TELKOMNIKA Indonesian Journal of Electrical Engineering
Vol.12, No.2, February 2014, pp. 1029 ~ 1036
DOI: http://dx.doi.org/10.11591/telkomnika.v12i2.4324

Checkpoint and Replication Oriented Fault Tolerant Mechanism for Map Reduce Framework

Yang Liu*[,a], Wei Wei[b], Yuhong Zhang[c]
College of Information Science and Engineering, Henan University of Technology, Zhengzhou, China
Corresponding author, e-mail: enjoyang@gmail.com[,a], weiwei_ise@haut.edu.cn[b], zhangyuhong001@gmail.com[c]

Abstract

MapReduce is an emerging programming paradigm and an associated implementation for processing and generating big data which has been widely applied in data-intensive systems. In cloud environment, node and task failure is no longer accidental but a common feature of large-scale systems. In MapReduce framework, although the rescheduling based fault-tolerant method is simple to implement, it failed to fully consider the location of distributed data, the computation and storage overhead. Thus, a single node failure will increase the completion time dramatically. In this paper, a Checkpoint and Replication Oriented Fault Tolerant scheduling algorithm (CROFT) is proposed, which takes both task and node failure into consideration. Preliminary experiments show that with less storage and network overhead, CROFT will significantly reduce the completion time at failure time, and the overall performance of MapReduce can be improved at least over 30% than original mechanism in Hadoop.

Keywords*: MapReduce, fault tolerant, cloud computing, checkpoint*

1. Introduction

MapReduce is an emerging programming paradigm that has gained more and more popularity due to its ability in supporting complex tasks execution in a large-scale and scalable way. In large-scale distributed computing environment such as cloud environment, node and task failure is no longer exceptional but a common feature. Research discovered that failure has a significant impact on system performance in large-scale systems [1]. Every year in a cluster, 1% to 5% hard disks will be scrapped, up to 20 racks and 3 routers will go down, and servers will go down at least twice with 2% to 4% scrap rate each year. It shows that failure also occurs daily even in a distributed system with up to ten thousand super reliable servers (MTBF of 30 years) [2]. For the cloud environment consisting of a large number of inexpensive computers, node and task failure become a more frequent and widespread problem, which must be handled by some effective fault tolerant method.

The MapReduce based programs generate a lot of intermediate data, which is critical for completion of the job. This paper views failover of intermediate data as a necessary component of MapReduce framework, specifically targeting and minimizing the effect of tasks and nodes failure on performance metrics such as job completion time. We propose new design techniques for a new fault tolerant mechanism called CROFT (Checkpoint and Replication Oriented Fault Tolerant Mechanism for MapReduce Framework), implement these techniques within Hadoop, and experimentally evaluate the resulting system.

2. Related Work

MapReduce is a programming model and an associated implementation for processing and generating big data [3]. It is initially designed for parallel processing of big data using mass cheap server clusters, and put scalability and system availability on the prior position. Within Google Company, more than 20PB of data is processed every day and 400PB every month. Yahoo implemented Hadoop - an open source MapReduce framework. Facebook uses it to process data and generate reports, while Amazon Company uses elastic MapReduce framework for large amounts of data-intensive tasks [4]. MapReduce has the obvious advantage over other programming models like MPI, and a single task failure does not affect the execution

Received August 4, 2013; Revised September 6, 2013; Accepted October 7, 2013

of other tasks because of the independence between Mapper and Reducer task. It has drawn a lot of attention for its benefits in simple programming, data distribution and fault tolerance, which has been widely used in many areas, like data mining, machine learning, information Retrieval and others.

