MDS Functionality Analysis

Lan Xue, Yong Liu

{lanxue,lyong}@cse.ucsc.edu

December 4, 2001

TABLE OF CONTENTS

1 INTRODUCTION

1.1 PURPOSE

This document describes the functions of Metadata Server (MDS) of the Object Based Storage Device (OBSD) Project.

The rest of report is organized as follow: section 1 describes the project briefly. Section 2 focuses on the analysis of MDS functions, problems and possible solutions. These issues are discussed in details starting from Section 3, which talks about the system access control. Section 4 addresses cache coherency. Section 5 discusses the system scalability. Section 6 describes versioning. Section 7 focuses on data availability, and Section 8 talks about metadata allocation among MDS cluster. Section 9 concludes and Section 10 is the acknowledgement.

1.2 PROJECT DESCRIPTION

The goal of the research is a storage system architecture that supports both high-speed capture of data from massive simulations and the interactive use of this data by investigators via visualizations [1]. Results of this research will be validated and demonstrated by creation of prototype implementations of this architecture [1].

The research is proposed to improve both file system performance and functionality by building a storage system from object-based storage devices (OBSDs) connected by high-speed networks. As a new storage strategy, the OBSD defines a new boundary subdividing the responsibility of File System. Shown in Figure 1, in traditional File System Model, file system deals with very detailed storage information of the data it wants to access, which is a heavy load to the file system and makes the data transfer between file system and storage device slow and problematic. In the OBSD Model, the storage devices have the ability to manage the data stored by themselves. The file system uses a high-level, object based interface to communicate with the OBSD. This strategy makes the communication between file system and storage devices much faster and easier to manage.

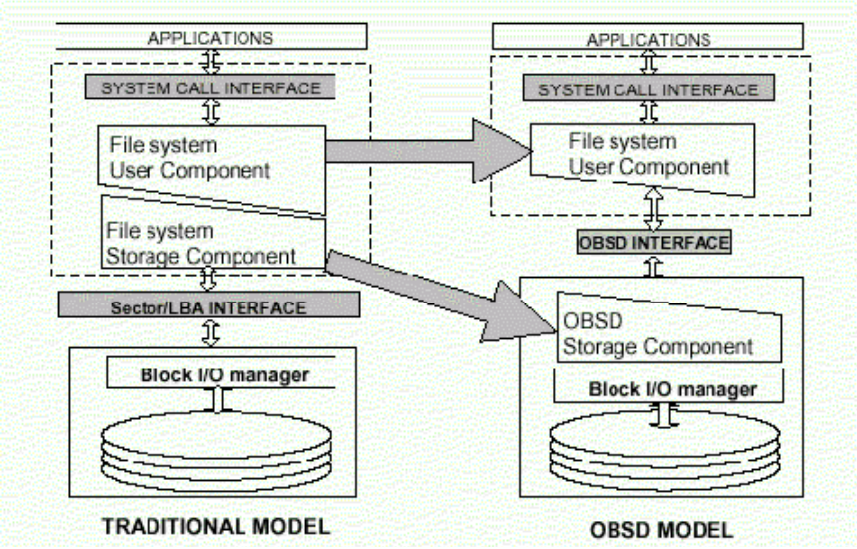

Figure 1. OBSD: A new boundary subdividing File System [2]

Figure 2 shows the big picture of the components. The data flow is illustrated in the Figure 3, and will be described in details in the rest of the proposal.

1.3 PERFORMANCE IMPROVEMENT

- The server is no longer the bottleneck of the system, since the system now is different from the traditional systems in that the server only deals with open/close requests from the client, leave the client talks directly to the storage devices with the more frequent read/write requests.
- Each OBSD is free to re-allocate its data
- File system does not need to care the storage details, e.g, how the data is allocated on the storage devices, what is the size of each block, etc.

