한국시뮬레이션학회 논문지
Vol. 17, No. 4, pp. 229-239 (2008. 12)

# SoFA: A Distributed File System for Search-Oriented Systems

**Eunmi Choi[1†] · Tran, Doan Thanh[1] · Bipin Upadhyaya[1] · Fahriddin Azimov[1] · Luu, Hoang Long[1] · Truong, Phuong[1] · SangBum Kim[2] · Pilsung Kim[2]**

# SoFA: 검색 지향 시스템을 위한 분산 파일 시스템

**최은미 · 쩐도안타인 · 비핀 우바디야 · 파흐릇딘 아지모프 · 루왕용 · 장옥향 · 김상범 · 김필성**

ABSTRACT

A Distributed File System (DFS) provides a mechanism in which a file can be stored across several physical computer nodes ensuring replication transparency and failure transparency. Applications that process large volumes of data (such as, search engines, grid computing applications, data mining applications, etc.) require a backend infrastructure for storing data. And the distributed file system is the central component for such storing data infrastructure. There have been many projects focused on network computing that have designed and implemented distributed file systems with a variety of architectures and functionalities. In this paper, we describe a complete distributed file system which can be used in large-scale search-oriented systems.

**Key words** : Distributed file system, Search-Oriented system

요 약

분산 파일 시스템(DFS)은 분산 환경에서 장애와 사본에 대한 투명성을 보장하며 파일을 다수의 물리적인 컴퓨터 노드들에게 저장할 수 있는 메카니즘을 제공한다. 검색엔진, 그리드 컴퓨팅, 데이터 마이닝 어플리케이션등과 같이 많은 양의 데이터를 처리하는 어플리케이션들은 데이터 저장을 위한 백엔드 인프라 구조를 제공할 필요가 있다. 분산 파일 시스템은 이러한 저장 데이터 기반을 위한 주요 구성요소가 된다. 많은 프로젝트의 관심사가 되는 네트워크 컴퓨팅은 이와 같이 설계 및 구현된 분산 파일 시스템을 갖추고 있으며, 다양한 아키텍처와 기능들을 시스템의 특성에 따라서 제공하고 있다. 이 논문에서는 대용량의 검색 지향적인 시스템에서 사용되는 SOFA 분산 파일 시스템, 메카니즘들과 성능들을 소개한다.

**주요어** : 분산 파일 시스템, Search-Oriented system

## 1. Introduction

Permanent Storage is a fundamental abstraction in computing. A permanent storage consists of a named set of objects that come into existence by explicit creation. The naming structure, the characteristics of the objects, and the set of operations associated with them characterize a specific refinement of the basic abstraction. A file system is one such refinement. A file system is organized as a hierarchical directory of files, and files are variable-length arrays of bytes. These elements (directories and files) are directly exposed to file system clients; clients are responsible for logically structuring their application data in terms of directories, files, and bytes inside those files.

* This research was supported by the SKT research project, the research program in Kookmin University, and partially the MKE under the ITRC support program (IITA-2008-C1090-0804-0015).

2008년 11월 17일 접수, 2008년 12월 6일 채택

[1)] 국민대학교 비즈니스IT학부
[2)] SK Telecom Convergence and Internet R&D Center
주 저 자 : Eunmi Choi (최은미)
교신저자 : Eunmi Choi (최은미)
E-mail: emchoi@kookmin.ac.kr

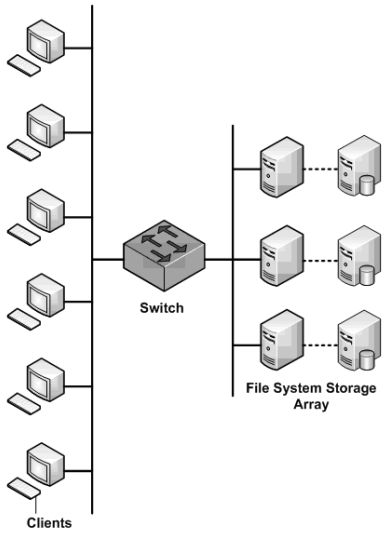



**Fig. 1.** Distributed File System Overview

A DFS is used to build a hierarchical view of multiple file servers and shared on the network. Instead of having to think of a specific machine name for each set of files, the user will only have to remember one name; which will be the 'key' to a list of shares found on multiple servers on the network as shown in figure 1. There are some features that should be considered to build a DFS such as: the architecture of a DFS, the communication technology, the naming and synchronization mechanism, data consistency, fault-tolerance and security. Even though considering these features, many DFS's have been developed over the years and almost two decades of research have not succeeded in producing a fully-featured DFS [11,12,15].

