A Reliable DHT-Based Metadata Server Cluster

Chang-Kuo Yeh, Tse-Ta Tseng, Yarsun Hsu
Department of Electrical Engineering,
National Tsing Hua University, HsinChu, 30013, Taiwan
{yachunggo, rogerable}hpcc.ee.nthu.edu.tw, yshsu@ee.nthu.edu.tw

Abstract- *This paper proposes a Distributed Hash Table-Based Metadata Server Cluster (DHT-MDSC), which provides an efficient routing strategy and an automatic reconfiguration protocol to eliminate the bottleneck of a centralized hash table. The design is based on a new concept merging p2p system with conventional metadata file system into a scalable and high performance distributed metadata server. A novel caching mechanism (LC-RIC) is implemented to improve the performance and scalability of the system. In addition, we have also implemented a reliable DHT-MDSC (RDHT-MDSC) which can tolerate the failure of multiple metadata servers. The system can reconstruct itself upon detecting any fault under normal operation. The time it takes to reconstruct the system is also short in our design.*

1. Introduction

Researches on networked storage have received great attention recently due to explosive data growth and widespread high speed network. As a result, the size of metadata is also getting bigger and bigger. Therefore the maintenance of metadata has also become more and more important. Many researches focus on the distribution of metadata since a single metadata server can't handle a very large number of metadata requests. Previous work has found that metadata operations can make up 50% traffic of file system communications [1]. Consequently metadata maintenance method must be carefully devised to improve the overall system performance.

There have been many papers on metadata distribution. For example, static subtree partitioning, dynamic subtree partitioning, pure hashing [2], equipotent subtree partition [3], and dynamic dir-gran (DDG) [4] have been proposed. These methods focus much on performance and "hot-spot" problems but neglect the maintenance overhead of manual configuration among metadata servers. In order to achieve load balance on directories, pure hashing is wildly used in a distributed file system. Although it is efficient to access a file, its hash table is not distributed across multiple metadata servers (MDSes). For a distributed hash table, an efficient routing strategy can significantly affect performance in a large scale metadata server cluster (MDSC). Furthermore, a high performance system should also provide a fast process to recover from a broken system. None of the above papers have dealt with these issues.

In this paper, we propose a Distributed Hash Table-Based Metadata Server Cluster (DHT-MDSC), which can provide an efficient routing strategy and an automatic reconfiguration protocol to eliminate the bottleneck of a centralized hash table. The communication protocol of DHT-MDSC is based on Chord [5, 12], a structured p2p implementation. Therefore, DTH-MDSC inherits the efficient routing strategy and the automatic reconfiguration from Chord [5, 12]. Besides, the hash table in DHT-MDSC is distributed to multiple metadata servers so that the accesses to metadata are no longer the performance bottleneck. To improve the reliability of the system, we also implement a Reliable Distributed Hash Table-Based Metadata Server Cluster (RDHT-MDSC) by adding a reliable mechanism to the DHT-MDSC. RDHT-MDSC can tolerate failure of multiple metadata servers and recover automatically except when two side by side metadata servers fail simultaneously.

The rest of this paper is organized as follows. Section 2 provides some related works. In section 3, we present the design and implementation of DHT-MDSC and RDHT-MDS. Performance evaluations of DHT-MDSC and RDHT-MDS are given in section 4. Finally, we make a conclusion in section 5.

2. Related works

There have been some researches studying the distribution of metadata. The most straightforward method is based on static subtree partitioning. For example, conventional network file systems like NFS[8], AFS[9] are based on static subtree partitioning. However, this policy suffers from overloading an individual server. Therefore, a smarter dynamic subtree portioning was proposed. However, dynamic subtree policy still suffers from the unbalance problem due to "hot spot" directories.

Another different approach uses pure hashing. For instance, Vesta[10], Intermezzo [11] are based on pure hashing. Although the hashing approach resolves the load balance problem, the central hash table is a performance bottleneck and can render the entire system unusable when it fails.

