An Adaptive Storage and Retrieval Mechanism to Reduce Response-Time in High Performance Computing Clusters

Amir Saman Memaripour, Ehsan Mousavi Khaneghah, Seyedeh Leili Mirtaheri, and Mohsen Sharifi
School of Computer Engineering, Iran University of Science and Technology, Tehran, Iran

Abstract - *Ever-increasing growth of high performance computing applications requires employment of novel methods in all aspects of computing systems. The response time of file storage and retrieval operations is one of the most important factors of storage systems and improving that will result higher computational power. Consequently, breathtaking efforts have been done and various file systems with different architectures have been proposed. Most of them are not aware of clusters' execution state and do not consider variety of I/O operations' response time on machines with different storage media, network traffic, and processing load. In this paper, we have proposed a mechanism to store and retrieve files with respect to the execution state of storage nodes and network topology of the cluster. Finally, the proposed architecture has been implemented and evaluated using Hadoop distributed file system.*

Keywords: File System, Response Time, Adaptive Storage and Retrieval, HPC

1 Introduction

Exponential growth of the volume of stored data has made conventional methods of storage and retrieval inefficient which require new ways with higher performance to deal with [1]. Performed estimations show that the size of electronic data was about 1.8 zeta bytes in 2011 that illustrates the need to store tremendous amounts of data [2]. Furthermore, improving the performance of these operations in various aspects such as scalability and reliability must be considered [3]. In this way, three different approaches including improving the performance of each computational element, revising the algorithms, and making use of a series of ordinary computing elements could be followed [4].

In the field of HPC computing, improving overall performance is mainly focused on reducing response time and increasing processing power of the system [4]. In order to fit in with the requirements of these environments and avoid becoming the bottleneck, file system designers have used several techniques to reduce the response time [5] [6]. Numerous file systems, such as HDFS and PVFS, have been proposed to cope with these issues and used novel algorithms and load balancing techniques to improve the overall system performance with the aid of distributed computing [7] [8].

Heterogeneous storage clusters are made of various computational elements with different performance levels. This diversity leads to different response times of computational nodes to a single request. Moreover, the response time of each node depends on other parameters such as processor load, network traffic, and remaining storage space changing during time and will affect the performance of the node [9]. Network topology and the distance between data consumer and data provider is another important parameter that can affect the performance of I/O operations. In many HPC clusters, applications and their respective data are placed as closest as possible and data access is performed using a high-bandwidth and low-latency communication network. However, communication cost between two adjacent nodes is far less than two nodes which are located in different parts of the communication network, i.e. two different racks [7].

In these clusters, data replication is performed in order to improve reliability and response time [7]. If the number and the place of replicas are determined with respect to the execution state of computational nodes, the response time and scalability of the cluster will be improved. In addition, the load will be distributed among the nodes and data will not aggregate in particular parts of the cluster. In this manner, an adaptive mechanism that replicates and distributes replicas in the cluster according to the execution state of cluster's nodes has been proposed in this paper.

2 Related works

Various usages of file systems are possible in computing systems. For example, resources could be represented, accessed, and managed by files in distributed computing systems [10]. Moreover, files could be used to store, retrieve, name, structure, protect, and manage user and system data. Our proposed mechanism considers file systems as the module which is responsible for the mentioned responsibilities in the second definition.

Remarkable efforts have been done in the field of file systems which can be categorized in three classes including centralized, decentralized, and distributed file systems. Traditional file systems like FAT belong to the first class and are inefficient for high performance applications [11]. Extremely low scalability and the limitations of storage media like disks' seek time lead to the appearance of the other two classes. Decentralized file systems were mainly proposed to increase the number of master nodes in file systems and avoid consequences of having a single point of failure. NFS is one of the centralized file systems that a decentralized version of it has been proposed to increase its scalability [12].

Computation requirements of today's applications need a level of scalability which is far from what decentralized approaches can reach. As a result, numerous file systems with distributed architectures have been presented including HDFS and GFS which are used to maintain organizational data at

Yahoo and Google respectively. These two are also good examples of the successful experiences in increasing scalability [7] [13].

