Volume 6, Issue 6, June 2016 ISSN: 2277 128X

International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Increasing the Fault Tolerance of NameNode: A Proposal for using DataNode as a Secondary Backup Node

[1]Somnath Paul, [2]Subho Chaudhuri, [3]Debasish Jana
[1, 2] Department of Computer Science and Engineering, Birla Institute of Technology, Mesra, Jharkhand, India
[3] TEOCO Software Private Limited

Abstract— Big Data is the need of hour. Hadoop helps to store and process Big Data in a distributed environment. NameNode is the centrepiece of HDFS. On failure of NameNode, the Backup Node takes its position until it is up again. The Backup Node stores a checkpointed image of the NameNode. However in a rare condition, if the Backup Node goes down while the NameNode is still not up, the system can collapse. The proposed system suggests electing a Secondary Backup Node from the DataNodes. An algorithm for election of DataNode to Secondary Backup Node has been developed considering the cost of failure. In the event of both NameNode and Backup Node failure, the Secondary Backup Node is pressed to action. Once the NameNode is up again, the NameNode synchronizes with the Secondary Backup Node, and starts functioning again. This implementation can help us by minimizing the downtime of the system.

Keywords— HDFS, Fault Tolerance, NameNode Failure, Backup Node, Election Algorithm

I. INTRODUCTION

The Hadoop Distributed File System is the file system of Hadoop Framework. HDFS is designed for storing huge data over a large number of commodity hardware in a distributable manner. HDFS stores data as metadata and the application data. The metadata is stored in the NameNode, which contains information about the placement of application data. The application data is stored block wise in multiple DataNodes [7].

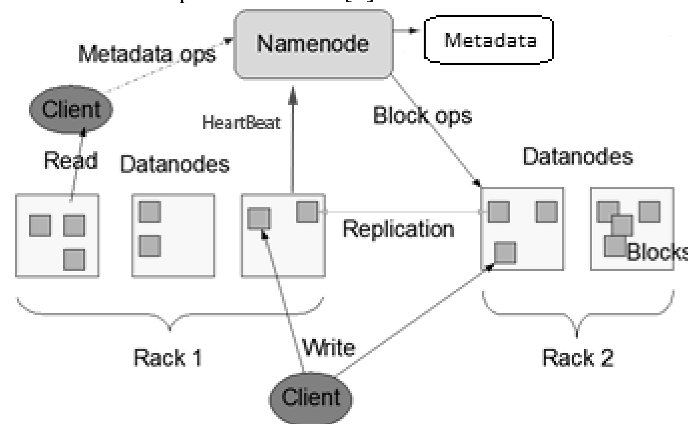

Figure 1: Architectural Backbone of HDFS (showing NameNode and DataNodes) [2]

Apart from storing metadata, the NameNode also monitors the performance of DataNodes. There is a single NameNode but multiple DataNodes. HDFS allows users to access the file system through HDFS client, which is a client machine that has Hadoop loaded in it and allows the user application in client to access HDFS interface. HDFS client support actions like read, write or delete files, add or delete directories. To read or write, the client requests the NameNode, which responds to the client by providing the data block placement in DataNode. Subsequently, the client communicates with the DataNode.

Another feature of Hadoop is Backup Node, which stores an up to date image of the NameNode, which is pressed to action on failure of NameNode. However if the Backup Node fails too before the recovery of the NameNode, then the whole system collapses [6]. In this paper we suggest an election algorithm to elect a DataNode which will be commissioned as Secondary Backup Node on failure of both NameNode and Backup Node, alleviating the problem of system non-performance on failure of NameNode and Backup Node.

The rest of the document is organized as follows. Section 2 provides a description of the related work done by other researchers regarding the fault tolerant mechanisms of HDFS. Section 3 reviews the problem we have at hand and how NameNode is prone to failure. Section 4 discusses about the solution methodology, how we intend to solve the problem. It deals with our proposal for modified architecture as well as the algorithm for selection of DataNode to Secondary

© 2016, IJARCSSE All Rights Reserved

Backup Node. Section 5 presents the benefit of the proposed system with the algorithm. Section 6 contains the conclusion and Section 7 has the references.

