2016 IEEE 2nd International Conference on Big Data Security on Cloud, IEEE International Conference on High Performance and Smart Computing, IEEE International Conference on Intelligent Data and Security

A New Replica Placement Policy for Hadoop Distributed File System

Wei Dai [*], Ibrahim Ibrahim[*], Mostafa Bassiouni [†]
[*] Department of Electrical & Computer Engineering
[†] Department of Computer Science
University of Central Florida
Orlando, USA
wdai@knights.ucf.edu , efahad@knights.ucf.edu , bassi@cs.ucf.edu

Abstract—**Today, Hadoop Distributed File System (HDFS) is widely used to provide scalable and fault-tolerant storage of large volumes of data. One of the key issues that affect the performance of HDFS is the placement of data replicas. Although the current HDFS replica placement policy can achieve both fault tolerance and read/write efficiency, the policy cannot evenly distribute replicas across cluster nodes, and has to rely on load balancing utility to balance replica distributions. In this paper, we present a new replica placement policy for HDFS, which can generate replica distributions that are not only perfectly even but also meet all HDFS replica placement requirements.**

Keywords—Hadoop Distributed File System; Replica Placement; Data Replication; Load Balance; Hadoop; MapReduce; Cloud Computing

I. Introduction

Today, Hadoop is widely used in many enterprises as a general purpose platform for distributed storage and processing of large data sets on commodity computer clusters. Prominent Hadoop users include Yahoo, Facebook, IBM, Twitter, and Adobe [1]. Many well-known enterprise vendors have been offering either commercial Hadoop products or technical support for Hadoop, including Amazon, Microsoft, Oracle and specialist Hadoop companies, such as Cloudera. The Hadoop Distributed File System (HDFS) is the storage part of the Hadoop framework, which is a distributed, scalable, and portable file system designed to run on commodity hardware. Although it has many similarities with other existing distributed file systems, HDFS is especially designed to be highly fault-tolerant, to provide high throughput access to application data, and to deal with very large data files (typically gigabytes to terabytes in size).

In HDFS, each data file is stored as a sequence of blocks which are replicated for both data reliability and performance improvement. By default, each block has three replicas. The placement of these replicas is crucial to the performance of HDFS. The current HDFS Replica Placement Policy (RPP) is a rack-aware policy which can improve write performance while maintaining data reliability and read performance. The drawback of the policy is that it cannot evenly distribute replicas to cluster nodes. Since an unbalanced HDFS cluster can seriously degrade the performance of Hadoop applications, HDFS provides a balancing utility to address the issue. The utility can analyze replica placement and rebalance replicas across the cluster nodes at the cost of extra system resources and running time.

In this paper, we present an innovative replica placement policy which addresses the above issue from a completely different perspective — it can distribute replicas to cluster nodes as evenly as possible, and also meet all replica placement requirements of HDFS. As a result, there is no need to run the balancing utility. To the best of our knowledge, our new policy is the first that addresses the load balancing issue by generating an even replica distribution in the first place.

The rest of the paper is organized as follows. Section II introduces the HDFS RPP and related work. Section III describes the proposed policy in detail. Evaluation results are presented in Section IV, and we conclude in Section V.

II. Background and Related Work

The current HDFS RPP (as of Hadoop 2.7.1) consists of the following three steps [2]. Assume the HDFS client runs outside the Hadoop cluster. The policy first places one replica on a random node. It then randomly selects another node on the same rack as the first node, and places the second replica on the node. Finally, the policy places the third replica on a random node on a rack that is different from the one where the first two replicas reside. Since the replicas of a block are placed on only two unique racks instead of three, this policy can cut down the inter-rack write traffic when distributing data to cluster nodes. On the other hand, replicas on two different racks are mostly sufficient for maintaining good read performance and high fault tolerance. However, this policy cannot generate an even replica distribution due to its inherent unbalanced placement logic.