In this paper, we describe about the structure of a DFS which is composed by many prominent features. There are many kinds of architecture for a DFS, such as: Client-Server Architectures, Cluster-Based Distributed File System, Symmetric Architecture (based on peer-to-peer technology), Asymmetric Architecture, and Parallel Architecture. We choose the Cluster-Based Distributed File System Architecture to implement because of its simplicity, scalability, reliability, and high-perfomance through single master controlling hundreds of chunk servers. For communication, we use Remote Procedure Call method to communicate as they make the system independent from underlying operating systems, networks and transport protocols. In a DFS, it is very important that each object has an associated logical path name and physical address. To achieve this naming capability, we use AVL tree data structure [2,3] for *mapping* of the file system abstraction onto physical storage media and keeping the transparency to the user. Finally, to integrate consistency and fault-tolerance into this system, we use *Caching* and *Replication* mechanism [10,13].

Based on those characteristics, we propose a number of DFS mechanisms and performance issues. In Cluster -based architecture, in order to manage the naming system we provide a metadata handling mechanism to optimize the performance of the metadata handling process by combining tree-based data structure and self-balancing data structure. The key improvment is our new metadata indexing mechanism having the better performance comparing to normal tree-based data structure used in HDFS [17] and faster performance of metadata processes comparing to B-tree implementation in GFS [18]. For synchronization issue, to provide a consistent and reliable Metadata Management System and support the persistency of cache in the client side, we employ Branch locking mechanism and leasing mechanism for access control of data objects. For Replication and Consistency, similar to Google File System (GFS) [18] and Hadoop [17], we use a pipeline replication mechanism with minor modification comparing to GFS in order to provide consistency among replications. In this mechanism, we employ passive replication method to transparently replicate data to multiple replica servers to reduce communication overhead between the client side and SoFA. Besides, we implement a persistent cache at client side ensuring Time-constraint Relaxing Data Storage.

The structure of the paper is as follows. Sections 2 cover related works in Distributed File Systems. In section 3, the system architecture of our application is presented. In Section 4 we showed Storing & Retrieving Mechanisms. In Section 5, Data Replication & Consistency for Search-Oriented System are presented and in Section 6 experimental results is outlined. We conclude our paper in section 7.

## 2. Related Work

First issue considered important to a DFS is the types of DFS architectures. Different DFS Architectures exists such as Client-Server Architectures (e.g. Sun Microsystem's Network File System (NFS) [16]) which provides a standardized view of its local file system. Advantage of this scheme is that it is largely independent of local file systems. Another type of Architecture is Cluster-Based Distributed File System such as GFS. It consists of a Single master along with multiple chunk servers and divided into chunks of 64 Mbytes each. The advantage is its simplicity and it allows single master to control a few hundred chunk servers. Third type of architecture is Symmetric Architecture that is based on peer-to-peer technology. It uses a DHT based system for distributing data, combined with a key based lookup mechanism. In contrast, an Asymmetric Architecture file system is a file system in which there are one or more dedicated metadata managers that maintain the file system and its associated disk structures. Examples include Panasas ActiveScale [21], Lustre [19] and NFS file systems. Finally, a Parallel Architecture file system is one in which data blocks are striped, in parallel, across multiple storage devices on multiple storage servers. Support for parallel applications is provided allowing all nodes access to the same files at the same time, thus providing concurrent read and write capabilities. An important note is that all of the above definitions overlap. Based on the architecture types provided by literature, we follow the implementation of the Cluster-Based Distributed File System with asymmetric and parallel architecture.

Next important issue is considered when implementing a DFS is the Naming mechanism. It plays an important role as each object has an associated logical path name and physical address. Its fundamental idea is to provide its clients complete transparent access to a remote file system. The currently common approach employs a central metadata server to manage file name space such as GFS, Hadoop, Lustre, Panasas, KFS [20]. Therefore decoupling metadata and data improve the file namespace throughput and relief the synchronization problem. Another approach is metadata distributed in all nodes resulting in all nodes understanding the disk structure. This approach is employed in Parallel Virtual File System (PVFS2) [22] and Red Hat Global File System (RGFS) [23]. But serious implication is users do not share name spaces due to security issues. It makes file sharing harder. Our approach is utilizing central metadata server to manage the naming system with important improvement to optimize the performance of the metadata handling process by combining tree-based data structure and self-balancing data structure. With this feature, we can overcome the linear searching in normal data structure as well as improving the performance in backup of the metadata information.