In a report given in 2006, Google Company pointed out that in a cloud environment with averagely 268 nodes, each MapReduce job is accompanied with failure of five nodes [5]. On the other hand, large number of data is usually accompanied with data inconsistency or even data loss, and incorrect data record will result in task failure or even failure of the entire job. MapReduce uses a rescheduling based fault-tolerant mechanism to ensure the correct execution of the failed task. But in the scenario of node failure, all the completed tasks on the failed node will start over, which shows severely reduced efficiency. And Rescheduling failed to fully consider the location of distributed data, the computation and storage overhead. And when Hadoop's node failure detection timeout is 10 minutes (the default), a single failure will cause at least 50% increase in completion time [6]. If each input split contains one bad record in the middle, the entire MapReduce job will have a 100% runtime overhead, which is not acceptable for those users with rigorous SLA requirements to process the MapReduce jobs. So it clearly shows the need for more effective algorithms that allow for reducing delays caused by these failures [7].

In [8], tests show that in seven types of cluster with different MTBF (Mean Time between Failure), MapReduce job with three replicas can achieve better performance than that with one replica, because more replicas can reduce the chances of data migration when rescheduling jobs at failure time. [9] discussed an alternative fault tolerance scheme - the state based Stream MapReduce (SMR), which is suitable for handling continuous data streaming applications with real-time requirements, such as financial and stock data. The key feature is a low-overhead deterministic execution which reduces the amount of persistently stored information. [10] proposed a method to replicate intermediate data to the reducer, but this method will produce a large number of I/O operations, and consume a lot of network bandwidth, and only supports recovery for single node failure. In [11] the author proposed a method to improve performance of fault tolerance by replicating data copies. In [12] author presented an intelligent scheduling system for web service, which considers both the requirements of different service requests and the circumstances of the computing infrastructure which consists of various resources. [13] described the priority of fairness, efficiency and the Balance between benefit and fairness respectively, then recompiled the CloudSim and simulate the three task scheduling algorithms above on the basis of extended CloudSim respectively.

We proposed a Checkpoint and Replication Oriented Fault Tolerant scheduling algorithm (CROFT), which can significantly reduce the average completion time of jobs. Unlike traditional MapReduce fault tolerance mechanism, this algorithm will reschedule tasks on the failed node to another available node without starting over again, but reconstruct intermediate results quickly from the checkpoint file. Under no failure, CROFT turns out to have little impact on the performance of Hadoop. The preliminary experiment shows that under a failure, CROFT outperforms Hadoop with a 30% increasing of performance and incurs up to 7% overhead compared to Hadoop.

3. Algorithms

3.1. MapReduce Programming Model

In MapReduce programming model, the calculation process is decomposed into two main phases, namely the Mapper stage and Reducer stage. For one piece of input data, the Reducer stage only starts when the Mapper stage is completed. A MapReduce job includes M Mapper tasks and R Reducer tasks. In Mapper stage, multiple Mapper tasks run in parallel, and one Mapper task will read an input split and perform a Mapper function, where Mapper tasks are independent from each other. Mapper tasks will produce a large number of intermediate results in local storage. Before Reducer function is called, the system will classify the generated intermediate result and shuffle result with the same key to Reducers. A reducer task will execute a reduce function and generates an output file, and eventually a MapReduce job will generate R output files which could be merged to get the final result. When programming, developers need to write a mapper and a reducer function:

$$map:(input\_data)\rightarrow\left\{(key_j, value_j)\mid j=1...k\right\} \quad (1)$$

$$reduce:(key,[value_1,...,value_m])\rightarrow(key,\ final\_value) \quad (2)$$

The MapReduce model is shown in Figure 1.

