise en œuvre de cet injecteur de faute. Au lieu de cela, MRBS synthétise automatiquement une nouvelle version du logiciel Hadoop MapReduce en utilisant des techniques de programmation par aspect (*AOP : Aspect-Oriented Programming*). La nouvelle version ainsi produite a la même interface API (*Application Programming Interface*) que le version originale de Hadoop, mais inclut des intercepteurs qui, lors de la création d'une tâche, vont servir à créer une exception qui représentera la faute logicielle injectée. L'intercepteur lèvera effectivement une exception à l'instant ou à la fréquence indiqués par le fichier de charge de fautes considéré.
Blocage de tâche. Ce type de faute est implanté par une pause prolongée lors de l'exécution d'une tâche map ou d'une tâche reduce. Le temps de pause provoqué ici est bien plus long que la période de temps nécessaire à détecter le blocage d'une tâche par le système MapReduce (ex. propriété *mapred.task.timeout* dans Hadoop). Comme précédemment, cet injecteur de faute est mis en œuvre par un intercepteur ajouté automatiquement au logiciel Hadoop MapReduce.
Ainsi, dans MRBS les informations de charge de fautes sont communiquées à un démon responsable de

l'orchestration de l'injection de fautes dans la grappe MapReduce. Les fautes de type panne de nœud ou arrêt de tâche sont injectées par ce démon et les fautes de type faute logicielle de tâche ou blocage de tâche sont injectées séparément dans les nœuds de la grappe MapReduce, tel qu'illustré par la Figure 2.

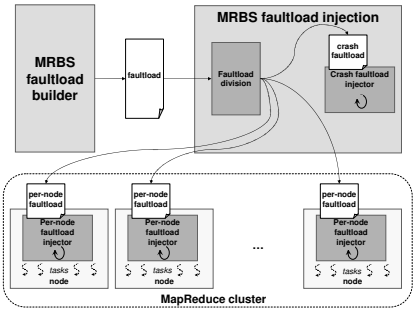


FIGURE 2 – Architecture de l'injecteur de fautes de MRBS

3.3. Bancs d'essai

En plus de l'injecteur de fautes, MRBS vient avec un ensemble de fonctionnalités décrites dans la Figure 3. En effet, afin d'évaluer la tolérance aux fautes d'un système MapReduce, il est nécessaire d'avoir des applications MapReduce munies d'ensembles de données à traiter, de créer une charge de travail réaliste sur ces applications (i.e. *workload*), de disposer d'outils de déploiement des expériences sur des infrastructures de grappes et de clouds publics ou privés, et de bénéficier d'outils de mesure et d'analyse de la tolérance aux fautes et de la performance du système MapReduce. Ces différentes fonctionnalités sont détaillées par la suite.

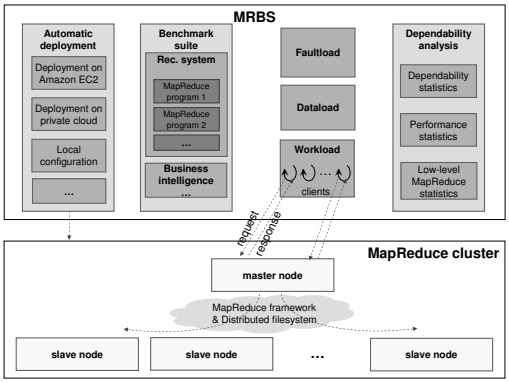


FIGURE 3 – Vue d'ensemble de MRBS

3.3.1. Applications

MRBS inclut un ensemble de bancs d'essais (i.e. *benchmark*) traitant de différents domaines d'applications : les systèmes de recommandation, la business intelligence, la bioinformatique, l'analyse et traitement de données textuelles, l'extraction de connaissances. L'objectif ici est d'avoir un panel de bancs d'essais aux comportements différents, certains étant orientés calcul et d'autres orientés données, tel

que nous l'avons présenté dans une étude précédente [34]. Chaque banc d'essai de MRBS représente un service en ligne fournissant différents types de requêtes. Le banc d'essai est déployé sur une grappe MapReduce et est accessible via le réseau par un ou plusieurs clients. Chaque client émet une requête au nœud maître de la grappe MapReduce, attend de recevoir sa réponse avant d'envoyer une autre requête. La requête d'un client est traitée par la grappe MapReduce via l'exécution d'un ou plusieurs *jobs* MapReduce successifs, en fonction de la complexité de la requête. Les différents bancs d'essais de MRBS sont décrits ci-dessous, plus de détails peuvent être trouvés dans [34].

