# A Content-Based Load Balancing Algorithm for Metadata Servers in Cluster File Systems*

Junho Jang, Saeyoung Han, Sungyong Park, and Jihoon Yang

Department of Computer Science and
Interdisciplinary Program of Integrated Biotechnology,
Sogang University, Seoul, Korea
{jj, syhan, parksy, yangjh}@sogang.ac.kr

**Abstract.** A metadata service is one of the important factors to affect the performance of cluster file systems. We propose a content-based load balancing algorithm that dynamically distributes client requests to appropriate metadata servers based on the types of metadata operations. By replicating metadata and logging update messages in each server rather than moving metadata across servers, we significantly reduce the response time and evenly distribute client requests among metadata servers.

## 1 Introduction

It is reported from SPEC that up to 60% of user requests in cluster files systems are metadata operations [1]. Due to the large amount of metadata operations, some cluster file systems use a separate metadata server or a cluster of metadata servers for scalability and availability [2][3][4][5].

A key question in the design of such systems is how to partition the metadata among metadata servers to maintain both high performance and scalability. The first approach, known as directory sub-tree partitioning, partitions the metadata along the directory sub-tree, which suffers from severe bottleneck due to the hot spots. As an alternative, a pure hashing approach [2] is introduced. This approach hashes the filename to distribute the namespace among the metadata servers evenly. This requires metadata servers to maintain the directory hierarchy, and further requires them to repartition the namespace among the servers whenever a metadata server is added or removed from the cluster. Another approach such as Lazy Hybrid (LH) [3] combines both approaches to address the problems above. However, all the approaches above are based on the static mechanism such that a metadata server is designated when a new metadata structure is created. This prevents client requests from being distributed fairly among the metadata servers based on current load conditions.

This paper proposes a content-based load balancing algorithm for metadata servers that dynamically distributes client requests to appropriate metadata servers based on the types of metadata operations. In order to distribute client requests dynamically, a dispatcher is used. In addition to distributing client requests dynamically, the

* This work was supported by grant No. R01-2003-000-10627-0 from the Basic Research Program of the Korea Science & Engineering Foundation.

Y. Pan et al. (Eds.): ISPA 2005, LNCS 3758, pp. 49 – 57, 2005.
© Springer-Verlag Berlin Heidelberg 2005

dispatcher also shares Indirect Metadata Table (ITL) with all the metadata servers and adjusts assigned entries among metadata servers, reflecting current load conditions. Although the capacity of the dispatcher is critical to the overall cluster system performance, emerging hardware technologies for switching reduces the relaying overhead significantly, which ensures us to assume sufficient capacity of dispatcher.

The rest of this paper is organized as follows. In chapter 2 we present an overview of metadata management schemes used in cluster file systems. Chapter 3 presents the detail mechanism of content-based load balancing algorithm. Its analysis and experimental result are presented in chapter 4. Chapter 5 summarizes our work and concludes this paper.

## 2 Related Work

The first approach to allocating metadata among metadata servers in cluster file systems is the hierarchical directory sub-tree partitioning. This approach partitions the file system namespace according to the structure of directory sub-tree and the metadata of each directory sub-tree is managed by individual metadata server. This technique suffers from severe bottleneck when a single file, directory, or directory hierarchy must be traversed to determine the permissions of each file that is accessed.

The second approach, pure hashing, distributes the namespace among metadata servers by hashing the file identifier, file name, or other related values. This results in more balanced workloads than directory sub-tree partitioning. Vesta parallel file system [2] is a representative method of pure hashing. The hash function of Vesta file system uses the full pathname as an input key, and outputs the identifier of the metadata server and the location of the metadata inside the server. This pure hashing guarantees direct accesses to metadata without traversing all the metadata servers along the directory hierarchy, but it does not support the directory path-based file permission using access control list. Moreover, for some expensive operations such as changing directory name, removing directory, and adding or removing of metadata servers, a large number of metadata should be moved across metadata servers, which leads to long response time and clients should wait for a long period of time for their requests.

