DESIGN AND IMPLEMENTATION OF AN OBJECT STORAGE SYSTEM

YAN JIE

(*B. Eng.(Hons.), Xi'an Jiaotong University*)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2005

Acknowledgments

The writing of a dissertation is a tasking experience. First and foremost, I would like to extend my deepest gratitude to my advisors Dr. Zhu Yaolong and Dr. Liu Zhejie for giving me the privilege and honor to work with them over the last 3 years. Without their constant support, insightful advice, excellent judgment, and, more importantly, their demand for top-quality research, this dissertation would not be possible. I am also grateful to my families. Without their long-lasting support and infinite patience, I cannot image how I could get through this process.

I would also like to thank Xiong Hui, Renuga Kanagavelu, Zhu Shunyu, Yong Kaileong, Sim Chinsan and Wang Chaoyang for giving a necessary direction to my research and providing continuous encouragement.

Furthermore, I would like to thank my friends Gao Yan, Zhou Feng, Meng Bin, So Lin Weon, and Xu Jun for always inspiring me and helping me in difficult times.

I am also thankful to SNIA OSD Technical Working Group and NASA/IEEE Conference on Mass Storage Systems and Technologies (MSST 2004) reviewers for providing their helpful comments on this work. Especially, I am grateful to Dr. Julian Satran from IBM, Dr. David Nagle from Panasas, Dr. Erik Riedel and Dr. Sami Iren from Seagate.

To

Mom and Dad

With Forever Love and Respect

Contents

Summary

Storage requirements continue to grow because of popularity of data intensive applications and rapidly increasing client performance. Application servers require a secure, scalable, highly-available, manageable, and high performance storage solution. However, the current file-level Network Attached Storage (NAS) solution is good at cross-platform, but poor in performance, while the block-level Storage Area Network (SAN) solution can achieve high performance, but lacks effective means to provide cross-platform data sharing. In order to address these issues, this thesis attempts to provide an intelligent storage solution based on the Object-based Storage Device (OSD) concept. Object, which is regarded as the convergence of file and block technologies, can provide the advantages of both of them. Based on the object access, BrainStor integrates the strengths of NAS and SAN technologies without inheriting their weaknesses. BrainStor can achieve the high performance from direct access and the cross-platform data sharing ability from high-level abstract.

This dissertation presents the design and implementation of BrainStor, a Fibre Channel OSD prototype. BrainStor introduces an OSD architecture with unique Object Cache Module and Object Bridge Module. There are six key components in BrainStor: Object Storage Client (OSC), Object Storage Module (OSM), Object Cache Module (OCM), Object Bridge Module (OBM), Object Manager Module (OMM) and Security Manager Module (SMM). The independent OMM and OSM clusters are adopted to separate the metadata path and data path. Hence the metadata server is removed from the data path and the OSM provides the direct data access to clients. Moreover, the OBM makes the BrainStor system compatible with the existing SAN components, such as the RAID systems from different vendors. In addition, Brainstor also offers a scalable cache solution. OCM, as a centralized cache for the entire BrainStor system, can be scaled to meet the increasing and unlimited performance needs of storage applications.

Through analyzing BrainStor test results, the dissertation demonstrates its strengths and further identifies some critical issues about object storage system design. Iometer and IOzone tests show that the storage scalability can greatly improve the overall performance of BrainStor. The PostMark test unveils the metadata management challenges in BrainStor design.

In order to address the metadata management issue, the dissertation further proposes a Hashing Partition (HAP) method in the OMM cluster design. HAP uses hashing method to avoid the numerous metadata accesses, and uses filename hashing policy to avoid the multi-OMM communication. Furthermore, based on the concept of logical partitions in the common storage space, the HAP method significantly simplifies the implementation of the OMM cluster and provides efficient solutions for load balancing, failover and scalability. Normally, the OMM cluster supports scalability without any metadata movement. However, if the OMM cluster scales to a number that is greater than the preset scalability capability, some metadata must be redistributed in the OMM cluster. The Deferred Update algorithm is proposed to improve the response time of this process and minimize its effects.