Both Equipotent subtree partition [3] and DDG [4] provide their own partition policy for directory hierarchy. They both make a uniform distribution of directories and files across metadata servers. However, scalability and reliability of metadata servers have not been addressed.

3. Implementation of DHT-MDSC and RDHT-MDSC

3.1 DHT-MDSC Architecture

DHT-MDSC can be divided into two parts, metadata servers and clients. Multiple metadata servers (MDS) form a metadata server cluster (MDSC) to service metadata requests. Clients send their requests to the metadata server they attach. Fig. 1 shows the MDSC architecture. Each server has an ID or hash key associated with it. The hash space is assumed to be 1024 in this example. A finger table and a hash table are also implemented on each metadata server as shown in the Figure. These metadata servers form a ring network by connecting two neighboring servers. In order to maintain the ring network, an MDS must record its two neighbors, predecessor (PD) and successor (SC). PD is the neighbor node in the counterclockwise direction and SC is the neighbor node in the clockwise direction. Following this rule, the next successor (N-SC) of an MDS is defined to be the successor of an SC, i.e. the second neighbor node in the clockwise direction. The front predecessor (F-PD) of an MDS is defined to be the predecessor of a PD, i.e. the second neighbor node in the counterclockwise direction. For example, the F-PD, PD, SC and N-SC of the MDS (ID = 5) are 761, 898, 135 and 273 respectively. Every hash key in a hash space also has its F-PD, PD, SC and N-SC. For example, the F-PD, PD, SC and N-SC of any hash key between 899 and 5 are the same as the MDS (ID =5). Hash functions like SHA1, SHA2, etc generate the hash key for a server according to its IP address and port. The finger table in each MDS contains part of the routing information for an MDSC. A finger table entry contains a hash key range and an MDS' id called SC-F (successor of a finger table entry). SC-F is used to indicate which MDS a metadata operation within the hash key range should be forwarded to. The hash table is composed of a hash key array ranging between an MDS' id and its PD's id. Each array entry has a linked list to store the metadata of a file hashed to the same key. Clients surrounding the ring structure also connect to this local network. But they do not know the existence of any MDS at the beginning. They use a discovery message to find an MDS they can use. A discovery message is a broadcast type packet containing a client's id, IP and port. A client broadcasts the discovery message to the network. It then picks the MDS with the shortest response time as its relayed server (RS). Once the RS is chosen, a client transmits all of its requests to this relayed server. Whenever a client issues a metadata command (such as read, write, delete, etc.) to its relayed server, the server will forward the command to a proper MDS according to the routing information contained in its finger table.

Join and *leave* are two major methods for an MDS to join to or leave from an MDSC. *Join* method is used when a new MDS wants to join an MDSC. First, it must broadcast a discovery message to its subnet. It then chooses the first MDS which sends back a reply message as the candidate to help it complete the join process. There are four steps for a new MDS to join an MDSC: updating its neighbors, initializing its finger table, updating others' finger table, and getting its own metadata from its PD. In step 1, the candidate is asked to find the two neighbors of the new MDS. After that, their neighbor relation is updated by the candidate. In step 2, the candidate establishes the finger table for the new MDS. In step 3, the candidate sends update messages to update finger tables of other MDSes. The update messages are sent to those MDSes which are the PD of the following hash keys:

$N - 2^i$, i = *0, 1, 2, 3,. …, log(hash space) -1*
if the generated hash key < 0,
generated hash key = generated hash key + hash space, where N is the hash key of the new MDS.

After receiving the update message, an MDS forwards the update message to its PD. The process terminates when a PD has been updated by the same update message. Thus, update messages propagate counterclockwise on the ring network of the MDSC. In step 4, the new MDS receives the metadata that it should control from its PD. Once these four steps have been performed, the joining process of the new MDS is completed and it becomes a member of the MDSC.

