International Journal of Database Theory and Application
Vol.7, No.1 (2014), pp.37-48
http://dx.doi.org/10.14257/ijdta.2014.7.1.04

Enhancing Fault Tolerance based on Hadoop Cluster

Peng Hu[1] and Wei Dai[*2]

[1]*School of Mathematics and Physics, Hubei Polytechnic University, Huangshi 435003, Hubei, P.R.China*
[2]*School of Economics and Management, Hubei Polytechnic University, Huangshi 435003, Hubei, P.R.China*

[1]*penghuit@163.com,* [*2]*dweisky@163.com (Corresponding Author)*

Abstract

Failures happen for large scale distributed systems such as Hadoop clusters. Native Hadoop provides basic support for failure tolerance. For example, data blocks are replicated over several HDFS nodes, and Map or Reduce tasks would be re-executed if they fail. However, simply re-processing the whole task decreases the efficiency of job execution, especially when the task is almost done. To this end, we propose a fault tolerance mechanism to detect and then recover from failures. Specifically, instead of simply using a timeout configuration, we design a trust based method to detect failures in a fast way. Then, a checkpoint based algorithm is applied to perform data recovery. Our experiments shows that our method exhibits good performance and is proved to be efficient.

Keywords: *Fault tolerance, Hadoop, Checkpoint*

1. Introduction

With the increasing development of cloud computing [1, 2] and the big data era [3-5], the distributed system and architecture for high performance computing [6] have been more and more complicated. Even though the designers pay much attention to the reliability of the application, the probability of failures for such a complicated system remains high. Therefore, in order to make it more efficient for distributed systems applications, it is necessarily urgent to enhance fault tolerance in distributed systems.

MapReduce programming model [7] provides more simple way to achieve large scale capability for distributed and parallel systems. As an open source implementation of MapReduce, Hadoop [8] has already been successfully applied in many applications such as search engine [9, 10], text processing [11], machine translation [12, 13] and so on.

Hadoop has two major components: HDFS (Hadoop Distributed File Systems) [14] and Hadoop MapReduce. Files are split into equal size of data blocks and replicated on multiple HDFS nodes. Also, jobs are divided into several tasks including Map tasks and Reduce tasks, and then are assigned to several task nodes for processing. Both HDFS and MapReduce employ a master/slave architecture, where the master is in charge of management and scheduling, and the slaves are responsible for data storage and task processing.

Indeed, Hadoop provides some fault tolerance mechanisms through both HDFS and MapReduce. First, HDFS provides storage layer of fault tolerance by replication. That is, HDFS keeps multiple replicas of each data block in several different nodes, so that if any one node is down, data could still be restored from other replicas. Second, Hadoop MapReduce provides job level fault tolerance. That is, if a Map or Reduce task fails, the

ISSN: 2005-4270 IJDTA

scheduler would re-assign the task to another node. Or, if a node fails, all the Map and Reduce tasks on that node would be re-scheduled to another node for re-execution. However, this simple redo solution increases the computation cost a lot. For example, suppose the total execution time of task i on node j is $\tau_{i,j}$, and task i encounters a failure at time t. If $\tau_{i,j} >> t$, the simple solution of re-scheduling and re-processing would be fine. Otherwise, a large number of computations that have been completed will be repeated elsewhere, especially when $\tau_{i,j}$ is big. Intuitively, we need to find a better solution to deal with this kind of failures instead of completely redo the whole task.

To this end, in this paper we propose to study on a fault tolerance mechanism for more efficient job scheduling, especially when the workload of each task is high, in which case the original Hadoop solution of re-processing the whole task would be quite inefficient. Specifically, we propose a trust based checkpoint algorithm to enhance fault tolerance of Hadoop. First, instead of simply use a timeout based rule to detect failure, we design a trust based method. The basic idea is to assign a trust value to each node, and if a task receive a fetch error from that node, the trust value would be reduced. Second, once we have detected a failure, we employ a checkpoint mechanism for recovery. Our checkpoint algorithm is coordinated by a centralized master node and the communication is made between replica nodes of specific data block. At the end of this study, our empirical experiments prove the efficiency of our proposed method.

