Fault tolerance in Hadoop MapReduce implementation

Matías Cogorno, Javier Rey, Sergio Nesmachnow

PER-MARE - Pervasive Map-Reduce
Project number 13STIC07
CAPES/MAEE/ANII STIC-AmSud collaboration program

Deliverable WP 1

Fault tolerance in Hadoop MapReduce implementation

PERMARE Work Package 1 Intermediate Report

Matías Cogorno, Javier Rey, Sergio Nesmachnow

UdelaR PERMARE Team

Partner	Author's information
Universidade Federal de Santa Maria SEDES SAPIENTIÆ 1960	
– PANTHÉON - SORBONNE – UNIVERSITÉ PARIS 1	
UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE	
UNIVERSIDAD DE LA REPUBLICA URUGUAY	*Matías Cogorno* *matiascogorno@gmail.com* *Javier Rey* *javirey@gmail.com* *Sergio Nesmachnow* *sergion@fing.edu.uy*

Abstract

This document reports the advances on exploring and understanding the fault tolerance mechanisms in Hadoop MapReduce. A description of the current fault tolerance features existing in Hadoop is provided, along with a review of related works on the topic. Finally, the document describes some relevant proposals about fault tolerance worth considering to implement in Hadoop within the PERMARE project in order to provide support for pervasive computing environments.

1 Introduction

Hadoop (White, 2009; Apache Hadoop, 2013) is an Apache open source framework developed in Java, which uses the MapReduce paradigm for parallel computing to divide the data processing among a set of available computing nodes, allowing to tackle large computing problems.

Currently (August 2013), Apache Hadoop has three different release branches: 1) the stable one, version 1.X for production use, 2) the preview release, version 2.X, currently in alpha (i.e. non-stable version), and 3) the 0.2X version which follows the original versioning and is not meant for production. The new version 2.X is a complete overhaul of Hadoop MapReduce and the Hadoop Distributed File System (HDFS) introducing YARN, a system which separates the resource management and job scheduling/monitoring (Apache Hadoop, 2013).

In the context of this specific Work Package 1 of the PERMARE project, all actual Hadoop releases were analyzed. Even though version 2.X improves significantly in many ways related to the objective of this Work Package, we concluded that it is not yet a solid base to work on and it would create a significant risk to our project.

Besides Apache Hadoop, there are many “distributions” analog to Linux distributions based on Apache Hadoop. Because of the Apache Hadoop license, Apache License 2.0, any derivative work on Hadoop does not necessarily require to have the same license and is often proprietary work. One of the most popular distributions is the one by Cloudera which is also open-source and extensively used instead of the Apache distribution.

According to the online meeting of the PERMARE project held on April 2013 and several other discussions, the research group decided to begin working with the release 1.0.4 of Hadoop, as it is known to be the most stable version and provides the best documentation creating the least amount of risk to the project.

Although there are many articles reporting works using MapReduce, Hadoop, and related technologies, and they have an increasing popularity in the last years, technical documentation is very scarce and often missing. For this reason, most of the knowledge acquired by the UdelaR PERMARE research team about the Hadoop MapReduce implementation, as well as fault tolerance and scheduling details and quirks was obtained directly from analyzing the source code (from the 1.0.4 distribution).

The rest of the document is organized as follows. Section 2 provides a description of the topics the research group has investigated about existing fault tolerance mechanisms in Hadoop; the information in this section was collected mostly from studying the source code. The focus was in the Heartbeat procedure, which is one of the main elements of fault tolerance already implemented in Hadoop. Section 3 reviews the related work on fault tolerance on Hadoop and MapReduce, showing that most of the previous research on this topic proposes to use some kind of redundancy to provide a fast recovery after a node failure. Finally, Section 4 presents some ideas to improve the performance and fault tolerance of Hadoop in order to provide support for pervasive environments in the context of the PERMARE project.

2 Fault tolerance in Hadoop

Due to the lack of specific literature about fault tolerance in Hadoop (see a review of the few articles found in the description of related works on Section 3), it was necessary to inspect the source code in order to understand how it works. This section provides a summary of the main features that have been analyzed so far.

Regarding the infrastructure of Hadoop, in the rest of the document the term worker will be used to refer to the computing elements in the network, and the term node will be used to refer to a unit composed by both a worker and its corresponding part of the HDFS.