II. RELATED WORK

Fault tolerance of HDFS is a much sought area. Research has been done to make the system stable (Evans; Singh & Singh; Mouliswaran & Sathyan; Goiri, Julia et. al).

Yahoo! identified a major problem of Hadoop as Scalability. NameNode is a single node and not distributed over multiple nodes. NameNode keeps the namespace in its memory and if that grows large enough then it may become unresponsive. A possible suggestion was to use distributed NameNode but it can cause communication problems.

Another workaround is to store only large files in HDFS. This will reduce the namespace size [5]. There has been suggestion to modify the NameNode to a ring of servers and distribute namespace equally among them. There will be balanced distribution based on hash table. The namespace will be abundantly replicated among them. So in case of any node being down, the system continues to remain up. This removes NameNode as the single point of failure [3].

It is also suggested to replicate the namespace of the NameNode. An idea was proposed for using multiple slave nodes and a primary node. During system initialization, the slave nodes register with the primary node. Later on if the primary node fails then the slave nodes conduct an election algorithm among themselves to elect a leader, which will act as a primary node [10].

Another proposal is to use checkpoint and recovery technique. In case of any failure the system rollbacks to the previous checkpoint and the transaction initiates again [4]. This technique might provide accurate results but availability is not guaranteed always.

III. THE PROBLEM OF FAILURE

The NameNode stores the HDFS namespace in its main memory and Backup Node always maintains an up to date image of NameNode namespace in memory. In case the NameNode fails, the system becomes unavailable as it is a single point of failure [5]. Therefore the Backup Node is bought to action in NameNode failure as it contains an up to date image of the NameNode. The Backup Node functions as the NameNode and services HDFS client's request till the NameNode recovers and is up again. One more scenario to be considered is when the Backup Node too fails and then NameNode is not up yet. In this situation the availability of the system may go down. The need is to present a fault tolerant HDFS architecture and remove NameNode as the single point of failure.

IV. SOLUTION METHODOLOGY

To counter the problem, we suggest electing a DataNode which will be functioning as a Secondary Backup Node and be pressed to action (to service client requests) if the Backup Node fails too.

A. Election of DataNode as Secondary Backup Node

An algorithm for the election of DataNode as Secondary Backup Node has been proposed keeping in mind all the associated cost due to performance overhead.

1) Election of DataNode as Secondary Backup Node:

- Scan the namespace for all DataNodes and assign priority to all with highest rating being given to the node with least data in it. This is because less data in the node means less quantity of data to be copied to different nodes for backup.
- Scan the namespace to find out the average replication value of all the DataNodes and assign priority to all with highest rating being given to the node with maximum replicated data. This is because highly replicat ed data need not be copied and the node can be directly used for Backup Node.
- Find out the DataNode which has the best priority in both the cases and use it to be the Secondary Backup Node.

The node which is elected to be Secondary Backup Node is checked if it contains any data which is not replicated. In that case the data is replicated. However if the data is already replicated, then the DataNode is cleared/erased and the deleted data is replicated from other sources. Once this is done, the DataNode is synchronized with the Backup Node using the checkpoint image and journal, and then takes the live stream and updates. Each DataNode belonging to the same cluster has the equal opportunity of being elected. The algorithm for the election of the DataNode for Secondary Backup Node is given below.

The first part of the algorithm is to find the priority of each DataNode based on its DataSize. Input is an array of DataNode along with its DataSize. Array DataSize is an associative array with DataNode as key and corresponding DataSize of node as value. The array is in sorted order of DataSize in increasing order. The algorithm processes the array and assigns a priority to each DataNode. The DataNode with minimum DataSize is ranked top. The output of the algorithm is two associative arrays, Array DataSizePriority (which stores priority as key and DataNode as value) and Array DataSizeElement (which stores DataNode as key and priority as value).

2) Steps for finding the priority of each DataNode based on the DataSize:

Step 1. Store all DataNode in array DataSize with increasing order of data size of the respective DataNode.
Step 2. Assign priority 1 to the least data size DataNode.
Step 3. Store array DataSizeElement with key as the 'DataNode' and value as the respective 'priority'.
Step 4. Increment priority value and repeat for all the DataNodes (from Step 3).