There has been large amount of research work conducted on the HDFS RPP due to its importance regarding improving the performance of HDFS. Shabeera et al. propose a bandwidth-aware RPP for Hadoop in [3], which measures and compares the bandwidth between the HDFS client and cluster nodes periodically, and places the replica on the node that has the maximum bandwidth to reduce the data transfer time. Zhang et al. address the dynamic block replication issue in [4] from the perspective of replica placement. They propose several constant-factor local search approximation algorithms, and present a dynamic replica distribution mechanism that implements the algorithms in HDFS. Xie et al. propose a replica placement mechanism for HDFS running on heterogeneous clusters in [5], which distributes replicas to nodes based on their different computing capacities to balance the data processing load across all cluster nodes. Eltabakh et al. present CoHadoop in [6], which is an HDFS implementation of a flexible, dynamic, and light-weight approach for collocating related data files. Compared with the HDFS RPP, CoHadoop can remarkably improve the performance of Hadoop applications that process data from multiple files. In our previous work [7], we address the load balancing issue from the perspective of task assignment of Hadoop, and propose an improved task assignment scheme that strives to balance the map task processing load of MapReduce jobs across all cluster nodes. Finally, the recent research work presented in [8] and [9] also helped with the formation of our research idea.

III. Partition Replica Placement Policy

The fundamental reason for the uneven replica distributions generated by the HDFS RPP is that it needs to place two replicas on one random rack and the third replica on another random rack. To overcome this problem, our new policy divides all available nodes into three sections. Section 1 has about two thirds of the nodes, and is used to store the first two replicas on the same rack. Sections 2 and 3 have about one third of the nodes, and are used to store the third replica. Since the partition scheme is the key of our new policy, which makes it possible to generate an even replica distribution, we call our new policy the Partition Replica Placement Policy (PRPP). Under PRPP, the whole replica placement process consists of two phases: section formation phase, and replica distribution phase.

In section formation phase, PRPP divides all nodes into three sections. If the total number of nodes is not a multiple of three, the policy will try to minimize the disparity between section 1 and the other two sections. Let $\boldsymbol{R}$ be a collection of $\boldsymbol{m}$ racks, each of which has $\boldsymbol{r_i}$ ($\boldsymbol{i}$ = 0,1, 2, …, $\boldsymbol{m}$-1) available nodes on it. Let $\boldsymbol{N}$ be the total number of all available nodes, $\boldsymbol{N_1}$ be the number of nodes to be assigned to section 1. $\boldsymbol{N}$ and $\boldsymbol{N_1}$ are calculated according to the following formulas:

$$N = \sum_{i=0}^{m-1} r_i \tag{1}$$

$$d = N/3, \quad r = N\%3, \quad N_1 = \begin{cases} 2d & (r = 0 \text{ or } 1) \\ 2d+2 & (r = 2) \end{cases} \tag{2}$$

, where "/" denotes integer division, and "%" integer remainder. (Note that $\boldsymbol{N_1}$ is always even.)

After figuring out $\boldsymbol{N_1}$, PRPP will assign nodes to section 1 from racks 0, 1, 2 and so on, until the number of nodes in section 1 has reached $\boldsymbol{N_1}$. During this process, PRPP only assigns even number of nodes from each rack to section 1, and it will assign nodes to section 2 instead of section 1 in the following two cases:

- During the process of assigning nodes to section 1, if it encounters a rack having odd numbers of nodes, PRPP will assign all the nodes on that rack to section 1, except for the last node which will be assigned to section 2 instead.
- At the end of the process of assigning nodes to section 1, when the number of nodes in section 1 has reached $\boldsymbol{N_1}$, PRPP will assign all the remaining nodes on current rack (if any) to section 2.

If neither of the above situations occurs, there will be no node in section 2. After $\boldsymbol{N_1}$ nodes have been assigned to section 1, all the remaining nodes will be assigned to section 3. Fig. 1 shows an example of the formation of the three sections. Assume there are three racks, rack 0 has ten available nodes, rack 1 ten, and rack 2 seven. In this example: $\boldsymbol{N}$ = 27, $\boldsymbol{d} = \boldsymbol{N}$ / 3 = 9, $\boldsymbol{r} = \boldsymbol{N}$ % 3 = 0, and so $\boldsymbol{N_1}$ = 2$\boldsymbol{d}$ = 18. PRPP first assigns all ten nodes on rack 0 to section 1, and then the first eight nodes on rack 1 to section 1. Now the number of nodes in section 1 has reached 18, thus the last two nodes on rack 1 are assigned to section 2, and all the nodes on rack 2 are assigned to section 3. The following is the pseudocode of the section formation algorithm.