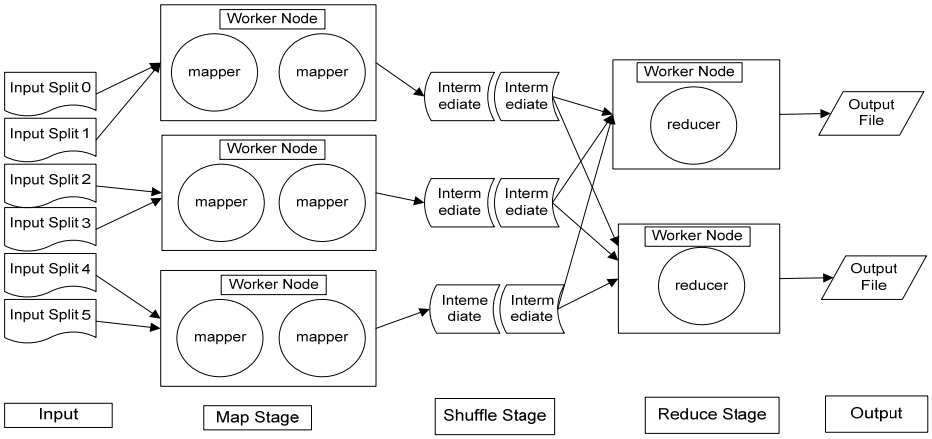

Figure 1. Execution Process of MapReduce Model

Node and task failure is prone to happen during a MapReduce job execution process. When a node fails, MapReduce will restart all the mapper tasks on available nodes. This kind of rescheduling method is simple but often introduces a lot of time cost, thus for users with high responsiveness requirements, the negative performance influence is not acceptable.

3.2. Improved Rescheduling Algorithm

In this paper, a Checkpoint and Replication Oriented Fault Tolerant scheduling algorithm (CROFT) is proposed, which use a checkpoint based active replication method to provide better performance with low overhead when failure happens. Both node and task failure can be supported and failure introduced delay is significantly decreased comparing with traditional rescheduling in Hadoop. Thus could improve overall performance of a MapReduce job.

Before the execution of one Mapper task, the algorithm will create a local checkpoint file and a global index file. The local checkpoint file is responsible for recording the progress of the current task, and could avoid re-execution from the beginning after task failure. If local task failure happened, the node could restart the task from recent status with the help of local checkpoint file. While the global index is responsible for recording the characteristics of the current execution, thereby help reconstructing the intermediate results in other nodes when the node failed, thus reduces the re-execution time. The global index file could be implemented as a checkpoint file saved in HDFS, thus could be accessible in case of node failure.

The CROFT algorithm is divided into two parts, one part works on the master node and the other works on the worker nodes. First, the master node will pre-assign all of Mapper tasks and Reducer tasks on the worker nodes. In addition, as the master node is important, it's necessary to maintain multiple fully consistent "hot backup", to ensure seamless migration when a fault occurs. Two parts of CROFT algorithm are shown in Table 1 and 2.

When a task failure occurs, simply read the checkpoint file saved in the local disk, restore task status to the checkpoint, and reload the intermediate results generated before failure, so the duplicated execution can be avoided.

When a node failure occurs, the scheduler on the master node is responsible for rescheduling the interrupted mapper tasks to available replica nodes. The replica node can

quickly construct the intermediate results of failed tasks according to the global index file, greatly reducing the re-execution time.

Note if tasks and nodes failed before saving checkpoint, the progress will continue from the recent checkpoint. And if the action of saving checkpoint failed, the progress will start from the recent checkpoint again. The simple failover strategy is effective in distributed computing environment with relatively low cost. The frequency of saving checkpoint is configurable. A high frequency will provide lower cost for failover and higher checkpoint saving cost for normal running, while a low frequency is in the opposite case. After careful adjusting, the frequency value in our experiment is set to one checkpoint per 105 key-value pairs.

If node failure happens in the Reducer stage, then tasks on the failed node are rescheduled to any available replica node. The needed intermediate results have been copied to the replica node when Mapper tasks finished, thus there is no need to repeat the mapper tasks on the failed node, so the overall completion time of the MapReduce job is greatly reduced.