3) Algorithm for finding the priority of each DataNode based on the DataSize:

© 2016, IJARCSSE All Rights Reserved

```
Assign value of 1 to priority;
For each element in Array DataSize
{
   Assign DataNode to DataSizePriority [priority];
   Assign priority to DataSizeElement [DataNode];
   Increment priority; // Unit Increment
}
```

The second part of the algorithm is finding the average replication value of each DataNode (which is the average replication value of each data element in the DataNode). Input is Array DataNodeReplication for each of the DataNode; DataNodeReplication is an array with Replication values of each data element of that DataNode. The average replication value of each DataNode is calculated and stored in an associative array DataReplication (key is DataNode and value is replication value). The array DataReplication is sorted based on replication value in descending order and then priority is assigned to each DataNode. The top priority is assigned to DataNode with highest replicated value, because then the DataNode can be selected and used as Secondary Backup Node without copying/replicating its content to another DataNode. The output of the algorithm is two associative arrays, Array DataReplicationPriority (which stores priority as key and DataNode as value) and Array DataReplicationElement (which stores DataNode as key and priority as value).

4) Steps for finding the priority of each DataNode based on the Average Replication Value:

Step 1. Calculate the sum of replication values of the DataNode and divide by the total number of values to obtain the average replication value of each DataNode.
Step 2. Store the average replication values of all the DataNode in array DataReplication. Sort the array based on average replication value in descending order.
Step 3. Assign priority 1 to the highest replicated DataNode.
Step 4. Store array DataReplicationElement with key as the ‘DataNode’ and value as the respective ‘priority’.
Step 5. Increment priority value and repeat for all the DataNodes (from Step 4)

5) Algorithm for finding the priority of each DataNode based on the Average Replication Value:

```
For each DataNode in HDFS
{
  For each element in Array DataNodeReplication
  {
     Assign sum of ReplicationSum and DataNodeReplication[element] to ReplicationSum;
  }
  Divide ReplicationSum by NumberofDataElements and Assign to DataReplication[DataNode];
}

Sort the array based on replication value in descending order

For each element in Array DataReplication
{
   Assign DataNode to DataReplicationPriority [priority];
   Assign priority to DataReplicationElement [DataNode];
   Increment priority; // Unit Increment
}
```

The third part of the algorithm deals with finding the best suitable DataNode based on highest priority in data size and data replication. The elected DataNode is Backup DataNode, which qualifies itself to be the new Secondary Backup Node.

6) Steps for finding the best suitable DataNode based on highest priority in Data Size & Data Replication:

Step 1. Assign sum of DataReplicationElement [DataNode0] and DataSizeElement [DataNode0] to MinCost.
Step 2. For each DataNode, assign sum of DataReplicationElement [DataNode] and DataSizeElement [DataNode] to Cost.
Step 3. If Cost is less than MinCost, the assign cost to MinCost and DataNode to Node.
Step 4. Repeat for all DataNodes (Step 2-3).
Step 5. Assign Node to BackupDataNode.

7) Algorithm for finding the best suitable DataNode based on highest priority in Data Size & Data Replication:

```
Assign sum of DataReplicationElement [DataNode0] and DataSizeElement [DataNode0] to MinCost;
Assign DataNode0 to Node;
For each DataNode in HDFS
{
  Assign sum of DataReplicationElement [DataNode] and DataSizeElement [DataNode] to Cost;
```

```
    If (Cost<MinCost)
    {
      Assign cost to MinCost;
      Assign DataNode to Node;
    }
  }
  Assign Node to BackupDataNode;
```

Figure 2 shows the existing Hadoop architecture. Figure 3 shows the elected DataNode replicating its data and preparing to act as Secondary Backup Node. Figure 4 show the elected Secondary Backup Node synchronizing with the Backup Node. Figure 5 shows the operation of the elected Secondary Backup Node on failure of NameNode and Backup Node.