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

In the information age, storage requirements continue to grow because of rapidly increasing client performance, popularity of data, such as video and music, and data intensive applications such as data mining and electronic commerce. Stored information is at least doubling every 24 months [1]. The growing demand of storage asks for a secure scalable, highly-available, manageable, and high performance storage solution.

Nowadays, there are three basic storage architectures commonly in use. They are Direct Attached Storage (DAS), Network Attached Storage (NAS) and Storage Area Network (SAN). In addition, based on the SAN architecture, SAN file system also emerges.

1.1.1 Direct Attached Storage (DAS)

Direct Attached Storage (DAS) refers to block-based storage devices, which directly connect to the I/O bus (e.g. SCSI or ATA/IDE) of a host[4]. In this topology, as shown in Figure 1.1, most of the storage devices such as disk drives and RAID systems are directly attached to a client computer through various adapters with

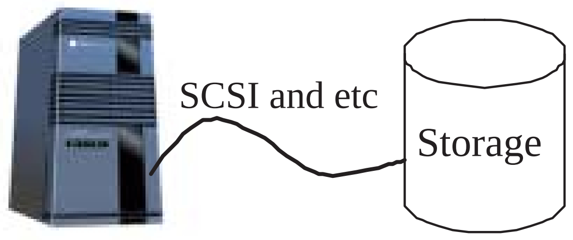


Figure 1.1: Direct Attached Storage (DAS)

standardized protocol, such as Small Computer System Interface (SCSI) [2].

Although DAS offers high performance and minimal security concerns, there are some inherent limitations. DAS can provide limited connectivity and scalability. It can only scale along with the server that it is attached to. DAS is an appropriate choice for applications, whose scalability requirement is low.

1.1.2 Network Attached Storage (NAS)

Network Attached Storage (NAS) [8] is a LAN attached file server that serves files using a network protocol such as Network File System (NFS) [9] or Common Internet File System (CIFS) [3]. Figure 1.2 shows a typical NAS architecture. NAS can also be implemented on top of a SAN or with DAS, which is often referred to as a NAS head, as shown in Figure 1.2.

NAS provides excellent capability for data sharing across multi-platform. All authorized hosts within the same network of the NAS server can access its storage. Different platforms, such as Windows and Linux, can access the same NAS server synchronously.

In terms of scalability, capacity of a single NAS server is limited by its direct attached storage. A NAS head enables better scalability solution from the SAN that it connects to.

However, NAS leads to an obvious bottleneck. The metadata about the file attributes and location on devices is managed by the file server, hence all I/O