The remain of this paper is organized as follows. Section 2 provides some preliminaries. Related work is discussed in Section 3. Then our proposed fault tolerant mechanism is proposed in Section 4. Empirical experiments are conducted in Section 5. Finally, the paper is concluded in Section 6.

2. Preliminaries

In this section, we will briefly introduce some background about Hadoop.

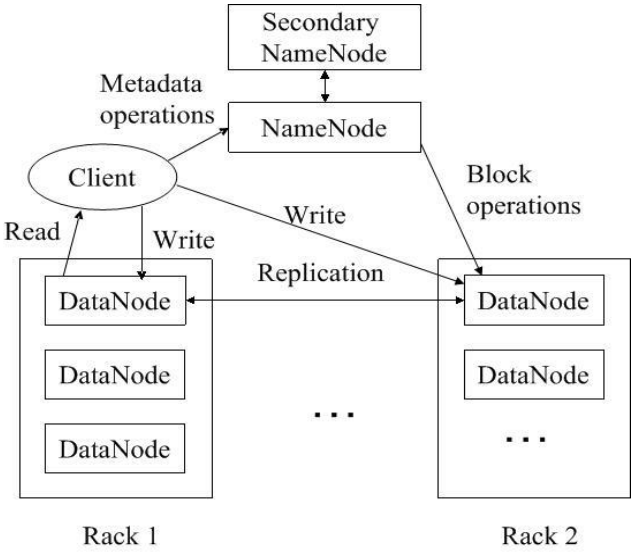

Figure 1. Architecture of HDFS

As an open source implementation of MapReduce, Hadoop [8] has two major components: HDFS (Hadoop Distributed File Systems) [14] and Hadoop MapReduce. The architecture of HDFS is illustrated in Figure 1, where NameNode is the master node of HDFS handling metadata, and DataNode is slave node with data storage in terms of blocks. Namenode holds metadata about the information of all DataNodes and is responsible of coordination and node management. Each data block is stored on one DataNode, and replicated to several other DataNodes. Every operation of writing data would update all the replicas of data blocks sooner or later. In this way, if any one of the data block is missing because of task failure or node failure, data can still be accessed from replicas on other DataNodes and even rebuilt from the replicas on other healthy nodes.

Figure 2 shows the process flow of MapReduce, which can be summarized as follows. First, the input data is partitioned into splits with appropriate size; and then Map procedure does the process and produces intermediate results, which are ready to be passed to a Reduce node by certain partition function; Later, after data sorting and shuffling, the Reduce procedure performs some aggregation on specific keys. The master node of Hadoop MapReduce is called JobTracker, which is in charge of managing and scheduling several tasks, and the slave node is called TaskTracker, where Map and Reduce procedures are actually performed.

A typical deployment of Hadoop is to assign HDFS node and MapReduce node on the same physical computer for the consideration of localization and moving computation to data [15]. As you will see in Section 4, we apply this deployment in our experiments.