One of the main components of Hadoop is the JobTracker, which is executed as a daemon process that executes on a master node. The JobTracker is the scheduler and the main coordinator of tasks. It is in charge of distributing the MapReduce tasks between the available computing nodes. Each time the JobTracker receives a new job to execute, it contacts a set of TaskTracker processes, which are daemons that execute on the working nodes (one TaskTracker exists for each worker in the infrastructure). MapReduce tasks are then assigned to those working nodes for which their TaskTracker daemons report that they have available slots for computation (several tasks assigned to the same worker are handled by a single TaskTracker daemon) (Apache Hadoop, 2013; Patil and Soni, 2013).

The JobTracker continuously monitors the TaskTracker nodes using control messages named heartbeat signals. These heartbeat signals are sent from the TaskTracker to the JobTracker and, after receiving this signal, the JobTracker sends a response including some commands (such as start or end a task, or if the TaskTracker needs to be restarted) to the TaskTracker called HeartbeatResponse (see a description of the heartbeat procedure in the diagram on Figure 1, from Kadirvel and Fortes (2013)).

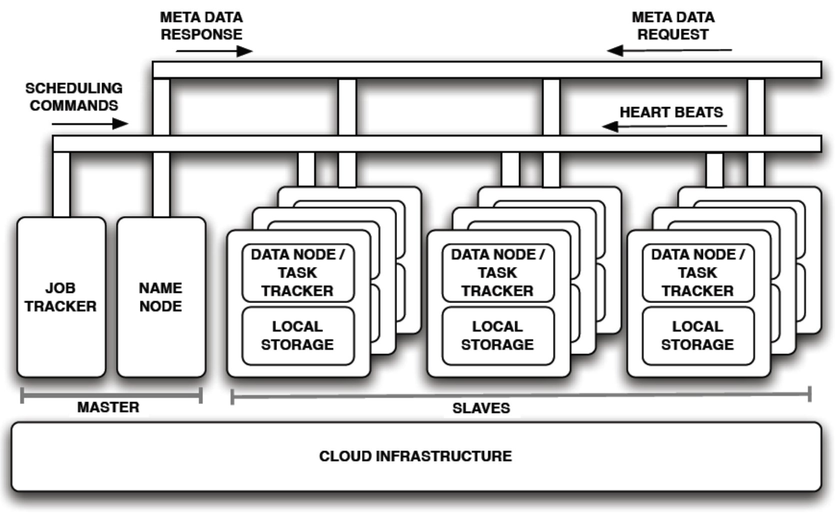

Figure 1: Communications between the TaskTrackers and the JobTracker in Hadoop (Kadirvel and Fortes, 2013)

In case the JobTracker does not receive any heartbeat from a TaskTracker for a specified period of time (by default, it is set to 10 minutes), the JobTracker understands that the worker associated to that TaskTracker has failed. When this situation happens, the JobTracker needs to reschedule all pending and in progress tasks to another TaskTracker, because the intermediate data belonging to the failed TaskTracker may not be available anymore. All completed map tasks need also to be rescheduled if they belong to incomplete jobs, because the intermediate results residing in the failed TaskTracker file system may not be accessible to the reduce task.

A TaskTracker can also be blacklisted. In this case, the blacklisted TaskTracker remains in communication with the JobTracker, but no tasks are assigned to the corresponding worker. When a given number of tasks (by default, this number is set to 4) belonging to a specific job managed by a TaskTracker fails, the system considers that a fault has occurred. Then, when a specific TaskTracker has more than a given number of faults (by default, this number is also set to 4) the TaskTracker is blacklisted. There are two possibilities for a TaskTracker to be removed from the blacklist: if one or several faults expires (by default, a fault expires in 24 hours) the TaskTracker is automatically removed from the blacklist, without need of any communication. The other possibility is for the TaskTracker to reboot, when this happens the JobTracker is notified in the Heartbeat sent by the TaskTracker, and it is removed from the blacklist.

Since one of the most important components of Hadoop related to fault tolerance features is the heartbeat procedure, after learning the basics about how Hadoop works, heartbeat became the focus of the research.

Some of the relevant information in the heartbeats the TaskTracker sends to the JobTracker are:

- The TaskTrackerStatus
- Restarted
- If it is the first heartbeat
- If the node requires more tasks to execute

The TaskTrackerStatus contains information about the worker managed by the TaskTracker, such as available virtual and physical memory and information about the CPU.

The JobTracker keeps the blacklist with the faulty TaskTracker and also the last heartbeat received from that TaskTracker. So, when a new restarted/first heartbeat is received, the JobTracker, by using this information, may decide whether to restart the TaskTracker or to remove the TaskTracker from the blacklist.