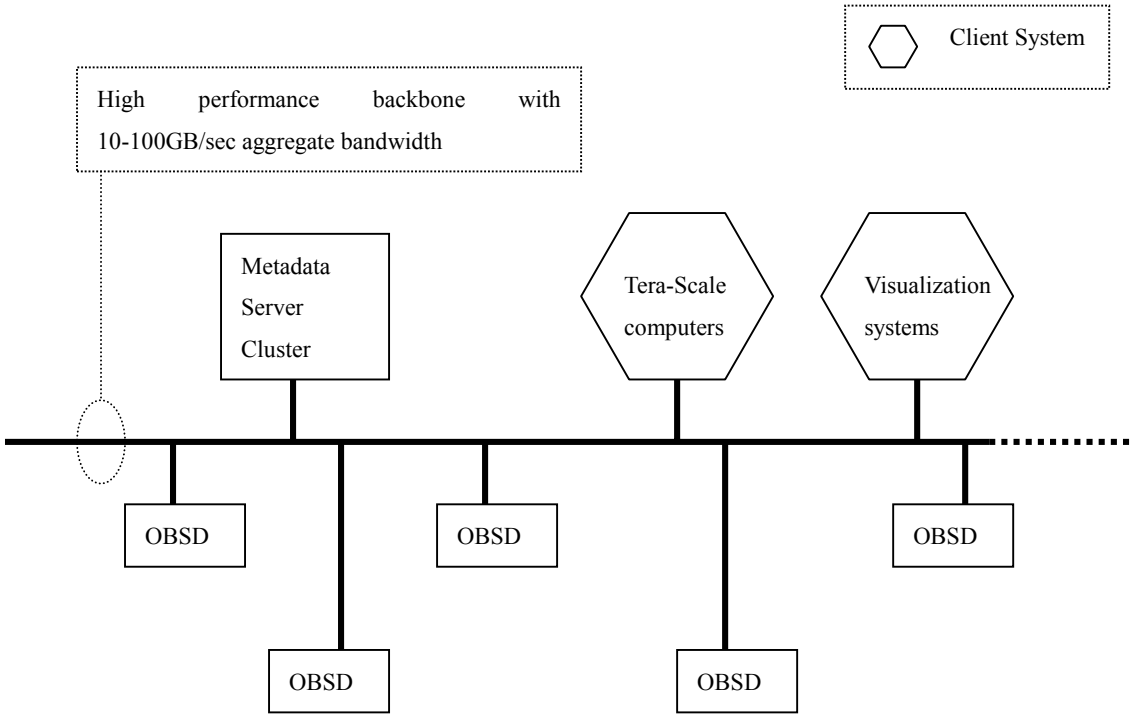

Figure 2. Architecture Overview [1]

2 METADATA SERVER FUNCTIONALITY

2.1 IMPORTANT ASSUMPTION

- The OBSD is always has enough free space for new data so that data can always be striped evenly among the OBSDs. There is no situation that any one of the OBSDs is full when data come.

Note: The assumption is an important design issue for OBS device, and will be addressed in another designing report about OBS devices.

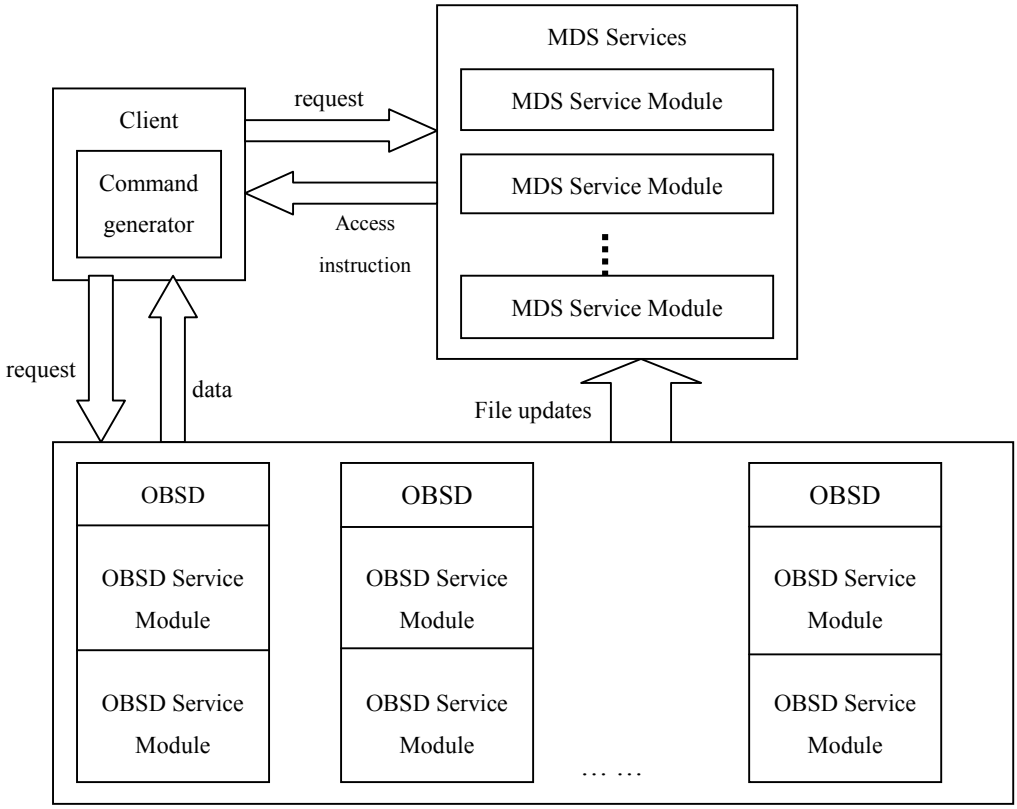

Figure 3. Function module strutures

2.2 FUNCTION MODULES FOR MDS

1. Access Control Service (ACS): Authorize the valid client, give permission based on the security policy. The security policy may be the policy that is used in the UNIX file system, which is per file access control based on users ID and user groups. But there may be other security policy options and the system should give the freedom to the user to choose the appropriate policy.
   Issues to solve in ACS:
   - Permission change
   - Permission expiring time
   - Permission encryption
2. Cache Coherency: Cache is important in that it improves the access time. Cache may exist in the clients, OBSDs, and MDSs. Cache coherency is especially hard and important for this architecture because data and metadata are stored separately on OBSDs and MDSs. So the caches in different places may have different cache content (data or metadata), and whenever update happens, the cache includes that part of data or metadata should be notified and invalidated.
3. System Scalability: The system is scalable only if it is easy to add or delete machines to the system. In this particular architecture, both of the numbers of OBSDs and MDSs should be free to change.
4. Versioning: Versioning is important in that it protects the users' data from losing.

Keeping the old data before each update, the system keeps the 'history' of the users data and thus users can retrieve any old 'version' of their data. It is no wonder that this will cost lots of storage space. Fortunately, the rapid increasing in the storage capacity and the decreasing in storage price makes versioning possible.