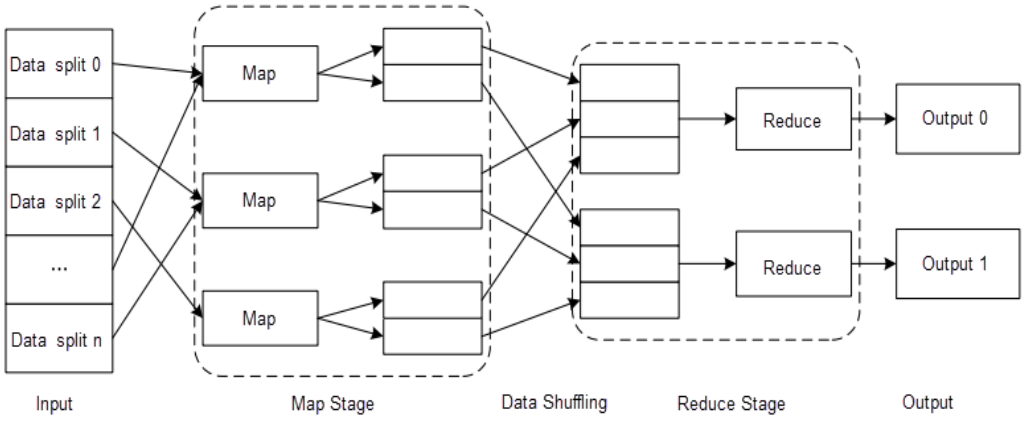

Figure 2. Description of MapReduce Programming Model

There are two ways to ensure the proper execution of the jobs. First, if a Map or Reduce task fails, the scheduler would re-assign the task to another node. Second, if a node fails, all the Map and Reduce tasks on that node would be re-scheduled to another node for re-execution. However, this simple redo solution increases the computation a lot, because the failed task might be a small step away from the completion. Therefore, finding a solution to deal with the failures instead of completely redo the whole tasks is in need.

3. Related Work

Aside from the native support of fault tolerance in Hadoop as mentioned earlier, there are some other efforts along this line.

Finding slow tasks that are lagging the whole job execution can help to ensure fault

tolerance. Matei Zaharia *et al.*, [16] discovered that the prediction mechanism might increase the total execution time in that the heterogeneous nature of a cluster with heterogeneous nodes. The reason behind is there is a fundamental assumption of Hadoop that the computing capability of computing nodes is equivalent. To this end, the authors proposed a scheduling strategy based on LATE (Longest Approximate Time to End) which employs the longest remaining time of each task as a measure for scheduling. Quan Chen *et al.*, [17, 18] presented to collect history data of each node along with the performance records, to assist the assignment of Map and Reduce tasks. However, unlike above methods, which try to predict the execution time of each node or task or the whole job and then help to avoid failures, in this paper, we propose solutions to detect and handle failures.

There are some efforts on avoiding Single Point Of Failure (SPOF) as well. The role of JobTracker in Hadoop MapReduce is significant in coordination, monitoring and job scheduling. Francesco Salbaroli [19] proposed to employ a Master-Slave architecture, where the master JobTracker provides services such as data access, and the slave JobTracker is responsible for synchronizing data with master JobTracker. In this case, if the master JobTracker is down, other slaves JobTracker would vote for a new master JobTracker. Besides, NameNode in HDFS can also be the single point of failure. Borthakur [20] presented a mechanism called AvatarNode to achieve high scalability, and Wang Feng *et al.*, [21] proposed to replicate metadata over other nodes. Unlike these researches, in our study, we focus on the failure during job execution. In fact, our method can be easily extended with the SPOF solutions from a big picture, which could be a future work.

4. Trust Based Checkpoint Algorithm

4.1 Trust Based Failure Detection

Typically, the process of MapReduce in Hadoop is composed of three stages: (1) after Map task finishes, the immediate results are saved to the local storage; (2) in shuffle stage, the local results are transferred to the Reduce task; (3) after the process of Reduce task, the results are saved in HDFS. Note that if a node fails during the Reduce stage, all the other Reduce tasks on other nodes will receive a fetch error, because the data shuffled from multiple Map tasks is missing. The original solution in Hadoop to detect the failure node is: given a pre-defined timeout parameter, if a node has no response after a specified timeout period, then that node is asserted to be dead.

However, complete dependence on timeout is an inefficient design, and the setting of timeout parameter is very important. For example, if the timeout is too small, and the execution time of a task happens to be longer than the timeout value, then that node would be reported as a failure while in fact it is just a lag processing task. Oppositely, if the timeout is too large, and there is one node fails much earlier than the pre-defined timeout moment, then the whole process would be waiting while the failure should be detected earlier.

Therefore, in this section, we propose a trust based failure detection method. The general idea is to make use of the fetch error that Reduce tasks receive when they attempt to get immediate data from other nodes. We assume that if multiple fetch errors are encountered, that node is reported as a failure.