The last important issue in a DFS is Consistency and Replication. To provide the consistency, most of DFS employ checksum to validate the data after sending through communication network. Besides, Caching and Replication play an important role in DFS, most notable when they are designed to operate over wide-area network. It can be done in quite few ways such as Client -side caching and Server-Side replication. There are two types of data need to be considered for replication: metadata replication and data object replication. Metadata is the most important part of the whole DFS. Thus, all DFS provide a mechanism to ensure the availability and recoverability of this data such as backup metadata server and snapshot of metadata with transaction logs. For data objects, there are different approaches depending on the purpose of applications. DFSs like Lustre and Panasas assume that data object is available as long as the physical devices are available. Hence, they consider a physical failure as an exception and the object data can be lost. In case of other DFSs like GFS and Hadoop, their applications require the availability of data as the critical condition and failure will be the norm rather than the exception. Thus, data objects are replicated in different servers. This high bandwidth consuming feature leads to the asynchronous replication method named "Replication in pipeline" which is employed in GFS and Hadoop. Similar to GFS and Haddop, we use a pipeline replication mechanism. In this mechanism, we employ passive replication method to transparently replicate data to multiple replica servers to

reduce communication overhead between the client side and SoFA. Besides, we implement a persistent cache at client side ensuring Time-constraint Relaxing Data Storage.

## 3. System Architecture

Our Distributed System is mainly targeted to large Search-Oriented Systems. This section presents the overall architecture mainly focuses on DFS for Search-Oriented systems.

### 3.1 Overall Search-oriented System Architecture

The following figure shows the system overview of our search system. The architecture includes Web Server, Delegator, Cache Server, DFS [3,4], Integrated Search Component Server and Management Station.

A user sends search queries to the Web Server. And these queries are passed to Cache Server. The Cache Server contains caching system of frequently searched contents. We have number of Cache Servers to maintain our search system without performance bottlenecks. The Cache Server activates operation of Delegator when it does not contain the searched contents in the caching system. Delegator invokes an ISC (Integrated Searching Component) Server. After generating search contents, we store Cached data of searched results into Distributed File System servers (DFS).

The DFS [1,9,12,14] provides fast access to files which located distributively in the network. Files are separated into chunks and locate in different chunk servers. Each chunk has its replication in other servers. This will provide data reliability in our system. DFS has been designed for Search-Oriented applications that process large volumes of data (such as: search engines, grid computing applications, data mining applications, etc.) require a backend infrastructure for storing data. Such infrastructure is required to support applications whose workload could be characterized as: primarily write-once/read- many workloads. We develop the DFS as a high performance distributed file system to meet this infrastructure need.

The Cluster Management station takes care of managing the whole system servers. The administrator through Cluster management can see the current status of the servers, control their balance, deploy or update applications, define unavailable servers and so on. Cluster management station observes processes search systems and status of system components.

ISC Servers are main sub-components for search query. In ISC Servers there are stored indexed documents of different category which retrieved from Crawler. Each ISC server deals with Delegator by getting query and returning search result.

### 3.2 Search-Oriented File System Architecture (SOFA)

We followed the implementation of the Cluster-Based Distributed File System with asymmetric and parallel