PVFS was introduced for parallel storage and retrieval of files in Linux clusters and parallel execution of applications which were previously performed using parallel machines. This file system was designed to support different file system interfaces like POSIX, facilitate installation and use of the file system, and provide features to store and retrieve files in a parallel and concurrent manner by several processes. Moreover, scalability and robustness were among other goals in the design process of this file system [8]. PVFS has been developed as a base for research and development in the field of parallel file systems in Linux [1] [5] [14]. Furthermore, a branch of PVFS called OrangeFS has been developed to consider issues like metadata operations, blocks replication, and access control in more depth [15].

In 2003, Google presented GFS which had been used before that time by Google's applications and designed according to their storage requirements. In addition to different aspects of performance which were addressed in other distributed file systems, GFS has paid special attention to efficient execution of its applications [13]. GFS do not support POSIX standard, has larger basic storage blocks comparing to traditional file systems, and is mainly focused on the optimization of append operations because of the nature of its applications [13] [16]. Moreover, using commodity and cheap hardware in the implementation process of GFS made hardware failure such as network devices and storage media a probable event rather than unexpected. In order to prevent the consequences of these failures, several innovative methods have been considered in the design of GFS.

Ceph is another distributed file system which is proposed to provide a scalable, reliable, and high performance storage cluster [17]. It separates the management of data and metadata and supports more than 250,000 metadata operations in every second by using a pseudo-random distribution function, called CRUSH, instead of allocation tables [18]. Moreover, it distributes the replication, fault detection, and recovery operations among storage nodes, where data and metadata are stored. Ceph is composed of three major parts including client, storage cluster, and metadata cluster. It also provides a near-POSIX file system interface which is a remarkable feature for a distributed file system.

Another project called HDFS is currently in progress in Yahoo that is similar to GFS in various aspects and is published as an open-source file system [7]. HDFS performs as a reliable and high performance storage cluster and used several concepts proposed in GFS. More detailed information about this file system is available in section 3, where implementation platform of our proposed mechanism has been introduced.

Common file systems in this area have not effectively considered execution state of the cluster and computational nodes. In addition, heterogeneity of storage media is another parameter that affects response time and has not properly addressed in these file systems. Our proposed mechanism, discussed in more detail in section 4, has considered these parameters and introduced a method to improve the response time of storage and retrieval operations.

3 Implementation platform

In order to implement and evaluate our proposed mechanism, we have used Hadoop Distributed File System as our implementation platform. HDFS is designed to reliably store large data sets and stream these data to user applications at high bandwidth. Distributing stored data and computations among several processing and storage nodes in HDFS leads to a scalable and economic cluster that can grow to any size on demand [7]. This file system has being used by Yahoo to manage 25 petabytes of application data and has been successful in this way [2]. We have stated our motivations to use it as our implementation platform in section 3.2 and after a brief introduction to HDFS.

3.1 Overview

Every HDFS cluster at least consists of a NameNode and one or more DataNodes. The cluster can grow to any size by adding more DataNodes to it [19]. Moreover, other nodes such as BackupNode and CheckpointNode can join the cluster to improve its reliability [7]. User applications, usually executed on DataNodes, are other pieces of the architecture and interact with HDFS using a programming library. This file system does not support POSIX standard and its interfaces are provided via HDFS client which is a code library [2].

In this architecture, NameNode maintains and manages namespace. Moreover, it manages DataNodes and replicas of each data block. The namespace is a hierarchical structure of directories and files which are represented by inodes. Each file is divided into large data blocks which could be replicated independently and stored on various DataNodes. Besides maintaining the file system namespace, NameNode stores required information to perform the mapping between data blocks and DataNodes [7].

DataNodes are responsible for maintaining the replicas of data blocks in their local storage media and store two files for each replica. One of the files is used to store the data and the other maintains block's metadata. Each DataNode performs a handshake with the NameNode at the startup. During this operation, namespace identifier and software version of the NameNode are compared with respective values of the DataNode. Consequently, the consistency of the file system will be preserved [7]. DataNodes send block reports to the NameNode after joining the cluster and periodically to inform it about their hosted replicas. Moreover, the NameNode receives another kind of control messages from DataNodes that indicates their presence in the cluster and accessibility to their hosted replicas [2].

3.2 Motivations in using HDFS

Availability of user manuals and books besides its open source implementation made it feasible to use HDFS as the implementation platform [2]. In addition, it is widely used in extraordinary computing environments and has the potential to be applied on high performance computing clusters.