Lazy Hybrid (LH) [3] addresses the above problems by combining the advantages of both approaches and adding capabilities such as global logging and delayed updates. The metadata location is determined by hashing the full pathname, which allows direct accesses to the metadata without traversing all of the metadata servers that stores directories along the path. However, hierarchical directories are maintained in order to provide standard directory semantics and operations such as *ls*. Lazy update policies allow for efficient metadata updates when the file/directory names or their permissions are changed or when metadata servers are added to or removed from the system. Moreover, a dual-entry access control list structure is maintained for any file permissions to be determined directly without traversing the entire path. When a large amount of metadata has to be moved at a time, the real location is globally logged in all the metadata servers, instead of moving metadata. Later, upon the first access after global logging, the metadata is actually moved. By using the delayed updates, the initial operation is very fast and only a little overhead is incurred at the

time when each of the modified metadata is accessed first. On the other hand, when the requests generated by the clients are bursty, this scheme leads to the concentration of the requests on a particular metadata server holding the real metadata, and suffers from the performance degradation due to the overhead incurred by forwarding client requests.

To address these shortcomings due to the static determination of metadata servers on each client, we propose a dynamic load balancing algorithm based on a dispatcher. The dispatcher periodically collects load information from the metadata servers and forwards client requests to appropriate server based on the content of each request.

## 3 Content-Based Load Balancing Algorithm

In this section, we present the detailed schemes used in the content-based dynamic load balancing algorithm.

### 3.1 Architecture

Fig. 1 shows the structure of the metadata server cluster. This cluster consists of several metadata servers and a dispatcher that relays the request from clients to appropriate metadata servers. Given the information of the file included in a request, the dispatcher hashes the full pathname of the file to produce a hash value indicating the index into the Indirect Lookup Table (ILT). The index found in the entry of the ILT specifies which metadata server currently stores the metadata for that file. After determining appropriate metadata server, the dispatcher forwards the requests to the selected metadata server or broadcasts it to all the metadata servers depending on the content of the request. The detailed operations will be described in the next section.

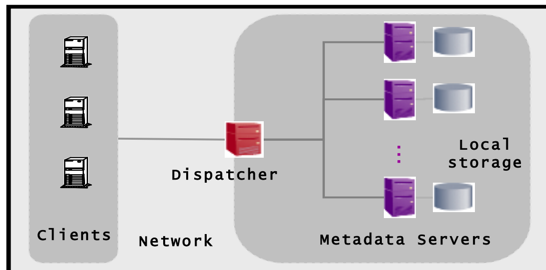


**Fig. 1.** Architecture for load balancing

In this architecture, all the metadata servers and the dispatcher should share the same ITL as well as the same hash function. Using these, each metadata server determines independently whether it is responsible for the requested file or not, and then stores, retrieves, or modifies the metadata of the file. Moreover, each metadata server caches the inode information of all the files and directories, and stores the directory hierarchies in order to improve the performance of metadata operations.

In order to efficiently distribute the load among metadata servers, all metadata servers report their load conditions to the dispatcher periodically. Based on this information, the dispatcher adjusts the ILT and then redistributes it to all the metadata servers.

### 3.2 Metadata Operations

To ensure the consistency of metadata among metadata servers, our algorithm writes and logs metadata write operations on every metadata servers. Since our algorithm uses a full pathname as an input into the hash functions, some operations, such as changing directory name, adding or removing of metadata server, and ITL adjustment, result in a large amount of metadata movement across the metadata servers. To reduce the overhead incurred by moving metadata, we replicate metadata among all the metadata servers, and log all the metadata modification messages. While the requests such as simply looking up metadata for files or directories are handled by one designated metadata server, the requests for writing metadata or logging some operations are broadcast to all the metadata servers concurrently. As a result, all the metadata servers have the same metadata information. For some retrieval operations for directories or file attributes that require metadata modification (i.e., update “last access time” field), we divide the operations into two steps: looking up metadata and updating the “last access time” field.