Leave message is used when an MDS must be shutdown or rebooted. Before one MDS can leave from an MDSC, it must move the metadata it controls to its SC and then sends a *leave* messages to notify those SC-Fs in its finger table. After receiving a leave message, an MDS relays this message to its SC. This process completes

when any SC has been updated by the same leave message previously. Therefore, the leave message looks like walking clockwise on the ring network of the MDSC to update other MDSes. Although the system can be reconfigured with the above two methods, the unpredictable failure of an MDS can destroy the whole system. We will present a reliable solution in section 3.3 and discuss it in section 4.

Although metadata delivering path is short, it still takes time to deliver on the network. The network speed is much slower than CPU and memory, so we need a mechanism to get desired metadata in one hop to achieve high performance. In section 3.2, we describe a caching method on a client side to improve the performance.

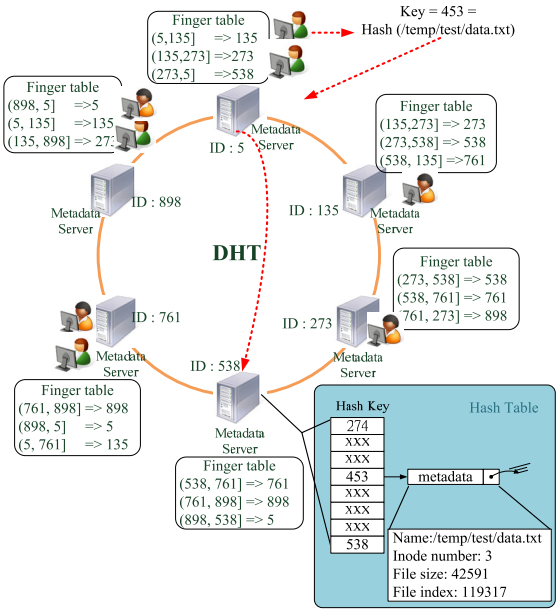


Fig. 1 MDSC architecture

3.2 Lazy Cache of Routing Information on Client End (LC-RIC) to Improve Performance

Although DHT-MDS has an efficient routing policy, its performance and scalability can still be improved further. This is because an operation packet may need to propagate through more than one hop to reach destination MDS and the network communication time is still much larger than CPU time, especially for small packets. This problem may be relieved by client side caching**.** We devise a lazy cache mechanism called Lazy Cache of Routing Information on Client End (LC-RIC). Lazy means that we only update it on demand. LC-RIC caches the id, ip, and port of an MDS on clients. If the destination MDS could be found in a client's cache table, only one hop is required to reach the destination MDS. Otherwise the request messages must go through routing processes described above. Once the request message returns, the destination MDS is found and added to the client's cache table. From here on, only one hop is needed to reach the MDS.

LC-RIC lazily updates a client's cache table when any MDS joins or leaves. It means a client doesn't update its cache table immediately when any MDS joins or leaves. For example, when the first time a client accesses its metadata on a new MDS, it can send the request to the closest MDS in its cache table. After that request returns, the new coming MDS will be added to its cache table. The new coming MDS is found now. If a client accesses an MDS which has left, a connection error exception will be returned to it. It deletes the MDS from its cache table and retransmits the unsuccessful request again to the closest MDS in its cache table. Thus, the client lazily deletes the leaving MDS from its cache table.