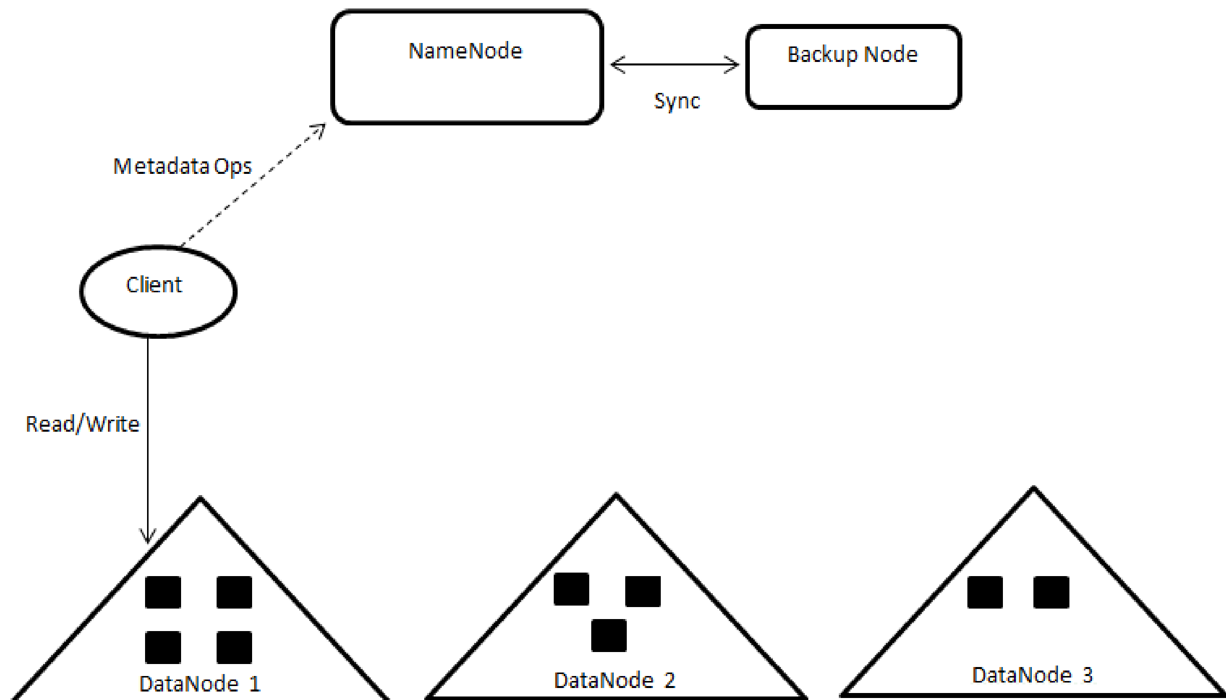

Figure 2: Existing Hadoop Architecture

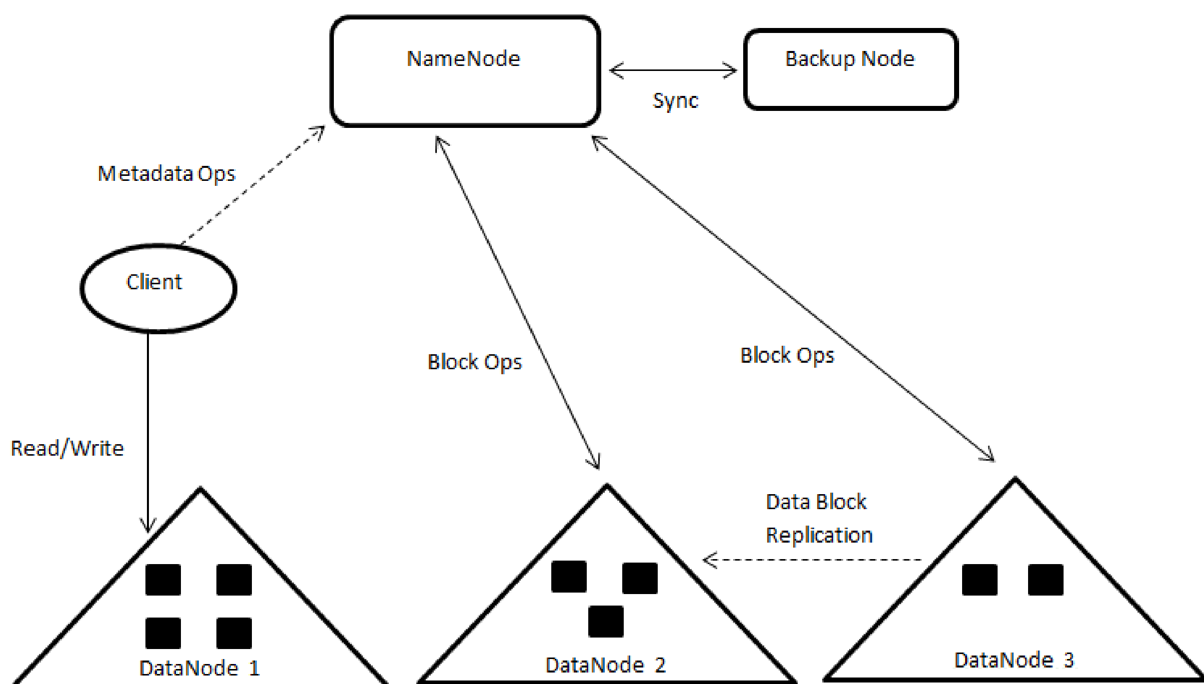

Figure 3: Elected DataNode replicates its data and prepares to act as secondary backup node