Suppose the set of tasks $T = \{t_1, t_2, ..., t_N\}$, and the set of processors (or nodes) $P = \{p_1, p_2, ..., p_M\}$, where M, N are the number of nodes and tasks respectively. Every processor is associated with a trust value indicating the reliability of that node. The corresponding trust value for P is $R = \{r_1, r_2, ..., r_M\}$, where $0 < r_j \le 1, j = 1,2,...,M$, and

initialized as 1 at the beginning. Note that the trust values will always be equal for multiple processors on a single node. If a Reduce task tries to get data from a Map task node p_j, and receives a fetch error, the corresponding trust value will be reset as $r_j - \delta$, where δ is the penalty value. Let β as the threshold of trust. If $r_j < \beta$, node j would be detected as a failure. The process of failure detection is described in Algorithm 1.

4.2 Checkpoint Based Recovery

Once a node is reported as a failure, the next step is to recovery. To achieve this, we employ a checkpoint based recovery strategy [22]. Checkpoint strategy can not only help to restore the system to a state before failures, but also avoid the completely re-processing, and therefore reduces the workload of data recovery. In fact, the checkpoints are set during the process of job processing, which is overlapped with the failure detection process. That is, while we are monitoring the trust value of nodes, we need to preserve the checkpoint information as well. The difference is that the former is used to report failures, and the latter is for data recovery from failures.

Algorithm 1 Trust based failure detection

```
1:  for p_j in P do
2:      initialize r_j = 1
3:      for t_i ∈ T do
4:          if r_j < β then
5:              return j as a failure
6:          end if
7:          if t_i gets a fetch error from p_j then
8:              r_j = r_j − σ
9:          end if
10:     end for
11:     return without any failures
12: end for
```

Figure 3. Algorithm Description of Trust Based Failure Detection

On Hadoop cluster, data is replicated over several nodes. Suppose each data block has r replica copies $c_1, c_2, ..., c_r$, and each c_k is processed by p_j. Note that we assume the deployment of our Hadoop cluster is localize the data storage and computation, which means that if p_j processes c_k then data c_k is stored on node j. The HDFS NameNode maintains a metadata about the locations and assignments of replicas.

On node p_j, the data structures include: (1) current sequence number of checkpoint CN_j, which is an integer increased by 1 if there is a checkpoint set on p_j, otherwise $CN_j = 0$; (2) current state ST_j, that is $NORMAL$ or CK (checkpoint) decided by if there

is any checkpoint settings; (3) the set of messages or logs in local data storage D_j, which is used to restore data; (4) state of other replicas RST_j, which is actually a list of replica c_k and current state ST_k pairs. Note that all the replicas of the same data block share one single sequence counter of CN_j to keep synchronized.

The communication between nodes is achieved by sending and receiving messages. Denote SM_j as the set of messages sent by p_j, and RM_j as the set of messages received by p_j. Each message m is composed of CN_j, ST_j and RST_j.

Suppose p_j has a local checkpoint CN_j. It will sent a message to the master node for coordination with the format of (CN_j, CK). Since the master node maintains the metadata of the locations and states of replicas, it will send messages to other replicas and update the checkpoint setting, shown as Lines 5-14 in Algorithm 2.

Algorithm 2 Set checkpoint

```
1:  while TRUE do
2:    if p_j sets a checkpoint then
3:      master receives m = (CN_j, CK, , )
4:      CN_j = CN_j + 1
5:      for replica c_k of data block on p_j do
6:        p_k which holds c_k receives m = (CN_j, , [(c_1, ST_1), (c_2, ST_2), ...])
7:        if CN_k ≥ CN_j then
8:          update CN_k = CN_k + 1
9:          send m = (CN_k, CK, , ) to master
10:       else
11:         update CN_k = CN_j
12:       end if
13:       update local RST_k
14:     end for
15:     update local RST_j
16:     update meta data in master
17:   end if
18: end while
```

Figure 4. Algorithm Description of Setting Checkpoint

If a node p_j is reported as a failure as described earlier, the recovery is processed as shown in Algorithm 3. First, information of other replicas is obtained from the master. Then, using the checkpoint data stored at each node, and also with the help of master's coordination, another replica is rebuilt on another live node.