When a file or a directory needs to be retrieved, the dispatcher uses a hash function (using the full pathname) to locate the appropriate metadata server in constant time and ask the designated server to reply with the metadata information related to the file or the directory. The modification message for the “last access time” field is then broadcast and all the servers update and log the information. On the other hand, except for the operations related to the attribute manipulation, all the metadata operations related to changing directory structure require the modification of directory hierarchy in addition to updating inode information. For example, the directory removal operation requires the deletion of all the subdirectories. Changing directory name should rearrange all the metadata for the files, subdirectories, and the files under the subdirectories across the metadata servers since the hash values need to be changed.

It should be noted that changing the directory hierarchy requires the movement of a large amount of metadata. In our approach, each metadata server is supposed to execute the operation at the same time and thereby eliminate the movement of metadata. Considering that the file system operations are mostly read operations (with the ratio of 9:1 in office environments), replication is much more reasonable than metadata movement in general cluster file system environment [8].

Unlike the directory write operations, the writing operations for files do not require any modification of the directory hierarchy. However, they are also carried out concurrently at each metadata server to ensure the metadata coherency.

### 3.3 Adjustment of Indirect Lookup Table (ILT)

Since each file system operation requires different amount of computational power and each file has different access frequencies, some metadata servers may be overloaded more than the others. This may cause longer response time and decrease overall system

performance. Moreover, since the entire metadata server may not have the same computing power, we should adjust the imbalance through reconstructing the ITL.

The goal of our algorithm is that all the metadata servers have similar load conditions approaching to the average load and minimize the change of designated metadata server. In order to do this, our algorithm should first determine the metadata servers whose load exceeds the overall average, and calculate the amount of extra load for each metadata server, $Extra(mds_i)$, by subtracting the average load from its own load. The metadata server with negative $Extra(mds_i)$ value can handle more metadata by assigning more ILT entries taken from the metadata server with positive $Extra(mds_i)$. In order to distribute the overloaded entries to other metadata servers, based on the load per entry $Load_e(mds_i)$, we determine the maximum number of ILT entries $EE_i$ for any overloaded metadata server $i$, satisfying that

$$Extra\ (mds_i) - Load_e\,(mds_i) \times EE_i \geq 0,$$

where $0 \leq EE_i \leq$ the number of ILT entries handled currently by $mds_i$.

Any metadata server $j$ with negative $Extra(mds_j)$ may take the entries from $i$ as many as maximum $EE_j$. That is, the following should be satisfied

$$Extra\ (mds_j) + Load_e\ (mds_j) \times EE_j \leq 0,$$

where $EE_j \geq 0$. In order to take the load more aggressively, we allow each metadata server with more available capacity than $Load_e\ (mds_j)\ /\ 2$ to take one more entry. Therefore, the above formula can be changed like this.

$$Extra\ (mds_j) - Load_e\ (mds_j) / 2 + Load_e\ (mds_j) \times EE_j \leq 0,$$

where $EE_j \geq 0$. Fig.2 shows an example of the adjustment of ITL so that all the metadata servers have quite evenly distributed load around the average load.

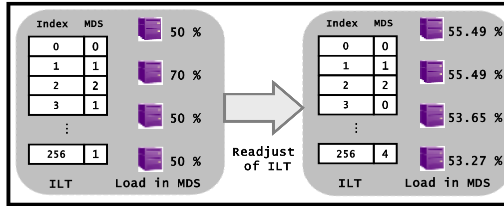


**Fig. 2.** Example of ILT Adjustment

# 4 Performance Evaluations

## 4.1 Experimental Environment

We evaluate our algorithm using CSIM 9.0, a process-oriented discrete-event simulator [8]. The simulations are performed on Intel Pentium-III (800 MHz dual CPU) running Linux Kernel 2.6. The detailed parameters are presented in Table 1.

In this evaluation, we measure the load of each metadata server to see how well the client requests are distributed. The average response time from the clients is also

measured. The ratio of read accesses and write accesses is 9:1. We evaluate our algorithm and compare it with those of Vesta and LH3.