Performing the replication under the supervision of the NameNode provides required information for adaptive distribution of replicas over the cluster and makes it a good choice to run the proposed mechanism in this node. Moreover, the execution state of computational nodes can be send along with control messages which are periodically sent by DataNodes to the NameNode.

The mechanism used by HDFS to provide user applications with the block replicas list can be changed to consider the execution state of DataNodes and perform adaptively. HDFS has also provided an interface to consider the network distance between data providers and consumers in order to sort hosts in the replica lists [7]. Furthermore, this file system is designed to use commodity hardware that makes evaluation feasible. HDFS is implemented using JAVA programming language that facilitates source code modification and makes it possible to run it on a wide variety of operating systems [2]. Considering these facts, we have chosen HDFS as our implementation platform in order to evaluate our proposed mechanism.

4 Proposed mechanism

The goal of our proposed mechanism is to consider the execution state of DataNodes and the topology of communication network in storing and retrieving data blocks. Adapting file system operations to the execution state of the cluster will improve the overall performance and distribute the I/O load across the storage cluster. Moreover, selecting least loaded nodes to host replicas can improve the response time of I/O operations.

The execution state of DataNodes including remaining storage space, available network bandwidth, and I/O speed of storage media and where it is placed in the communication network are considered with regard to selecting an appropriate node. Besides adapting I/O operations to execution state of nodes, our proposed mechanism introduces a method to automatically detect the network topology that will help data blocks distribution in the storage cluster.

4.1 Architecture

Adapting reading and writing data blocks to execution state of DataNodes and the network topology is performed by three modules. One of these modules concerned about the network topology and is responsible for maintaining the Network Tree which is used by the other modules. Construction of the Network Tree is performed according to relative distance between computational nodes and communication switches. For instance, direct connection of two nodes to a switch means both of them belong to a same rack. In the same way, relative distance between different racks is estimated using the communication delay between their computational nodes. Section 4.3 contains detailed information about this module.

Adaptive generation of the list of DataNodes hosting replicas of a data block is performed by another module which is described in more detail in section 4.2. This module estimates transfer time of the replica according to the last known load of the DataNodes and the network distance between each DataNode and the client.

The other module of the proposed mechanism selects destination DataNodes to host replicas of each data block. In order to select more appropriate nodes, a priority value is assigned to each DataNode and is continually being updated based on its workload. Using these priority values and the network distance between DataNodes and the client, this module will choose the best node to host the replicas. More information about this module has been stated in section 4.4.

4.2 Adaptive Generation of Replica-Hosts List

Once the NameNode receives a request from a client for reading a data block, it provides the client with a list of DataNodes hosting replicas of the requested data block. HDFS only considers the network distance between the client and replica hosts in construction of this list and ignores the execution state of DataNodes [7]. Our proposed mechanism estimates the time that C_i will spend for reading B_k from D_j and orders the replica-hosts list by these estimated values. As a result, the overhead of read operations will distribute across DataNodes that will improve the overall response time of the cluster. Moreover, some required computations of this mechanism are performed by DataNodes that avoids addition of a considerable overhead to the NameNode.

$$EstReadTime(C_i, D_j, B_k) = size(B_k) \times ReadLatency(D_j) + NetDist(C_i, D_j) \quad (1)$$

Equation 1 is used to estimate the time required for reading a data block called B_k from D_j by the client that is running on C_i in seconds. The value of $size(B_k)$ is stored on the NameNode and can be fetched by one memory access. Furthermore, required computations for $ReadLatency(D_j)$, the amount of time in seconds to read one megabyte of data from D_j, is performed by the respective DataNode and the result is sent to the NameNode via heartbeat messages. Equation 2 shows how this value is calculated. The network distance between C_i and D_j is estimated based on the network topology using $NetDist(C_i, D_j)$. Section 4.3 contains detailed information about this function and the way it detects the network topology.

$$ReadLatency(D_j) = \frac{1}{ReadSpeed(D_j)} + \frac{1}{(1 - NetUt(D_j)) \times netSpeed(D_j)} \quad (2)$$