© 2016, IJARCSSE All Rights Reserved

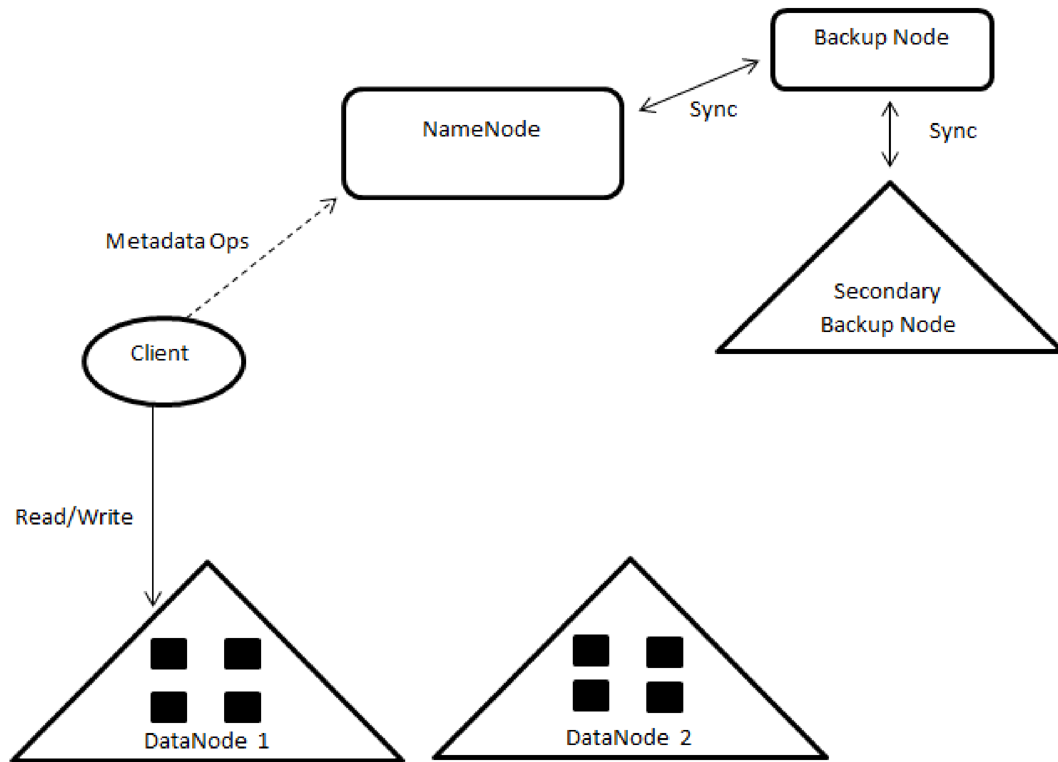

Figure 4: Elected Secondary Backup Node synchronizes with Backup Node

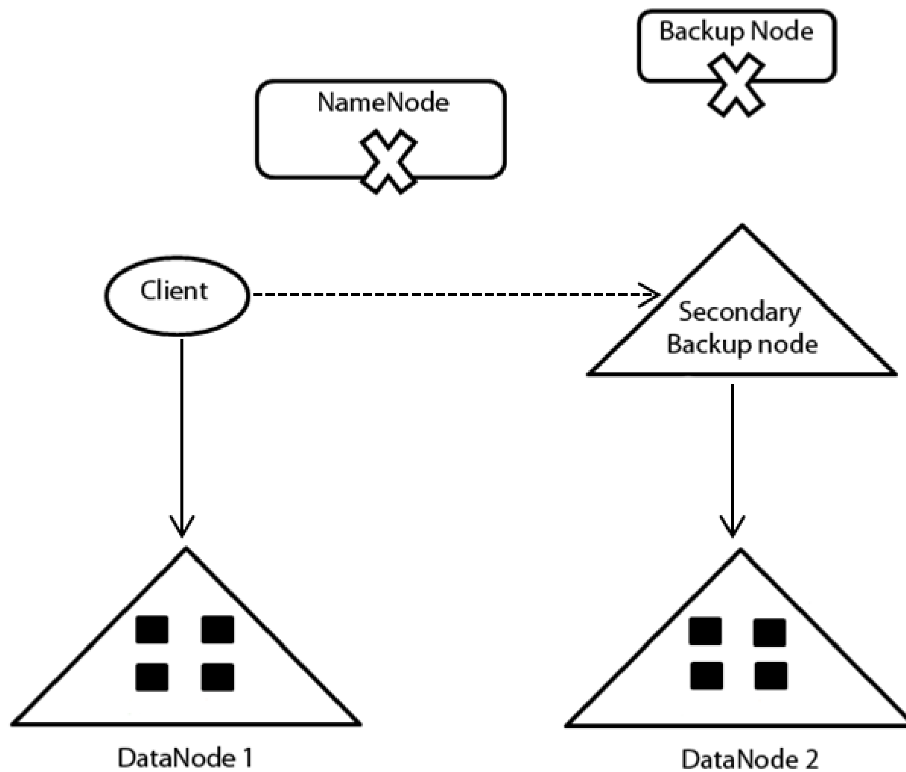

Figure 5: Secondary Backup Node in action (Failure of NameNode and Backup Node)

B. Threshold value and Cost of failure

The important issue that arises in the election of DataNode is whether the election of DataNode for Backup Node will be done prior to failure or after failure of NameNode. The cost of electing a DataNode may be high and therefore an optimized decision must be taken. An algorithm has been devised for finding the cost of selecting the DataNode.

The optimized decision will be taken based on a variable 'threshold'. The threshold value will range from 0 to 99. A value towards 99 tends to be of critical application with heavy data or compute intensive task, whose failure can be catastrophic. A value towards 0 tends to be a mild application, whose failure may not be much concern. Probability of failure will range from 0 to 1, indicating chances of a failure to occur. Impact of failure will range from 0 to 99, indicating the damage cost due to failure of application. A value of 0 denotes no loss and 99 denote maximum loss.

© 2016, IJARCSSE All Rights Reserved

If the cost of failure is less than the threshold value then the election of DataNode for Backup Node will be done after failure of NameNode, otherwise it will be done prior to failure.

```
Assign product of Probability of failure and Impact of failure to Cost of failure;
If (CostOfFailure >= ThresholdValue)
The election of DataNode will be done prior to failure
Else
The election of DataNode will be done after failure
```

C. Risk Management

Risk management will involve identification, assessment and prioritization of risks. The identification will involve finding out major risk factors of the system. These factors will include the various cause/reasons for the failure for the system. Assessment will involve identifying which risks are active and which are passive. Active risks are the risks which have higher probability of occurring and Passive risks are the risks with low probability. Prioritization will involve giving priority to active risks and taking steps to ensure the minimization of occurrence of active risks.

D. Frequency of Election Algorithm

The election among DataNodes will be initiated in the NameNode, in case the election happens before failure of NameNode (for critical applications). This is so chosen because the NameNode already has the updated block report of each DataNode. Thus it can take an informed decision without any overhead. However the election must be repeated on a periodic basis, whose frequency depends on the rate of update of namespace of the system. This is because more frequently the namespace is updated greater the chances that any other DataNode has more priority of becoming the Secondary Backup Node, as files are updated/deleted/added in the DataNodes. In case the election happens after NameNode failure, it will be initiated in the Backup Node, as it has the most updated replica of NameNode.

