2010 IEEE International Conference on Cluster Computing

CDRM: A Cost-effective Dynamic Replication Management Scheme for Cloud Storage Cluster

Qingsong Wei
Data Storage Institute, A*STAR,
Singapore
WEI_Qingsong@dsi.a-star.edu.sg

Bharadwaj Veeravalli, Bozhao Gong
National University of Singapore
Singapore
{elebv, bozhao}@nus.edu.sg

Lingfang Zeng, Dan Feng
Huazhong University of Science & Technology
China
{lfzeng, dfeng}@hust.edu.cn

***Abstract*—Data replication has been widely used as a mean of increasing the data availability of large-scale cloud storage systems where failures are normal. Aiming to provide cost-effective availability, and improve performance and load-balancing of cloud storage, this paper presents a cost-effective dynamic replication management scheme referred to as CDRM. A novel model is proposed to capture the relationship between availability and replica number. CDRM leverages this model to calculate and maintain minimal replica number for a given availability requirement. Replica placement is based on capacity and blocking probability of data nodes. By adjusting replica number and location according to workload changing and node capacity, CDRM can dynamically redistribute workloads among data nodes in the heterogeneous cloud.**

We implemented CDRM in Hadoop Distributed File System (HDFS) and experiment results conclusively demonstrate that our CDRM is cost effective and outperforms default replication management of HDFS in terms of performance and load balancing for large-scale cloud storage.

***Keywords* — Dynamic Replication Management; HDFS; Cost-effective; Load Balance**

I. INTRODUCTION

Cloud storage is emerging as a powerful paradigm for sharing information across the Internet, which satisfies people's mobile data demand anywhere and anytime. Rather than relying on a few central large storage arrays, such a cloud storage system consolidates large numbers of geographically distributed computers into a single storage pool and provides large capacity, high performance storage service at low costs in unreliable and dynamic network environment [1].

Striping is one of the performance enhancing techniques for cloud storage system that has been widely used. Files are divided into multiple blocks and distributed across data nodes to enable parallel data transfer, thus achieve high aggregate bandwidth. However, data nodes may be unreachable in a heterogeneous cloud storage system, in which failure is the norm rather than the exception. If one of the blocks is unavailable, so as the whole file. Efficient data sharing in such a cloud storage system is complicated by erratic node failure, unreliable network connectivity and limited bandwidth. Some applications such as scientific application strongly require data to be available when they need it, at least with high probability. As large numbers of data nodes are used in the cloud storage system, the probability of some data nodes failure increases, and therefore, improving data availability in a cloud storage system becomes a significant challenge for the system designers.

Data replication has been widely used as a mean of increasing the data availability of storage systems such as RAID 1, Google file system [1], Ceph file system [13] and HDFS [3]. If multiple copies of block exist on different data nodes, the chances of at least one copy being accessible increase. When one data node fails, the data is still available from the replicas and service need not be interrupted. While replication has above advantage, the management cost will significantly increase with the number of replica increasing. Too much replicas may not significantly improve availability, but bring unnecessary spending instead. *Hence, how many minimal replicas should be kept in the system to satisfy availability requirement*?

By keeping all replicas active, the additional copies may be used not only to improve availability, but also to improve load balance and overall performance if replicas and requests are reasonably distributed[12,14]. However, replica placement in such a large-scale heterogeneous storage system becomes more complicated when compared to a small-scale system, where each data node may have different capability and can only admit a restricted number of requests. Evenly assigning replicas to the data nodes in such a storage system does not balance the workload and maximize the overall performance. Access skew may happen that the requests to a few intensive data nodes may be rejected, or blocked due to their capacity constrains, while other data nodes are idle. This may result in load imbalance across cloud storage, as well as poor parallelism and low performance. This significant deficiency raises a key concern: *how to place these replicas to effectively distribute workloads data node cluster*?

Therefore, in this paper, we address the above mentioned two issues by designing a cost-effective dynamic replication management scheme for large-scale cloud storage system refereed to as CDRM. We first build up a model to capture the relationship between availability and replica number. Based on this model, lower bound on replica reference number to satisfy availability requirement can be determined. We further place replicas among data nodes in a balance way, taking into account capacity (CPU power, memory capacity, network bandwidth, etc.) and blocking probability of each data node. Proposed CDRM can adapt to the changes of environment and maintains a rational number of replica, which not only satisfies availability, but also improves access latency, load balance, and keeps the whole storage system stable. We implemented CDRM in HDFS and experiment results prove that proposed CDRM can significantly improve availability, performance and load balance for real-life applications.

The rest of this paper is organized as follows. Section II provides an overview of background. We present cost-effective dynamic replication management scheme (CDRM) in Section III. Evaluation results are presented in Section IV. In Section V, we give a brief study of related works in the literature and conclusions and possible future work are summarized in Section VII.

II. BACKGROUND

In this section, we shall present three basic concepts that are essential to our work: cloud storage, replication management and replica placement.

A. Cloud Storage

With the rapid growth of Internet services, many large server infrastructures have been set up as data centers and cloud storage platforms, such as those in Google, Amazon and Yahoo!. Compared with traditional large scale storage systems built for HPC (High Performance Computing), they focus on providing and publishing storage service on Internet which is sensitive to application workloads and user behaviors. Moreover, they provide both scalable data management and an efficient MapReduce [2] programming model for data-intensive computing.