Table 1. The Algorithm on Master Node

1. The master node pre-assigns Mapper and Reducer task to different worker nodes.
2. Choose K replicas for every worker node.
3. Wait for results of all worker nodes.
    a) If all results received, merge these results and complete job.
    b) Or, go to 3 and keep waiting.
4.Periodically send probing packets to all worker nodes for their status, then in a round:
    a) If all worker nodes give responses, then go to 4 and keep probing.
    b) Or, if one node didn't respond in given time interval, then mark the node as failed:
        i. Get the worker id of the failed node, with all the unfinished tasks on the node.
        ii. Put all unfinished Mapper tasks into the global queue, and reschedule them on the available replica node.
        iii. If there are failed tasks on the failed node, then reschedule these tasks on the replica node which have the intermediate results, without re-executing the Mapper tasks.
    c) If one node has finished all tasks, then re-assign other unexecuted tasks to the node.

Table 2. The algorithm on worker node

1. Check the given task: Is it a Mapper task or a Reducer task.
  a) If it's a Mapper task, check it's a new task or a re-execution of failed task.
    i. If it's a new task, initialize and execute it.
    ii. If it's a re-execution of local failed task, then get its progress from the local checkpoint file and continue execution.
    iii. If it's a re-execution of the failed task from other nodes, then read global index file for the task, rapidly reconstruct intermediate result from the offsets in global index file.
  b) If it's a Reducer task, check it's a new task or a re-execution of the failed task, or unexecuted tasks from other nodes.
    i. If it's a new task, initialize and execute it.
    ii. If it's a re-execution of the failed task from other nodes, read backup intermediate data from the local disk and execute it.
    iii. If it's a new assigned unexecuted task from other nodes, then read intermediate data from a given worker node and execute it.
2. Create a local checkpoint file and a global index file for the given Mapper task.
3. Start a Mapper task, in the process:
  a) When the Mapper's memory buffer is full, dump intermediate data into local files. After dumping finished, record the position of the input stream and Mapper ID (Position, Mapper_ID) into local checkpoint file.
  b) According to the location distribution of input key-value pairs which contributed to output key-value pairs, two different strategies are employed to record these distributions into the global index file.
    i. For input Key-Value pairs producing output key-value pairs, record their position (T1, offset) into global index file. Which means only pairs in these offsets need to be processed when re-execution. Here T1 means this is the type 1 record.
    ii. Or, for input pairs which give no output, record the range as (T2, offset1, offset2), which means the input pairs between offset1 and offset2 have no output and could be skipped when re-execution. T2 means this is the type 2 record.
4. When a Mapper task finished, shuffle the intermediate results and send to corresponding Reducer nodes. Copy the intermediate data needed by local Reducer task to replica nodes, in case of node failure. Then notify the completion of Mapper task to master node and delete the local checkpoint file and the global index file.

3.3. System Analysis

For a given Mapper task m, the question is: for a given input range of an input split, how to decide the kind of records (T_1 or T_2) in global index file, thus to minimize storage overhead. For the given input range, set the total number of key-value pairs as N_m, and the number of output key-value pairs as N'_m. The size of type 1 and type 2 records is L_1 and L_2:

$$L_2 = 2L_1 \tag{3}$$

Set $S_{m,1}$ and $S_{m,2}$ as the range's storage overhead in type 1 and type 2 record. Set V_m as the count of input sub-ranges which give no output, and the O_i-th input key-value pair produces the i-th outputting key-value pairs.

$$V_m = \sum_{i=2}^{N'_m} \begin{cases} 0, O_i - O_{i-1} = 1 \\ 1, O_i - O_{i-1} > 1 \end{cases} \tag{4}$$

We have,

$$S_{m,1} = L_1 N'_m \tag{5}$$

$$S_{m,2} = L_2 V_m = 2L_1 V_m \tag{6}$$

Set R_m as the decision we make, when $R_m = 1$, we store type 1 record to global index file, or if $R_m = 2$, we use type 2 record. Then we can use next equation to decide which kind of record to use.