Algorithm 3 Failure recovery

1: get replicas $c_1, c_2, \ldots, c_{k-1}$ from master
2: another node a is randomly assigned as a permanent storage by master
3: **for** i in $i, 2, \ldots, k-1$ **do**
4: **if** $ST_i = CK$ **then**
5: send (CN_i, CK, RST_i) to master
6: **end if**
7: **end for**
8: rebuild data on a

Figure 5. Algorithm Description of Failure Recovery

5. Experiments

In order to evaluate our fault tolerant mechanism, we first need to set up a Hadoop cluster. We have a Hadoop cluster deployed on 4 PCs with 2.11 GHz dual-core CPU, 1G RAM and 200G hard disk. According to the best practices of Hadoop [23], we assign twice the number of Map tasks and the same number of Reduce tasks as the number of cores on each PC. That is, we have 4 Map tasks and 2 Reduce tasks on each physical node. On each physical node, both a HDFS and MapReduce nodes are deployed. We let one of them as HDFS *NameNode* and MapReduce *JobTracker* (*i.e.*, master), and the remaining nodes act as HDFS *DataNode* and MapReduce *TaskTracker* (*i.e.*, slave).

In our study, we choose two different types of jobs, i.e., java sort and monsterQuery [24]. The former job is a small one, which consists of 10 Map tasks and 15 Reduce tasks, while the latter one is large, which is composed of 80 Map tasks and 170 Reduce tasks. In our experiments, we will evaluate the execution time of above two jobs.

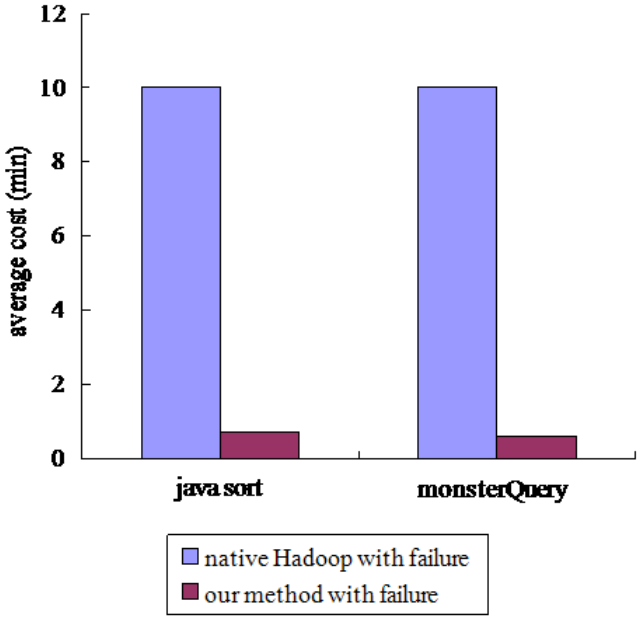

Figure 6. Evaluation of Failure Detection

First, we evaluate the time cost of detecting failure. Note that the timeout settings in the native Hadoop is set to 10 minutes. We compare the execution time of detecting failure for both java sort and monsterQuery jobs, as shown in Figure 6. First, for small job java sort, the time cost of detecting failure for native Hadoop is 10 minutes, because only the timeout condition is satisfied the system would be aware of the task failure. On contrary, it just takes less than 1 minute for our method to report the failure. Second, for large job monsterQuery, the time cost of native Hadoop is also 10 minutes. The reason might be that even though the execution time of job monsterQuery is much longer than that of java sort, it is still shorter than the 10 minutes timeout setting. As you can see, our method outperforms native Hadoop in detecting failures for both small and large jobs.