The key components of cloud storage infrastructures are distributed file systems. Three famous examples are Google file system [1] (GFS), Hadoop [3] distributed file system (HDFS) and Amazon Simple Storage Service (S3) [4]. Among them, HDFS is an open-source implementation, so more details can be discovered. Working mechanism of HDFS is similar to that of GFS but it's light-weighted. In GFS, three components are client, master and chunk server, while in HDFS they are client, Name node and Data node, as shown in the Fig. 1.

An HDFS cluster consists of a single Name node, a master server that manages the file system namespace and regulates access to files by clients. In addition, there are a number of Data nodes, usually one per node in the cluster, which manages storage attached to the nodes that they run on. Internally, a file is striped into one or more blocks and these blocks are stored in a set of Data nodes. The Name node executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to Data nodes. The Data nodes are responsible for serving read and write requests from the clients. The Data nodes also perform block creation, deletion, and replication upon instruction from the Name node. HDFS does not support concurrent writes, only one writer is allowed at time.

B. Data Replication

In the cloud storage system, files are usually striped into blocks across multiple data nodes to enable parallel accesses. However, striping decreases file availability. Let say the probability of each data node available is p ($0<p<1$), and file is stripped into n ($n>0$) blocks distributed into different data nodes. The whole file will be available if and only if all the n blocks belonging to this file are available. Then, the probability of the file available is p^n, and obviously, $p^n<p$. From above analysis, we can see that availability degrades due to stripping.

Data replication has been widely used as a mean of increasing the data availability of distributed storage systems in which failures are no longer treated as exceptions [14]. Replica number is a key issue of replication management. To further understand how replica number influences availability and performance, we took experiments on our prototype. The Fig. 2 shows the relationship of availability and replicas number when node failure ratio is 0.2 and 0.1. From results, we observe that availability improves along with the increase of replica number. When replica number reaches a certain point, the file availability is equal to 1, and adding more replicas will not improve the file availability any more. The lower the node failure ratio, the less replica number it requires for file availability to reach 1. Therefore, we can only maintain minimum replicas to ensure a given availability requirement.

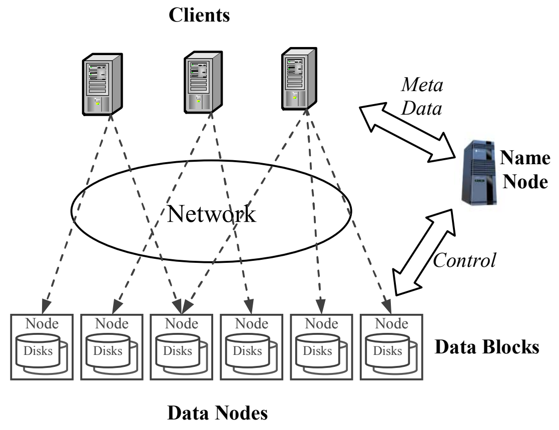


Fig. 1. HDFS Architecture.

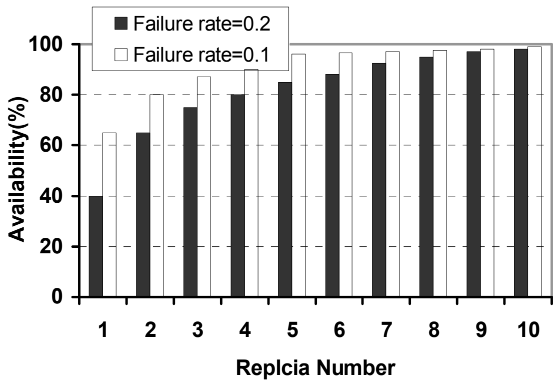
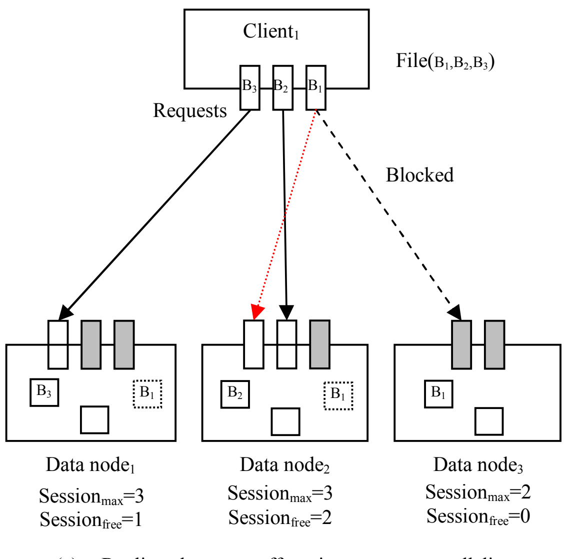
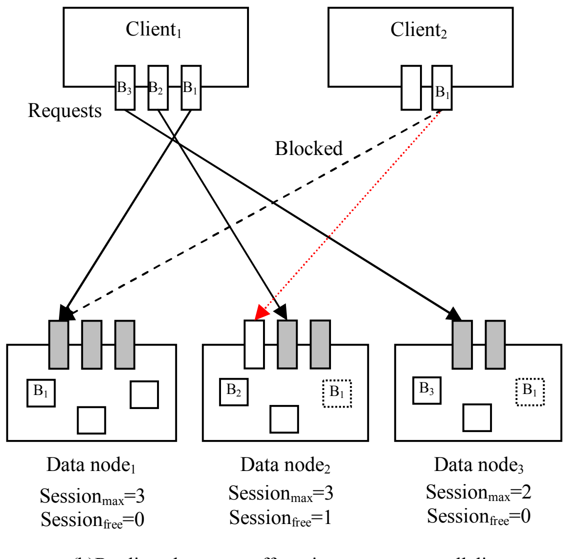


Fig. 2. Availability varies with Replica Number.