– *Recommendation System.* Ce banc d'essai met en œuvre un service de recommandation de films en ligne. Les utilisateurs de ce service soumettent leurs évaluations et préférences de films et le service produit des recommandations de films en fonction des profils des utilisateurs. Ce banc d'essai fournit quatre types de requêtes : (i) les dix meilleurs recommandations pour un utilisateur, (ii) les évaluations données à un film, (iii) les évaluations faites par un utilisateur, et (iv) la recommandation d'un film pour un utilisateur donné. Ce banc d'essai s'appuie sur les données réelles d'un site web de recommandation de films [9].
– *Business Intelligence.* Ce banc d'essai implante un service de support à la décision ; il est compatible avec le standard industriel TPC-H [13]. Il propose 22 types de requêtes à ses clients, des requêtes pouvant être complexes et qui sont mises en œuvre avec le langage de requête Hive QL . Ce banc d'essai utilise huit tables de données, dont le contenu est généré de manière synthétique et en respectant les spécifications de TPC-H [13]. Ainsi, il est possible d'utiliser ce banc d'essai avec des volumes de données plus ou moins importants, en fonction des scénarios considérés.
– *BioInformatics.* Ce service effectue un séquençage de l'ADN de génomes réels. Il fournit actuellement trois types de requêtes au client, pour analyser des génomes plus ou moins complexes. Les données utilisées par ce service sont des génomes réels disponibles publiquement [7].
– *Test Processing.* Ce banc d'essai met en œuvre un service de traitement et d'analyse de documents textuels. Il fournit trois types de requêtes qui permettent de rechercher un mot ou un pattern de mot dans des documents textes, de calculer le nombre d'occurrences d'un mot dans ces documents, ou de trier le contenu d'un document. Ces opérations sont classiquement utilisées par les moteurs de recherche et les analyseurs de fichiers journaux (*logs*). Ce banc d'essai utilise des jeux de données de taille variable provenant de Wikipedia [14].
– *Data Mining.* Ce banc d'essai est un service proposant des algorithmes d'extraction de connaissances, tels que des algorithmes de classification et de regroupement (i.e. *clustering*). Il fournit deux types de requête au client : une requête de classification des données dans différentes catégories et une requête de regroupement des données basée sur le *canopy clustering*. Ce banc d'essai utilise des jeux de données synthétiques et des données publiques [1].

3.3.2. Données

Les bancs d'essai de MRBS incluent des applications MapReduce qui traitent et analysent des ensembles de données. Ces données se caractérisent par leur taille et leur nature qui diffère d'un banc d'essai à l'autre. Par exemple, pour le système de recommendation de films, les données se composent de films, d'utilisateurs, et de notes données par des utilisateurs aux films. MRBS permet ainsi de choisir, pour un banc d'essai donné, la taille des données utilisées tel que décrit dans le Tableau 1.

TABLE 1 – Volume de données

Banc d'essai	**Volume de données**
Recommendation system	dataload 100,000 ratings, 1000 users, 1700 movies dataload+ 1 million ratings, 6000 users, 4000 movies dataload++ 10 million ratings, 72,000 , 10,000 movies
Business intelligence	dataload 1GB dataload+ 10GB dataload++ 100GB
Bioinformatics	dataload genomes of 2,000,000 to 3,000,000 DNA characters
Text processing	dataload text files (1GB) dataload+ text files (10GB) dataload++ text files (100GB)
Data mining	dataload 5000 documents, 5 newsgroups, 600 control charts dataload+ 10,000 documents, 10 newsgroups, 1200 control charts dataload++ 20,000 documents, 20 newsgroups, 2400 control charts

3.3.3. Charge de travail

La charge de travail dans MRBS se caractérise par différents aspects. Une fois un banc d'essai MRBS sélectionné, le nombre de clients concurrents accédant au service MapReduce peut être paramétré pour générer une charge plus ou moins importante. Il est également possible de choisir le mode d'exécution du service MapReduce entre : le mode interactif où les requêtes des clients concurrents s'exécutent en (pseudo-)parallélisme sur la grappe MapReduce, ou le mode batch où les requêtes des clients sont exécutées sur la grappe MapReduce les unes à la suite des autres. Enfin, une charge de travail dans MRBS est caractérisée par la distribution (i.e. fréquence d'occurrence) des requêtes des clients. Différentes lois de distribution peuvent être considérées, telle que la distribution aléatoire qui est la valeur par défaut. MRBS fournit un émulateur de clients qui permet à l'utilisateur de MRBS de générer différentes charges de travail, qui sont plus ou moins importantes.