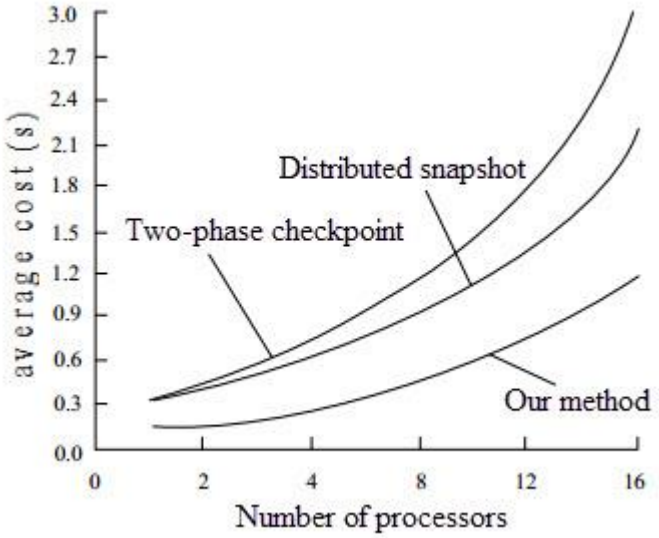

Figure 7. Evaluation of Checkpoint Cost

Second, we evaluate the time cost of setting checkpoints. We compare our checkpoint algorithm with two popular checkpoint algorithms: traditional two-phase checkpoint algorithm [25] and distributed snapshot algorithm [26]. As shown in Figure 7, which gives the average cost of checkpoints with different numbers of processors. We can observe that our checkpoint algorithm is better than the other two. The reasons include: (1) in two-phase checkpoint algorithm, there are three synchronization steps which would block the whole process; and (2) in distributed snapshot algorithm, although no block operations, the communication cost between processors increases up to approximately $O(n^2)$.

Third, we evaluate the recovery time with various number of processors. Figure 8 provides the average cost of recovery with different numbers of processors when there are failures. The cost includes both the communication cost and the replication cost. The axis denotes the number of processors the failure node needs to notify, which is typically depended on the dfs. replication in Hadoop settings file for HDFS configuration. We can see that if a failure is detected, the more replicas we have, the more cost it would take for recovery.

Last, we evaluate the total execution time for both the java sort and monster Query jobs as well. As shown in Figure 9, we have the following observations. First, for small job java sort, the execution time without any failure is small, while the large job monster Query costs more than 9 times of processing. Second, as discussed earlier, the time cost of detecting failure of native Hadoop is determined by the timeout setting, which is set to 10 minutes in

our experiment. Therefore, the execution time of both jobs in native Hadoop with failure is more than 10 minutes. Third, for our method, the execution time with failure for both jobs is slightly longer than the situation of without any failures. The extra cost is caused by the checkpoint mechanism and communication between processors.