**Table 1.** Parameters for the simulation

| The number of MDS | 8 |
|---|---|
| Metadata size | 256 Bytes |
| Average memory cache search time | 0.155 msec for 10MB |
| Memory cache hit ratio | 90% |
| Disk access time | 1.561 msec for 1 metadata |
| Network transfer time | 0.209 msec for 1 metadata |

### 4.2 Results

Figures 3 through 5 show the load condition of each metadata server for Vesta, LH3, and our approach, respectively. In order to obtain current load at each metadata server, we measure the number of requests waiting to be processed at each server for a period of 20,000 milliseconds.

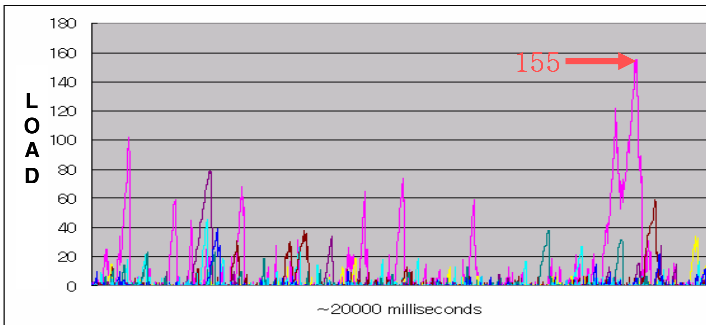


**Fig. 3.** Number of requests waiting for services in each metadata server (Vesta)

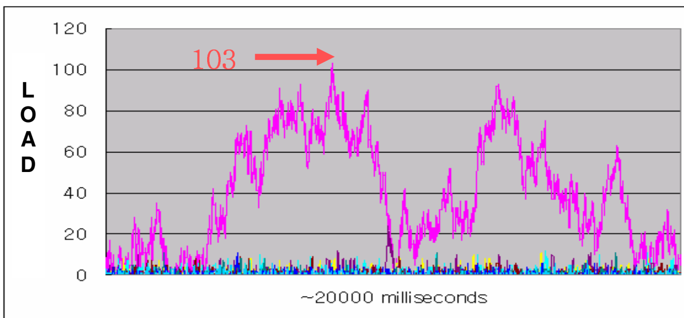


**Fig. 4.** Number of requests waiting for services in each metadata server (LH3)

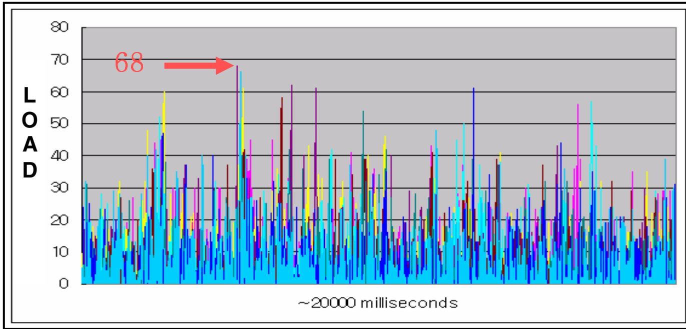


**Fig. 5.** Number of requests waiting for services in each metadata serer (proposed approach)

As you can see from Fig. 3 and Fig. 4 (Vesta and LH3 cases), for some of metadata servers, the number of waiting requests is much larger than those of the others. This implies that the client requests are forwarded heavily onto some metadata servers and the load is not fairly distributed among all the metadata servers. On the other hand, Fig. 5 (our approach) shows that the requests are well distributed all over the metadata servers. Moreover, while the average load of our approach is a little bit higher than that of Vesta, the variance is remarkably smaller (see Table 2). This also indicates that replicating metadata is more efficient for distributing client requests than moving metadata throughout the network.

Table 2 shows the average response time of all three approaches. As the table shows, our approach has minimum average response time although it doesn't include the processing time at the dispatcher. Under the assumption that we can implement the dispatcher with quite good performance, the processing time at the dispatcher can be ignored. Table 3 also shows that our approach significantly outperforms other approaches.