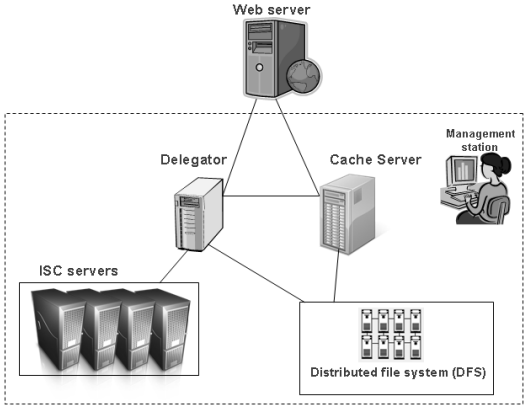



**Fig. 2.** Overall System Architecture

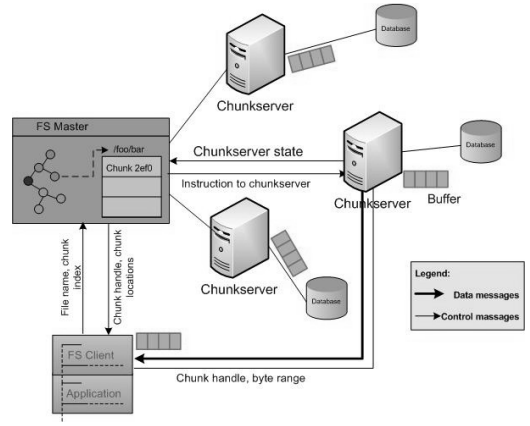



**Fig. 3.** The overall flows of communication between FS Client, Master and Chunk Server

architecture. It consists of three major components: File System (FS) Client, FS Master Server and FS Chunkserver that construct our DFS. We assume that the DFS will run on homogeneous computers. The brief description of each component is as follows.

FS Master Server: FS Master Server maintains all file system metadata that includes the file and chunk namespace, access control information, the mapping from files to chunks, and the current locations of chunks.

FS Client: FS Client is the host for applications running on DFS and it interacts with the FS Master Server for metadata operations, but all data-bearing communication goes directly to the Chunkserver. It sends a request to one of the replicas, most likely the closest one. FS Clients never read and write file data through the Master Server, instead it asks the FS Master Server which Chunkserver it should connect. It caches this information for a limited time and interacts with the Chunkserver directly for subsequent operations.

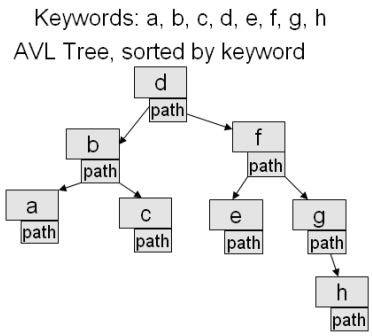
Chunkserver: Chunkserver store chunks on local disks as Linux files. Read or write chunk data specified by a chunk handle and a byte range. Files are divided into fixed-size chunks (64 MB), each chunk is identified by an immutable and globally unique 64 bit chunk handle. Chunk is replicated on multiple Chunkservers on different groups of computers.

## 4. Storing And Retrieving Mechanism

In this section, we describe the storing and retrieving mechanisms and related structures.

### 4.1 Indexing Structure for Mapping

There are different data structures for storing metadata of files such as array, list or tree. Array and List are very popular because they can give a visual vision about storing mechanism and very easy to implement. However, one considerable disadvantage of arrays and lists is inefficient time in searching. Using arrays and lists can take long time for searching because of linear traversal. Therefore, considering the most appropriate mechanism to deal with the demands for short time searching, tree




An example of AVL tree data structure

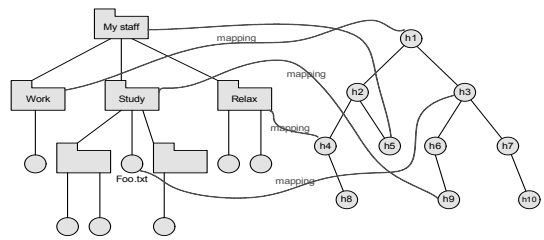



**Fig. 5.** An Using AVL tree for storing File Namespace. Mapping between Organizing Structure (left) and Performance-oriented Structure (right)

data structure is more appropriate than others.

After making comparison (results are shown in section 6.1), we see that AVL tree has higher performance than other trees (such as: 2-3-4 tree [5,6] and Red-Black tree [7]). From these results, we decide to use AVL tree data structure for storing File Namespace in our Distributed System.

The AVL tree is a binary search tree that rearranges its nodes whenever it becomes unbalanced. A node in an AVL tree has only two child-nodes and each node contains the balance factor where the factor of a node is the difference between the height of its right sub-tree and the height of its left sub-tree. This factor is used to make decision to rotate tree whenever the tree is unbalanced as a result we obtain a balanced tree. In an AVL tree, the heights of the two child subtrees of any node differ by at most one; therefore, it is also said to be height-balanced. An imbalance at a node of an AVL tree can be corrected by a single rotation or a double rotation. Figure 4 shows an example of AVL tree. In this example, we construct an AVL tree from a list of

input files. With each file's path, we generate a keyword according to that file's path and this keyword will be used to insert into an AVL tree. Each node, hence, contains a keyword, a file's path and two pointers for two child nodes.

As a result, an approach using a mechanism to mapping the File Namespace with nodes of a tree to solve this problem is proposed as in Figure 5.