3.3 Reliable DHT-MDSC (RDHT-MDSC)

DHT-MDSC can become useless if any metadata server fails. In order to improve its reliability, we have also implemented a reliable DHT-MDSC (RDHT-MDSC). RDHT-MDSC can tolerate the failure of any metadata server. Our design replicates each MDS' metadata to its successor and records some additional information to recover the ring structure of DHT-MDSC if broken. An MDS must record its N-SC and F-PD. Each successor of SC-F in its finger table also needs to be recorded. In the following we will illustrate how to reconstruct the ring structure when it is broken because of a failed MDS. If an MDS fails (no responses from ping message), its SC will receive a connection abort exception. These ping messages are maintained between two neighboring MDSes. There are two steps to handle the reconstruction process: reconstruct the neighboring relation and update the finger tables of affected MDSes. In the following, we call the SC of a failed MDS SF-MDS (successor of the failed MDS) and its PD PF-MDS (predecessor of the failed MDS). In step 1, the PD and F-PD of SF-MDS are changed to PF-MDS and its PD respectively. After that, SF-MDS sends an "update front-predecessor" message to its SC. Then, the F-PD of its SC is changed to PF-MDS. The same process is required for PF-MDS and its PD. In step2, SF-MDS sends failure messages to all affected MDSes stored in its finger table. When the failure message arrives at an MDS, it deals with the message (updating its finger table entry if needed) and forwards the message to its SC. The process terminates when any SC has received the same failure message previously. After all of the MDSes in SF-MDS's finger table return acknowledgements, the ring structure has been reconstructed and the MDSC can handle new requests as usual again. The time it takes to

reconstruct a ring is called “reconstruction time”. In order to tolerate future MDS failure, we need to duplicate each MDS’s metadata to its SC. The time taken to copy metadata depends on the size of metadata stored in an MDS.

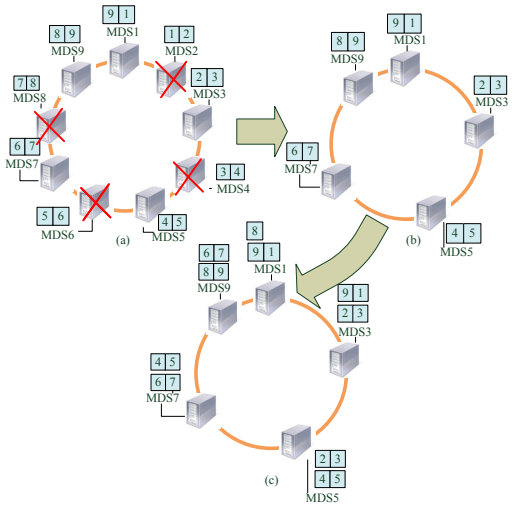


Fig. 2 Reconstruction of RDHT-MDSC

Fig. 2 shows the metadata relocation processes of our RDHT-MDS. Originally, there are nine MDSes with their metadata represented on top of them. Each metadata has a number that represents where it comes from. Now suppose MDS2, MDS4, MDS6 and MDS8 fail simultaneously, the metadata controlled by them can be found on MDS3, MD5, MDS7 and MD9 respectively. Therefore the system can tolerate multiple failed MDSes and continue to function. However, currently the system can not tolerate the failure of two side-by-side metadata servers since their metadata can not be recovered elsewhere. We feel it is less likely that two neighboring nodes fail simultaneously. Finally, all of the survival MDSes will transmit their metadata to their SCs in order to tolerate future failure of metadata servers as shown in Fig. 2 (c). Therefore the system can tolerate not only the usual failure of one MDS but also the simultaneous failures of multiple MDSes as long as no side-by-side metadata servers fail simultaneously.

4 Performance Evaluations

In this section we study the performance of DHT-MDSC and RDHT-MDSC. Even though we think that hash key based system has better performance than traditional subtree system, it is important to quantify their performance gap. We use five IBM e-servers as metadata servers (MDSes), five Supermicro 1UTwin servers as clients, and connect them with 3com Gigabit Ethernet. The hardware configuration of MDSes and clients is listed in Table 1. All of them use Windows Server 2003 operating system.