Algorithm 1: section formation algorithm

Input:
A collection of $\boldsymbol{m}$ racks, each of which has $\boldsymbol{r_i}$ ($\boldsymbol{i}$ = 0, 1, 2, …, $\boldsymbol{m}$-1) available nodes on it.
Output:
$\boldsymbol{S_1}$: Node section 1; $\boldsymbol{S_2}$: Node section 2; $\boldsymbol{S_3}$: Node section 3.

```
1.  calculate $N$ and $N_1$ according to (1) and (2) respectively;
2.  $TN = N_1$;      // Total number of nodes to be assigned to node section 1.
3.  for ( $i$ = 0 to $m$ -1 )     // Assign m racks of nodes to three node sections.
4.      if ($TN > 0$)          // Node section 1 still needs more nodes.
5.          if ($TN \ge r_i$)
6.              if ( $r_i$ is even )
7.                  assign all $r_i$ nodes on rack $i$ to $S_1$ ;
8.                  $TN = TN - r_i$;
9.              else
10.                 assign the first ($r_i$ -1) nodes on rack $i$
                    to $S_1$ , and the last node to $S_2$ ;
11.                 $TN = TN - (r_i -1)$ ;
12.             end if
13.         else
14.             assign the first $TN$ nodes on rack $i$ to $S_1$ ,
                and the remaining nodes on rack $i$ to $S_2$ ;
15.             $TN = 0$ ;
16.         end if
17.     else       // The number of nodes in section 1 has reached $N_1$ .
18.         assign all the nodes on rack $i$ to $S_3$ ;
19.     end if
20. end for
```

In the following replica distribution phase, PRPP distributes replicas to all cluster nodes in the three sections formed in the previous phase. PRPP first constructs three replica assignment tables, one for each node section as shown in Fig. 1. Each column in the replica assignment tables corresponds to the replica assignment of one node. All three tables have the same number of lines $\boldsymbol{k}$, which is calculated according to the following formula:

$$d = (n \times f)/N, \; r = (n \times f)\%N, \; k = \begin{cases} d & (r = 0) \\ d+1 & (r \neq 0) \end{cases} \tag{3}$$

, where $\boldsymbol{n}$ is the total number of blocks, $\boldsymbol{f}$ the duplication factor of Hadoop (3 in this case), $\boldsymbol{N}$ the total number of nodes.

PRPP then uses a tabular scheme to distribute the two replicas that are supposed to be placed on the same rack to nodes in section 1 (table 1). As shown in Fig. 2, replicas are filled into table 1 in block number order, from left to right, and from top to bottom. Each column in table 1 corresponds to the replica assignment of one node. For example, Node 0 has the replicas of blocks 0 and 9 assigned to it. Since the number of nodes on

Table	1									2		3			
Section	1									2		3			
Rack	0				1					1		2			
Node	0	1	…	9	10	11	…	17		18	19	20	21	…	26
Block No. of Replica															
	……									……		……			

Fig. 1. Example of section formation and replica assignment tables.

Table	1																	
Section	1																	
Rack	0										1							
Node	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Block No. of Replica	0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8
	9	9	10	10	11	11	12	12	13	13	14	14	15	15	16	16	17	17
	……										……							

Fig. 2. Replica placement before randomization.

Table	1																	
Section	1																	
Rack	0										1							
Node	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Block No. of Replica	2	0	3	1	3	4	1	0	2	4	6	5	5	8	6	7	7	8
	11	10	13	13	12	12	10	9	9	11	15	16	14	16	17	17	15	14
	……										……							

Fig. 3. Replica placement after randomization.

each rack in section 1 is even, the two replicas of all the blocks filled into table 1 will be in two different columns within the same rack portion, which means they are on two different nodes on the same rack. The replica placement in Fig. 2 already meets all HDFS requirements except for that the adjacent nodes on each rack have identical replica assignment. To reduce the impact of the failure of one node on other nodes, the replica placement needs to be randomized so that no nodes have the same replica assignment. In the example shown in Fig. 2, PRPP first randomizes each line in the rack 0 portion of table 1, then each line in the rack 1 portion. The result is shown in Fig. 3. The randomization is done within each line and within each rack portion. Therefore, after the randomization, the two replicas of the same block will remain in two different columns in the same rack portion, or on two different nodes on the same rack per the HDFS replica placement rules. The following is the pseudocode of the replica distribution algorithm for node section 1.