By combining tree-based data structure and self-balancing data structure, we can overcome the linear searching which results in the complexity of *O(n)* as in normal data structure with the searching complexity of *O(log n)*, where n is the number of elements in the tree. Section 6 in this paper presents a state-of-the-art investigation on tree data structures and compares the pertinent characteristics of B, Red-Black and AVL trees in the context of DFS with large scale data. Our experiments and analysis show that neither data structure totally dominates the other. The decision as to which is performance-wise efficient is a function of the application.

### 4.2 Caching Mechanism

One of the concerns for our system was the memory of the master server. The master server maintains about 64 bytes of each 64 MB chunk. The number of requests the masters gets from FS Client about the chunk local can be one of the bottleneck of the system. The FS client caches the result it retrieved from master server for future use.

We employ a symmetric design for persistent cache at client side of SoFA ensuring Time-constraint Relaxing Data Storage. In this design, we add an In Memory File System (IMFS) to the client side to provide applications use an Interface to interact with the SoFA. Metadata operations are executed on IMFS, then through a RPC, they transparently executed to Metadata Server. Data operations are executed on IMFS. Normal data are buffered in IMFS and are transferred to Data Object Servers when buffers are full or their lease is near to end. IMFS automatically revokes Locks and Leases when they are time-out or revoked by the Master Server.

### 4.3 Fragmentation Control Mechanism

The chunks are stored in local file system of chunk servers in DFS. We have fixed of chunks size to be 64 MB. So irrespective to the size of actual chunk we are writing we allocate each chunk a 64 MB size. This gives each file a continuous space in the disk for the files to be stored which reduces seek and latency time of disk.

When a file is deleted by a user or an application, it is not immediately removed from DFS. Instead, in our DFS, the chunks corresponding to the file is marked the deleted. The file can be restored quickly as long as it the space is not reclaimed by other chunks. The time when the chunk can be used by other chunks can be configured. The deletion of a file causes the blocks associated with the file to be freed. Note that there could be an appreciable time delay between the time a file is deleted by a user and the time of the corresponding increase in free space in DFS.

### 4.4 Load Balancing Mechanism

Single Master Server simplifies our design. The master server knows about the load each chunkserver has. When the FS Client requests for the file it can give its knowledge to the FS Client so that the chunk servers are not overloaded. Replicating each chunk into three chunkservers does not allow the chunkserver to be overloaded.

## 5. Data Replication/Consistency

Replication is one of the oldest and most important topics in distributed systems. Replication is the process of sharing data to ensure consistency between redundant resources, to improve reliability, fault-tolerance, or accessibility.

In SoFA, all the operations of a replication will be controlled by a Master Deamon in Master Server, and it will be controlled at each replica by a Replica Deamon. This replication mechanism itself will be transparent to an external user. Also, in a failure scenario, a failover of replicas is hidden as much as possible.

We use locking and leasing mechanism to ensure the consistency of DFS. A lease is a contract that gives its

holder specified property for limited amount of time and lock means avoiding anyone except the person holding the lock to make changes in the resources.

## 6. Experiment Results

We setup a distributed system environment on a LAN network for experimentation purposes. All the machines in our implementation use Linux Fedora Core 8 (kernel version: 2.6.25.4-10).

Computer specifications for one system are Intel E8400 Dual Core CPU 3.00 GHz, 2Gb 6400 PC RAM, 500Gb/7200RPM HDD with 32Mb cache, gigabit Ethernet card and for network experiment 1000Mbps switch is used.

### 6.1 Experiment for DFS Indexing Structures

We implemented the trees in DFS environment using C++ programming language and conduct the performance comparison tests. In this test, we use B tree of order 4 (also called 2-3-4 tree). We run test many times to insert, delete and search a node into a 2-3-4 tree, a Red-Black tree and an AVL tree. We have collected the execution time to compute the average time to insert, delete and search data structures. The test data ranged from small scale (100,000 records) to large scale (10,000,000 records). We also conduct a comparison in two operations: writing a tree into files and loading a tree from files. Finally, we analyze the tree size in memory (the size of memory needed to hold the tree). It is an important factor to assert the quality of a tree.

#### 6.1.1 Storing & Retrieving Performance

The insertion or deletion of an element into a node in these trees may cause imbalance at that node. In order to solve this problem, each tree has its own mechanism. For instance, 2-3-4 trees use splitting to divide node when that node is overflow or use merging to merge nodes when one node is underflow. These steps can be recursive upward to the root node. Similarly, Red-Black trees use one color-flipping operation or two rotation operations to deal with imbalance at a node and in AVL tree only two rotation operations are used.