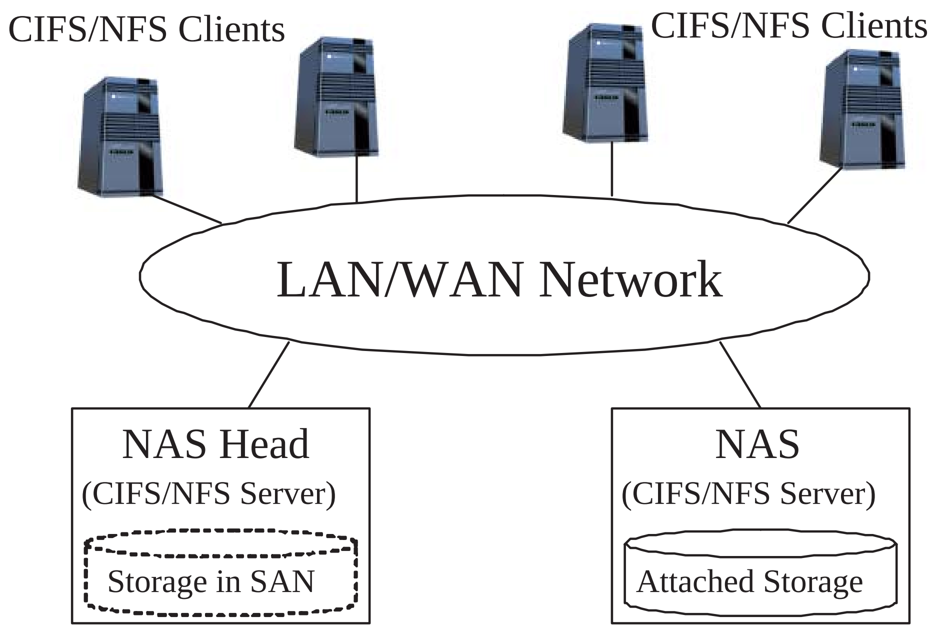


Figure 1.2: Network Attached Storage (NAS)

requests must go through the single file server. No matter NAS is used as a single file server or NAS head, clients' access performance is limited by the performance of the file server.

1.1.3 Storage Attached Network (SAN)

Storage Area Network (SAN) is a high-speed network (or sub-network) that is dedicated to storage. SAN interconnects all kinds of data storage devices with associated application servers [4]. In a SAN, application servers access storage at block level.

SAN addresses the connectivity limits of DAS and thus enables the storage scalability. New storage devices can be easily connected to a SAN in order to improve the capacity as well as performance. With this added connectivity, SAN also needs a better security solution. Therefore, SAN introduces concepts such as zoning and host device authentication to keep the fabric secure [5]. Figure 1.3 shows a typical SAN setup. All kinds of servers centralize their storage through a dedicated storage area network. Storage systems, such as RAID subsystem and