5. Data Availability: This is an important metric to evaluate the systems' robustness. The data and the metadata should still be available whenever there is some storage devices or MDS down in the system. In other words, single or small amount of storage devices or servers failure should not cause the data unavailable to the users.
6. Metadata Allocation: This is the most difficult issue of the design. To distribute metadata among servers in the MDS cluster, the allocating method must be able to distribute the workload evenly, avoiding any possible hot spot, reasonable and easy for the clients to locate and allocate their metadata on the server, and possible to scale the MDS cluster. The metadata will not be striped across the servers, as the data.
7. Sharing mechanism (to be finished): provide a sharing mechanism (could be file based sharing or block based sharing). But should it be the responsibilities of MDS or OBSD? Since client does not talk with MDS about read/write operations, it's hard for MDS to decide when to apply locks. Moreover, it may cause deadlock if the clients who hold the lock suddenly down, without freeing the locks to the system, so that the resources being locked is not usable until the system find out deadlock happen and frees those resources. So locking is not suitable here. Instead, leasing is a good way, the clients have the accessing right to the resources during the lease, and write back any modification before the lease ends. It is good in that even if clients shut down accidentally, the resource will still be available when the lease expires. There's no need for the system to take care of the possibility of the permanent unavailable resources.
8. Encryption (future work): encode important data transferred to the client or OBSDs

3 ACCESS CONTROL

3.1 INTRODUCTION TO THE PROBLEM

In OBSD System, each time a client accesses a file, first it will send access request to certain MDS to get permission through its agent. If access is granted, the client receives some location information together with the appropriate capability to access the file. Using the permission, a client can create a virtual secure channel with OBSDs, in which the client directly interacts data with the OBSDs according to the allocation information.

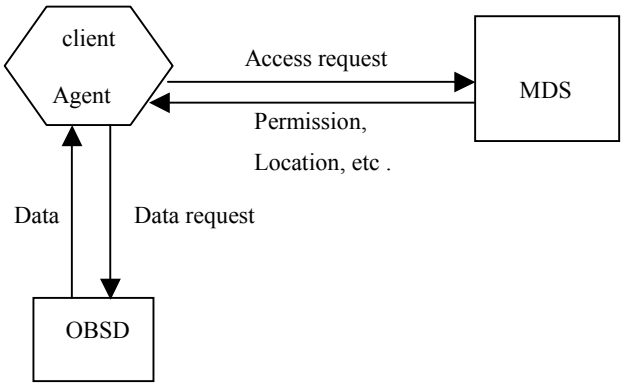

Figure 3.1. Access Control model

3.2 SECURITY POLICY IN THE OBSD SYSTEM

According to the requirement of different workloads or file types, we can use different access control models and permission policies.

Corresponding to the access control model in Figure 3.1, we provide a simple and efficient permission policy to implement it (Encryption is not considered here):

- For each pair of open/close operations to a file, a client is only required to get permission when opening. After this point, the client can directly access the file in OBSDs using the proper capability until the file is closed.
- Access Control Bits (ACB) is used to control the access capability. ACB is included in each request. If an access to a certain file is granted, MDS will update the ACB correspondent to it . Hence, whenever the client accesses the file in OBSDs, OBSDs will check the correspondent ACB to authenticate it.
- If the permission for a client to access a certain file changes during a single pair of open/close operation, MDS will inform the agent of this client to change its capability. So to adapt the change, the granularity of the permission verification in OBSDs is per-operation. The problem is, what is the agent is a bad-behaved one and it refuses to change its ACB? How to provide a way for the OBSDs so that they know the client is using an out-of-date permission code? Obviously, it is not feasible for the MDS to notify the OBSD each time it changes the permission. A possible way is to use a time stamp on each permission, and a mechanism for the OBSD to decide the permission expiring time.
- Permission type: read or write; time-limited or no time-limited; etc.

Some more complicated access control models such as the Encryption Key Policy are in consideration.

4 CACHE COHERENCY

4.1 INTRODUCTION TO THE PROBLEM

To decrease the speed gap between CPU and storage devices, an OBSD system uses cache to improve the general read/ write performance as other file systems do. However,

using cache will be involved in some important issues such as cache consistency, and replacement algorithm.

Due to the differences between the architecture of OBSD systems and that of other file systems, cache consistency becomes an outstanding problem in OBSD systems. In other existing file systems, we only need to consider cache consistency between clients and servers, but in OBSD systems cache consistency may exist among clients, MDSs and OBSDs. And things are even more complex because data and metadata are stored separately in the system.

4.2 CACHE POLICY IN THE OBSD SYSTEM

According to the access control model of OBSD System, we implement the cache consistency as follows:

- When a client wants to open a file, it has to get permission and the metadata of the file from MDS first. Since the client uses the metadata to allocate data in OBSDs, it assumes that the version of the metadata the client got is the up-to-date version (by default the MDS will return the metadata of the latest version to the client, otherwise the client should specify the version it wants). So any time the location information of the file is updated, the client should inform MDS at once and keep the metadata of the file up-to-date [3]. The client can get the metadata from MDS's cache but the MDS must take full responsibility of making its cache holds the newest version of metadata.
- When a client read/write a file, after getting the newest version of the metadata, it checks whether the file is in its cache and its version is up-to-date. If so, it just gets the file from its cache; if not, it will send request to OBSDs. Henceforth, OBSD will also check if the fragment of this file is in its cache and its version is up-to-date. If so, it just gets the fragment from its cache; if not, it will get the fragment from its disk. After getting data, OBSD sends back the required data to the client. Especially for write operations, the client must notify MDS the updated metadata when it closes the file.

4.3 SOME IMPORTANT NOTES ABOUT CACHE POLICY

- Note that the MDS caches metadata information, while client and OBSD cache both metadata and data information.
- Note what happens when the metadata is updated: it is the client's responsibility to notify the MDS of this change. Why? Because it is easier and can reduce lots of extra traffic. But this moves some functionalities of the file system to the client agent, which may not be good.
- When the data is updated, the client and OBSD invalidate their caches, respectively, if necessary.