Algorithm 2: replica distribution algorithm for node section 1

```
Input:
S_1[0, 1, …, N_1 -1]: Node section 1, which has N_1 nodes in it.
n data blocks to be distributed, which are numbered 0 through (n - 1).
Output:
T_1[k][N_1]: Replica assignment table for node section 1.

1.  calculate k according to (3);
2.  for (i = 0 to N_1 -1)
3.      CL[i] = 0;    // Reset current line number in table T_1 for each node.
4.  end for
5.  P = 0;            // Reset current node pointer.
6.  for (i = 0 to n -1)
7.      CN = S_1[P];          // Retrieve the next node in S_1 as current node.

8.      T_1[CL[CN]][CN] = i;          // Put the first replica of
                                      // current block on current node.
9.      CL[CN] = CL[CN] + 1;          // Increment current line number of
                                      // current node.
10.     P = P + 1;                    // Increment current node pointer.
11.     CN = S_1[P];          // Retrieve the next node in S_1 as current node.
12.     T_1[CL[CN]][CN] = i;          // Put the second replica of
                                      // current block on current node.
13.     CL[CN] = CL[CN] + 1;
14.     P = P + 1;
15.     if (P == N_1)
16.         P = 0;                    // Reset current node pointer
17.     end if                        // when it reaches the end.
18. end for
```

The following is the pseudocode of the randomization algorithm for node section 1.

Algorithm 3: randomization algorithm for node section 1

```
Input:
S_1[0, 1, …, N_1 -1]: Node section 1, which has N_1 nodes in it.
T_1[k][N_1]: Replica assignment table before randomization.
Output:
T_1[k][N_1]: Replica assignment table after randomization.

1.  TN = N_1;        // Total number of nodes to be processed.
2.  P = 0;           // Current node pointer.
3.  CN = S_1[P];     // Retrieve the first node in S_1 as current node.
4.  while (TN > 0)
5.      CR = rackNumberOf(CN)         // rackNumberOf(CN) is the rack
                                      // number of CN.
6.      C = 0;       // The count of nodes retrieved that are on the same rack.
7.      RP = 0;      // Retrieved nodes pointer.
8.      do           // Retrieve nodes in S_1 that are on the same rack.
9.          S[RP] = CN;          // Save current node in S.
10.         RP = RP + 1;
11.         C = C + 1;
12.         TN = TN - 1;
13.         P = P + 1;           // Increment current node pointer.
14.         if (P == N_1)        // Reset the pointer when it reaches the end.
15.             P = 0;
16.         end if
17.         CN = S_1[P];         // Retrieve the next node in S_1 as current node.
18.     while (CR == rackNumberOf(CN))
19.     for (i = 0 to k -1 )      // Randomize the replica placement on the
                                  // retrieved nodes.
20.         for (j = 0 to C-1)    // Retrieve one line of replicas from the
```

```
21.             $L[j] = T_1[i][S[j]]$;  // portion of $T_1$ that corresponds to the above
22.         end for                     // retrieved nodes and save it in $L[0,1, ..., C-1]$.
23.         randomize the elements of $L[0,1, …, C-1]$;
24.         for ($j$ = 0 to $C$-1)             // Copy the randomized result back to $T_1$.
25.             $T_1[i][S[j]] = L[j]$;
26.         end for
27.     end for
28. end while
```