After that, the status of the TaskTracker is updated in the JobTracker and a HeartbeatResponse is created. This HeartbeatResponse contains the next actions to be taken by the TaskTracker .

If there are tasks to perform, the TaskTracker requires new tasks (this is a parameter of the Heartbeat) and it is not in the blacklist, then cleanup tasks and setup tasks are created (the cleanup/setup mechanisms have not been further investigated yet). In case there are not cleanup or setup tasks to perform, the JobTracker gets new tasks. When tasks are available, the LunchTaskAction is encapsulated in each of them, and then the JobTracker also looks up for:

- Tasks to be killed
- Jobs to kill/cleanup
- Tasks whose output has not yet been saved.

All this actions, if they apply, are added to the list of actions to be sent in the HeartbeatResponse.

The fault tolerance mechanisms implemented in Hadoop are limited to reassign tasks when a given execution fails. In this situation, two scenarios are supported:

1. In case a task assigned to a given TaskTracker fails, a communication via the Heartbeat is used to notify the JobTracker, which will reassign the task to another node if possible.

2. If a TaskTracker fails, the JobTracker will notice the faulty situation because it will not receive the Heartbeats from that TaskTracker. Then, the JobTracker will assign the tasks the TaskTracker had to another TaskTracker.

There is also a single point of failure in the JobTracker, since if it fails, the whole execution fails.

The main benefits of the standard approach for fault tolerance implemented in Hadoop consists on its simplicity and that it seems to work well in local clusters. However, the standard approach is not enough for large distributed infrastructures such as the ones considered in the PERMARE project, as the distance between nodes may be too big, and the time lost in reassigning a task may slow the system. This approach neither considers a pervasive environment where new nodes can appear in the system, so if a new node is available tasks will not be assigned to it, this means that the system will be wasting available resources.

3 Related work

Related works proposing fault tolerance mechanisms on Hadoop are scarce, mainly because the standard fault tolerance features included in the MapReduce implementation are usually enough for the (dedicated) computational systems where the applications are run (Dean and Ghemawat, 2008). As the infrastructure is not supposed to change dramatically, most papers deal with improving performance or recovering from software errors by using additional computational resources. Existing articles about this subject are more related with HDFS features than with failures in the processing execution or environment.

Two main methods are used in the related works to implement fault tolerance in HDFS: i) data duplication and ii) checkpoint and recovery (Patil and Soni, 2013). Data duplication consist in duplicating the data in multiple DataNodes. Then, each time a client wants to write a file to the HDFS, he first contacts the NameNode and then the NameNode nominates a number of (three by default) different DataNodes that can be used to replicate the data. The number of replicas can be increased, improving the fault tolerance and the bandwidth to reading the file (Evans, 2011). The checkpoint and recovery technique is similar to the concept of rollback. If a failure occurs, the system rollbacks to the last saved synchronization point, and the transaction starts again. This method is slower than data duplication, but on the other hand, it needs less additional resources (Goiri et al., 2010).

Fault tolerance of the MapReduce paradigm in cloud systems was first investigated by Zhen (2010), who proposed a mechanism for provisioning redundant copies for tasks. On top of the standard re-execution schema provided by the Amazon Elastic MapReduce, a passive replication is proposed to improve MapReduce fault tolerance in the cloud. Replication is performed by allocating backups of the running task, which are executed in case of a failure. Specific heuristics to schedule, move and select backups are proposed. Event driven simulations and experiments on a local Hadoop testbed were performed by executing medium-size MapReduce applications, and significant improvement were obtained on the fault tolerance metrics studied (recovery time, delayed completion time, and completion time), while fulfilling strict SLA requirements.