$$R_m = \begin{cases} 1, N'_m < 2V_m \\ 2, N'_m \geq 2V_m \end{cases} \tag{7}$$

4. Experiment

We validate our algorithm on Hadoop cluster. CROFT is implemented as a patch of Hadoop. The comparison is measured from the performance aspect and the overhead aspect in the case of tasks failure and nodes failure. The performance of the algorithm is evaluated by delay which is a very important factor for user experience, and the pursuit of low-latency is a main target in large-scale cloud environment. The implementation of the algorithm is based on Hadoop 0.20.1, Java1.6 and HDFS file system with HDFS data block size of 256MB. The underlying infrastructure is a 20-node HP blade cluster node with quad-core Xeon 2.6GHz CPU, 8G memory, 320G hard drive, with two pieces of Gigabit NICs. Every node is configured to hold four Xen virtual machines. Thus it has 80 virtual nodes where 40 nodes for Hadoop cluster and 40 nodes for Hadoop with CROFT. For each cluster, one node is deployed as the master node and remaining 39 nodes are deployed as the worker node. A single worker node can simultaneously run two Mapper tasks and one Reducer task. The job for experiment is a typical filter job, which is to filter out certain entries in the huge amounts of data. This kind of job is computationally intensive and has less intermediate results. We use 1.2 million English web pages with an average page size of 1MB for test. By adjusting the split size, one Mapper handles an average of approximately 120M input data, and each node is assigned with an average of about 250 mapper tasks.

There are three kinds of MapReduce job used in the experiment, according to the distribution of query words in input data: the aggregated job, the sparse job and the mixed job. In an aggregated job, the location of query words is gathered in the target data; in a sparse job, the location of the query words are more dispersed; while in a hybrid job, above two situations co-exists.

The MapReduce job execution performance is usually influenced by factors as below: (1) the size of the data set, usually every mapper task will deal with a subset of data (split); (2) worker nodes' computation performance; (3) MapReduce job type; (4) The number of bad records in each split, each of these bad records will result in restarting of the failed task; (5) The number of failed worker nodes. By adjusting these factors, the effectiveness and the overhead of given fault tolerant algorithms can be evaluated and compared in the same criterion.

In scenario with only task failure, Figure 2 shows the comparison of a MapReduce job completion time between Hadoop and CROFT. The x-axis is the task error probability in the form of error task number per 100 tasks; the y-axis is the total completion time. There is no upper limit on the error number of each mapper task. We can see, with CROFT, along with the increase in the probability of errors, the execution time of the job will increase, but has significant improvement compared to that of Hadoop without CROFT. Especially when the number of bad record in input split increased, the local checkpoint file will be read more frequently, thus saved more execution time, and performance here will be more superior to that of Hadoop.