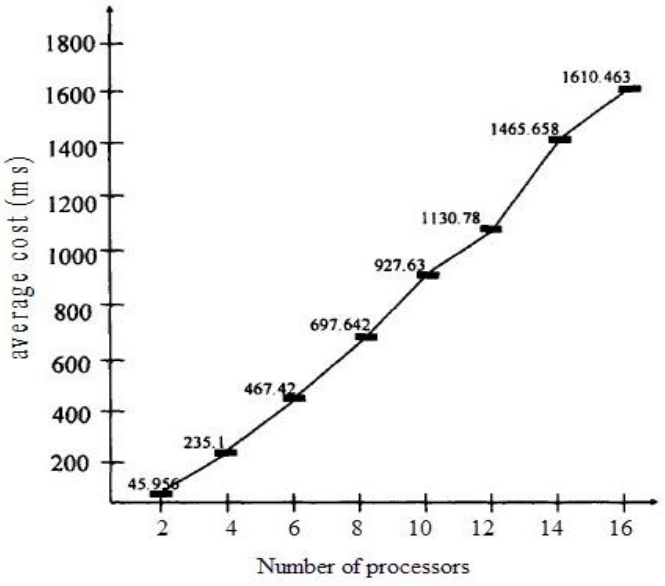

Figure 8. Evaluation of Recovery Cost

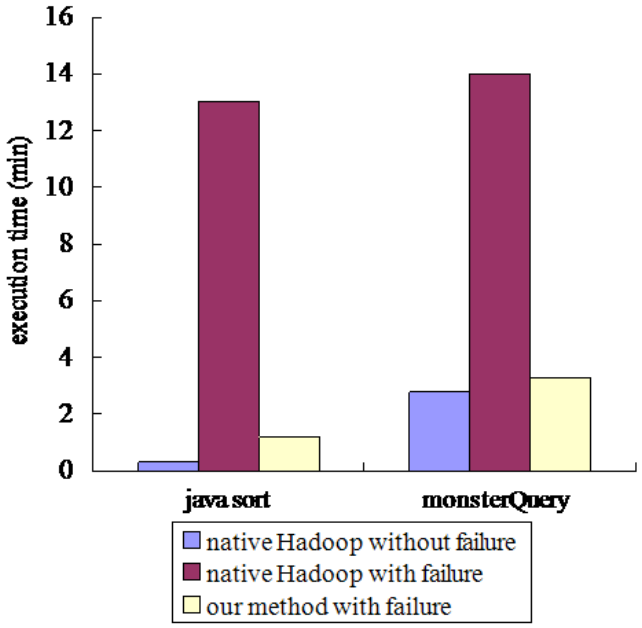

Figure 9. Evaluation of Total Execution Time

6. Conclusions

In this paper we propose a fault tolerance mechanism based on Hadoop. The only support of fault tolerance on native Hadoop is replication on HDFS and re-execution of failed Map or Reduce tasks. However, this increases the cost by re-execution the whole task no matter how far it proceeds. The situation would be worse if a long-last task fails in a large job.

To this end, we propose to first detect failure at a early stage through a trust based method, and then use a checkpoint algorithm for failure recovery. First, instead of simply applying a timeout solution, we assign a trust value to each node, which decreases if a Reduce task gets a fetch error from it. In this way, we dramatically reduces the cost of failure detection.

Second, for our checkpoint algorithm, we employ a non-block method by sending and receiving messages with sequence numbers of specific replica of data blocks. And the data is written to the local storage first, and then combined at the centralized master. Besides, according the metadata information on master, it can choose an appropriate location for rebuilding the missing data block using checkpoint data from other replicas.

Besides, we conduct extensive experiments on a Hadoop cluster, and compare our proposed method with the native configuration of Hadoop. Our empirical results indicate that proposed method exhibits good performance and efficiency.

Acknowledgements

This study has been supported by Humanities and Social Science Youth Fund Project of Ministry of Education (No.13YJCZH028), Hubei Polytechnic University Innovative Talents Project (No.12xjz20C) and Hubei Polytechnic University Youth Project (No.13xjz07Q).

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica and M. Zaharia, "A view of cloud computing", Communications of the ACM, vol. 53, no. 4, **(2010)**, pp. 50-58.
[2] P. Mell and T. Grance, "The NIST definition of cloud computing (draft)", NIST special publication 800-145, **(2011)**.
[3] D. Howe, M. Costanzo, P. Fey, T. Gojobori, L. Hannick, W. Hide, D. P. Hill, R. Kania, M. Schaeffer, S. S. Pierre, S. Twigger, O. White and S. Y. Rhee, "Big data: The future of biocuration", Nature, vol. 455, no. 7209, **(2008)**, pp. 47-50.
[4] S. LaValle, E. Lesser, R. Shockley, M. S. Hopkins and N. Kruschwitz, "Big data, analytics and the path from insights to value", MIT Sloan Management Review, vol. 52, no. 2, **(2011)**, pp. 21-31.
[5] P. Zikopoulos and C. Eaton, "Understanding big data: Analytics for enterprise class hadoop and streaming data", McGraw-Hill Osborne Media, **(2011)**.
[6] K. Dowd, C. R. Severance and M. K. Loukides, "High performance computing", vol. 2. O'Reilly, **(1998)**.
[7] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters", Communications of the ACM, vol. 51, no. 1, **(2008)**, pp. 107-113.
[8] T. White, "Hadoop: the definitive guide", O'Reilly, **(2012)**.
[9] B. Pratt, J. J. Howbert, N. I. Tasman and E. J. Nilsson, "MR-tandem: parallel X! tandem using hadoop MapReduce on amazon Web services", Bioinformatics, vol. 28, no. 1, **(2012)**, pp. 136-137.
[10] S. Goel, J.M. Hofman, S. Lahaie, D. M. Pennock and D. J. Watts, "Predicting consumer behavior with Web search", Proceedings of the National Academy of Sciences, vol. 107, no. 41, **(2010)**, pp. 17486-17490.
[11] J. Lin and C. Dyer, "Data-intensive text processing with MapReduce", Synthesis Lectures on Human Language Technologies, vol. 3, no. 1, **(2010)**, pp. 1-177.
[12] C. Dyer, A. Cordova, A. Mont and J. Lin, "Fast, easy, and cheap: Construction of statistical machine translation models with MapReduce", Proceedings of the Third Workshop on Statistical Machine Translation, Association for Computational Linguistics, **(2008)**, pp. 199-207.
[13] D. Vilar, D. Stein, M. Huck and H. Ney, "Jane: Open source hierarchical translation, extended with reordering and lexicon models", Proceedings of the Joint Fifth Workshop on Statistical Machine Translation and Metrics MATR, Association for Computational Linguistics, **(2010)**, pp. 268-276.