The Byzantyne Fault Tolerance (BFT) MapReduce runtime system by Costa et al. (2011) was proposed to tolerate system faults that corrupts the results of computation of tasks (e.g. DRAM and CPU errors or faults). A replication approach is applied over a Hadoop implementation, and the authors proposed several methods to reduce the number of replicas needed: i) deferred execution, which allows reducing the number of replicas from 3f +1 to f+1, where f is the number of errors to tolerate; ii) tentative reduce execution, which starts executing the reduce tasks when the first replicas complete their execution; iii) digest outputs, which fetches the (usually large) outputs from replicas and compare them to hashes, thus reducing the network traffic; and iv) tight storage replication that implies writing only one replica for the outputs of both map and reduce tasks. A prototype of BFT was implemented by modifying the Hadoop version 0.20.0 source code, including changes into the JobTracker class to implement the replica management using queues, and the Jobinprogress class to store information (into a VotinSystem object) about each running replica. The Heartbeat class was slightly modified to include the digest and task replica identifiers. The experimental evaluation performed over Grid'5000 compared the BFT system and the traditional Hadoop implementation, for the case of f=1 errors. Executions of MapReduce using BFT are about 15% larger than using Hadoop, but they provide support for single execution errors at a cost of roughly twice the consumption of resources. Similar to other fault tolerance mechanisms using replication, the proposed BFT system is more expensive than using the original MapReduce runtime implemented on Hadoop.

In their distributed Hadoop platform (Hadoop on the Grid - HOG), He et al. (2012) introduced a third level for fault-tolerance in Hadoop: the site failure level, extending the traditional two already provided in the standard implementation (node level and rack level). Three components are included in the HOG architecture: the grid submission/execution system, the HDFS and the MapReduce framework. The grid submission/execution system uses Condor and GlideinWMS to allocate and manage tasks, as well as initializing the environment to execute on the Open Science Grid (OSG) organization and sending the Hadoop executable packages.The rack level is implemented in both the HDFS and Mapreduce components, and it provides the load balancing and improved fault tolerance features. A site availability script is included, and an increased number of replicas (10, determined by experimental evaluation) is used, while at any time a task has at most two copies of execution in the system. The site awareness fault-tolerance method affects the data placement and also the placement of Map tasks. HTTP communications are used between the tasktracker and the jobtracker, which have a negative impact in the times for the startup and data transfer initialization. The experimental evaluation compared a HOG system running over OSG against a traditional Hadoop implementation in a cluster. The results demonstrated that the proposed HOG system was able to finish MapReduce jobs faster than the traditional Hadoop implementation, even when some nodes are missing. HOG showed a good scalability behavior, but the response times for workloads did not always decrease when increasing the number of nodes in the system.

In line with the main objectives of the PERMARE project, the recent work by Kadirvel and Fortes (2013) proposes a dynamic resource scaling approach for improving the fault tolerance capabilities of the MapReduce paradigm for a set of job characteristics and framework parameters. The authors focused on Hadoop, and the proposed system is based on applying a scaling method that dynamically increase the number of slave nodes assigned to a given Hadoop job when a node failure occurs. The new nodes are used to execute failed tasks but also to apply a load balancing schema to share the load of healthy tasks. By implementing the resource scaling approach as a separated module to be integrated in Hadoop, the authors provide an easy-to-implement solution that accounts for scalability and minimum overhead. The experimental analysis evaluated the penalties on the execution time of tasks associated to node failures, and the proposed approach was able to reduce up to 78% the penalties associated to single-node failures when compared with the traditional re-execution-based fault tolerance mechanism implemented in Hadoop. As the authors state, "the proposed approach is more beneficial for Hadoop jobs executing on a small number of nodes", so the applicability of this kind of solutions for large environments (such as the one to build in the context of the PERMARE project) remains to be studied. The work by Kadirvel and Fortes did not tackle the fault-tolerance at the data storage level, and no checkpointing mechanism is applied.

Other strategies have been proposed for fault tolerance in Hadoop, but they are not appropriate for handling pervasive infrastructures. Among other proposals, researchers have presented fault tolerance mechanisms at the virtual machine level (VMWare, 2012), high availability using metadata replication (Wang et al., 2009), and also dataset division and replication for parallel data-intensive algorithms (Kutlu et al, 2012).

The details of the main related works on fault-tolerance methods for MapReduce applications which are relevant for the PERMARE projet are summarized in Table 1.

Author(s)	**Year**	**Fault tolerance approach**	**Platform**
Zhen	2010	passive replication	Hadoop/Amazon EC
Costa et al.	2011	replication	Hadoop/Grid'5000
He et al.	2012	site failure level/replication	Hadoop/OSG
Kadirvel and Fortes	2013	resource scaling	Hadoop

Table 1: related work on fault tolerance approach for MapReduce/Hadoop

4 Ideas and key features for the fault tolerance implementation

As already mentioned in Section 2, there are some topics to be investigated, but so far the basic issues related to fault tolerance seems to be properly handled by Hadoop, as when one node fails, its jobs are assigned to other node. Nevertheless, a new mechanism needs to be proposed for handling pervasive and volunteer computing infrastructures. In addition, two main improvements have been considered to make Hadoop more reliable and efficient in the context of the PERMARE project.