3.4. Analyse de la tolérance aux fautes et de la performance

MRBS permet de conduire des expériences sur une grappe MapReduce, de générer une charge de travail dans un banc d'essai et d'injecter une charge de fautes dans le système en cours d'exécution. MRBS permet alors d'analyser la tolérance aux fautes du système MapReduce. Différentes métriques et statistiques sont produites, telles que le nombre de requêtes clients réussies vs. échouées, le nombre de jobs et tâches MapReduce réussis vs. échoués, la fiabilité du service MapReduce et la disponibilité du service telles que perçues par le client du service MapReduce [19]. La *fiabilité* de service est le ratio entre le nombre de requêtes réussies et le nombre total de requêtes émises par les clients, pendant une période de temps donnée. La *disponibilité* de service est le ratio entre le temps durant lequel le service MapReduce est capable de répondre avec succès au client et le temps total d'exécution du service MapReduce. De plus, MRBS produit des statistiques de performance et de coût, telles que le temps de réponse d'une requête client, le débit de traitement des requêtes par le système MapReduce (en nombre de requêtes par unité de temps) ou encore le coût financier d'une requête d'un client. MRBS produit également des statistiques de plus bas niveau telles que le nombre, la durée et le statut (réussite ou échec) des jobs MapReduce, des tâches map ou reduce, ainsi que la taille des données lues ou écrites dans MapReduce, etc. Ces statistiques bas niveau sont produites hors-ligne, après l'exécution du banc d'essai. MRBS peut également, et de manière optionnelle, générer des graphiques de ces statistiques.

3.5. Déploiement automatique d'expériences sur des environnements de cloud

MRBS permet un déploiement automatique d'expériences sur des grappes et clouds publics ou privés. En effet, une fois que l'utilisateur de MRBS a choisi une charge de travail, un volume de données et une charge de fautes (ou utilisé les valeurs par défaut de ces paramètres), MRBS déploie automatiquement le système MapReduce sur grappe de l'infrastructure de cloud considérée et lance automatiquement l'expérience souhaitée en injectant les charges dans la grappe MapReduce. Les données utilisées sont également automatiquement déployées et le générateur de charge de travail crée autant de thread qu'il y a de clients concurrents, ces derniers envoyant des requêtes au nœud maître de la grappe MapReduce qui se chargera d'ordonnancer les jobs MapReduce. Les clients envoient des requêtes et reçoivent des réponses en continu jusqu'à ce que l'exécution se termine. L'exécution d'une expérience comprend trois étapes successives : l'étape de *warm-up* qui permet une stabilisation initiale du système , l'étape de *run-time* consacrée à l'exécution propre de l'expérience et l'étape de *slow-down* qui permet une terminaison propre de l'expérience. Les durées de ces étapes sont paramétrables. Les statistiques produites à la fin de l'expérience concernent la phase de *run-time*. Une expérience peut également être paramétrée pour être exécutée un certain de nombre de fois, pour fournir des moyennes et autres valeurs statistiques.

Afin de rendre MRBS flexible, un fichier de configuration est fourni, permettant de configurer certains paramètres tel que la durée de l'expérience, la taille des données, etc. Mais pour garder MRBS simple d'utilisation, ces paramètres viennent avec des valeurs par défaut qui peuvent être modifiées par l'utilisateur.

La version courante de MRBS permet un déploiement automatique d'expériences sur des infrastructures telles que que Amazon EC2 [2], ou Grid'5000 [25]. Un nœud est dédié à MRBS et à ses injecteurs de charge et les autres nœuds hébergent la grappe MapReduce. A chaque fin de test, MRBS libère les ressources utilisées. L'implantation actuelle de MRBS est proposée pour Apache Hadoop.

4. Évaluation expérimentale

4.1. Environnement d'évaluation

Les expériences présentées dans les Sections 4.2 et 5 ont été réalisées dans le cloud public Amazon EC2 et sur deux grappes de la l'infrastructure Grid'5000. Ces expériences utilisent plusieurs bancs d'essai de MRBS avec les volumes de données par défaut. Les expériences sont exécutées en mode interactif, avec plusieurs clients concurrents. Dans ces expériences, la distribution des requêtes des clients est aléatoire. La disponibilité et la fiabilité sont mesurées sur une période de 30 minutes, le coût se base sur le tarif actuel d'instances de Amazon EC2, soit de 0.34$ par heure et par instance. Chaque expérience est exécutée trois fois afin de produire la moyenne et l'écart-type des résultats.