(a) Replica placement affects intra-request parallelism



(b)Replica placement affects inter-request parallelism

Fig. 3. Replica placement influences access parallelism and performance.

With replica number increasing, the management cost including storage and network bandwidth will significantly increase. Because hundreds of thousands of blocks may be stored in each data node, even a small increase of replica number can result in a significant increase of management cost in the overall storage system. In a cloud storage system, the network bandwidth resource is very limited and crucial to the overall performance. Too much replicas may not significantly improve availability, but bring unnecessary spending instead.

In view of these issues, we developed a model to express availability as function of replica number. This model is used to determine how much minimal replica should be maintained to satisfy availability requirement, which will be discussed in Section III.

C. Replica Placement

A data node can simultaneously support a limited number of sessions due to capacity constraint. When the number of sessions has reached its upper bound, connection requests from application servers will be blocked, or rejected. In large scale cloud storage system, data nodes are heterogeneous, with different types of disks, network bandwidth, CPU speed, etc., so the maximal number of sessions that a data node can support is different. Blocking probability in each data node may be different due to different workload intensity and capacity [9].

Replica placement influences *intra-request* parallelism. Let us take HDFS as an example. One advantage of HDFS is that file can be transferred between clients and data nodes in parallel. This may not be true if one of *intra-request* sessions is blocked by data node. As shown in the Fig. 3 (a), request for block B_1 can be immediately served if it has been replicated in Data node$_2$ instead of Data node$_3$. Replica placement also influences *inter-request* access parallelism. Suppose two clients request same block B_1, shown as Fig. 3 (b). Two requests can be immediately served if the Data node$_1$ has enough free sessions. Otherwise, the request from client$_2$ will be blocked and delayed. Even if the block B_1 has been replicated in the Data node$_3$, the request still can not be served immediately because it does not have free sessions at that time. If replica of block B_1 is placed in the Data node$_2$ which has free sessions, the requests from client$_2$ can be served immediately, gaining improved access latency.

From above analysis, we can see that replica placement has significant influence on blocking probability and access skew. Efficient replica placement can significantly boost the *inter-request* and *intra-request* parallelism, and overall performance and load balance of the HDFS cluster. These motivate us to propose a novel replica placement policy to efficiently distribute workload across cloud nodes, thus improve blocking probability and access skew.

III. COST-EFFECTIVE DYNAMIC REPLICATION MANAGEMENT

We should now present the cost-effective dynamic replication management scheme for cloud storage cluster. In this paper, without loss of generality, we make the assumption that all data nodes in the system are equally secure. Our proposed model will answer following key questions: how many replicas the system should keep at least to satisfy availability requirement? How to place these replicas efficiently to maximize performance and load balancing?

A. System Model

The cloud storage system is composed of N independent heterogeneous data nodes storing a total of M different blocks $b_1, b_2, \ldots, b_M$, where $M>>N$. We use B to denote the set of blocks in the data node cluster. For every data node S_i, there are M_i blocks saved in it and $B_i=\{b_{i1}, b_{i2}, \ldots, b_{iMi}\}$ is denoted as the set of blocks belonging to S_i. Obviously, $B_i \subset B$ and $B = B_1 \cup B_2 \cup \ldots \cup B_N$ hold. An block b_j is modelled as attribute tuple $b_j=(p_j, s_j, r_j, \tau_j)$, where p_j, s_j, r_j and τ_j are popularity (i.e. block b_j be requested with probability p_j.), size, replica number, and access latency requirement of block b_j respectively. We model data node S_i as $S_j=(\lambda_i, \tau_i, f_i, bw_i)$, where λ_i, τ_i, f_i, and bw_i are request arrive rate, average service time, failure probability, and network bandwidth of data node S_i respectively.

When retrieving a block b_j from data node S_i with performance requirement (i.e. access latency less than τ_j.), bandwidth s_j/τ_j should be assigned to this session to guarantee performance. Obviously, in S_i, the total bandwidth used to serve different requests should be no more than bw_i at any time.

$$bw_i \geq \sum_{j=1}^{c_i} \frac{s_j}{\tau_j} \qquad (1)$$

where c_i is maximal network sessions data node S_j can serve concurrently, and can be calculated from Eq. (1).

B. Availability

Followings are some definitions that will be used in this section.

Node Available: refers to the event of data node reachable, denoted as *NA*, and *P(NA)* is the probability of node available.

Node Unavailable: refers to the event of data node unreachable, denoted as $\overline{NA}$. $P(\overline{NA})$ is the probability of node unavailable, and $P(\overline{NA}) = 1 - P(NA)$.

Block Available: refers to the event of block B_i available, denoted as BA_i, and $P(BA_i)$ is the probability of the block B_i available.

Block Unavailable: refers to the event of block B_i unavailable, denoted as $\overline{BA_i}$. $P(\overline{BA_i})$ is the probability of the block B_i unavailable, and $P(\overline{BA_i}) = 1 - P(BA_i)$.