5. SYSTEMATIC SCALABILITY

5.1 INTRODUCTION TO THE PROBLEM

Some file systems use Hashing method to implement data striping and allocate data. However, when adding some new storage devices, traditional Hashing policy will be hard to adapt this change.

If the number of OBSDs changes, it is infeasible to just simply move the affected parts of all files according to the new Hash function because of too many costs. To keep the OBSD system go ahead with little intervention and reduce the cost of this change as possible as we can, we provide two possible solutions to it. The real performance of these solutions will be determined by many simulation experiments.

5.2 FRAGMENT MAPPING

Since the Hashing method can not adapt the change of the number of OBSD devices directly, we use the mapping policy to avoid the scalability problem of Hashing. The mapping policy will just use the simple sequential method to implement data striping and data allocation, and try to make the data of a file be distributed in each OBSD equally. It works as follows:

MDS holds the fragment-mapping information for each file. When a client sends the open request for a file to MDS, MDS feedback the fragment-mapping information of this file to the client besides the permission key, then the client can use the mapping to allocate the data directly and read/write; when the client closes the file, if the locations of some fragments of the file are changed, the correspondent Agent of certain client will inform specific MDS to update its fragment-mapping. Figure 5.1 shows an example of the fragment-mapping table.

Frag1	Frag2	Frag3	Frag4

(a) The fragments components of a requested file

Fragments	Frag1	Frag2	Frag3	Frag4
OBSD No.	1	2	4	5

(b) The fragment-mapping information of the file

Figure 5.1. The format of the fragment-mapping

The mapping policy has two apparent advantages. First, it simplifies the clients' operations so that the clients do not need to care the allocation policy. Second, it has good scalability because the change of OBSDs does not affect the existing mapping information in MDS. However, it brings many communication costs between clients and MDS.

Especially for the workloads in which write operations account for the primary parts, the costs may be unacceptable. In addition, the overhead to maintain the mapping table is also huge.

5.3 A SELF-ADAPTIVE HASHING

When striping the data, traditional Hashing policies use certain Hash function to pseudo-randomly distribute the file data to different storage devices, which can achieve good workload balance and provide rather high effectiveness in allocating data. However, traditional Hash policies often assume that the storage devices are not changeable because the module of the Hash function is determined by the number of storage devices. So if the number of storage device is changed, Hash function can not correctly allocate the existing files or adapt the change of the existing files in the future.

It is unbelievable if we just move all the necessary data to adapt this change since the volume of the existing files is huge. If instead of equal to the number of storage devices, we make the module of the Hash function very large (much larger than the current number of storage devices), it still can not solve the problem. For example, say we make the module to 100, and now the system only has 40 OBSDs. So the positions for these 40 OBSDs are from 0 to 39, respectively. When there are some data whose Hash value is 40, but there is no OBSD in that position, then we have to re-Hash the data until it finds a place with OBSD. So in face, the large module does not help us, it is still the number of actual OBSDs in the system that takes effect.

To reduce the costs of adaptation and continue to exploit the high effectiveness of Hash function, we propose the self-adapt Hashing policy to solve this problem of scalability. Its main idea includes two aspects: First, keep two different (an older and the newest) versions of Hash function in the metadata of each file; Second, re-allocate the data of the existing files on-demand. Its work mechanisms are described as the following:

- Whenever a new OBSD joins the system, it notifies the MDS cluster. So the MDS cluster always keeps the most updated OBSD number.
- MDS holds the information for the different versions of Hash functions, which may be like Figure 4.
- The metadata of each file has two bits to keep the version No. of Hash function. One (NewVer) keeps the newest version, the other (OldVer) may keep an older version, but not necessarily to be. When system initiation, the two bits are the initial version.
- Any time a client requests a file, MDS will check the two bits in the metadata of this file; if NewVer is less than the newest Hashing version of the system, it is

replaced by the newest version No.

- The client will check the two variables when it gets the feedback from the MDS. If OldVer is different from NewVer, it then knows the system scales since its last modifying the file.
- If the client creates a new file, it just uses the NewVer given by MDS to allocate the data.
- If it is a 'read' operation, or 'write' the file but no change to the allocation of the file's fragments, the client will just use the oldest version of Hash function to allocate the data.
- If any client writes the file and changes the distribution of the file in OBSDs, for example, create new fragment for an existing file, or the fragment of the file splits, then it uses the newest version of Hash function. The client needs to reallocate all the data of the file to form a new distribution. Henceforth, the client informs MDS to change OldVer in the metadata of the file as NewVer.

Num. of OBSDs	5	7	8
Version No. of Hash	1	2	3

Figure 4. Versions of Hash function maintained by MDS

Although in many workloads write operations make up the main composition, many of them are concentrated on some specific data [4]. Moreover, after a few days users created a file, they will seldom update the file and just read it [4].

So according to the characteristics of file access, we guess that self-adapt Hashing can use the least costs to implement the scalability smoothly.

6 VERSIONING

6.1 INTRODUCTION TO THE PROBLEM

In a traditional file system, users control what is stored on disk by explicitly creating, writing, and deleting files [5]. The key weakness of this model is that user actions have an immediate and irrevocable effect on disk storage [5]. If a user mistakenly deletes or overwrites a valuable file, the data it stores is immediately lost and, unless a backup copy of the file exists, lost forever [5].

Today, information is valuable and storage is cheap. It is possible that the file system provides better protection to the users' data by making copies of the old versions of data before the data are modified.

There are some file systems provide versioning. Elephant is the one that has a quite good design idea, however, its scheme is still based on the traditional file system architecture — FreeBSD file system. The design is thus not suitable for the parallel file system, which fits the requirement of a high-performance, massive storage system better.