La configuration logicielle sous-jacente est comme suit. Les nœuds utilisés dans Amazon EC2 sont des larges instances, sous Fedora Linux 8 avec un noyau v2.6.21. Les nœuds dans Grid'5000 utilisent Debian Linux 6 avec un noyau v2.6.32. Le système MapReduce utilisé est Apache Hadoop v0.20.2 et Hive v0.7, avec Java 6. MRBS utilise la bibliothèque de data mining Apache Mahout v0.6[5], la bibliothèque de séquençage d'ADN CloudBurst v1.1.0[35]. La configuration matérielle utilisée dans nos expériences est décrite dans le Tableau 2.

TABLE 2 – Configuration matérielle des grappes MapReduce

Infrastructure	CPU	Mémoire	Disque	Réseau
Amazon EC2	4 EC2 Compute Units in 2 virtual cores	7.5 GB	850 MB	10 Gbit Ethernet
G5K I	4-core 2-cpu 2.5 GHz Intel Xeon E5420 QC	8 GB	160 GB SATA	1 Gbit Ethernet
G5K II	4-core 1-cpu 2.53 GHz Intel Xeon X3440	16 GB	3200 GB SATA II	Infiniband 20G

4.2. Résultats d'évaluation

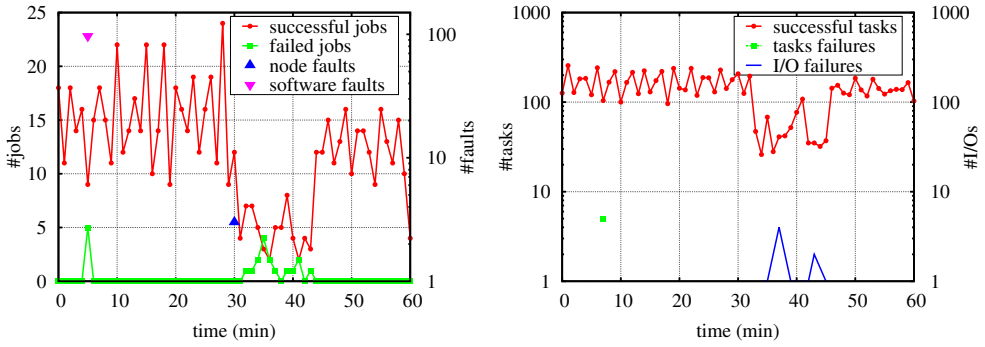
Dans cette section, nous présentons les résultats de l'utilisation de MRBS pour évaluer la tolérance aux fautes de Hadoop MapReduce. Dans cette expérience, 20 clients concurrents accèdent au service en ligne MRBS de bioinformatique, s'exécutant sur une grappe Hadoop de dix nœuds avec la configuration G5K I (voir Tableau 2). L'expérience comprend une étape de *warm-up* de 15 minutes, suivie d'une étape de *run-time* de 60 minutes. La charge de fautes injectée dans le service MapReduce en ligne est la suivante : 100 fautes logicielles surviennent dans les tâches MapReduce 5 minutes après le début de la phase de *run-time*, puis 3 pannes de nœud MapReduce surviennent 25 minutes après. Ici, nous constatons que bien que la charge de fautes soit importante, la grappe Hadoop atteint une disponibilité de service de 96% au cours de ce temps, et est capable de traiter avec succès 94% des requêtes des clients (voir Tableau 3). Ceci a un impact sur le coût des requêtes, avec une augmentation de 14% du coût, en comparaison avec un système où il n'y a pas de fautes.

TABLE 3 – Fiabilité, Disponibilité et Coût

Fiabilité	Disponibilité	Coût (dollars/requête)
94%	96%	0.008 (+14%)

La Figure 4 présente une analyse plus détaillée du comportement de l'expérience. La Figure 4(a) présente l'évolution des jobs MapReduce échoués ou réussis au cours du temps. Nous constatons que lorsque des fautes logicielles surviennent, peu de jobs échouent. Au contraire, lorsque des pannes de nœuds surviennent, cela provoque une hausse du nombre de jobs échoués. Cela provoque une chute du débit de jobs réussis de 16 jobs/minute avant panne à 5 jobs/minute après panne.