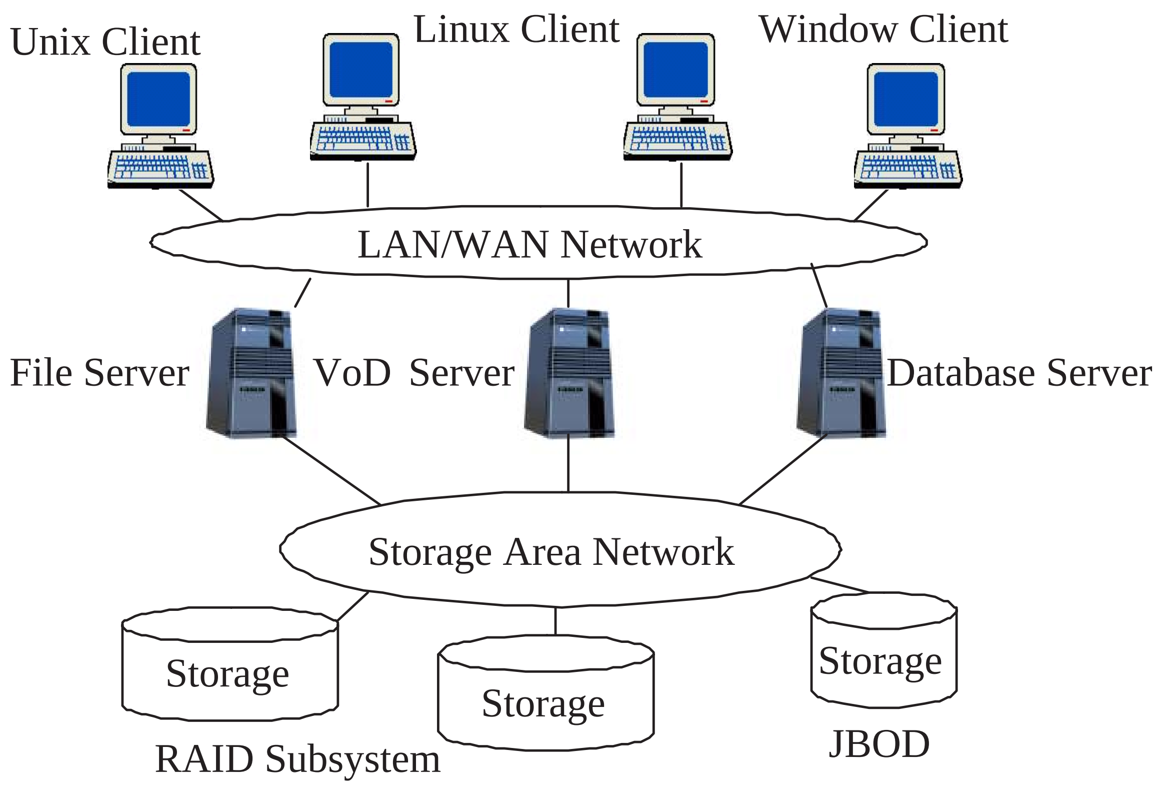


Figure 1.3: Storage Area Network (SAN)

JBOD, connect to SAN and make up a high performance storage pool.

1.1.4 SAN File System

In order to address the performance and scalability limitations of NAS, especially NAS head, some SAN file systems have emerged in recent years. A SAN file system architecture is shown in Figure 1.4. Separated servers are built to provide metedata services. SAN file system can remove the bottleneck at the file server from the data path and have the direct block-level access to storage. And the SAN file system can provide the ability of cross-platform data sharing.

In the SAN file system architecture, storage are exposed to all the application servers. At block level, there is no accordingly security mechanism for each request. Thus, security is one important issue in SAN file system. Currently, many high-end storage systems adopt this kind of architecture. For example, IBM's StorageTank [6], EMC's High-Road, Apple's XSAN and Veritas' SANPoint Direct.