In order to estimate $ReadLatency(D_j)$ in Eq. 2, we need to be aware of the execution state of D_j including the transfer speed and the utilization of its NIC besides the I/O speed of its storage media. The momentary I/O speed of the storage media can be estimated based on the current data streams of the DataNode. In order to rate the maximum I/O speed of the storage media, a shell script will be executed at each startup of the DataNode. Using this information, Eq. 3 estimates $ReadSpeed(D_j)$ in megabytes by calculating a weighted-average of the capacity of all local storages in D_j. Similarly, network transfer speed is evaluated by multiplication of network utilization and total transfer speed of the NIC in MB/s. In this manner, the total transfer speed of each network

interface is determined by parsing the respective output of *ethtool* in Linux. Finally, the addition of inverse of these values represents the time required for reading one megabyte of data from D_j by any other node in the cluster.

$$ReadSpeed(D_j) = \frac{\sum_{i=1}^{n} UsedSpace(S_i) \times (TotalSpeed(S_i) - UsedSpeed(S_i))}{\sum_{i=1}^{n} UsedSpace(S_i)} \forall S_i \in D_j \quad (3)$$

4.3 Dynamic network topology detection

In order to estimate the network distance between DataNodes, an n-ary tree called Network Tree is used. The interior nodes of this tree represent communication switches, while its leaves contain DataNodes of each rack. The branching factor of this tree is the maximum number of ports of all communication switches and is denoted by *BF*. As a result, the height of this tree can be calculated by $h=\lceil \log_{BF} n \rceil$ for some positive *n* that reflects the number of racks in the cluster. Moreover, every leaf of the Network Tree is assigned with a unique identifier called RackID which is used for indexing all the DataNodes belonging to a same rack [20].

Once a DataNode joins the cluster, it is considered to be a member of a new rack and is added to the Network Tree as the immediate successor of the root. After that, the Network Tree and the indexed list of DataNodes will be sent to the newly joined DataNode to find its place in the network. Fig. 1 contains the pseudo-code of the search algorithm a DataNode follows to find its place in the network along with the structure of the Network Tree. The result of the performed search in the newly joined DataNode will be sent to the NameNode in order to update the Network Tree.

INPUT: N_c = current DataNode
INPUT: p = the root of NetworkTree
OUTPUT: R_{id} = null
while (R_{id} == null) do
  $last\_RTT$ = 0
  for each $node_i$ in childs(p)
    N_D = select one of the descendent DataNodes of $node_i$ in a random manner
    new_RTT = eval_RTT(N_c, N_D)
    if ($new\_RTT < last\_RTT$) {
    if (childs($node_i$) == null) $R_{id} = node_i$
    $p = switch_i$
    break
    }
  $last\_RTT = new\_RTT$

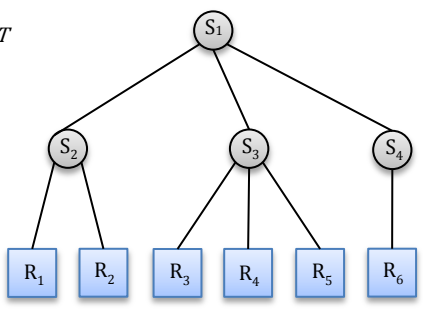

Figure 1: The structure of Network Tree and pseudo-code of its search algorithm

By identifying the nearest common predecessor of two DataNodes in the Network Tree and counting the number of interior nodes in their communication path, the NameNode can estimate the communication latency between every two DataNodes in the cluster. The NameNode assumes a constant communication delay for each switch in the communication path in order to estimate the total communication latency. The running time of this algorithm is $O(\log^{BF}{}_n)$. Considering the large values of *BF*, its overhead could not be remarkable. As an example, assuming *n*=350 and *BF*=16, the height of the Network Tree would be 3 that supports our point.