After distributing the first two replicas to nodes in section 1, PRPP continues to distribute the third replicas to nodes in sections 2 and 3. The distribution to section 3 nodes is fairly simple as any replica can be placed on any node. However this is not true for the nodes in section 2. Each section 2 node is on the same rack as certain section 1 nodes where the first two replicas of certain blocks reside. Thus, the third replica of those blocks cannot be put on that section 2 node. Under PRPP, the replica distribution to all nodes will be as even as possible. If the total number of blocks is a multiple of the total number of nodes in sections 2 and 3, each node in sections 2 and 3 will have the same number of replicas assigned after the distribution process ends. Otherwise, PRPP will have to assign one more block to certain nodes, and it will choose section 3 nodes first as any replica can be distributed to section 3 nodes. Before the distribution process starts, PRPP calculates $\boldsymbol{n_2}$ and $\boldsymbol{n_3}$, which are the total number of replicas to be distributed to the nodes in sections 2 and 3 respectively, according to the following formulas:

$$d = n / \left(N_2 + N_3\right) \quad , \quad r = n \% \left(N_2 + N_3\right)$$

$$n_2 = \begin{cases} d \times N_2 & \left(r \le N_3\right) \\ d \times N_2 + r - N_3 & \left(r > N_3\right) \end{cases} \tag{4}$$

$$n_3 = n - n_2 \tag{5}$$

, where $\boldsymbol{n}$ is the total number of blocks, $\boldsymbol{N_2}$ the number of nodes in section 2, $\boldsymbol{N_3}$ the number of nodes in section 3.

Since not all replicas can be distributed to section 2 nodes, PRPP starts with section 2. The replicas are filled into the replica assignment table for section 2 in block number order, from left to right, and from top to bottom. Before it assigns the third replica of current block to current node, PRPP checks whether the first two replicas of current block reside on the same rack as current node. If so, PRPP discards current block and keeps trying the next block in block number order until it finds one that meets the requirement and assigns it to current node. Otherwise, PRPP assigns the third replica of current block to current node, and proceeds to next block and next node. The process continues until PRPP has assigned $\boldsymbol{n_2}$ replicas. Finally, PRPP fills the third replicas of the remaining $\boldsymbol{n_3}$ blocks into the replica assignment table for section 3 in the same way without performing the check it does for section 2 nodes. The following is the pseudocode of the replica distribution algorithm for node sections 2 and 3.

Algorithm 4: replica distribution algorithm for node sections 2 and 3

Input:
$S_2[0, 1, …, N_2-1]$: Node section 2, which has N_2 nodes in it.
$S_3[0, 1, …, N_3-1]$: Node section 3, which has N_3 nodes in it.
n data blocks to be distributed, which are numbered 0 through (n - 1).
Output:
$T_2[k][N_2]$: Replica assignment table for node section 2.
$T_3[k][N_3]$: Replica assignment table for node section 3.

```
1.  calculate $n_2$ and $n_3$ according to (4) and (5) respectively;
2.  for ($i$ = 0 to $n$ -1)
3.      $A[i]$ = false;   // $A[i]$ indicates whether block $i$ has been assigned.
4.  end for
5.  for ($i$ = 0 to $N_2$-1)
6.      $CL[i]$ = 0;      // Reset current line number in table $T_2$ for each node.
7.  end for
8.  $P$ = 0;             // Reset current node pointer.
9.  $CB$ = 0;            // Reset current block number.
10. for ($i$ = 1 to $n_2$)          // Distribute $n_2$ replicas to section 2 nodes.
11.         $CN = S_2[P]$;          // Retrieve the next node in $S_2$ as current node.
12.         $P = P + 1$;
13.         if ($P == N_2$)          // Reset current node pointer when it reaches
14.             $P$ = 0;              // the end.
15.         end if
16.         while (rackNumberOf($CB$) == rackNumberOf($CN$)
                    or $A[CB]$ == true)
            // Find the CB that meets the requirement; rackNumberOf(CB) is
            // the number of the rack where the first two replicas of CB reside;
            // rackNumberOf(CN) is the rack number of CN.
17.             $CB = CB + 1$;
18.             if ($CB == n$)                // Reset current block number
19.                 $CB$ = 0;                  // when it reaches the end.
20.             end if
21.         end while
22.         $T_2[CL[CN]][CN] = CB$;            // Place the above found CB on
                                               // current node.
23.         $CL[CN] = CL[CN] + 1$;
24.         $A[CB]$ = true;                    // CB has been assigned.
25.         $CB = CB + 1$;
26.         if ($CB == n$)
27.             $CB$ = 0;
28.         end if
29. end for
30. $P$ = 0;                 // Reset current node pointer.
31. $CB$ = 0;                // Reset current block number.
32. for ($i$ = 1 to $n_3$)      // Distribute the remaining $n_3$ replicas to
                             // section 3 nodes.
33.     $CN = S_3[P]$;          // Retrieve the next node in $S_3$ as current node.
34.     $P = P + 1$;
35.     if ($P == N_3$)     // Reset current node pointer when it reaches the end.
36.         $P$ = 0;
37.     end if
38.     while ($A[CB]$ == true)       // Find the next unassigned block CB.
39.         $CB = CB + 1$;
40.     end while
41.     $T_3[CL[CN]][CN] = CB$;       // Place the above found CB on
                                    // current node.
42.     $CL[CN] = CL[CN] + 1$;
43.     $CB = CB + 1$;
44. end for
```