File Available: refers to the event of file available, denoted as *FA*, and *P(FA)* is the probability of the file available.

File Unavailable: refers to the event of file unavailable, denoted as $\overline{FA}$. $P(\overline{FA})$ is the probability of the file unavailable, and $P(\overline{FA}) = 1 - P(FA)$.

Let say file *F* is stripped into *m* blocks denoted as $\{b_1, b_2, ..., b_m\}$, and distributed into different data nodes. The block b_j with r_j replicas will be obviously unavailable if all the data nodes storing this block are not available. So the probability of block b_j unavailable is

$$P(\overline{BA_i}) = P(\overline{NA_1} \times \overline{NA}_2 \times ... \times \overline{NA}_{r_j})$$

Because the data nodes are independent, we get

$$P(\overline{BA_i}) = P(\overline{NA_1}) \times (P(\overline{NA_2}) \times ... \times P(\overline{NA}_{r_j}) = \prod_{i=1}^{r_j} f_i \qquad (2)$$

To retrieve whole file *F*, we must get all the *m* blocks. Any block unavailable will cause file unavailable. So we have

$$\begin{aligned} P(\overline{FA}) &= P(\overline{BA_1} \cup \overline{BA_2} \cup ... \cup \overline{BA_m}) \\ &= \sum_{j=1}^{m} P(\overline{BA_j}) - \sum_{1 \leq j < k \leq m} P(\overline{BA_j} \cap \overline{BA_k}) + \\ &\quad ... + (-1)^{m-1} P(\overline{BA_1} \cap \overline{BA_2} \cap ... \cap \overline{BA_m}) \end{aligned} \qquad (3)$$

Generally, there are two types of distribution policies: blocks independent and blocks dependent. Here we only consider block independent distribution policy in this paper, where block b_j unavailable does not result in block b_k unavailable. Then, we get

$$P(\overline{BA}_j \cap \overline{BA}_k) = P(\overline{BA}_j) \times P(\overline{BA}_k) \qquad j \neq k$$

Substituting Eqs. (2) into Eq. (3) gives

$$P(\overline{FA}) = \sum_{j=1}^{m} (-1)^{j+1} C_m^j (\prod_{i=1}^{r_j} f_i)^j$$

Hence, the availability of file *F* is

$$P(FA) = 1 - P(\overline{FA}) = 1 - \sum_{j=1}^{m} (-1)^{j+1} C_m^j (\prod_{i=1}^{r_j} f_i)^j$$

Suppose the expected availability for file *F* is A_{expect}, which defined by users. To satisfy the availability requirement for a given file, we get

$$1 - \sum_{j=1}^{m} (-1)^{j+1} C_m^j (\prod_{i=1}^{r_j} f_i)^j \geq A_{\exp ect} \qquad (4)$$

Name node uses Eq. (4) to calculate minimum replica number r_{min} to satisfy expected availability with average data node failure rate. In case of data node failure and current replica number of a block is less than r_{min}, additional replicas will be created into data node cluster.

C. Blocking Probability

After determining how many replicas the system should maintain at least to satisfy availability requirement, we shall explain how to place these replicas efficiently to maximize performance and load balancing in this section.

Blocking probability is used as criterion to place replicas among data nodes to reduce access skew, so as to improve load balance and parallelism. We shall present how to calculate blocking probability of data node.

If block b_j is assigned to data node S_i, some requests will access S_i for this block. Because the block b_j is replicated and assigned to r_j different data nodes (r_j>=1), the probability that a request accesses data node S_i for block b_j is $\frac{p_j}{r_j}$. Then, the probability that a request accesses data node S_i is $\sum_{j=1}^{M_i} \frac{p_j}{r_j}$. We assume that the requests arrive at the data node cluster according to a Poisson process with total arrival rate λ. The request arrival rate served on data node S_i can be given to be

$$\lambda_i = \sum_{j=1}^{M_i} \frac{p_j}{r_j} \lambda.$$

The average service time of data node S_i can be calculated as

$$\tau_i = \frac{1}{M_i}\sum_{j=1}^{M_i}\tau_j \ .$$

After admitting a request, data node S_i serves it for an average duration τ_i. When data node S_i is serving c_i sessions, it does not have free channel to serve any additional requests and so any request accessing this data node must be rejected or blocked. Thus, data node S_i can be modelled as $M/G/c_i$ system with arrival rate λ_i and service rate $1/\tau_i$ [9], and accordingly, the blocking probability of data nide S_i can be given to be

$$BP_i = \frac{(\lambda_i\tau_i)^{c_i}}{c_i!}\left[\sum_{k=0}^{c_i}\frac{(\lambda_i\tau_i)^k}{k!}\right]^{-1} \qquad (5)$$

From Eq. (5), we can see that blocking probability BP_i captures the workload intensity and performance of data node S_i by the value of λ_i and τ_i, respectively. So, blocking probability is good criterion to be used for replica placement for load balance.

Data node S_i leverages its computational ability to record history access information of each block within a sliding time window T and calculate the average value of λ_i and τ_i. Then, blocking probability BP_i can be calculated with input of λ_i and τ_i periodically. Our replica placement policy works like that replica will be placed into data node with lowest blocking probability to dynamically maintain overall load balancing.