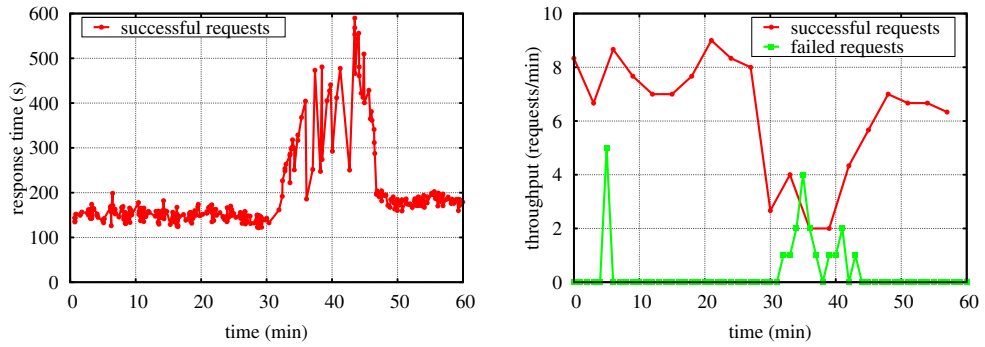
La Figure 4(b) présente l'évolution des tâches MapReduce échouées ou réussies au cours du temps. La figure différencie les tâches échouant à cause d'un problème d'accès aux données du système de fichiers sous-jacent (i.e. *I/O failures*) des tâches échouant à cause d'erreur d'exécution dans la tâche. Nous remarquons ici que les jobs échoués suite aux fautes logicielles sont dus à des erreurs d'exécution des tâches, alors que les échecs de jobs suite à des pannes de nœuds sont dus à des erreurs d'entrées/sorties et des problèmes d'accès aux données et ceci, pendant une quinzaine de minutes après la survenue des pannes. En effet, lorsque des nœuds d'une grappe Hadoop s'arrêtent inopinément, Hadoop doit reconstruire l'état du système de fichiers, en re-dupliquant les blocs de données qui étaient sur les nœuds



(a) Jobs échoués ou réussis (b) Tâches échouées ou réussies

FIGURE 4 – Impact des fautes sur les jobs et tâches MapReduce

défaillants vers d'autres nœuds de la grappe. Cette phase de reconfiguration de Hadoop implique le délai durant lequel on observe des erreurs d'entrées/sorties.
La Figure 5 présente une analyse de l'impact des fautes sur les performances du système MapReduce. La Figure 5(a) présente l'évolution du temps de réponse des requêtes réussies des clients au cours du temps. Nous remarquons que les fautes logicielles n'ont pas de réel impact sur le temps de réponse aux requêtes qui reste le même. A l'inverse, le temps de réponse augmente sensiblement lorsque des nœuds tombent en panne et ce, durant toute la période de reconfiguration de Hadoop suite aux pannes. La Figure 5(b) montre l'impact des fautes sur le débit des requêtes réussies ou échouées. Ainsi, après les pannes de nœuds, Hadoop peut encore satisfaire les requêtes des clients, mais avec une augmentation du temps de réponse 30% et une diminution du débit de traitement des requêtes de 12%.



(a) Temps de réponse aux requêtes (b) Requêtes réussies ou échouées

FIGURE 5 – Impact des fautes sur performances

5. Étude de cas

Pour illustrer l'utilisation de MRBS, nous considérons l'étude de cas suivante. Nous considérons le cas d'un fournisseur de service hébergeant un service MapReduce sur une grappe composée de 10 nœuds dans le cloud public sur Amazon EC2. Une question à laquelle le fournisseur de service devrait répondre est : *combien de pannes de nœud peuvent être tolérées dans une grappe MapReduce tout en garantissant une disponibilité de service d'au moins 85% ?*

Nous avons mené des expériences avec trois bancs d'essai de MRBS pour illustrer trois comportements différents : le service *Business Intelligence* traitant des données volumineuses, le service *Recommendation System* orienté calcul et le service de *Bioinformatics* qui mixe les deux comportements. Nous considérons une charge de travail constituée de 5 clients concurrents accédant aux services et les ensembles de données par défaut de ces bancs d'essai. Différentes charges de fautes sont considérées pour injecter plus ou moins de pannes de nœuds et caractériser leurs impact. Ces fautes sont injectées dans la grappe Hadoop MapReduce au début de la phase de *run-time*. La phase de *run-time* dure 30 minutes, après une phase de *warm-up* de 15 minutes.
La Figure 6 présente les mesures de disponibilité de service pour différentes charges de fautes. Pour garantir une disponibilité de service d'au moins 85%, la grappe MapReduce qui héberge le service *Business Intelligence* orienté données ne tolère pas plus de 2 pannes de nœuds. En comparaison, le service *Bioinformatics* qui manipule des données moins volumineuses peut tolérer jusqu'à 4 pannes de nœuds tout en atteignant le même objectif de disponibilité de service. Le service *Recommendation System* quant à lui est orienté calcul et peut tolérer jusqu'à 6 pannes de nœuds sur les dix nœuds présents dans la grappe. En résumé, Hadoop MapReduce est capable de tolérer de manière transparente une panne de nœud. Lorsque d'autres pannes de nœuds se produisent, Hadoop peut faire face à ces dernières avec un taux acceptable de disponibilité si le service MapReduce hébergé est plus orienté calcul que données.
D'autres études de cas illustrant l'utilisation de MRBS ont été menées et sont décrites dans [33, 34].