4.1 Support for pervasive and volunteer infrastructures

As mentioned in Section 2, and also in the conclusion of the related work section, Hadoop was not designed to support pervasive and volunteer infrastructures. In theory, nodes can come and go as they please, but the system is not optimized to handle such dynamic nodes behavior, and performance would be far from optimum in highly dynamic scenarios.

In order for Hadoop to support this new kind of pervasive infrastructures, some changes need to be implemented. To keep a record of the existing nodes, the JobTracker will need to have a specific data structure to store the information about all the nodes that have previously contacted it. This data structure will have information sent from each node that will help the JobTracker to know the availability of each node, and how reliable each node is through time.

A new node appears. When a new node is available in the system, it will start execution by sending a Heartbeat to the JobTracker. During the analysis of the Heartbeat, the JobTracker will add the node to the list of available nodes and initialize the reliability field with the minimum value.

Reliability. The data structure will have information sent from each node that will help the JobTracker to know how reliable the node is through time. The information will include a register of successful and unsuccessful tasks executed by the node, the online time, the average online time and other information that may help to identify how reliable the node is.

Availability of an already known node. When an already known node that previously left the system is available once again, it will send Heartbeats with any available useful information it may know about the causes of the period of time it was unavailable. The JobTracker will then set the reliability of the node according to the information it has about it, adjusting the information it knows about the node.

Task scheduling. The assignment of the tasks to the nodes will also be modified. As Hadoop will execute in a pervasive and volunteer infrastructure, nodes should be expected to come and leave frequently, and new nodes may become part of the system. Tasks will be assigned taking into account the reliability of the node. This procedure may cause that new nodes will not be able to get tasks until a certain period of time (threshold_in) has passed. In order to reduce the impact of this situation, after assigning a certain number of tasks to reliable nodes, the JobTracker will try to assign a task to one of the new nodes, disregarding the threshold_in constraint, but taking into account the availability and reliability status. This way, new reliable nodes may be found, thus improving the performance of the system.

One of the main objectives of the PERMARE project is to allow any device to be part of the system, this feature includes allowing to participate in the system devices that may frequently change its geographic location, for example a notebook. For this reason, the scheduler is deeply related with the results expected from Work Package 3 in the PERMARE project. This Work Package aims to get context information of the nodes, such as their location, and from this information the scheduler would be able to assign tasks more efficiently, for example by assigning tasks to nodes that are close to the data they need.

4.2 JobTracker: improving the single point of failure

As described in some papers (Evans, 2011, Patil and Soni, 2013) about fault tolerance in Hadoop, there is a single point of failure in the system, which is the JobTracker: Hadoop is not able to recover if the JobTracker fails. For this reason, a worthy idea to improve the fault tolerance is to have more than one JobTracker. There are two options to analyze in this case.

The first option is to have more than one JobTracker daemon for a single Hadoop instance, and distributing the MapReduce tasks among them. In order to accomplish this goal, a protocol of communication between the JobTrackers has to be designed, to allow them to synchronize their work and know if the other JobTracker are working properly. The idea is to include a new Heartbeat mechanism between the JobTrackers, in which they exchange information about the jobs they are working on and if they are able to start new jobs. This Heartbeat mechanism will also let each JobTracker know if the others are still working properly, and used to provide fault tolerance as if a JobTracker realizes that another is not working, it will be able to create a new JobTracker to continue with the work assigned to the faulty one.

The second option is to have a set of backup JobTrackers for the main JobTracker. There would also be a Heartbeat protocol between them, but in this case all the backup JobTrackers are waiting for the main JobTracker to fail, if this happens, one of the backups will became the main JobTracker and it will create a new backup JobTracker. In case of one of the backup JobTrackers fails, then the main JobTracker will create a new backup JT.

These two options are yet to be analyzed. On one hand, the first option will distribute the load of execution between the several existing JobTrackers, but the messages exchanged between them will be larger and communications can increase, and the interconnection medium may become a bottleneck on the system. On the other hand, the second option will need a simpler communication protocol, but the main JobTracker could become the bottleneck itself. As the papers we found in the related work deal mainly with the performance of the TaskTrackers and the distribution of the work, and not with the performance of the JobTracker, it will seem that it is not necessary to distribute the work of the JobTracker in more nodes, so using the second option could be enough to tackle this single point of failure, there will be a certain and novel contribution, but this alternative should be analyzed more carefully.