Table 1 Hardware configuration of MDSs and clients

Metadata Server	CPU	One Intel Xeon CPU 2.8G
	Memory	512MB DDR 266
	NIC	Broadcom NetXtreme Gigabit Ethernet
Client	CPU	Two Xeon CPU 2.33G (Core2 Quad)
	Memory	2GB DDR2-667 X 2
	NIC	Intel(R) Pro/1000 EB Gigabit Ethernet
Network connection	Gigabit switch	3C16479 3Com 4226T 3C17300 Switch

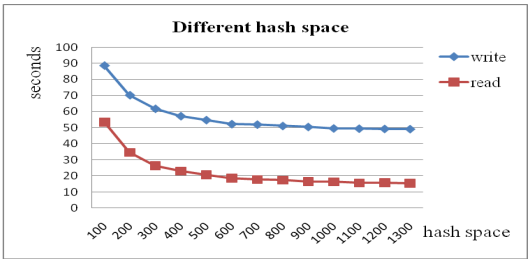
In order to select a suitable hash space for our tests, we measure read and write latencies for a hash space ranging from 100 to 1300. We use one IBM e-server as an MDS and one Supermicro 1UTwin with eight physical CPU cores on it as clients. Therefore eight client threads can run on eight different physical CPU cores. Each client issues 25,000 metadata write requests (each with different file pathname), and then read them back. Once an operation is issued, the client must wait until the acknowledgement returns. The results are shown in Fig. 3. Fig. 3 (a) plots the write and read latency as a function of hash space size. Fig. 3 (b) plots the write and read latency as a function of average linked list length corresponding to each hash space size. The average linked list length is computed as total metadata numbers (8 x 25,000 = 200,000) divided by the specified hash space. Therefore, as shown in Fig. 3 (b), when the average length of each linked list is less than 500, the time required to search the linked list is insignificant. Hence, the length of linked lists used in all of the measurements is less than 500.

To study the performance and scalability of the system without caching, we disable LC-RIC on DHT-MDSC in the following measurements. The range of hash keys controlled by each server is the same. We run eight client threads on each Supermicro 1UTwin, so the number of clients in our test bench is between 8 and 40. Each client issues 12,500 metadata write requests (each uses a different file pathname), and then read them back. We do the measurement 10 times and average the total time taken on each client. In this test the hash space is set to 1024 and the average length of each linked list is 488. As we can see from Fig 4, the result is disappointing and distressful since both write and read latencies increase as more metadata servers are used. This is opposite to what we see in Fig. 5 (to be described in the next paragraph) where caching (LC-RIC) is used. This is because every metadata request must go through the routing path to reach destination metadata server. Hence, metadata servers are always busy in relaying requests on the ring network. Furthermore, the more metadata

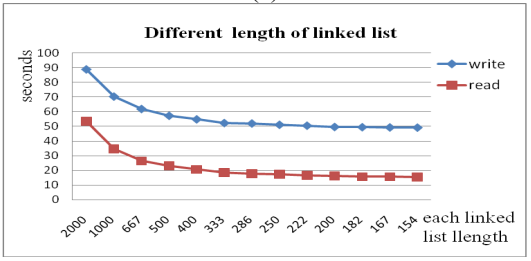
servers we add, the longer routing path requests must go through. Therefore, the performance is negatively impacted as more metadata servers are used.

The scalability for our DHT-MDSC and RDHT-MDSC are also studied as follows. All of the settings are the same as described in the previous paragraph except LC-RIC is turned on here. In Fig. 5, we can see that the performance of our DHT-MDSC scales very well as the number of metadata servers is increased. When the number of clients is 40, both write and read performances improve by a factor of about 4.5 as we increase the number of metadata servers from one to five. This is in contrast to the results shown in Fig. 4 where LC-RIC is turned off. Therefore, LC-RIC is a critical mechanism to make DHT-MDSC scale with the number of metadata servers. Both the write and read latencies of using only one MDS increase nonlinearly when more clients are added. As the number of clients is increased, more requesting packets are generated and sent to the MDS. However, the server interface card may not be able to handle all of them and some packets may get dropped. This is relieved when more MDSes are used.

Fig. 6 shows the write and read performances of our RDHT-MDSC. For write performance using two metadata servers, the latency grows nonlinearly when the number of client is larger than 24. We can use this result to suggest the number of MDS that should be used in the system if write performance is a critical design factor. For example, if the number of clients is 24, then at least two metadata servers should be used if write performance is critical. The read performance is almost the same as in DHT-MDSC due to the same read mechanism.