IV. Evaluation

In statistics, the number of replicas on one single node in the replica distributions generated by the HDFS RPP is a Discreet Random Variable (DRV). To examine the distribution of a DRV, theoretical simulation is more appropriate than actual implementation because much more distribution samples can be obtained by simulation. Therefore, we conducted large scale simulation to examine the replica distribution generated by HDFS RPP. The simulation was conducted in three scenarios as shown in TABLE I. Scenarios one and two resemble the two common situations Hadoop applications encounter in practice. In scenario one, the application has a dedicated cluster where all nodes on all racks are available to it. While in scenario two, the application shares the cluster with other applications, and hence only partial cluster nodes are available to it.

Let X be the number of replicas on one single node in the replica distributions generated by HDFS RPP. Fig. 4 and Fig. 5 show the probability distribution of X in Scenarios one and two respectively, based on 10,000 simulation runs. The simulation results confirm that, in both scenarios one and two, HDFS RPP generates uneven replica distributions. The number of replicas on one node spreads over a wide range from 12 to 79 in scenario one and from 9 to 77 in scenario two. TABLE II shows the statistics results calculated based on all the samples generated by HDFS RPP in the simulation. It can be observed that the two probability distributions shown in Fig. 4 and Fig. 5 are very close. This is because HDFS RPP randomly selects cluster nodes directly instead of selecting a cluster rack first, and hence the node distribution across racks doesn't significantly affect the replica distribution. If the policy randomly selects a rack first, the replica distribution would be more uneven when the node distribution is heterogeneous, because in general the nodes on racks with less nodes would have more replicas than the nodes on racks with more nodes.

TABLE I. SIMULATION SETTINGS

Simulation Settings	**Scenario One**	**Scenario Two**	**Scenario Three**
Total Number of Blocks	8,000		54
Duplication Factor	3		
Total Number of Replicas	24,000		162
Total Number of Nodes	600		27
Average Number of Replicas on One Node	40		6
Total Number of Racks	30	45	3
Number of Available Nodes on Rack *i*	20	20 ($i = 0, 3, 6, \ldots, 42$) 15 ($i = 1, 4, 7, \ldots, 43$) 5 ($i = 2, 5, 8, \ldots, 44$)	10 ($i = 0, 1$) 7 ($i = 2$)

TABLE II. STATISTICS RESULTS OF HDFS RPP SAMPLES

Statistics Results	**Scenario One**	**Scenario Two**
Minimum Value	12	9
Maximum Value	79	77
Sample Mean	40.0	40.0
Sample Variance	39.83	39.80
Sample Standard Deviation	6.31	6.31