Because hundreds of data nodes might coexist on a single cluster, sorting and searching blocking probability of all data nodes is challenging. An attractive choice for managing index and list structures is by using B+tree implementation because it maintains records in sorted order and scale efficiently in both time and space. In this paper, Name node uses B+tree to sort data nodes in descending order using their blocking probability BP_i as the key, as shown in the Fig. 4. When we want to find a candidate data node to place replica, the Name node quickly searches the B+tree and return a data node with lowest blocking probability. With this mechanism, the search is very quick and computation workload is extremely low. Blocking probability of each data node is calculated locally in data node side and updated to the Name node periodically, which can reduce Name node management workload. With updated value of blocking probability, Name node quickly rebuilds the B+tree and makes decision of replica placement.

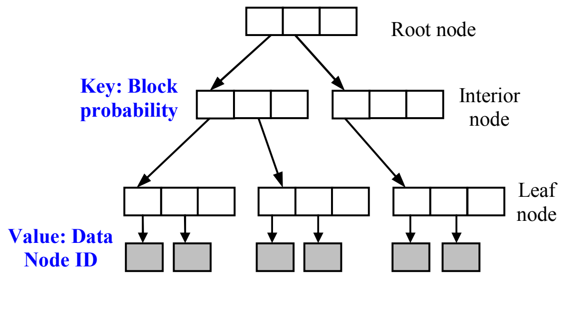


Fig. 4. Data node B+Tree.

IV. Implementation in HDFS

Besides default replication management, HDFS provides flexible API for us to extend and implement efficient replication management. The replication factor is configurable per file in HDFS. An application can specify the number of replicas of a file. The replication factor can be specified at file creation time and can be changed later. However, HDFS does not provide policy to determine the replication factor. Our algorithm can be used to determine minimal replica number for a given availability requirement. HDFS also provides interface for data migration among Datanodes to balance workload. Therefore, HDFS is good platform to implement and evaluate our cost-effective dynamic replication management scheme.

To implement proposed CDRM in HDFS, we use replication factor to denote minimal replica number in the metadata of Name node. The Fig. 5 shows the framework of CDRM in HDFS. Since blocks number is required for CDRM to calculate minimal replica number for a given availability requirement, we modify HDFS client to cache the whole file data into a temporal local file instead of first block. It is reasonable to buffer whole middle size file in local file, whose size ranges from hundreds Megabyte to Gigabyte. This simplifies our prototype implementation in HDFS, but may not be good for very large file whose size is Terabyte. Finding a best way to efficiently integrate CDRM into HDFS is an interesting issue for future work.

After whole data is written into local file, the client contacts the NameNode with availability setting and block number. The NameNode inserts the file name into the file system hierarchy, calculates the replication factor based on Equation (4) and quickly searches the DataNode B+tree to obtain a list of DataNodes for each data block. The NameNode responds to the client request with a list of DataNodes for each block, the destination data blocks and replication factor. Then the client flushes each block of data from the local temporary file to the specified DataNode and its replicas to selected DataNodes in pipelining way.

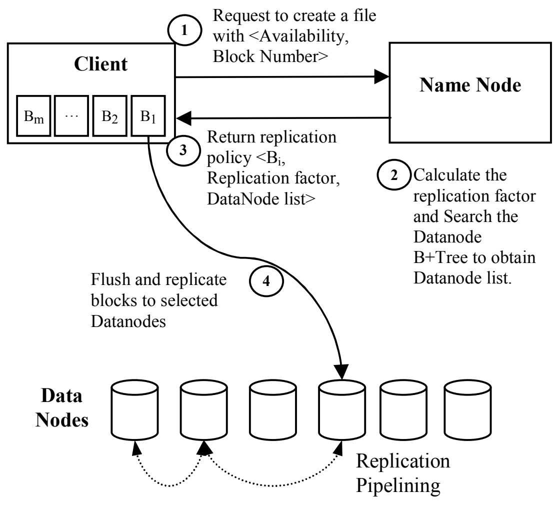

1 Request to create a file with <Availability, Block Number>
Client
Name Node
B_m … B_2 B_1
3 Return replication policy <B_i, Replication factor, DataNode list>
2 Calculate the replication factor and Search the Datanode B+Tree to obtain Datanode list.
4 Flush and replicate blocks to selected Datanodes
Data Nodes
Replication Pipelining

Fig. 5. Framework of cost-effective dynamic replication management.

A. *Control Strategies*

In real application, workloads may change frequently. To adaptively satisfy availability and load balance according to dynamic environment in terms of node failure and access pattern changes, we develop a *Dynamic Replica Control Strategy* (referred to as DRCS) running on Name node, as illustrated in Fig.6. By monitoring the workload changes and evaluating the efficiency of control in current time window T_i, DRCS dynamically adjusts replica number and replica location in next time window T_{i+1} with updated parameters input from Data nodes. DRCS not only maintain reasonable replica number in the system, but also dynamically adjust replica location according to workload changes.

In case of data node unreachable and current replica number less than minimal replica number r_{min}, a new replica will be added to a data node with lowest blocking probability to guarantee the availability requirement.