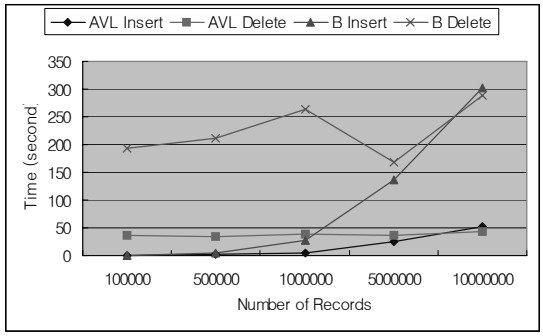



**Fig. 6.** Insert (Insert the whole tree) and Delete (10,000 sample records) Test

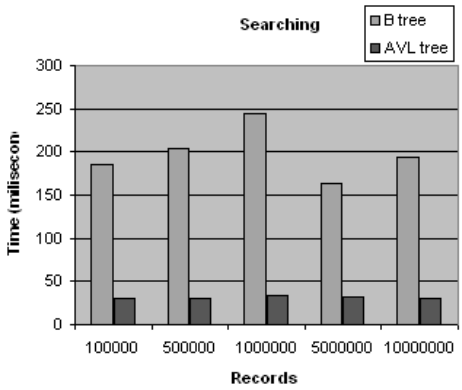



**Fig. 7.** Searching Test

In this paper, we performed insertion and deletion test with data ranged from small scale (100,000 records) to large scale (10,000,000 records).

#### 6.1.2 Searching Performance

The searching result in AVL tree illustrates that AVL tree is more effective than in Red-Black tree. The height of a 2-3-4 tree is the smallest compared to others two data structure. But we can not say that searching in 2-3-4 trees are absolutely faster than the others because the linear traversal applied to all elements in one node while searching can take a lot of time.

The searching operation in AVL tree takes *O(log n)* time in both the average and worst cases, where n is the number of nodes in the tree prior to the operation. However, we found that AVL trees perform better than B trees for search-intensive applications.

It is observed that by combining tree-based data

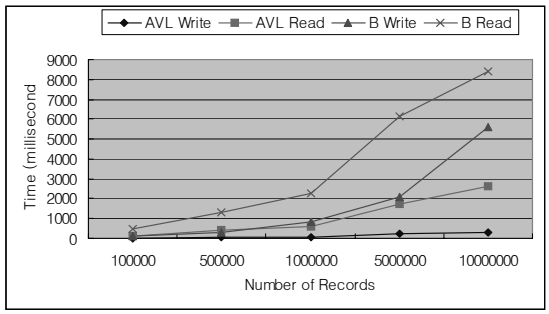



**Fig. 8.** Writing and Reading Tree Test

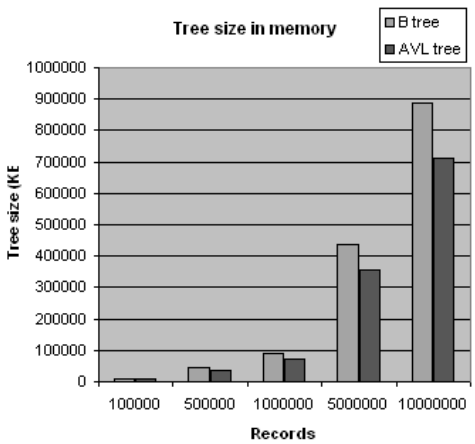



**Fig. 9.** Memory Size of Trees Comparison

structure and self-balancing data structure, we can overcome the linear searching which results in the complexity of *O(n)* as in normal data structure with the searching complexity of *O(log n)*, where n is the number of elements in the tree.

#### 6.1.3 Building Index & Saving Configuration Performance

Based on our experiments with respect to find memory -efficiency that we performed by inserting records range from 100,000 to 10,000,000, we found number of disadvantages. One of the disadvantages of the 2-3-4 tree is space waste in memory because of lots of empty nodes in leaf level. Our approach to this problem is writing a tree into the file and reloading the tree from that file when needed [3]. With this approach, we can reduce considerable amount of time for creating trees than creating trees from raw-data files.

In the result figures, the horizon axe shows the number of records and the vertical axe shows the execute time of these test case. In figure 6, the execution time of B trees increases dramatically when the number of records increases above one million records. However, the execution time of AVL trees in this case steady increases.