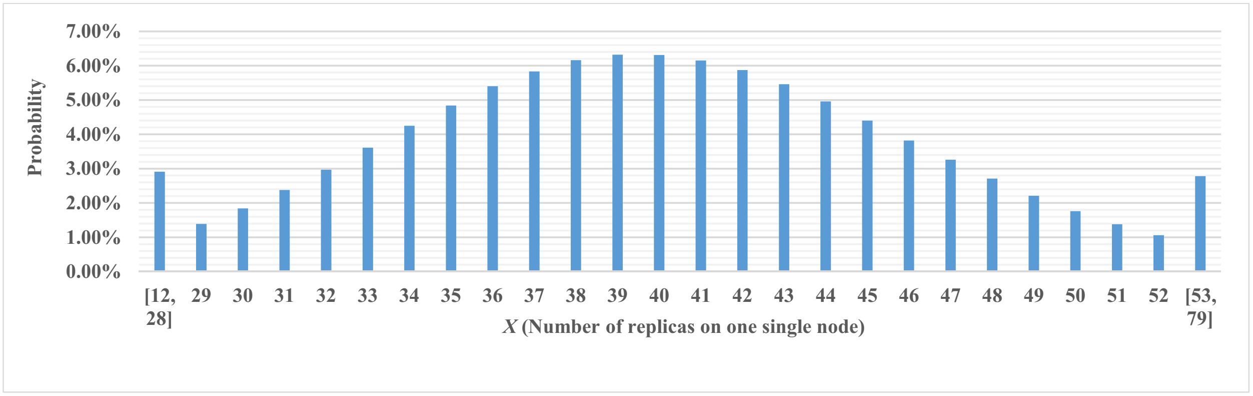

Fig. 4. Probability distribution of ***X*** in scenario one.

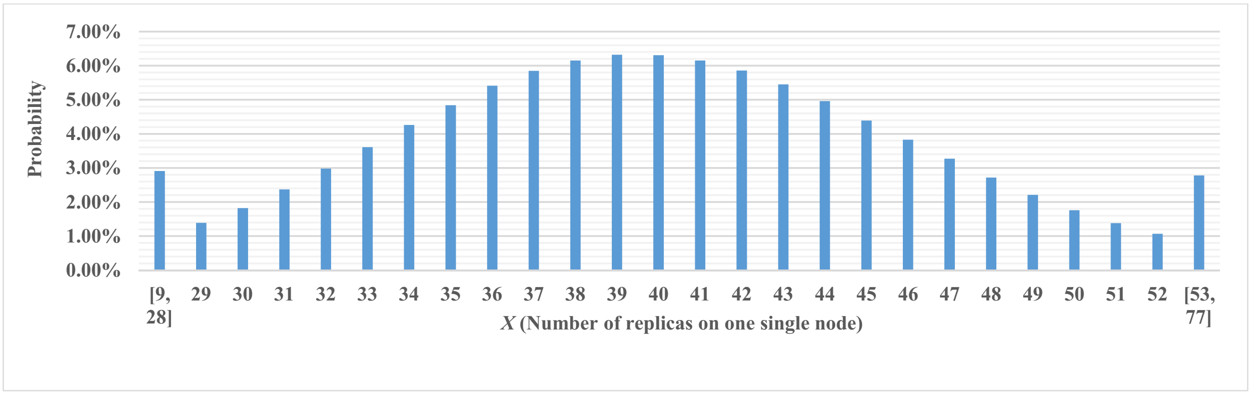

Fig. 5. Probability distribution of ***X*** in scenario two.

Rack	0										1										2						
Node	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
Block No. of Replica	7	2	7	9	20	9	0	3	3	4	8	15	1	20	4	17	1	6	13	6	6	0	16	2	0	9	11
	16	11	8	12	40	16	2	32	22	8	10	25	5	27	17	20	7	28	14	15	14	22	18	5	1	24	19
	26	18	24	25	41	27	4	52	53	11	13	26	19	30	23	23	21	32	40	19	21	31	24	10	3		22
	29	49	29	31		30	12	53		18	26	36	34	47	35	28	30	36	46	29	45	35	28	12	5		39
	33	52	43	33		31	13			23	38	42	37		44	46	37	41	48	39		36	50	17	10		45
	34		51	50		45	25			34	47	48	40			53	38					43	51	21	14		51
	37					49	27			38		49					42					46		35	15		
	41						32			43												50		39	33		
							42			47														44	44		
										48														52			

Fig. 6. Replica distribution generated by HDFS RPP in scenario three.