In case of accesses to a popular block exceeding *threshold for migration* (th_{mig}) and access skew happening, replica migration is firstly used to re-distribute workloads of popular block from hot data node to cool data node. If the access latency requirement of this popular block still can not be satisfied, a new replica will be added to share the workloads. Consequently, system will maintain more replicas for popular blocks, while less replicas for unpopular blocks. Several previous studies [10] claimed that the most popular files are typically small in size, while the large files are relatively unpopular. Then, migration and replication cost of popular blocks is relatively smaller than that of unpopular blocks.

```
Input: $A_{expect}, m, f_i, th_{del}, th_{add}, th_{mig}$

1.  for each time window $T$ do
2.    for each file
3.      if availability reset by user
4.        Calculate $r_{min}$ based on Eq.(2)
5.      end if
6.    end for
7.    New $BP_i$ reported by Datanodes
8.    Reorder data node B+tree using $BP_i$ as key
9.    for each block
10.     $r_j$=current replica number of block$_j$
11.     $a_i$ =Access frequency of block$_j$
12.     if $r<r_{min}$
13.       Add ($r_{min}-r$) replicas to data nodes with lowest $BP_i$
14.     else
15.       if $a_i< th_{del}$
16.         Delete a replica from data node with highset $BP_i$
17.       else if $th_{mig}<a_i< th_{add}$
18.         Migrate replica from hot data node to cool node
19.       else if $a_i>th_{add}$
20.         Add one replica to data node with lowest $BP_i$
21.       end if
22.     end if
23.   end for
24. end for
```

Fig. 6. Dynamic Replica Control Strategy running on NameNode.

Access to a block will not always keep high. Once the average access frequency for the block drops down to *threshold for deletion* (th_{del}) and current replica number is more than r_{min}, the extra replicas will be marked as invalid gradually. All invalid replicas will not be involved in consistency update. Once the access frequency rebounds, we will validate the invalidated replica if its version is enough new. The other invalid replicas will be eventually overwritten according to local storage policy. The data node with highest blocking probability will be selected to delete replica.

V. EVALUATION

Our test platform is built on a cluster with one name node and twenty data nodes of commodity computer. Each node has an Intel Pentium 4 CPU of 2.8GHz, 1GB or 2GB memory, 80GB or 160GB SATA disk. The operating system is Red Hat AS4.4 with kernel 2.6.20. Hadoop version is 0.16.1 and java version is 1.6.0.

We developed a benchmark for evaluation. We used the AUSPEX file system trace [18] as the workload for our evaluation. The trace follows the NFS activity of 237 clients serviced by an AUSPEX file server over the period of 1 week, and was gathered by snooping Ethernet packets on four subnets in University of California, Berkeley. The traces are used by the synthesizer, which is developed to create workloads with different characteristics, such as data sets of different sizes, varying data rates, and different popularities. These characteristics reflect the differences among various workloads to the cloud storage cluster.

To change request arrival rate, the synthesizer reduces or enlarges the time interval between any two consecutive accesses. The sizes of the data sets are enlarged by replacing one access in the traces by multiple accesses. Data popularity reflects how uniformly different files are accessed. We define the popularity as the ratio between the size of the most popular data receiving 90% of total accesses and the size of the total data set. A large ratio indicates sparse popularity: the data are more uniformly accessed. A small ratio means dense popularity: accesses concentrate in a small amount of data. To obtain denser popularity, we vary the accesses in the original traces by replacing the accesses to less popular pages with the accesses to more popular pages.

A. *Availability*

One way to check the accuracy of our availability model Eq. (4) is to fix the number of replicas existent in the system at any time and measure file availability. In this experiment, file is stripped into 3 blocks. The Fig. 7 compares the numbers of replicas corresponding to expected availability. For example, for Data node failure probability of 0.1 and a required availability of 0.8, minimal replica number is 2 based on calculation of the Eq. (4). When the system maintains the number of replicas per block at 2, we measure the average availability per file to be around 0.93. Though the model predictions are not this accurate at all points in the graph, this validates the general trend pointed to by the model.

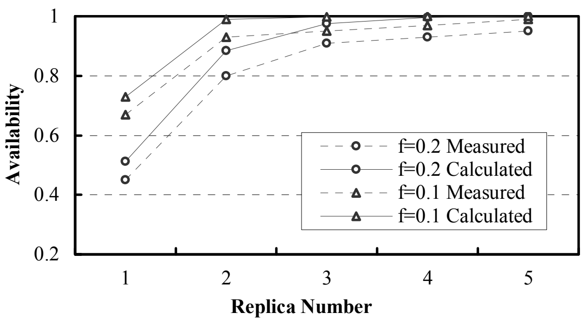


Fig.7. Accuracy comparison of availability model.

To evaluate the advantage of CDRM as a dynamic algorithm, we measured the replica number as experiment running. Result is plotted in the Fig.8. We set A_{expect} as 0.8 and one replica per block initially. From the results, we can see that replica number increases at beginning of the experiments. After reaching a certain point, the number of replicas is maintained at a constant level during experiments. This indicates that CDRM dynamically adds more replicas if current replica number does not satisfy availability requirement, while only maintains minimal replicas to save cost.

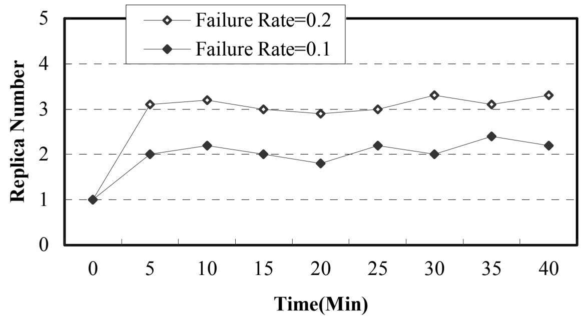


Fig.8. Dynamic replication with Data node failure rate of 0.1 and 0.2, A_{expect}=0.8.