Figures 7 shows the results when randomly deleting\ searching 10,000 nodes from\in B trees and AVL trees. The execution time in AVL tree is unchanged while B trees have less stability by changing four times higher or more.

Figures 8 depict the execution time when writing\ reloading a B trees and AVL trees into\from file. The B trees execution time always much longer than AVL trees especially when the number of records increases. This result shows that the structure in AVL trees is the better solution in the DFS.

The memory size which each B tree and AVL tree holds according to each number of records are shown in figure 9. With the small number of records, the difference in size of B trees and AVL trees is small. Nevertheless, the difference increases rapidly with large data (>5,000,000 records).

### 6.2 Comparison with Local File System and Hadoop

In this section, we compare our performance with the Hadoop Distributed File System (HDFS) with 6 servers. For this experiment we read and write 1GB files in HDFS and in SoFA. Multiple read/write clients were used while reading and writing in the systems without caching.

The result shows that reading/writing big size files in distributed file system has considerable performance gain as compared to HDFS. The graph shows the increase in performance as we add more clients in our DFS. Compared to HDFS, SoFA has achieved 2 times faster in reading and 1.5 times faster in writing performance. Reading is much better performance since search oriented systems require heavy and fast reading for its right-once -read-many mode. The performance is better than HDFS because of the high-performance metadata processing

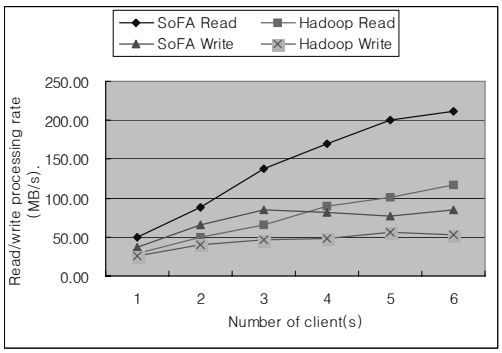



**Fig. 10.** File System Access Speed Performance of SoFA and Hadoop

and optimization of logging and synchronizing during the process. With the use of AVL tree for metadata indexing, SoFA can process thousands of metadata operations in one second compare to Hadoop with only around 10-70 operations. Besides, HDFS' accessing disk rate for logging is too high (3 times logging for one 0-byte file creation) and there are too many used synchronization protections. Those degrade the HDFS performance while SoFA reduces them as much as possible. This result approves the performance improvement of SoFA System Architecture.

## 7. Conclusion

SOFA is designed to support very large files of search-oriented systems. Applications that are ideal to use our file system are those that deal with large datasets. SOFA applications need a write-once-read-many access model for files. A file once created, written, and closed need not be changed. This assumption simplifies data coherency issues and enables high throughput data access. Our system provides fault tolerance by constantly monitoring, replicating the critical data. We compare our system with HDFS. The results showed the significant improvement in file access performance. In future work, we consider optimizing the file system for better performance. Using UDP for transferring reduces the overhead time to maintain communication and the correctness of the file will be checked by comparing the CRC of the received block at application level.

## Reference


1. Tran Doan Thanh, et.al. "A Taxonomy and Survey on Distributed File Systems", in Proceedings of NCM, 2008.
2. Erin K, "Lectures on AVL Tree", http://inst.eecs.berkeley.edu
3. Luu Hoang Long, Eumi Choi, "Data Structure for Distributed File System", in Proceedings of NCM, 2008.
4. Bipin Upadhyaya, et. al., "Distributed File System: Efficiency Experiments for Data Access and Communication", in Proceedings of NCM, 2008.
5. Goetz G, "Write-Optimized B-Trees", in the Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004.
6. Theodore J, Dennis S, "Utilization of B-trees with Inserts, Deletes and Modifies", in the Proceedings of PODS Conference. 235-246, 1989.
7. Lyn Turbak, "Lecture on Red-Black Tree", Wellesley College, 2001.
9. Chandramohan A. Thekkath, et al, "Frangipani: A scalable Distributed File System", System Research Center, Digital Equipment Corporation, Palo Alto, CA, 1997.
10. Barbara Liskov, et al, "Replication in the Harp File System", Laboratory of Computer Science, MIT, Cambridge, CA, 1991.
11. John Douceur and Roger Wattenhofer, "Optimizing file availability in a server-less distributed file system" In Proceedings of the 20th Symposium on Reliable Distributed Systems, 2001.
12. Eliezer levy and Abraham silberschatz, "Distributed File Systems: Concepts and Examples", ACM Computing Surveys, Vol. 22, No. 4, December 1990.
13. Yasushi Saito and Marc Shapiro, "Optimistic Replication", ACM Computing Surveys, Vol. 37, No. 1, March 2005, pp. 42-81.
14. Satyanarayanan, M., "A Survey of Distributed File Systems," Technical Report CMU-CS-89- 116, Department of Computer Science, Camegie Mellon University, 1989
15. Howard, J.H., et al, "Scale and Performance in a Distributed File System," ACM Transactions on Computer Systems, Vol. 6, Issue 1, February 1988.
16. Callaghan, B., et.al, "NFS Version 3 Protocol Specification", Technical Report RFC 1813, IETF, June 1995.
17. The Hadoop Distributed File System http://hadoop.apache.org/core/docs/current/hdfs_design.html
18. Ghemawat, S., Gobioff, H., Leung, S.T., "The Google file system", ACM SIGOPS Operating Systems Review, Volume 37, Issue 5, pp. 29-43, December, 2003.
19. Braam, P.J, "The Lustre storage architecture", White Paper, Cluster File Systems, Inc., October, 2003.