4.4 Adaptive selection of replication hosts

Once a new data block is created or an existing one is going to be replicated, the NameNode should select some DataNodes to host the newly created replicas. This selection can be performed based on various considerations such as reducing replication time, improving data distribution, reducing the distance between data provider and consumer, and hosting replicas in the nearest racks. Our proposed mechanism assumes that reader processes are uniformly distributed across the cluster, so it tries to select the more appropriate DataNodes from the nearest racks to the data source. In addition, it obeys the replication rules of HDFS and stores a data block in a rack that contains at most one of its replicas. At the final step of selection, qualified DataNodes are ordered by their remaining storage space and available network bandwidth using Eq. 4. After that, the most proper nodes will be chosen to host the recently created replicas.

$$NodeRate(D_j) = Min\left(\alpha \times \frac{FreeSpace(D_j)}{maxStorageSize}, \beta \times \frac{AvailBW(D_j)}{maxNetBW}\right) \forall D_j \in Cluster \quad (4)$$

The selection priority of every DataNode is determined by comparing its free storage space and available network bandwidth with the total storage space and maximum network bandwidth of the cluster. *FreeSpace*(D_j), *AvailBW*(D_j), *maxStorageSize* and *maxNetBW* are calculated using the information which is being periodically received from DataNodes. Moreover, the effect of free storage media and available network bandwidth on *NodeRate*(D_j) can be adjusted using α and β constants. Here, we have used α=1 and β=1 for the sake of simplicity.

The respective statistics and information of all DataNodes of each rack is stored in a max-heap according to their selection priority value [20]. These max-heaps are referenced by the leaves of the Network Tree and are accessible in this way. As a result, selection of the most proper DataNode of a rack can be performed in O(1). Fig. 2 shows the pseudo-code of the algorithm for selecting the DataNodes to host the recently created replicas of the data block hosted by *Di*.

Once a new data block is created in a DataNode, the first replica is stored in the same node. The other two DataNodes to store replicas are selected using the algorithm shown in Fig. 2 and a pipeline will be organized by HDFS to transfer replicas to the selected hosts. By default, this algorithm stores the replicas in the nearest racks to the data source. However, other policies to select replication hosts could be used in order

to improve reliability of the cluster. For example, storing a replica in another part of the cluster can keep the data block accessible on the event of communication switch failures.

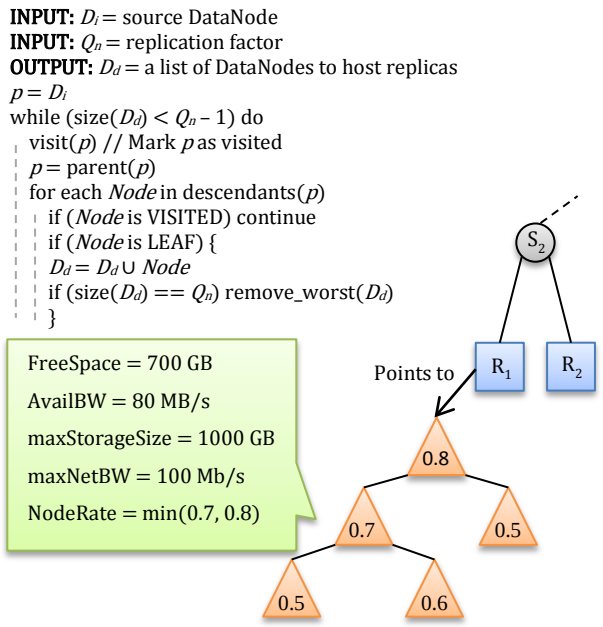

Figure 2: The structure of leafs of Network Tree and the psuedo-code of host selection algorithm

Replication of an existing data block is performed in a similar way. In order to find the proper DataNode to host the new replica, every DataNode that hosts one of the replicas of the data block selects a host using an algorithm similar to the one shown in Fig. 2 with Q_n=1. Next, the proper host for the new replica will be chosen from the selected hosts in the previous step and replication will be performed by the nearest DataNode to it. HDFS performs the replication whenever one of the replicas of a data block becomes corrupted, so the execution rate of this mechanism cannot be remarkable. Moreover, this mechanism can be used to adapt the number of replicas with a dynamic replication factor in storage clusters.

5 Evaluation

We have measured the performance of the proposed mechanism using three distinct criterions. The first two evaluations were performed using HDFS benchmarks in order to demonstrate the improvements in I/O performance of DataNodes and response time of the NameNode. The third one was performed to verify the low computational cost, network traffic and execution time of the algorithm proposed for Network Tree construction. According to the evaluation results, the proposed mechanism can improve the response time of file storage and retrieval operations in heterogeneous clusters with long-term I/O operations.