4.3 Supernodes to improve efficiency in the cloud

Another idea to improve the fault tolerance and the performance is to use context-aware nodes to handle some faults without needing the JobTracker to get involved. In this new infrastructure configuration, neighboring nodes will behave as a supernode, and each node will know the other nodes in the supernode and the tasks they have assigned. Thus, in case one of the nodes fails, another node in the supernode can take the role of the failed one, without needing the JobTracker to know about it or take any action.

As the main goal of the PERMARE project is to develop a version of Hadoop in a pervasive cloud, the time spent due to a fail in a node may have a considerably impact in the performance of the system, because the JobTracker and the node that failed may be far away geographically, so the JobTracker will not notice the failure instantly. However, using the supernodes approach, the impact of these failures will be smaller, since each single failure will be noticed by a node in the neighborhood, which will be able to take the role of the failed node in the supernode.

The implementation of this approach will be significantly more complex than other ideas for fault tolerance, because it will need a major change in the communications between the master node and the other nodes, and a new communication protocol between neighbouring nodes has to be proposed and implemented. Algorithms to create, manage, and update the supernodes have to be implemented too.

In order to continue developing this idea, we will investigate what has been done about P2P in Hadoop in the PERMARE project in Work Package 2. We need to know whether the P2P protocol is ready to be used or not, and how it should be modified to fit the requirements of the supernodes approach.

References

Apache (2013). The Apache Software Foundation: All you wanted to know about Hadoop, but were too afraid to ask: genealogy of elephants. Available at https://blogs.apache.org/bigtop/entry/all_you_wanted_to_know, accessed August 2013.

Apache Hadoop (2013). Hadoop documentation. Available at http://hadoop.apache.org/, accessed August 2013.

P. Costa, M. Pasin, A. Bessani, and M. Correia (2011). Byzantine Fault-Tolerant MapReduce: Faults are Not Just Crashes. In Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science (CLOUDCOM '11). IEEE Computer Society, Washington, DC, USA, pp. 32-39.

J. Dean and S. Ghemawat. (2008). MapReduce: simplified data processing on large clusters. Communications ACM 51, 1, pp. 107-113, January 2008.

J. Evans (2011). Fault Tolerance in Hadoop for Work Migration, Technical Report CSCI B534 (Survey Paper), Indiana University, November 2011.

I. Goiri, F. Julià, J. Guitart, and J.Torres (2010). Checkpoint-based fault-tolerant infrastructure for virtualized service providers. IEEE/IFIP Network Operations and Management Symposium, April 2010. IEEE. Osaka, Japan, pp. 455-462.

C. He, D. Weitzel, D. Swanson, and Y. Lu. 2012. HOG: Distributed Hadoop MapReduce on the Grid. In Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis (SCC '12). IEEE Computer Society, Washington, DC, USA, pp. 1276-1283.

S. Kadirvel and J. Fortes (2013). Towards self-caring MapReduce: a study of performance penalties under faults. Concurrency and Computation: Practice and Experience. Online first 28 May 2013. DOI: 10.1002/cpe.3044.

M. Kutlu, G. Agrawal, and O. Kurt (2012). Fault tolerant parallel data-intensive algorithms. In Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing (HPDC '12). ACM, New York, NY, USA, pp. 133-134.

V. Patil and P. Soni (2013). Hadoop skeleton & fault tolerance in Hadoop clusters. International Journal of Application or Innovation in Engineering & Management, Volume 2, Issue 2, February 2013, pp. 247-250.

VMware (2012). Protecting Hadoop with VMware vSphere 5 Fault Tolerance. Technical White Paper. Available at http://www.vmware.com/files/pdf/techpaper/VMware-vSphere-Hadoop-FT.pdf. Accesed on July 2013.

F. Wang, J. Qiu, J. Yang, B. Dong, X. Li, and Y. Li (2009). Hadoop high availability through metadata replication. In Proceedings of the first international workshop on Cloud data management (CloudDB '09). ACM, New York, NY, USA, 37-44.

T. White. (2012). Hadoop: The Definitive Guide (3rd ed.). O'REILLY. 2012, United States.

Q. Zheng (2010) Improving MapReduce fault tolerance in the cloud. In Proc. of the IEEE International Symposium on Parallel & Distributed Processing. Atlanta, USA. IEEE Press, 2010, pp. 1-6.

Table of Contents