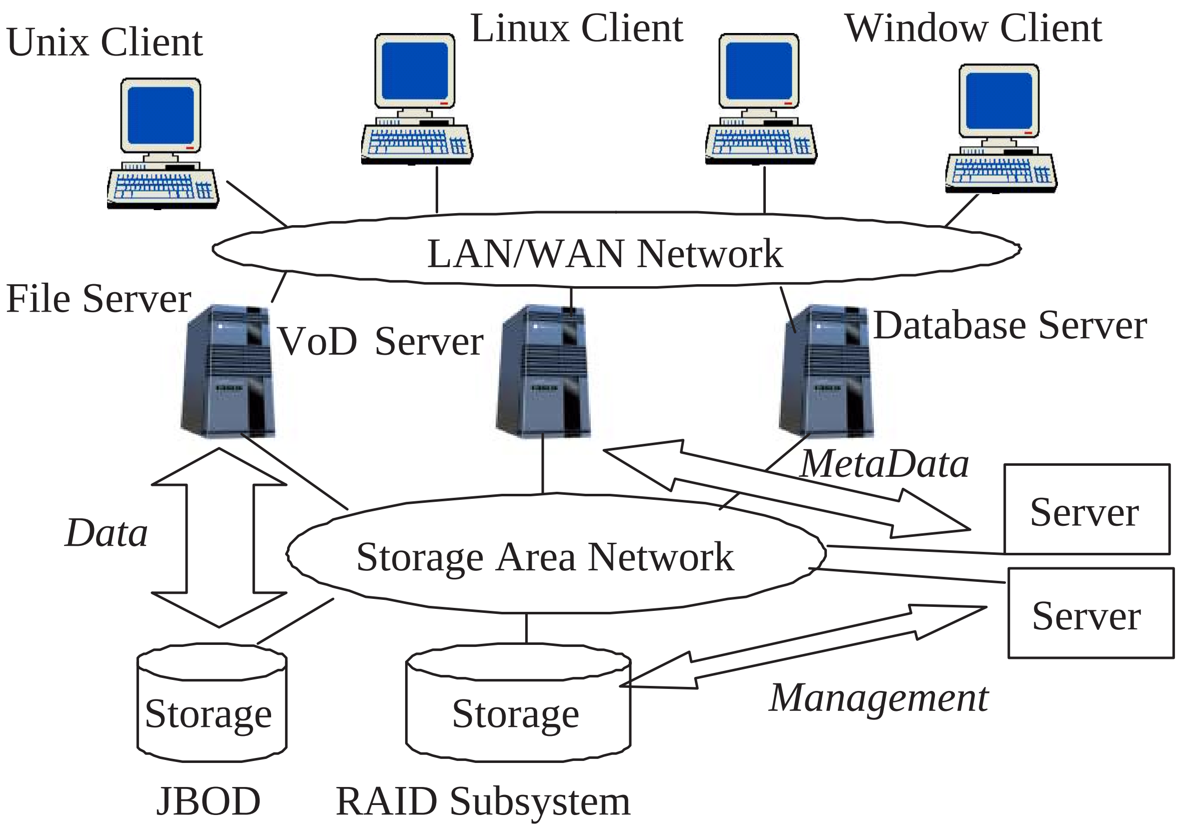


Figure 1.4: Architecture of SAN File System

1.1.5 Evolution of Storage

Each of DAS, NAS, and SAN can be used to solve problem specific to particular applications. Several studies have been conducted on the performance of these three network storage architectures [10, 11, 12, 13]. Some researchers even explore the iSCSI-based SAN performance in wireless environment [14].

At enterprise level, DAS is fading due to its limitation of scalability. NAS achieves cross-platform by providing a centralized server and well know interfaces such as CIFS and NFS. However its performance is poor due to queuing delay at the central file server and poor performance of TCP. SAN can achieve great performance through direct access, low latency fabric and aggregation techniques, such as Redundant Array of Independent Disks (RAID) [15]. However SAN does not perform well in cross-platform data sharing. The trade-off in today's architectures is therefore among high performance (blocks), security, and cross-platform data sharing (files). While files allow one to securely share data among systems, the overhead imposed by a file server can limit performance. On the other hand, increasing file serving performance by allowing direct client access comes at the cost of security. Building a scalable, high-performance, cross-platform, secure data sharing architecture requires a new interface that provides both the direct access nature of SANs and the data sharing and security capabilities of NAS. OSD [16], as a next generation interface protocol, is proposed to meet this goal.