5.1 I/O performance

In order to evaluate the I/O throughput of HDFS in presence and absence of our proposed mechanism, we have used TestDFSIO benchmark, one of the benchmark packages of hadoop-test-1.0.1. In this manner, a cluster of one master node and three slave nodes is used. Hardware configuration and OS specification of these systems is shown in Table 1.

Table 1: Hardware and OS specification of cluster nodes

	OS	CPU	RAM	Disk (RPM)
Master	Mac 10.6.8	Intel Core 2 Due 2.8	4 GB	5400
Slave	Ubuntu 10.04	Intel Pentium IV 3 GHz	2 GB	5400

TestDFSIO is a MapReduce job and proceeds in two distinct phases. First, it creates some map and reduce tasks to write a number of files which is specified as an input parameter. After that, some other tasks are created to estimate the read performance of the cluster by reading specified number of files. In each phase, the benchmark prints some statistical information about the I/O operation performance of the cluster. In order to examine the effect of our proposed mechanism on HDFS I/O performance, TestDFSIO is executed in the presence and absence of the mechanism using various input parameters and the results are shown in Table 2.

Considering the presented evaluation results, our proposed mechanism is more beneficial for storage clusters with long-term I/O operations. As a result, increasing the size of files will magnify the effect of adaptive selection algorithms on the performance of storage clusters.

Table 2: Evaluated I/O throughput using TestDFSIO

Number of files		10	100	10	100
File size (MB)		100	100	1000	1000
Read (MB/s)	HDFS	7.006	7.421	8.983	9.603
	HDFS + AM	7.043	7.492	9.217	10.039
Write (MB/s)	HDFS	4.893	5.161	6.037	6.958
	HDFS + ASM	4.923	5.202	7.686	7.278

5.2 Cluster scalability

Execution overhead of the proposed mechanism can degrade the number of metadata operations that the NameNode is capable of performing in every second. Once the NameNode reaches its resource limits as a result of an extraordinary number of metadata operations, presence of the proposed mechanism can increase response time of metadata operations and decrease overall performance of the cluster. For investigating this issue, one of the benchmark packages of hadoop-test-1.0.1 called NNBench is used. This benchmark performs in four distinct phases including *create_write*, *open_read*, *rename*, and *delete*. Moreover, NNBench supports a wide variety of input parameters including *sizeOfBlocks* and *numberOfFiles*. Fig. 3 shows the execution results of NNBench in the presence and absence of our proposed mechanism.

As it can be seen in Fig. 3, execution overhead of the proposed mechanism can have a slight impact on the NameNode performance only when the NameNode is overwhelmed with user requests. Indeed, this performance degradation is a direct outcome of centralize nature of the

NameNode in Hadoop clusters and can be avoided on the first sight by downscaling the cluster or improving hardware specification of the NameNode. However, using the proposed mechanism looks economical and effective with regards to how much it improves the system performance by spending a slight amount of processing power and memory space.