(a)



(b)

Fig. 3 (a) Write and read latency vs. hash space size, (b) latency vs. length of linked list

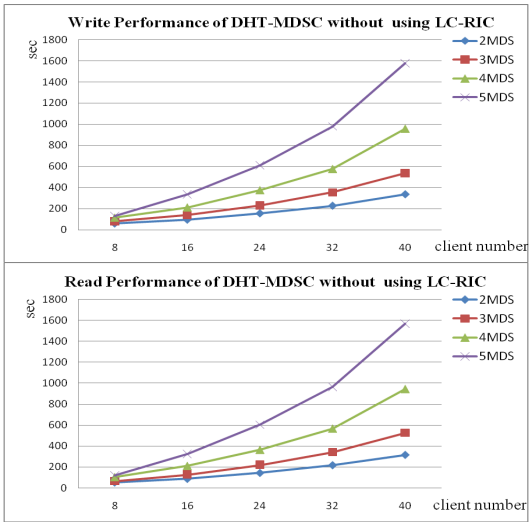


Fig. 4 Write and read performance of DHT-MDSC without using LC-RIC

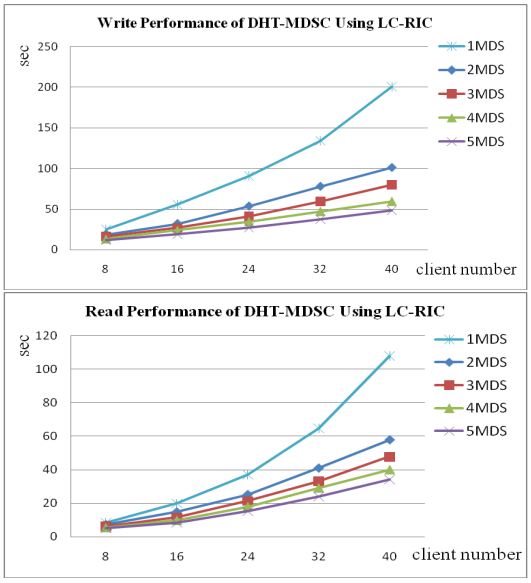


Fig. 5 Write and read performance of DHT-MDSC using LC-RIC

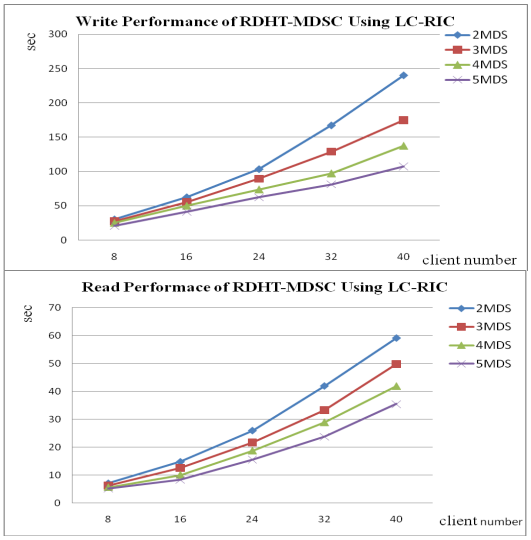


Fig. 6 Write and read performance of RDHT-MDSC using LC-RIC

In order to measure the "reconstruction time" described in section 3.3, we use five IBM e-server as metadata servers (MDSes). One of them is shut down suddenly in the middle of processing requests. The reconstruction time is equal to end time minus start time. Start time is initialized in SF-MDS once the ping message is not returned by the failed MDS. After the process of updating the failed MDS's neighbors, SF-MDS sends failure messages to those MDSes in its finger table. After all of them return acknowledgements to SF-MDS, the end time is recorded. The "reconstruction time" of recovering from one MDS failure in our RDHT-MDSC is shown in Fig. 7. It can be seen that the metadata server cluster can reconstruct itself in less than 18 milliseconds and be ready to service requests from clients again.