OBSD storage system is different from the traditional system in that the server is no longer handling and forwarding all the requests and data between the clients and storage devices, in which case the server is the bottleneck in the whole system. Instead, servers and storage devices are separated in the OBSD system, where servers dealing only with metadata and some system management, and clients talk directly with the storage devices about data transferring. This idea is helpful in improving the system performance by offloading the workload of servers and balancing work among multiple storage devices. On the other hand, it makes difficulty for the servers to manage the data in the system, since the servers no longer take charge of all the data transfers.

In this section, two possible versioning designs for OBSD storage system are proposed. Neither of them, however, is perfect for the system, each of them has its advantages and disadvantages. This section will address them later.

6.2 A PSEUDO-CENTRALIZED VERSIONING CONTROL

The name means, first, the versioning control is not distributed on all the nodes in the system; second, it is not really ‘centralized’, since there a cluster of Metadata Servers (MDS) sharing the responsibility of versioning control.

In addition to maintain the traditional metadata in MDS, it also includes the version information of the file. The clients access data flow is almost no different with that without versioning service. The clients send requests for open/close to the MDS, the MDS finds the right version of file, returning the location information with that specific version to the clients. And clients compute the location for each object they need according to the retrieved information.

The apparent advantages of MDS cluster managing versioning control are:

- The data structures do not need to change much compared with the OBSD design without versioning
- There is no problem if the system scales in a certain version, which causes different versions distributing differently among the system. This can be solved in the same way as we addressed in the report of OBSD System Scalability.
- Clients may have variant access levels over different versions.

The main disadvantage, however, relies on:

- Since MDS maintains the versioning information, it must keep track of each time the file is updated. In other words, each time the clients update the file, somebody (maybe the clients, or maybe the OBSD) should mention the change to the MDS that holds the metadata of the file. This moves lots of traffic back to the MDS cluster, which does not comply the principle (according to my understanding) of the OBSD storage system design — the servers should participate the data transfers as little as possible. Though in this case, MDS only cares about writes, however, the MDS cluster still has the potential to be the bottleneck of the system, depending on whether of the workload is a write-intensive one.

6.3 A DISTRIBUTED VERSIONING CONTROL

This solution is to remedy the disadvantages of the pseudo-centralized versioning.

Instead of centralizing the versioning information in the MDS cluster, the versioning information is stored on OBSDs. There are two variations of this design.

First, the versioning can be distributed on each OBSD that holds a fragment of the file. So every time the fragment is changed, a new version is created. It is important to decide how to synchronize the versions maintained on different OBSDs. Versioning based on time is a good way to solve the problem. So that the versions on each OBSD don't need to be coherent, but the condition is, there is a global synchronized clock for all the OBSDs.

Second, the versioning information can be stored only on the starting OBSD. So there is no versioning or time synchronization needed. But it brings more traffic to the system, in that each time the client writes anything, it needs to notify the change to the starting OBSD, after it successfully finishes writing.

The distributed versioning control works well for simple system. It means, the system does not scale, so that all versions have the same distributing among the system. And also, for all the versions of the same file, clients have the same access level. These are usually not true, however, in the real systems.

Also, there is question about what version will MDS holds? It does not make sense that MDS holds the oldest version. But to keep the latest version, we meet the same problem mentioned in section 2, that is, the MDS always needs to be notified about the file update, which may cause lots of more traffic to the MDS cluster.

6.4 COMPARISON AND CONCLUSION

The goal of versioning in OBSD storage is, protecting the users' data in a way that can still keep the parallel system scalable and high performanced. The design principle that the servers participate as little data communication as possible earns performance for the system, but that is also the reason that we can not come up with a perfect solution for versioning. Both pseudo-centralized and distributed versioning have trade-offs. They are somehow a complementary solution for each other.

7 DATA AVAILABILITY

7.1 INTRODUCTION TO THE PROBLEM

With the number of disks increasing in the system, the probability of disk failure gets higher. Data reliability is as important a metric to the I/O systems as performance and cost, and it is perhaps the main reason for the popularity of redundant disk arrays [6].
For the OBSD storage system, the data redundancy exists both in MDS cluster and OBS devices.

This section focuses on data reliability of the OBSD system. Section 7.2 discussed solutions for data reliability for OBSDs. Section 7.3 proposed mirroring for MDS data backup.

7.2 OBJECT BASE STORAGE DEVICE (OBSD)

For OBS devices, data reliability is needed in two levels. Firstly, as shown in Figure 2, each OBSD is an I/O node with a single CPU and a local disk array. This structure is good

in that each OBSD thus is autonomic, since the clients and MDS only locate data on OBSD basis. The OBSD can allocate or reallocate the data inside itself freely.

To achieve good data reliability on OBSD basis, it is reasonable that each OBSD maintains a local RAID. The local RAID may just be the RAID-5 working in the same way as that described in [6]. The local RAID protects data loss from disk failure inside a single OBSD.

It is trickier for data redundancy among OBSDs. Because a single OBSD may have problems causing it suddenly down from the system, it is necessary to have data redundancy on OBSD basis. It is obviously not a good idea to have a backup OBSD for each OBSD in the system. It makes the system too expensive and also not effective.

One possible solution to the inter-OBSD data reliability is to use two levels of RAIDs, with one inside OBSD, the other one among OBSDs. However, the expense for writing is huge, since for each write, one level of RAID requires four disk I/Os: one to write the new data, two to read the old data and old parity for computing the new parity, and one to write the new parity [6]. The number for I/O doubles or even more than double if there are two RAID levels. The inter-OBSD RAID overhead can be improved (here improve means make the overhead less.) by making parity group size small among OBSDs. One RAID is implemented on one parity group.