Section	1																		2		3						
Rack	0										1										2						
Node	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
Block No. of Replica	2	0	3	1	3	4	1	0	2	4	6	5	5	8	6	7	7	8	0	1	5	6	7	8	14	15	16
	11	10	13	13	12	12	10	9	9	11	15	16	14	16	17	17	15	14	2	3	17	20	21	22	23	24	25
	21	21	19	19	20	22	20	22	18	18	23	24	24	26	23	26	25	25	4	9	26	27	28	29	30	31	32
	28	31	31	28	27	29	30	27	30	29	33	35	34	34	33	32	32	35	10	11	33	34	35	36	37	38	39
	36	37	39	40	40	39	36	38	37	38	41	42	42	43	44	43	41	44	12	13	40	41	42	43	44	45	46
	49	48	49	47	47	48	45	46	45	46	52	52	51	50	51	53	50	53	18	19	47	48	49	50	51	52	53

Fig. 7. Replica distribution generated by PRPP in scenario three.

On the other hand, our simulation also confirms that the proposed PRPP can generate perfectly even replica distributions in both scenarios one and two, where each node has 40 replicas on it. Due to the large scale of the first two scenarios, the replica distributions generated in them cannot be presented in this paper. Therefore, we include scenario three, which is in reduced scale as shown in TABLE I, to present the replica distributions generated by both HDFS RPP and PRPP.

Fig. 6 shows one replica distribution generated by HDFS RPP in scenario three. The replica distribution is uneven with number of replicas on one single node ranging from 2 to 10. In contrast, as shown in Fig. 7, the replica distribution generated by PRPP in scenario three is not only perfectly even, but also meets all HDFS replica placement requirements.

V. Conclusion and Future Work

In this paper, we propose a new replica placement policy for HDFS, which addresses the load balancing issue by evenly distributing replicas to cluster nodes, and hence eliminates the need for running any load balancing utility. The simulation results confirm that PRPP can generate replica distributions that are perfectly even and also comply with all HDFS replica placement rules. PRPP is devised for homogeneous cluster environments where all cluster nodes have the same computing capabilities. In future research work, we plan to adapt PRPP for heterogeneous environments where multiple generations of hardware coexist.

References

[1] Hadoop Wiki, http://wiki.apache.org/hadoop/PoweredBy, Accessed November 28, 2015.

[2] HDFS Architecture, http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html, Accessed November 28, 2015.

[3] T. P. Shabeera and S. D. Madhu Kumar, “Bandwidth-aware data placement scheme for Hadoop,” in Proceedings of the 2013 IEEE Recent Advances in Intelligent Computational Systems, pp. 64-67 , Trivandrum, Kerala, India, 2013.

[4] Q. Zhang, S. Q. Zhang, A. Leon-Garcia, and R. Boutaba, “Aurora: adaptive block replication in distributed file systems,” in Proceedings of the 2015 IEEE 35th International Conference on Distributed Computing Systems, Columbus, USA, 2015.

[5] J. Xie, S. Yin, X. Ruan, Z. Ding, Y. Tian, J. Majors, A. Manzanares, and X. Qin, “Improving MapReduce performance through data placement in heterogeneous Hadoop clusters,” in Proceedings of the 2010 IEEE International Symposium on Parallel & Distributed Processing, pp. 1-9, Atlanta, USA, 2010.

[6] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson, “CoHadoop: flexible data placement and its exploitation in Hadoop,” in Proceedings of the VLDB Endowment, Vol. 4, Issue 9, pp. 575-585, June 2011.

[7] W. Dai and M. Bassiouni, “An improved task assignment scheme for Hadoop running in the clouds,” in Journal of Cloud Computing, Springer Publishing, Vol. 2:23, pp. 1-16, December 2013.

[8] M. Qiu, Z. Ming, J. Li, K. Gai, and Z. Zong, “Phase-Change Memory Optimization for Green Cloud with Genetic Algorithm,” in IEEE Transactions on Computers, Vol. 64, Issue 12, pp. 3528-3540, December 2015.

[9] Y. Li, W. Dai, Z. Ming, and M. Qiu, “Privacy Protection for Preventing Data Over-Collection in Smart City”, in IEEE Transactions on Computers, Vol. PP, Issue 99, August 2015.