[14] D. Borthakur, "HDFS architecture guide. Hadoop Apache Project", http://hadoop.apache.org/common/docs/current/hdfs_design.pdf, **(2008)**.
[15] L. Pan, L. F. Bic and M. B. Dillencourt, "Distributed sequential computing using mobile code: Moving computation to data", IEEE International Conference on Parallel Processing, **(2001)**, pp. 77-84.
[16] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz and I. Stoica, "Improving MapReduce Performance in Heterogeneous Environments", OSDI, vol. 8, no. 4, (2008), pp. 7.
[17] Q. Chen, D. Zhang, M. Guo, Q. Deng and S. Guo, "Samr: A self-adaptive mapreduce scheduling algorithm in heterogeneous environment", IEEE 10th International Conference on Computer and Information Technology, IEEE, **(2010)**, pp. 2736-2743.
[18] H. Lin, X. Ma, J. Archuleta, W. C. Feng, M. Gardner and Z. Zhang, "Moon: Mapreduce on opportunistic environments", Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, ACM, **(2010)**, pp. 95-106.
[19] K. Hwang, J. J. Dongarra and G. C. Fox, "Fault tolerant Hadoop Job Tracker: Apache Hadoop", Distributed and Cloud Computing, Elsevier/Morgan Kaufmann, **(2012)**.
[20] D. Borthakur, "Hadoop avatarnode high availability", Facebook, http://hadoopblog.blogspot.com/2010/02/hadoop-namenode-high-availability.html, **(2010)**.
[21] F. Wang, J. Qiu, J. Yang, B. Dong, X. H. Li and Y. Li, "Hadoop high availability through metadata replication", Proceedings of the first international workshop on Cloud data management, ACM, **(2009)**, pp. 37-44.
[22] J. S. Long, W. K. Fuchs and J. A. Abraham, "Forward Recovery Using Checkpointing in Parallel Systems", ICPP, no. 1, **(1990)**.
[23] C. Lam, "Hadoop in action", Manning Publications Co., **(2010)**.
[24] P. Costa, M. Pasin, A. N. Bessani and M. Correia, "Byzantine fault-tolerant MapReduce: Faults are not just crashes", (CloudCom), IEEE Third International Conference on Cloud Computing Technology and Science, **(2011)**, pp. 32-39.
[25] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems", IEEE Transactions on Software Engineering, vol. 1, **(1987)**, pp. 23-31.
[26] E. N. Elnozahy, L. Alvisi, Y. M. Wang and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems", ACM Computing Surveys (CSUR), vol. 34, no. 3, **(2002)**, pp. 375-408.

Authors

Peng Hu. He received his B.S. in Mathematics (2003) from Hubei Normal University and M.Sc. in Information Sciences (2010) from Hubei University. Now he is full researcher of informatics at Mathematics and Physics Department, Hubei Polytechnic University. His current research interests include different aspects of Artificial Intelligence and Information Coding.

Wei Dai. He received his M.S.E. in Computer Science and Technology (2006) and PhD in Computer Application Technology (2012) from Wuhan University of Technology. Now he is full researcher of Economics and Management Department, Hubei Polytechnic University. His current research interests include different aspects of Intelligence Computing and Information Systems.