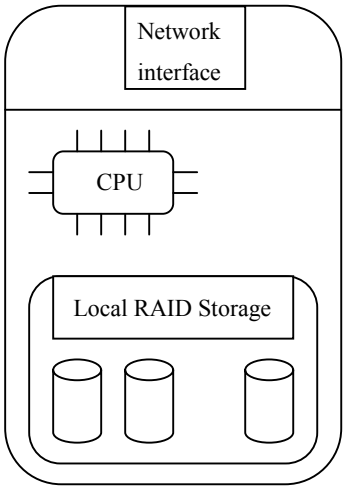

Figure 7.1. A single OBSD [1]

The safest way is to have everything copied to the massive tertiary storage. That is one of the reasons that we include the second level of storage in our architecture, besides OBSD. A daemon should be triggered to copy data from OBSD to tertiary storage whenever the OBSD is idle. The including of tertiary storage is a good for data availability in that when an OBSD is down, other OBSD can also access the data stored in the tertiary storage, and make the data still available to the system.

7.3 METADATA SERVER (MDS) CLUSTER

Data reliability for MDS cluster is extremely important, since it keeps the metadata records for the whole system. The problem of any one of the servers can cause the data of multiple OBSDs unavailable to the clients.

Mirroring is the best and simplest idea for data redundancy for MDS. This relies on two reasons:

- The number of servers in the system is much smaller, compared with the number of OBSDs. The cost of doubling MDSs is reasonable.
- Since a single MDS can have significant effect on the data availability of the system, the faster the metadata is back to work, the better performance the system will have. MDS mirroring can make the server crash transparent to the clients by simply replacing the ‘sick’ server by its mirror server.

8 METADATA ALLOCATION IN MDS CLUSTER

8.1 INTRODUCTION TO THE PROBLEM

There are two kinds of architectures in the current file systems to allocate the metadata. The first architecture has a centralized MDS in the system, which holds all the metadata, as shown in Figure 8.1. The centralized MDS makes the metadata easy to manage, and the system easy to control. But the disadvantage is obvious, the MDS will be the bottleneck.

The other architecture, as shown in figure 8.2, is to distribute metadata with data among the storage devices. There is no server existing in the system. To completely distributing the metadata, the system can provide high parallelism. On the other hand, the system is easy to be lost control and the metadata is hard to manage. For example, it is hard to have file access control, and almost impossible to have versioning without an MDS in the system.

Figure 8.3 shows, in the OBSD project, we are combining these two architectures. We are using an MDS cluster that is responsible for metadata management, as well as providing a fairly good parallelism.