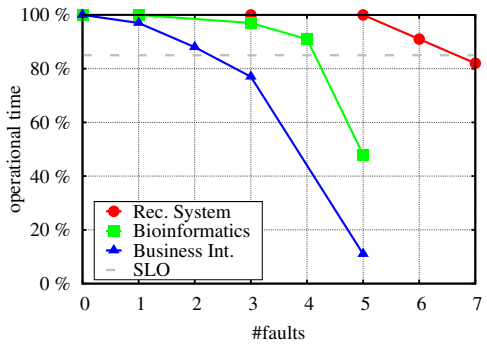


FIGURE 6 – Disponibilité en fonction des charges de fautes

6. État de l'art

Le *benchmarking* est une problématique importante pour l'évaluation de systèmes répartis. Plusieurs travaux de recherche et de standards industriels existent dans le domaine, tels que TPC-C qui évalue en ligne des systèmes transactionnels (OLTP) [11], TPC-H pour tester les systèmes d'aide à la décision [12] et YSCB pour l'évaluation de différents systèmes de stockage dans des environnements de clouds [20]. Des systèmes de *benchmarking* ont également été proposés pour évaluer la fiabilité de systèmes répartis et ont été utilisés pour l'analyse de systèmes RAID [17], pour des systèmes OLTP [36], et pour les serveurs web [22].
Le besoin d'évaluer la fiabilité de systèmes MapReduce est motivé par plusieurs travaux récents consacrés à l'étude et à l'amélioration de la tolérance aux fautes de MapReduce. Les travaux menés dans le domaine concernent la tolérance aux fautes sur demande [27], la tolérance aux fautes adaptative [29, 23], et l'extension de MapReduce avec d'autres modèles de tolérance aux fautes [16, 31]. L'évaluation des solutions proposées de tolérance aux fautes est malheureusement faite de manière ad-hoc. Bien que quelques initiatives existent [6], elles restent de très bas niveau et ne permettent pas de décrire une charge de fautes ni de mesurer la fiabilité et la disponibilité de grappes MapReduce. MRBS est le premier système permettant d'évaluer la tolérance aux fautes de MapReduce.

D'autres travaux se sont intéressés à l'évaluation de la performance de MapReduce, tels que HiBench [28], MRBench [30], PigMix [10], Hive Performance Benchmarks [8], GridMix3 [3] et SWIM [18]. HiBench se compose de huit jobs MapReduce (ex. *sort*, *word count*, etc.) [28]. Il mesure la performance en termes de temps de traitement d'un job et du débit tâches et d'entrées/sorties. Bien que constitué de huit jobs, HiBench ne considère pas l'exécution de jobs concurrents dans la grappe MapReduce, ce qui inhibe la consolidation du cluster. De plus, il ne permet pas d'avoir différentes charges de travail et ne permet pas l'injection de fautes pour l'évaluation de la tolérance aux fautes de MapReduce.
MRBench est spécifique au domaine d'aide à la décision [30], et est dérivé de TPC-H [12]. Comme HiBench, MRBench ne permet pas la concurrence de jobs ni la définition d'une charge de travail plus ou moins complexe. Il ne considère par les fautes et pannes et ne permet pas d'évaluer la tolérance aux fautes.
De manière similaire, PigMix et Hive Performance Benchmarks utilisent un ensemble de requêtes pour évaluer respectivement les performances des plates-formes Pig et Hive [37]. Pig et Hive sont des solutions logicielles au-dessus de Hadoop MapReduce, le premier fournissant un langage haut niveau pour l'analyse de données volumineuses, le second étant un système d'entrepôt de données pour des requêtes ad-hoc.
GridMix3 prend en entrée la trace de précédents jobs Hadoop MapReduce et émule de manière synthétique les jobs depuis cette trace. Ainsi, il permet de re-exécuter les jobs synthétiques en générant la même fréquence d'arrivée de jobs et la même quantité d'entrées/sorties. Cependant, GridMix ne permet pas la capture des fautes et ne peut donc pas reproduire les échecs et les temps de réponses des jobs dans une grappe MapReduce. SWIM est un environnement similaire qui permet de synthétiser une charge de travail MapReduce spécifique en échantillonnant les traces d'une grappe MapReduce [18]. SWIM ne permet pas de prendre en compte les jobs échoués, ni de considérer la tolérance aux fautes de MapReduce.