B. Performance

In order to investigate performance of proposed CDRM under different workload distribution, we change the popularity from 10% to 100%. We compared average access latency of proposed CDRM and default replication management used in HDFS (called HDRM) under different popularity distribution and access arrival rate. HDRM statically maintains configured number of replicas among system and adopts rack-aware replica placement policy [3]. When replication factor is three, HDRM places two replicas in the same rack and one replica in a different rack. However, Datanode in the same rack is randomly selected without considering performance and workload difference among Datanodes. We set A_{expect}=0.8, and average failure rate is 0.1 and system maintains 2 replicas per block initially. The test result is plotted in the Fig. 9.

From result, we can observe that as access popularity increases, the access latency is reduced for both CDRM and HDRM. But, performance of CDRM is much better than that of HDRM when popularity is small. This is because the CDRM can dynamically redistribute workload among data nodes by increasing replica number for popular blocks and adjusting replica location according to workload changing and data node capacity. From result, we also can find that when we increase access arrival rate from 0.2 to 0.6, CDRM performs better than HDRM. This indicates that CDRM can evenly distribute heavy workloads among data node cluster and utilize resource of whole cluster to achieve overall performance improvement.

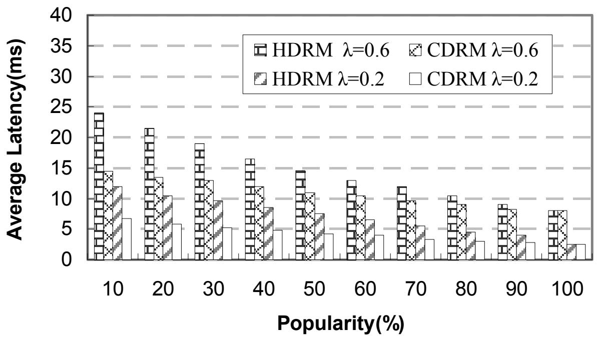


Fig.9. Effect of popularity and access arrival rate, 20 data nodes.

C. Load Balance

Replica placement among data nodes has significant impact on load balance, thus system performance. In this test, we compared system utilization rate of each data node $\rho_i = \lambda_i \tau_i$ of CDRM and HDRM under popularity=10%. HDRM utilizes rack-aware replica placement policy, which randomly places two replicas in the same rack and another replica in the different rack. We set A_{expect} as 0.8, and average failure rate is 0.1 and system maintains 3 replicas per block initially. The test result is plotted in the Fig. 10. The figure shows the difference of system utilization of each data node comparing to the average system utilization of the cluster.

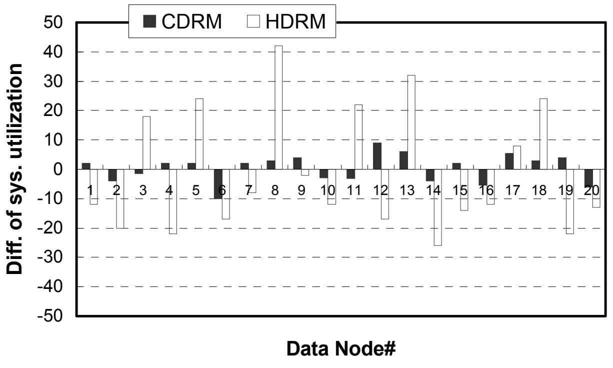


Fig.10. System utilization among data nodes, popularity=10%, λ=0.6.

From the result, we can see that with HDRM mechanism, data node #14 received the least system utilization with a difference of about -26%, whereas data node #8 received the most loads with +42% above means. In contrast, with proposed CDRM mechanism, data node #6 received the least number of workload with a difference of about -10% and data node #12 received the most loads with only +9% above means. This is because CDRM evenly distributes imbalanced workload to whole cluster by dynamically replicating and migrating, while HDRM can not dynamically conduct on-line replication and suffer imbalance issues.

The above experimental results demonstrate that proposed CDRM is very efficient to boost system performance and improve availability and load balance.

VI. RELATED WORK

There have been a number of research efforts in recent years to address the replication management issues in large-scale storage system. Google file system treats failures as common events. In the Google file system, a single master makes decisions on data chunk placement and replications. A data chuck is re-replicated when the number of its replicas falls below a limit specified by users. In Ceph[13], files are striped across many blocks, grouped into placement groups (PGs), and distributed to Object-based Storage Devices[8] via CRUSH[20], a specialized replica placement function. CRUSH maps objects to storage devices without relying on a central directory. However, both of Google file system and Ceph file system just maintain fixed number of replicas and can not dynamically distribute workloads by replicating and migrating replicas among storage cluster.

HDFS[3] stores each file as a sequence of blocks. The blocks of a file are replicated for fault tolerance. The block size and replication factor are configurable per file. An application can specify the number of replicas of a file. The replication factor can be specified at file creation time and can be changed later. The NameNode makes all decisions regarding replication of blocks. It periodically receives a Heartbeat and a Blockreport from each of the DataNodes in the cluster. HDFS adopts rack-aware replica placement policy, which is a work in progress. For the common case, when the replication factor is three, HDFS puts one replica on one node in the local rack, another on a different node in the local rack, and the last on a different node in a different rack. However, Datanode in the same rack is randomly selected for replica placement. The purpose of a rack-aware replica placement policy is to cut the inter-rack write traffic which generally improves write performance.