**Table 2.** Average numbers of requests waiting for services and the variances

| | Vesta | LH3 | Our approach |
|---|---|---|---|
| Average # of requests waiting | 2.11 | 4.35 | 3.13 |
| Variance | 10.56 | 95.10 | 0.14 |

**Table 3.** Average response time for each approach

| | Vesta | LH3 | Our approach |
|---|---|---|---|
| Average response time (msec) | 11.93 | 32.04 | 6.39 |

In order to explain the relationship between the performance of dispatcher and the response time of client requests, we introduce a formula using queuing theory. For example, the response time at the dispatcher R can be written as

$$R = \frac{1}{C - \lambda},$$

where C is the service rate at the dispatcher and λ is the arrival rate of client requests [10]. When λ is unchanged, the only factor that affects the response time is C. If C is much larger than λ, a dispatcher can forward the client requests to appropriate metadata server immediately on receiving a request. If C is approximately equal to λ but is not smaller than λ, the response time increases rapidly because of the processing delay at the dispatcher. If C is smaller than λ, the arrival rate of client requests exceeds the capacity of a dispatcher, and thereby the response time can't be measured.

Based on the fact described above, we measure the average response time including the processing time at the dispatcher. As you can see from Fig. 6, the response time increases exponentially as we increase 1/C values. The average response time of our approach is lower than those of Vesta and LH until 1/C is up to 0.8. However, our approach suffers from long response time when 1/C goes close to λ, which implies that the performance of dispatcher becomes the bottleneck of overall cluster system. On the other hand, we can expect performance improvement when the arrival rate of client requests is below 93% of service rate of the dispatcher in this experiment.

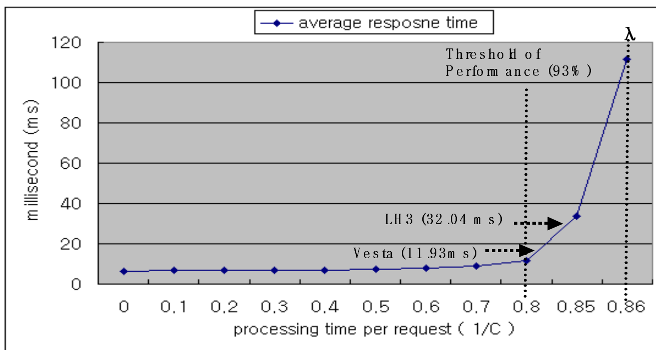


**Fig. 6.** Effect of the performance of dispatcher

## 5 Conclusion

In this paper, we have proposed a content-based load-balancing algorithm for metadata servers in cluster file system, where the client requests are handled differently according to their contents, and the loads of the metadata servers are redistributed by dynamically adjusting the indirect metadata table periodically. By replicating the metadata and logging update messages, all the metadata servers concurrently execute the update operations on metadata, which minimizes the metadata movements.

Through our performance evaluation, we have showed that our dynamic load balancing algorithm outperformed existing metadata management schemes used in traditional cluster file systems. We are currently investigating further about the effect of the performance of dispatcher on the overall system performance in the metadata cluster.

## References

1. SPEC, SFS 3.0 Documentation Version 1.0, Standard Performance Evaluation Corporation, 2001.
2. Peter F. Corbett et al., The vesta parallel file system, ACM Transactions on Computer Systems(TOCS), 14(3), pp.225-264, Aug. 1996.
3. Scott A. Brandt et al., Efficient Metadata Management in Large Distributed Storage Systems, Proceedings of the 11th IEEE NASA Goddard Conference on Mass Storage Systems and Technologies, Apr. 2003.
4. Peter J. Braam et al., The Lustre Storage Architecture, Cluster File Architecture, Cluster File System. Inc, Mar. 2003.
5. Jin Xiong et al., Design and Performance of the Dawning Cluster File System, IEEE International Conference on Cluster Computing(Cluster'03), Dec. 2003.
6. Bourke T., Server Load Balancing, O'Reilly and Associates, Sebastopol, 2001.
7. Daniel P. Bovet et al., Understanding the Linux Kernel, O'Reilly and Associates, Sebastopol, 2003.
8. http://www.mesquite.com