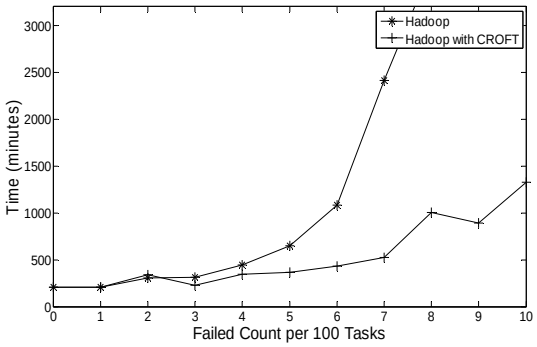

Figure 2. Comparison of Execution Time with Task Failure

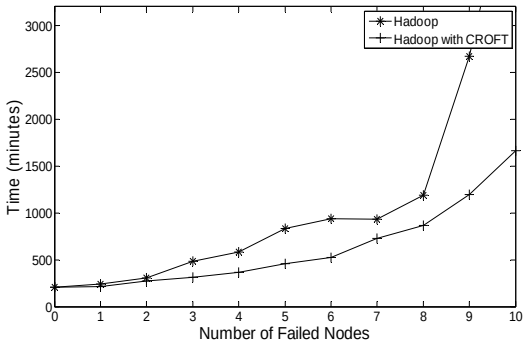

Figure 3. Comparison of Execution Time with Node Failure

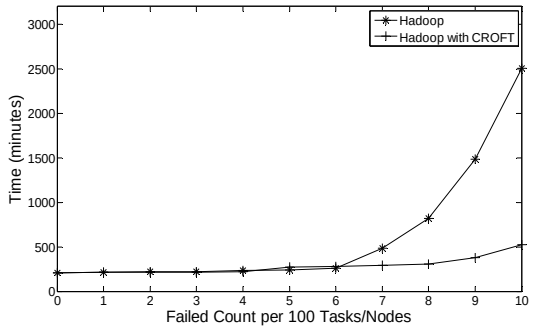

Figure 4. Comparison of Execution Time with task and Node Failure

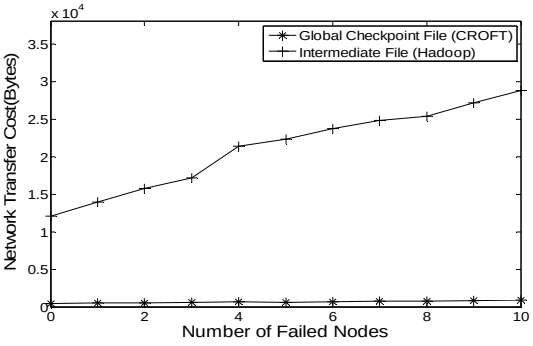

Figure 5. Comparison of Average Network Overhead with Node Failure

In scenario with only node failure, Figure 3 shows the comparison of execution time of a MapReduce job between Hadoop and CROFT. The x-axis is the number of failed nodes; the y-axis is the total completion time. When node failure happens, the failed node will be removed from the cluster. In CROFT, it is shown that along with more nodes failed, the increase of job execution time is not obvious. CROFT's performance is better than the original Hadoop. CROFT will significantly decrease the re-execution time, because the simple rescheduling mechanism of Hadoop will start all failed task from the beginning on replica nodes, which defer shuffling and reducing. While CROFT could finish more tasks in the same period of time by re-executing the mapper tasks quickly and efficiently.

In scenario with both task failure and node failure, Figure 4 shows the comparison of a MapReduce job completion time between Hadoop and CROFT. The x-axis is the probability of task failure and node failure, in the form of failed task/node count per 100 tasks/nodes; the y-axis is the total completion time. It's shown that under the joint influence of task and node failure, the relation between the job execution time and the failure probability is more than linear, while the overhead in CROFT is mainly introduced by task migration and task progress restore, thus a lot of re-execution time is saved.

On the other hand, the CROFT algorithm does not bring too much overhead. Network overhead is ignored because there is no extra network transferring under the condition of task failure. Figure 5 shows the extra network overhead of CROFT which is introduced by rescheduling of a MapReduce job under the condition of node failure. X-axis is the number of failure nodes, and the y-axis is the average network overhead. It is shown that, compared with the network overhead of intermediate results replication in Hadoop, the extra network overhead of CROFT is quite limited, because in the case of node failure, network overhead is mainly from the active replication of the global index file.

In the task failure scenario, the storage overhead is the size of the local checkpoint file, which is negligible due to the limited position information stored in it. Figure 6 shows the comparison of storage overhead of three different types of MapReduce job, under the scenario of the 10 failed nodes. Because increased storage overhead in CROFT is mainly for storing of global index file, it is very limited compared with the intermediate results stored in Hadoop.