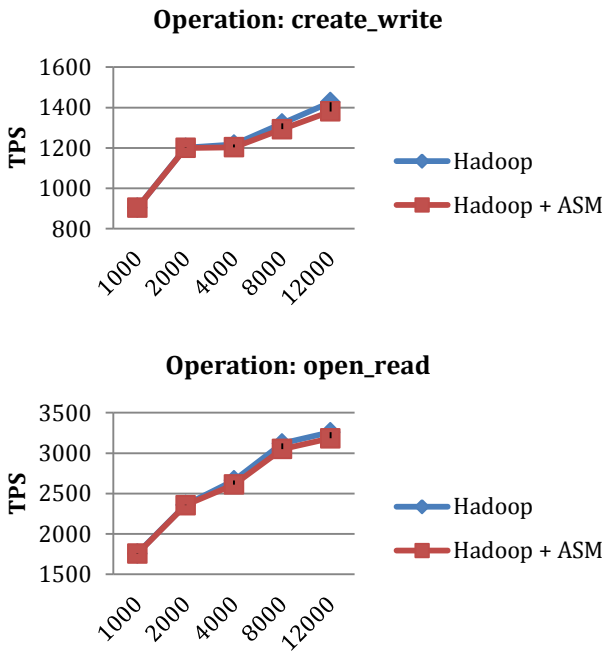

Figure 3: NameNode TPS evaluation using NNBench

5.3 Construction cost of Network-Tree

Construction of Network Tree is performed using ICMP network messages and the Ping utility. To evaluate the performance of our proposed method, we have examined the total network traffic and construction time of this method. Total network traffic of each run is monitored using Wireshark [21]. Searching the Network Tree is a one-time operation and every DataNode have to perform it just after joining the cluster. D-Link switches with a total bandwidth of 100 Mbps have been used to from the communication network and connect computational nodes running a Linux distribution. The height of the Network Tree is two that demonstrates a two level network topology including the core switch and rack switches which are directly connected to the core switch. Fig. 4 shows our experimental results, in which total network traffic and the execution time are in KB and seconds respectively.

The execution time of this algorithm, which is used for searching the Network Tree, directly depends on the branching factor and the height of the tree. Height of the tree does not vary much and even in extra-large clusters, it can hardly ever reach a value more than 3. For instance, using 16-port switches in building a communication tree with the height of 3 will lead to a cluster of size 16^3. The largest operational Hadoop cluster at Yahoo contains 3500 DataNodes which is far less than 4096. Moreover, the branching factor does not exceed 16 most of the times. In most of the clusters, network switches with more than 16 ports are used as rack switches that have no effect on the branching factor of the Network Tree. Considering how this method can facilitate joining of new DataNodes to the cluster, it can be inferred that its execution time and network traffic are reasonable.

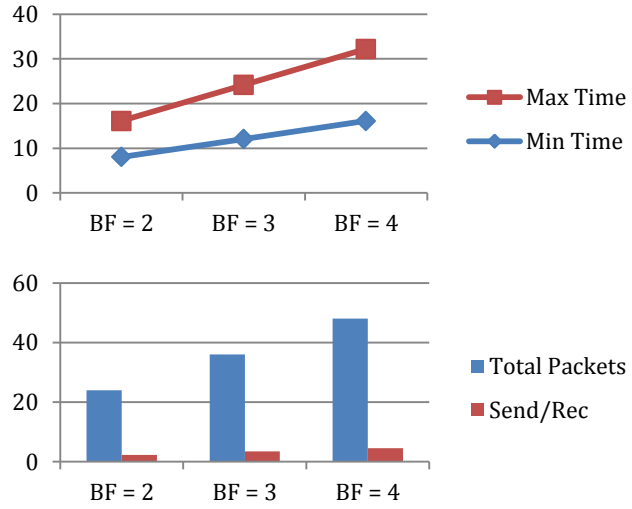

Figure 4: Evaluation results of the tree construction algorithm

6 Further works

Besides the proposed mechanism, distributed file systems can adapt with the execution state of the cluster in some more ways. Using dynamic replication factors and popularity domains for files are two of these ways that can benefit the storage cluster in both reliability and response time.

Considering the concept of files' popularity which is widely used in P2P file sharing systems, the replication policy can be improved. Thus, each file will be replicated in every neighborhood that contains more reader or writer nodes. In this way, a popularity domain will be assigned to each file containing the DataNodes that will probably read or write from/to it in the future. This mechanism can be used along with the task distribution mechanism of Hadoop which executes every task near to its required data. The cooperation of these two can lead to better distribution of load in Hadoop clusters. As a result, the behavior of file system users will affect the replication of data blocks and the file system will adapt with users' activity.

Besides being used for determining proper replication factor of each file, the concept of popularity domains is beneficial for improving reliability of the NameNode in Hadoop clusters. As each popularity domain normally contains the list of DataNodes sharing the respective file, the global namespace can be divided into many popularity domains and distributed across several NameNodes. Consequently, reliability of the central mechanism which is currently in use in Hadoop clusters to maintain file system metadata will be improved by increasing the number of metadata servers. This mechanism follows the same idea as namespaces in Plan 9 and uses popularity domains in exchange of per-process namespaces. Reducing the computational overhead of the NameNode will directly improve the response time of storage and retrieval

operations of files. Indeed, this mechanism can improve both reliability and response time of distributed file systems.