7. Conclusion

Cet article présente MRBS, le premier système d'évaluation de la tolérance aux fautes de MapReduce. MRBS considère différents modèles de fautes et permet de caractériser une charge de fautes, de la générer et l'injecter dans un système MapReduce en cours d'exécution. Ainsi, MRBS permet d'effectuer une évaluation empirique de la disponibilité et de la fiabilité de services MapReduce. Il permet également de mesurer l'impact des fautes sur la performance et le coût de services MapReduce.
MRBS est disponible en prototype logiciel pour aider les chercheurs, les développeurs et les utilisateurs de MapReduce à mieux analyser et évaluer la performance et la fiabilité de ces systèmes. Le prototype actuel est proposé pour Hadoop MapReduce, un environnement MapReduce très populaire et largement utilisé dans les clouds publics. MRBS permet, par ailleurs, le déploiement automatique d'expériences sur des plates-formes de clouds, facilitant ainsi son utilisation. Cet article illustre l'utilisation de MRBS pour évaluer la tolérance aux fautes de grappes Hadoop exécutés sur Amazon EC2 et sur Grid'5000. L'article présente également une étude de cas de MRBS. Nos travaux futurs incluent le déploiement automatique sur d'autres infrastructures de clouds et l'exploration d'autres cas d'utilisation.

8. Remerciements

Ces travaux ont été, en partie, soutenus par l'Université de Grenoble et l'ANR (Agence Nationale de la Recherche) à travers le projet MyCloud (http ://mycloud.inrialpes.fr/). Une partie des expériences décrites dans cet article ont été conduites sur l'environnement d'expérimentation Grid'5000, soutenu par l'action de développement INRIA ALADDIN avec un soutien du CNRS, RENATER et différentes Universités (http ://www.grid5000.fr).

Bibliographie

1. 20 Newsgroups. – `http://people.csail.mit.edu/jrennie/20Newsgroups/`.
2. Amazon Elastic Compute Cloud (Amazon EC2). – http ://aws.amazon.com/ec2/.
3. Apache GridMix. – `http://hadoop.apache.org/mapreduce/docs/current/gridmix.html`.
4. Apache Hadoop. – http ://hadoop.apache.org/.