K. Ranganathan proposed a dynamic model-driven replication scheme, in which each peer in the P2P storage system runs a model to determine how many replicas of a file are needed to maintain desired availability [19]. This scheme can ensure file availability in dynamic P2P environment, but it does not consider access efficiency and management costs.

Data Grid manages large numbers of read-only scientific data and provides data sharing for globally distributed user communities. To save latency and bandwidth, K.Ranganathan and I.Foster presented dynamic replication strategies for Data Grid in [15], by applying different replication strategies for different kinds of access patterns. The most common distributed storage systems, such as Napster [16] and Freenet [17], dynamically manage replicas based on file popularity degree, in which once access frequency to a file exceeds a threshold, a new replica of the file will be created. However, these works did not consider replica placement.

This paper differs from the above mentioned studies in a number of ways. First, CDRM satisfies availability requirement at low cost by dynamically maintaining minimum replicas among the system. A second major feature of this study is that we dynamically place replicas to distribute workload across cluster according to data node capacity and activity intensities. It also should be noticed that besides HDFS, our approach can also be applied to other distributed file systems such as PVFS [5], pNFS [6], Gpfs[7] and LusterFS [11].

VII. CONCLUSION

In this paper, we design a cost-effective dynamic replication management scheme for large-scale cloud storage system refereed to as CDRM. We first build up a model to capture the relationship between availability and replica number. Based on this model, lower bound on replica reference number to satisfy availability requirement can be determined. CDRM further places replicas among cloud nodes to minimize blocking probability, so as to improve load balance and overall performance. We implemented CDRM in HDFS and experiments demonstrate that CDRM can adapt to the changes of environment in terms of data node failure and workload changes and maintains a rational number of replica, which not only satisfies availability, but also improves access latency, load balance, and keeps the whole storage system stable.

ACKNOWLEDGMENT

This work was partially supported by the National High Technology Research and Development Program ("863"Program) of China under Grant No.2009AA01A401, 2009AA01A402, Changjiang innovative group of Education of China No. IRT0725.

REFERENCES

[1] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung, The Google File System, *Proceedings of 19th ACM Symposium on Operating Systems Principles(*SOSP 2003), New York, USA, October, 2003.

[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. pages 137 – 150, December 2004.

[3] The Apache Software Foundation. Hadoop. http://hadoop.apache.org/core/, 2009.

[4] Amazon-S3. Amazon simple storage service (amazon s3). http://www.amazon.com/s, 2009.

[5] Philip H. Carns, Walter B. Ligon, III, Robert B. Ross, and Rajeev Thakur. Pvfs: A parallel file system for linux clusters. *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317 – 327. USENIX.

[6] Peter Honeyman, Dean Hildebrand, Dean Hildebrand, Lee Ward, and Lee Ward. Large files, small writes, and pnfs. *Proceedings of the 20th*

ACM International Conference on Supercomputing, pages 116-124, 2006.

[7] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. *Proceedings of the 2002 Conference on File and Storage Technologies* (FAST), pages 231 – 244, 2002.

[8] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. SIGPLAN Not., 33(11):92–103, 1998.

[9] Dan Feng, Lingjun Qin. Adaptive Object Placement in Object-Based Storage Systems with Minimal Blocking Probability. *Proceeding of the 20th international conference on Advanced Information Networking and Applications* (AINA'06), 2006.

[10] C. Cunha, A. Bestavros, and M. Crovella, Characteristics of WWW Client-Based Traces, Technical Report 1995-010, Boston Univ., 1995.

[11] Lustre: A Scalable, High-performance File System. Whitepaper, Cluster File System, Inc. http://www.lustre.org/docs/lustre.pdf.

[12] Hai Huang, Wanda Hung, Kang G. Shin: FS2: dynamic data replication in free disk space for improving disk performance and energy consumption. *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (SOSP 2005), Brighton, UK, October, 2005.

[13] Sage A. Weil Scott A. Brandt Ethan L. Miller Darrell D. E. Long, Ceph: A Scalable, High-Performance Distributed File System, *Proceeding of 7th conference on operating system design and implementation* (OSDI'06), November, 2006.

[14] Thanasis Loukopoulos, Ishfaq Ahmad and Dimitris Papadias, An Overview of Data Replication on the Internet, *Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN.02).*

[15] K. Ranganathan and I. Foster, Identifying Dynamic Replication Strategies for a High Performance Data Grid, presented at *International Workshop on Grid Computing, Denver, CO, 2001.*

[16] Napster project home page. http://ww.napster.com.

[17] FreeNet home page. http://freenet.sourceforge.net.

[18] http://now.cs.berkeley.edu/Xfs/AuspexTraces/auspex.html

[19] K.Ranganathan, A.Iamnitchi and I.Foster, Improving Data Availability through Dynamic Model-Driven Replication in Large Peer-to-Peer Communities, *Proceedings of the Workshop on Global and P2P Computing on Large Scale Distributed Systems*, Berlin, May 2002.

[20] Sage Weil, Scott A. Brandt, Ethan L. Miller, Carlos Maltzahn, CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data, *Proceedings of SC '06*, November 2006.