7 Conclusion

Our proposed mechanism considers the execution state of the storage cluster in performing data blocks replication. Various aspects of the cluster such as network topology and different state variables of computational nodes like network traffic and remaining storage space are considered to find the proper host for storing replicas. As the first step in the replication process, the NameNode specifies the required number of replicas for each data block. Next, the replication roadmap is designed based on the performed calculations and the selected host will store the newly created replica. The host selection algorithm will choose the proper nodes to store the new replica by looking at their execution state, available resources, and distance to the source of data. As a result, the load of storage and retrieval operations will be uniformly distributed across the cluster.

A similar method is used in order to find proper replica hosts to provide read operations with requested data blocks. In this manner, our proposed mechanism estimates the cost of reading a data block from every host and uses the results to generate an ordered list of hosts. Some execution variables of storage nodes including available network bandwidth, disk read speed, and current data streams are considered in evaluating the cost of reading every data block. The network distance between the replica host and reader node is also considered in this selection algorithm.

HDFS has been used as our implementation platform. Several facts like easy to use interfaces and open source implementation of HDFS lead to this decision. Performed evaluations show that our proposed mechanism improves the response time of file storage and retrieval operations in heterogeneous clusters running long-term I/O operations. Furthermore, its computational overhead will not have a remarkable effect on cluster scalability.

8 References

[1] R. Ross, D. Nurmi, A. Cheng, and M. Zingale, "A Case Study in Application I/O on Linux Clusters," in *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, New York, NY, USA, 2001, pp. 11-11.

[2] T. White, *Hadoop: The Definitive Guide, 2nd Edition*. O'Reilly Media / Yahoo Press, 2010.

[3] B. Witworth, J. Fjermestad, and E. Mahinda, "The Web of System Performance," *Communications of the ACM*, vol. 49, no. 5, pp. 93-99, May 2006.

[4] R. Buyya, *High Performance Cluster Computing*. Prentice Hall, 1999.

[5] W. Yu, S. Liang, and D. K. Panda, "High Performance Support of Parallel Virtual File System (PVFS2) over Quadrics," in *Proceedings of the 19th Annual International Conference on Supercomputing*, New York, NY, USA, 2005, pp. 323-331.

[6] B. Nicolae, G. Antoniu, L. Bougé, D. Moise, and A. Carpen-Amarie, "BlobSeer: Next-Generation Data Management for Large Scale Infrastructures," *Journal of Parallel and Distributed Computing*, vol. 71, no. 2, p. 169–184, Feb. 2011.

[7] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *26th Symposium on Mass Storage Systems and Technologies (MSST)*, Incline Village, NV, 2010, pp. 1-10.

[8] P. H. Carns, W. B. Ligon, and R. T. Robert B. Ross, "PVFS: A Parallel File System for Linux Clusters," in *Proceedings of the 4th Annual Linux Showcase*, Berkeley, CA, USA, 2000, pp. 28-28.

[9] J. Montes, B. Nicolae, G. Antoniu, A. Sánchez, and M. S. Pérez, "Using Global Behavior Modeling to Improve QoS in Cloud Data Storage Services," in *Second International Conference on Cloud Computing Technology and Science (CloudCom)*, Indianapolis, IN, 2010, pp. 304-311.

[10]D. Presotoo, R. Pike, K. Thompson, and H. Trickey, "Plan 9, A Distributed System," AT&T Bell Laboratories, 1991.

[11]W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer, "Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs," in *Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Santa Clara, California, United States, 2000, pp. 34-43.

[12]T. E. Anderson, et al., "Serverless Network File Systems," *ACM SIGOPS Operating Systems Review*, vol. 29, no. 5, pp. 109-126, Dec. 1995.

[13]S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 29-43, Oct. 2003.

[14]A. Devulapalli, D. Dalessandro, P. Wyckoff, N. Ali, and P. Sadayappan, "Integrating Parallel File Systems with Object-Based Storage Devices," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, New York, NY, USA, 2007, pp. 1-10.

[15]OrangeFS. (2011, Dec.) Orange File System. [Online]. http://www.orangefs.org/

[16]J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107-113, Jan. 2008.

[17]S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System," in *7th Symposium on Operating Systems Design and Implementation*, Washington, 2006, pp. 307-320.

[18]S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, New York, NY, USA, 2006, pp. 31-31.

[19]J. Venner, *Pro Hadoop*. Apress, 2009.

[20]T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009.

[21]Wireshark Foundation. (2012, Feb.) Wireshark. [Online]. http://www.wireshark.org/