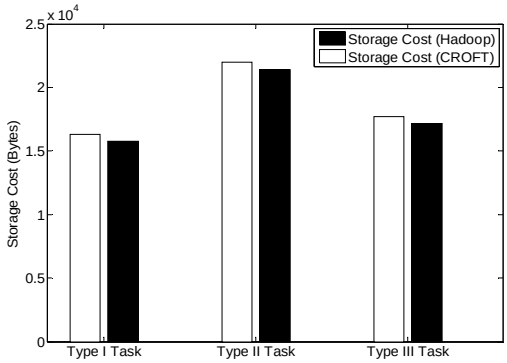

Figure 6. Comparison of Average Storage Overhead with Node Failure

5. Conclusion

Hadoop's simple rescheduling mechanism significantly increased the MapReduce job's overall completion time. The paper proposed a Checkpoint and Replication Oriented Fault Tolerant scheduling algorithm (CROFT), which is fully implemented and tested on Hadoop. Experimental results show that CROFT is suitable for multiple types of MapReduce job and could greatly reduce the overall completion time under the condition of task and node failure. The runtime performance can be improved by 30% or more than that in Hadoop. In the case of a large number of nodes failed, the improvement could be 80% or more.

Acknowledgement

This paper is supported by the National Natural and Science Foundation of China (No. 61003052 No. 61103007), Natural Science Research Plan of the Education Department of Henan Province (No. 2010A520008), Henan Provincial Key Scientific and Technological Plan (No. 102102210025), Program for New Century Excellent Talents of Ministry of Education of China (No. NCET-12-0692), Doctor Foundation of Henan University of Technology (No.150126), Key Science and Technology Research Project of Education Department Henan Province (13A413001).

References

[1] Schroeder B, Gibson A. A large-scale study of failures in high-performance computing systems. *IEEE Transactions on Dependable and Secure Computing*. 2010; 7(4): 337-350.

[2] Dean J. *Designs, Lessons and Advice from Building Large Distributed Systems*. ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS), Big Sky. 2009; 3: 1-5.

[3] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. *Communications of the ACM*. 2008; 51(1):107-113.

[4] Amazon Elastic Mapreduce, http://aws.amazon.com/elasticmapreduce/.

[5] Dean J. *Experiences with Mapreduce, an Abstraction for Large-Scale Computation*. Internet Conference on Parallel Architectures and Computation Techniques (PACT), Seattle. 2006; 8: 5-10.

[6] Ko SY, Hoque I, Cho B, Gupta I. *On Availability of Intermediate Data in Cloud Computations*. The USENIX Workshop on Hot Topics in Operating Systems (HotOS), Monte Verità. 2009; 5: 32-38.

[7] Quiané-Ruiz J, Pinkel C, Schad J, Dittrich J, *RAFTing Mapreduce: Fast Recovery on the Raft*. IEEE International Conference on Data Engineering, hannover. 2011; 3: 589-600.

[8] Hui J, Kan Q, Xian-He S, Ying L. *Performance under Failures of Mapreduce Applications*. IEEE/ACM International Symposium on Cluster Cloud and Grid Computing, Newport Beach. 2011; 5:608-609.

[9] Martin A, Knauth T, Creutz S, Becker D, Weigert S, Fetzer C, Brito A. *Low-overhead fault tolerance for high-throughput data processing systems*. International Conference on Distributed Computing Systems, Minneapolis. 2011; 3: 689-699.

[10] Ko SY, Hoque I, Cho B, Gupta I. *Making cloud intermediate data fault-tolerant*. ACM symposium on Cloud computing, Indianapolis. 2010; 1: 181-192.

[11] Zheng Q. *Improving Mapreduce Fault Tolerance in the Cloud*. IEEE International Symposium on Parallel Distributed Processing Workshops and Phd Forum (IPDPSW), New York. 2010; 1: 1-6.

[12] Jing L, Xingguo L, Bainan L, Xingming Z, Fan Z. An Intelligent Job Scheduling System for Web Service in Cloud Computing. *TELKOMNIKA Indonesian Journal of Electrical Engineering.* 2013; 11(12): 2956-2961.

[13] Hong S, Shiping C, Chen J, Kai G. Research and Simulation of Task Scheduling Algorithm in Cloud Computing. *TELKOMNIKA Indonesian Journal of Electrical Engineering.* 2013; 11(11): 1923-1931.