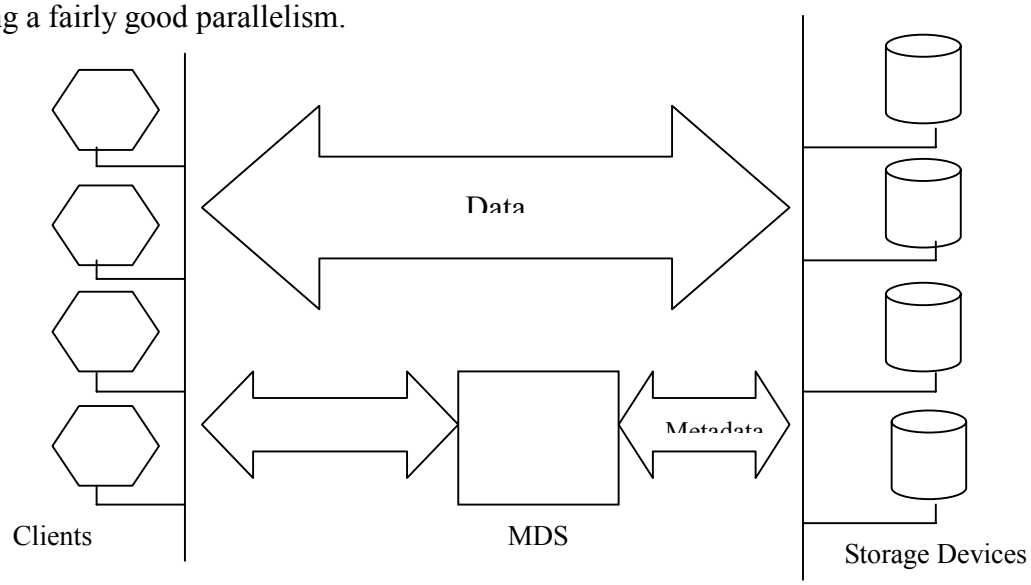

Figure 8.1. Centralized MDS

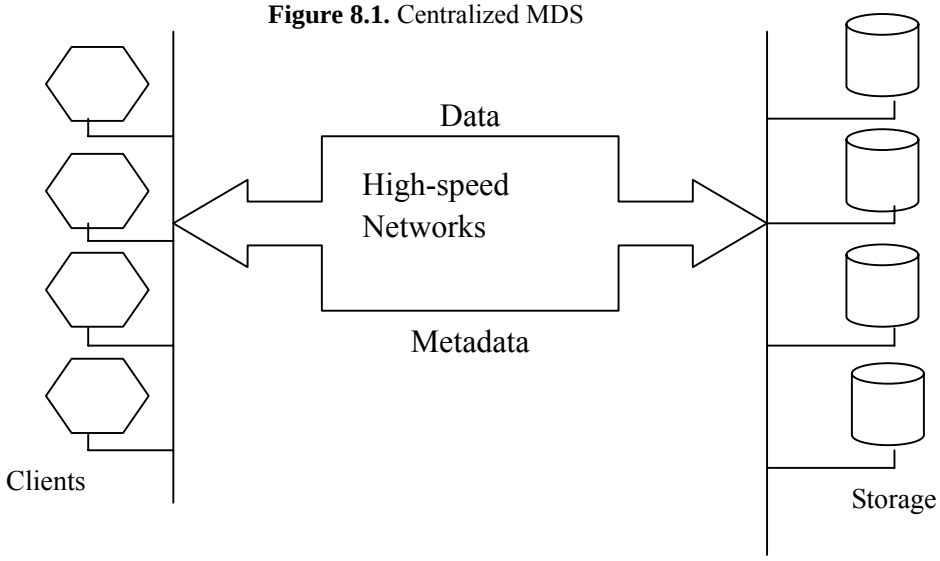

Figure 8.2. Distributed Metadata among the storage devices

8.2 STATIC ALLOCATION

Our motivation for this architecture is to allocate the metadata evenly on MDS, without any obvious hot spot. This is a rather complex issue, though. We come out two solutions that can solve the problem in some degree and they are easy to implement. However, they are not the perfect solutions.

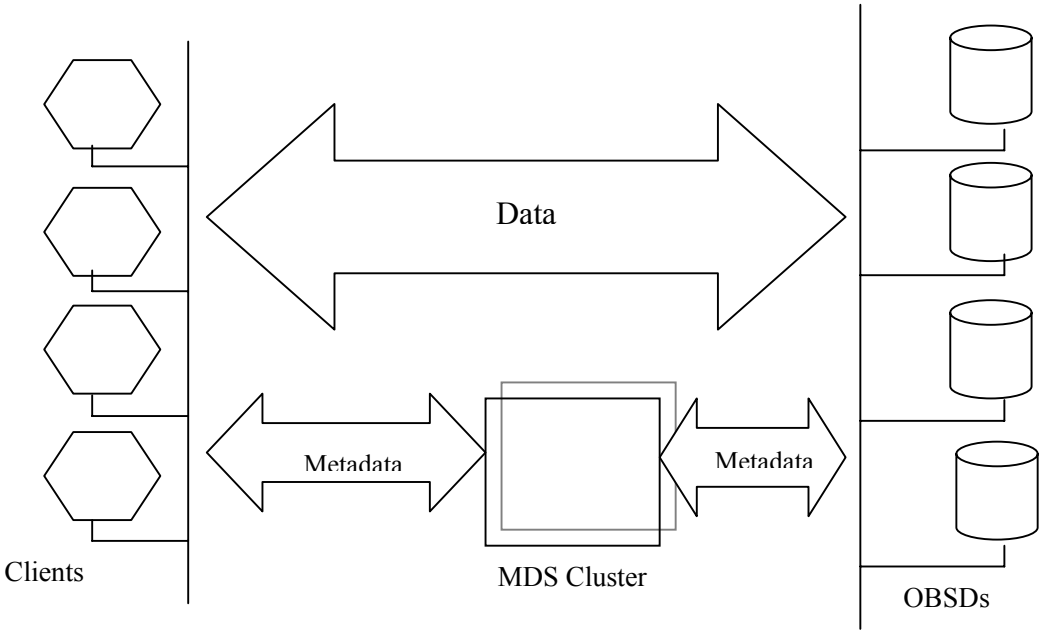

Figure 8.3. MDS cluster structure in OBSD architecture

The idea of static allocation is to partition the metadata allocation among MDSs in advance, according to some partition policy. It is very important to choose a good policy that can accurate reflect the system's file access pattern. A good enough policy can successfully avoid the hot spot among MDSs.

The file type could possibly be the partition policy. As an example, shown in Figure 8.4, if we get the distribution relation of file types vs. access times per minute, we can easily partition the work on the servers. In this example, binary and library files are the most possibly accessed files, while the other three types are much less frequently accessed. So we can span the metadata of binary and library files across more MDSs than those for the other three file types. Figure 8.5 shows the partition.

The problems of static allocation are: first, each client needs to keep a copy of the global allocation partition table, and each time it needs to access some metadata, it needs to look it up in the table, which may be slow and not efficient. Second, though the result is good if we find an ideal partition policy. However, it is hard to actually find a policy that can be totally independent of the workload that generates it. Third, the cluster is hard to scale. Once the partition is made, it is hard to re-balance the workload after new MDS enters the cluster or some MDS suddenly down.