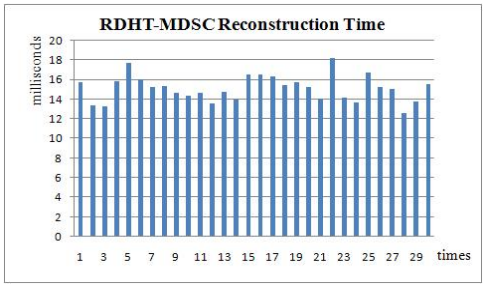


Fig. 7 The reconstruction time of RDHT-MDS when one MDS fails

5 Conclusions

In this paper we present the design and implementation of our DHT-MDSC and RDHT-MDSC. The architecture supports a scalable, reliable, and high performance metadata file system. The system is also very flexible to allow an MDS to join to or leave from the metadata server cluster (MDSC). Therefore, the system does not need to be shut down for hardware upgrade, periodical check or planned maintenance. LC-RIC is a smart and efficient caching mechanism to significantly improve the performance of RDHT-MDSC. The reconstruction time of RDHT-MDSC is less than 18 milliseconds. The other advantage of RDHT-MDSC is that it can recover itself from the failure of multiple metadata servers as long as no two side-by-side metadata servers fail simultaneously.

6 Acknowledgements

The authors would like to thank the support from National Science Council under grant 96-2221-E-007-131-MY3.

References

[1] D.Roselli, J.Lorch, and T.Anderson, "A Cmparison of File System Workloads," *Proceedings of the 2000 USENIX Annual Techical Conference*, pp. 41-54, Jun 2000.

[2] Scott Brandt, Ethan L. Miller, Darrell D. E. Long, and Lan Xue, "Efficient Metadata Management in Large Distributed File Systems," *NASA/IEEE Symposium on Mass Storage Systems and Technologies (MSST 2003)*, pp. 290–298, San Diego, California, April 7–10, 2003.

[3] Zhou Gongye, Lan Qiuju and Chen Jincai, "A Dynamic Metadata Equipotent Subtree Partition Policy for Mass Storage System," *Frontier of Computer Science and Technology FCST*, 1-3 November, 2007. Los Angeles: *IEEE Computer Society*, 2007.29-34

[4] Jin Xiong, Rongfeng Tang, Sining Wu, Dan Meng, Ninghui Sun, "An Efficient Metadata Distribution Policy for Cluster File Systems," *IEEE International Conference on Cluster Computing (Cluster2005)*, September 26-30, 2005, Boston, USA.

[5] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, Hari Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications," *IEEE/ACM Transactions on Networking*, Vol. 11, No. 1, pp. 17-32, February 2003.

[6] Carns, P.H., Ligon III, W.B., Ross, R.B., Thakur, "PVFS: A parallel file system for linux clusters," *4th Annual Linux Showcase and Conference, Atlanta, GA*, pp. 317–327, 2000

[7] Mesnier, M., Ganger, G.R., Reidel, E., "Object-Based Storage," *IEEE Communications Magazine* 41(8), 84–90, 2003

[8] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz, "NFS version 3: Design and implementation," *Preceedings fo the Summer 1994 USENIX Technical Conference*, pp. 137-218, Apr. 2003.

[9] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. H. Rosenthal, and F.D. Smith, "Andrew: A distributed personal computing environment," *Communications of the ACM*, 29(3):184-201, Mar, 1986.

[10] P. F. Corbett and D. G. Feitelson, "The Vesta parallel file system," *ACM Transactions on Computer Systems*, 14(3)225-264, 1996.

[11] P. Braam, M. Callahan, and P. Schwan, "The intermezzo file system," *Proceedings of the 3rd of the Perl Conference,* O'Reilly Open Source Convention, Monterey, CA, USA, Aug, 1999.

[12] Chord: http://pdos.csail.mit.edu/chord/