5. Apache Mahout. – `http://mahout.apache.org`.
6. Fault injection framework. – `http://hadoop.apache.org/hdfs/docs/r0.21.0/faultinject_framework`.
7. Genomic research centre. – http ://www.sanger.ac.uk/.
8. Hive Performance Benchmarks. – `https://issues.apache.org/jira/browse/HIVE-396`.
9. MovieLens web site. – http ://movielens.umn.edu/.
10. PigMix. – `http://cwiki.apache.org/confluence/display/PIG/PigMix`.
11. TPC-C : an on-line transaction processing benchmark. – http ://www.tpc.org/tpcc/.
12. TPC-H Benchmark Standard Specification. – http ://www.tpc.org/tpch/.
13. TPC-W : a transactional web e-Commerce benchmark. – http ://www.tpc.org/tpcw/.
14. Wikipedia Dump. – `http://meta.wikimedia.org/wiki/Data_dumps`.
15. Ananthanarayanan (G.), Agarwal (S.), Kandula (S.), Greenberg (A.), Stoica (I.), Harlan (D.) et Harris (E.). – Scarlett : Coping with Skewed Content Popularity in MapReduce Clusters. *In : EuroSys*.
16. Bessani (A. N.), Cogo (V. V.), Correia (M.), Costa (P.), Pasin (M.), Silva (F.), Arantes (L.), Marin (O.), Sens (P.) et Sopena (J.). – Making Hadoop MapReduce Byzantine Fault-Tolerant. *In : IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN), Fast abstract*.
17. Brown (A.) et Patterson (D. A.). – Towards Availability Benchmarks : A Case Study of Software RAID Systems. *In : USENIX Technical Conf.*
18. Chen (Y.), Ganapathi (A.), Griffith (R.) et Katz (R.). – The Case for Evaluating MapReduce Performance Using Workload Suites. *IEEE Int. Symp. on Modeling, Analysis, and Simulation of Computer Systems (MASCOTS)*, 2011.
19. claude Laprie (J.). – Dependable computing and fault-tolerance : Concepts and terminology. *In : 25th International Symposium on Fault-Tolerant Computing*.
20. Cooper (B. F.), Silberstein (A.), Tam (E.), Ramakrishnan (R.) et Sears (R.). – Benchmarking Cloud Serving Systems with YCSB. *In : ACM Symp. on Cloud Computing (SoCC)*.
21. Dean (J.) et Ghemawat (S.). – MapReduce : Simplified Data Processing on Large Clusters. *In : USENIX Symp. on Operating Systems Design and Implementation (OSDI)*.
22. Duraesa (J.) et et. al. – Dependability Benchmarking of Web-Servers. *In : Int. Conf. on Computer Safety, Reliability, and Security (SAFECOMP)*.
23. Ekanayake (J.), Li (H.), Zhang (B.), Gunarathne (T.), Bae (S.-H.), Qiu (J.) et Fox (G.). – Twister : A Runtime for Iterative MapReduce. *In : ACM Int. Symp. on High Performance Distributed Computing (HPDC)*.
24. Eltabakh (M.), Tian (Y.), Ozcan (F.), Gemulla (R.), Krettek (A.) et McPherson (J.). – CoHadoop : Flexible Data Placement and Its Exploitation in Hadoop. *In : Int. Conf. on Very Large Data Bases (VLDB)*.
25. et. al. (F. C.). – Grid'5000 : A Large Scale and Highly Reconfigurable Grid Experimental Testbed. *Int. Journal of High Performance Computing Applications (IJHPCA)*, 2006.
26. Fadika (Z.), Dede (E.), Govindaraju (M.) et Ramakrishnan (L.). – Benchmarking mapreduce implementations for application usage scenarios. *In : IEEE/ACM Int. Conf. on Grid Computing (GRID)*.
27. Fadika (Z.) et Govindaraju (M.). – LEMO-MR : Low Overhead and Elastic MapReduce Implementation Optimized for Memory and CPU-Intensive Applications. *In : IEEE Int. Conf. on Cloud Computing Technology and Science (CloudCom)*.
28. Huang (S.), Huang (J.), Dai (J.), Xie (T.) et Huang (B.). – The HiBench Benchmark Suite : Characterization of the MapReduce-Based Data Analysis. *In : IEEE Int. Conf. on Data Engineering Workshops (ICDEW)*.
29. Jin (H.), Yang (X.), Sun (X.-H.) et Raicu (I.). – ADAPT : Availability-Aware MapReduce Data Placement in Non-Dedicated Distributed Computing Environment. *In : IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*.
30. Kim (K.), Jeon (K.), Han (H.), Kim (S.-g.), Jung (H.) et Yeom (H. Y.). – MRBench : A Benchmark for MapReduce Framework. *In : IEEE Int. Conf. on Parallel and Distributed Systems (ICPADS)*.
31. Ko (S. Y.), Hoque (I.), Cho (B.) et Gupta (I.). – Making Cloud Intermediate Data Fault-Tolerant. *In : ACM Symp. on Cloud computing (SoCC)*.
32. Lin (H.), Ma (X.), Archuleta (J.), Feng (W.-c.), Gardner (M.) et Zhang (Z.). – MOON : MapReduce On Opportunistic eNvironments. *In : ACM Int. Symp. on High Performance Distributed Computing (HPDC)*.
33. Sangroya (A.), Serrano (D.) et Bouchenak (S.). – Benchmarking Dependability of MapReduce Systems. *In : 31st IEEE International Symposium on Reliable Distributed Systems (SRDS)*.
34. Sangroya (A.), Serrano (D.) et Bouchenak (S.). – *MRBS : A Comprehensive MapReduce Benchmark Suite*. – Research Report nRR-LIG-024, Grenoble, France, LIG, 2012.
35. Schatz (M. C.). – CloudBurst : Highly Sensitive Read Mapping with MapReduce. *Bioinformatics*, 2009.
36. Vieira (M.) et Madeira (H.). – A Dependability Benchmark for OLTP Application Environments. *In : Int. Conf. on Very Large Data Vases (VLDB)*.
37. White (T.). – *Hadoop : The Definitive Guide*. – O'Reilly Media, 2009. http ://hadoop.apache.org/.