20. “KOSMOS DISTRIBUTED FILE SYSTEM”, http://kosmosfs.sourceforge.net/
21. Nagle, D., Serenyi, D., Matthews, A., “The Panasas Active-Scale Storage Cluster: Delivering Scalable High Bandwidth Storage”, Proceedings of the 2004 ACM/IEEE conference on Supercomputing, pp. 53-62, 2004.
22. Yu, W., Liang, Sh., Panda, D.K., “High performance support of parallel virtual file system (PVFS2) over Quadrics”, Proceedings of the 19th annual international conference on Supercomputing, pp. 323-331, 2005.
23. “Red Hat Global File System”, White Paper, www.redhat.com/whitepapers/rha/gfs/GFS_INS0032US.pdf.

**최 은 미** (Eunmi Choi) (emchoi@kookmin.ac.kr)

1988 고려대학교 컴퓨터학과 학사
1991 Michigan State University, Computer Science, M.S.
1997 Michigan State University, Computer Science, Ph.D.
1998～2004 한동대학교 전산전자공학부 조교수
2004～현재 국민대학교 비즈니스IT학부 부교수

관심분야 : 분산시스템, 미들웨어, 유비쿼터스 컴퓨팅, 소프트웨어 메타 모델링, 대용량 검색 시스템, 그리드 컴퓨팅

**Tran, Doan Thanh** (thanhtd@kookmin.ac.kr)

2002 Hochiminh National University of Science, Telecommunication and Networking, B.S.
2006 Kookmin University, School of Business IT, M.S.
2006～Now Kookmin University, School of Business IT, Ph.D. Candidate

Areas of Interest: Grid computing, Ubiquitous Computing, Ubiquitous Sensor Network, Web Service Orchestration

**Bipin Upadhyaya** (bipin_upd@yahoo.com)

2006 Advanced College of Engineering and Management, Tribhuvan University. B.E
2008 Kookmin University , School of Business IT, M.S.

Areas of Interest: Distributed Computing, Peer-to-Peer Systems, Grid Computing, Social Network

**Fahriddin Azimov** (fahriddin@hotmail.com)

2007 Tashkent University of Information Technologies, E-commerce, B.S
2007～Now Kookmin University, Graduate School of Business IT. M.S Candidate

Areas of Interest: Ubiquitous Sensor Network, Distributed File Systems

**LuuHoangLong** (luuhoangdragon@gmail.com)

2007 Hochiminh National University of Science, Computer Science, B.S.
2008～Now Kookmin University, School of Business IT, M.S.

Areas of Interest: Grid computing, Ubiquitous Sensor Network, MIC, Peer 2 Peer Systems, Security

**Truong, Thi Ngoc Phuong** (ngocphuongtruong@gmail.com)

2005 Ho Chi Minh National University of Science, Software Engineering, BS
2007～Now Kookmin University, School of business IT, Master

Area of interest: P2P computing, semantic web technology

**김 상 범** (amzang@sktelecom.com)

1997 동국대학교 컴퓨터 공학과 학사 졸업
2009 한양대학교 Global MBA 경영학 석사 졸업
1998 ㈜평창정보통신 (알타비스타 코리아)
2000 ㈜네띠앙
2001 야후! 코리아
2007～SK Telecom

**김 필 성** (pskim11@sktelecom.com)

1997 연세대학교 컴퓨터과학과 졸업
1997～현재 SKtelecom C&I 기술원 연구원