V. BENEFITS OF PROPOSED SOLUTION

The main concern addressed in this paper is the fault tolerance of Hadoop. The NameNode as the Single point of failure is eliminated. In normal circumstances, on failure of NameNode the system can go down. But in the proposed architecture, the NameNode is fault tolerant. The Backup Node stores checkpointed image of NameNode and there is Secondary Backup Node (elected from DataNodes) to support the failure of both NameNode as well as Backup Node. Thus the problem of system non-performance is alleviated.

The algorithm presented to elect Secondary Backup Node from DataNodes takes in consideration of data size as well as replication value of DataNode. It makes an optimized decision of electing the DataNode which has fewer data block and maximum replicated data so that the overhead in copying data blocks is minimized and the Secondary Backup Node election is completed and pressed in action quickly, to prevent any unavailability of system.

The threshold value is application specific which helps in delaying the election of Secondary Backup Node in case the failure of the system is not very catastrophic. However for critical application, the election is done before any failure occurs to alleviate the issue of unavailability.

VI. CONCLUSION

The main purpose of the paper is to propose a fault tolerant NameNode in Hadoop architecture. Electing a Secondary Backup Node helps in alleviating the unavailability of system, as the Secondary Backup Node is put to service the clients request on failure of the NameNode and Backup Node. The algorithm for electing the DataNode as Secondary Backup Node takes an optimized decision and reduces the cost of overhead. The future work involves finding more optimized and cost-efficient techniques to handle NameNode as the Single Point of Failure. We are also studying to find out the frequency of the election algorithm in NameNode and how it relates with the update rate of namespace data.

REFERENCES

[1] D. Laney. 3D Data management: Controlling Data Volume, Velocity and Variety. [online]. Available: http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf, Retrieved 17th Apr 2016.

[2] Dhruba Borthakur. HDFS Architecture Guide. [online]. Available: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html, Retrieved 1st Nov, 2015.

[3] Harcharan Jit Singh and V. P. Singh. "High Scalability of HDFS using Distributed Namespace." *International Journal of Computer Applications in Technology*, Vol. 52, No. 17, Aug 2012, pp. 30-37.

[4] I. Goiri, F. Julia, J. Guitart, and J.Torres. "Checkpoint based Fault Tolerant Infrastructure for Virtualized Service Providers." *Proceedings of the IEEE Network Operations and Management Symposium*, Apr 2010, pp. 455-462.

[5] Jared Evans. "Fault Tolerance in Hadoop for Work Migration." *CSCI B34 Survey Paper 3*, No. 28, Mar 2011.

[6] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." *Communications of the ACM*, Vol. 51, No. 1, Jan 2008, pp. 107–113.

[7] Konstantin Shvachko, Hairong Kuang, Sanjay Radia and Robert Chansler. "The Hadoop Distributed File System." *Proceedings of the 26th Symposium on Mass Storage Systems and Technologies (MSST)*, May 2010, pp. 1-10.

© 2016, IJARCSSE All Rights Reserved

[8] Venkata Swamy Martha, Weizhong Zhao and Xiaowei Xu. "h-MapReduce: A Framework for Workload Balancing in MapReduce." *Proceedings of the 27th International Conference on Advanced Information Networking and Applications (AINA)*, Mar 2013, pp. 637-644.

[9] Pooja Malikwade and S. B. Jadhav. "Boosting the Performance of MapReduce by Better Resource Utilization in Cluster." *International Journal of Computer Applications*, Vol. 112, No. 16, Feb 2015, pp. 29-33.

[10] S. Chandra Mouliswaran and Shyam Sathyan. "Study on Replica Management and High Availability in Hadoop Distributed File System (HDFS)." *Journal of Science Information Technology*, Vol. 2, No. 2, 2012, pp. 65-70.

[11] Tao Gu, Chuang Zuo, Qun Liao, Yulu Yang and Tao Li. "Improving MapReduce Performance by Data Prefetching in Heterogeneous or Shared Environments." *International Journal of Grid & Distributed Computing*, Vol. 6, No. 5, 2013, pp. 71-82.

[12] Zhenhua Guo and Geoffrey Fox. "Improving MapReduce Performance in Heterogeneous Network Environments and Resource Utilization." *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, May 2012, pp. 714-716.

© 2016, IJARCSSE All Rights Reserved