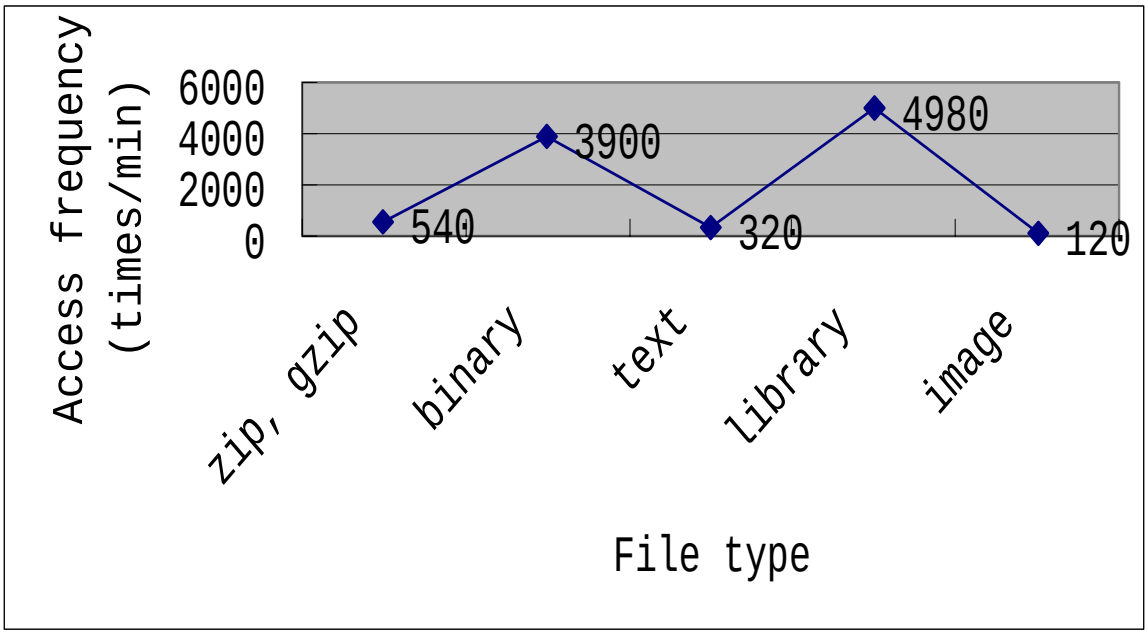

Figure 8.4. an example distribution of file types vs. access frequency

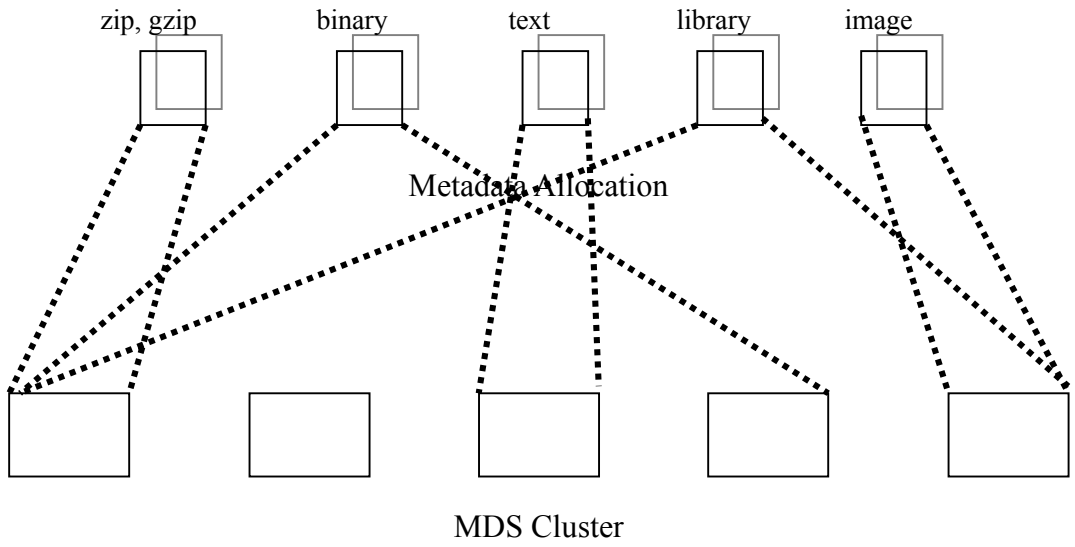

Figure 8.5. An example metadata partition in the MDS cluster

8.3 DYNAMIC ALLOCATION

The other solution is dynamic allocation. Instead of partition the allocation work beforehand, the client allocates or locates metadata by Hashing. It is easier and faster for the client. However, the same as static allocation, it is crucial to decide a good hash policy that is accurate and workload independent. In addition, partition by hashing is also not easy to scale. The scaling problem for hashing exists when allocating data among OBSDs, but the problem can be solved by doing hash versioning controlled by the MDS cluster. Obviously, the same strategy does not work for the MDSs themselves.

9 CONCLUSIONS

The rapid increasing computing capability, giant amount of information, and the fast speed of networks have spawned an urgent need to a high performance, massive distributed storage system. In this document, we presented the Object-base Storage Device System, a highly available, scalable, and secure system.

The distinguishing properties of the OBSD Storage System includes: the separation of data and metadata management as well as storage, the division of clients accessing flow to the servers and storage devices, and the distributed allocation of metadata among metadata servers. These characters make the OBSD Storage System perform better than other systems, but also make the design very complex.

The OBSD Storage System is still in its designing phase. This document addressed and analyzed many design issues of the MDS cluster. Some of the solutions are not perfect and need to improve in future. And also, we will do simulations very soon, to prove the feasibleness of some ideas and get the tradeoffs.

10 ACKNOWLEDGEMENTS

We owe great thankfulness to the course instructor, Professor Scott Brandt. Without his guide and suggestions, we could not achieve this design. And we would also thank Professor Darrell Long and Professor Ethan Miller for their advices and helping us to refine our thoughts about the design.

REFERENCE

[1] Darrell D. E. Long, Scott A. Brandt, Ethan L. Miller, Patrick E. Mantey, Alexandre Brandwajn and Katia Obraczka. *Scalable File Systems for High Performance Computing*, June 1, 2001.

[2] D. Anderson. *Object Based Storage Devices Presentation,* November 29, 1999.

[3] G. Gibson et al. *File server scaling with network-attached secure disks*, Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (Sigmetrics '97), 1997.

[4] M. Spasojevic and M. Satyanarayanan, "An Empirical Study of a Wide-Area Distributed File System," ACM Transactions on Computer Systems 14(2), May 1996, pages 171-199.

[5] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. *Deciding when to forget in the Elephant file system*, 17th ACM Symposium on Operating Systems Principles (SOSP '99), Published as Operating Systems Review, 34(5): 110-123, Dec. 1999