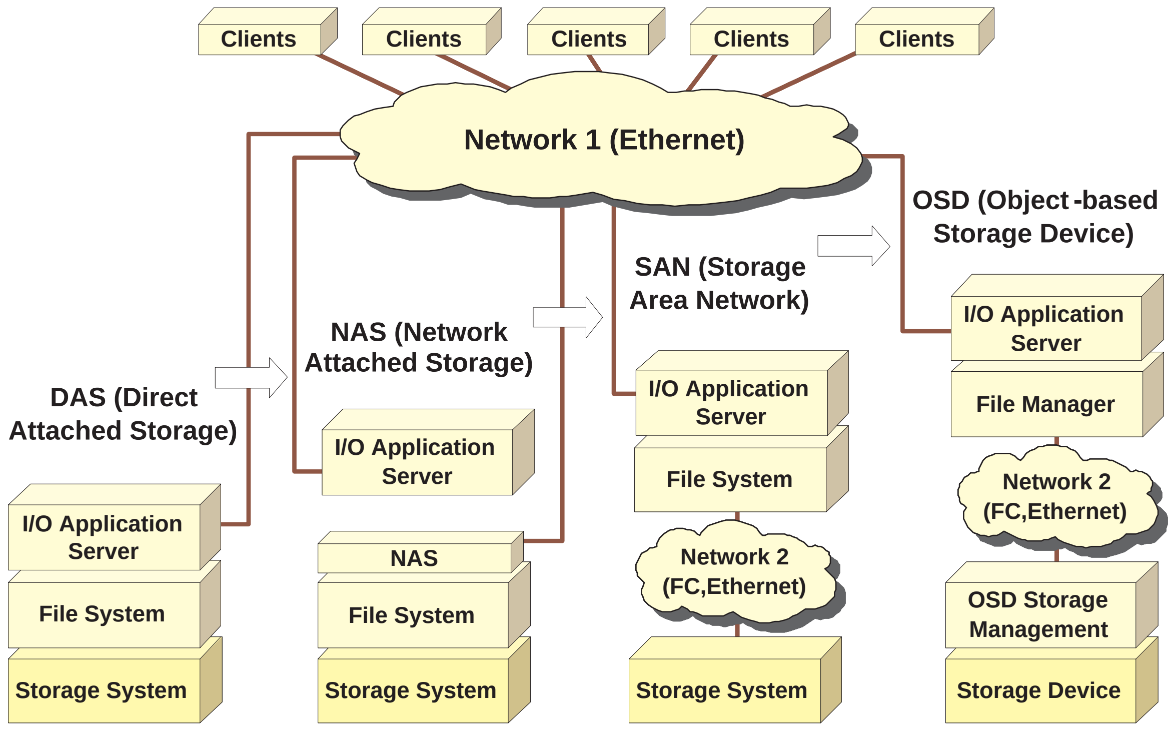


Figure 1.5: Evolution of Storage

The evolution of storage follows the steps shown in Figure 1.5. The first step is from the direct connected DAS to the networked storage: NAS, which puts storage server on the user network. Then a dedicated storage network, SAN, emerged. In SAN, online server can access the storage at block level through another high speed network, which is normally based on Fibre Channel [17] or iSCSI [18]. In this way, all the traditional local file systems can be adopted in a SAN infrastructure easily.

Now, storage is moving to Object-based Storage Device (OSD). In OSD, the storage management component of normal file system is moved to the storage system. Storage is accessed at object level. OSD is designed to integrates the strengths of NAS and SAN technologies without inheriting their weaknesses.

The strength and weakness of DAS, NAS, SAN and OSD can be summarized in Table 1.1 [21].

Table 1.1: Comparison of DAS, NAS, SAN and OSD

Storage Architecture	DAS	NAS	SAN	OSD
Access Layer	Block	File	BLock	Object
Security	High	Medium	Low	High
Storage Management	High/Low	Medium	High	High
Device and Data Sharing	Low	High	Medium	High
Storage Performance	High	Low	High	High
Scalability	Low	Medium	Medium	High
Device Functionality	Low	Medium	Low	High

1.2 Object-based Storage Device (OSD): Future Intelligent Storage

1.2.1 Object Storage

Nowadays, industry has begun to place pressure on the storage interface, demanding it to do more. Since the first disk drive in 1956, disks have grown by over seven orders of magnitude in density and over four orders in performance. However the block interface of storage has remained largely unchanged [19]. As storage architectures becoming more and more complex, the functions that storage system can perform, are limited by the stable block interface.

In addition, storage devices can be a far more useful and intelligent devices with the knowledge of data stored on them. Even with the integrated advanced electronics, processors, and buffer caches, today's hard disks are still relatively "dumb" devices. Disks perform two functions: read data and write data, and know nothing about the data that they store. The basic premise of OSD concept is that the storage device could be an intelligent device if it knew more information about the data it stores.

OSD is the device that stores, retrieves and interprets objects, which contains user data and their attributes. An object can be looked as a logical collection of raw user date on a storage device, with well-known methods for access, metadata describing characteristics of the data, and security policies that prevent unauthorized access [19].

Unlike blocks, objects are of variable size and can be used to store entire data structures, such as database tables or multimedia. A single object can be used to store an entire database or part of a file. The storage application decides what is stored in an object. And the object storage device is responsible for all internal space management of the object.

Objects can be regarded as the convergence of two technologies: files and blocks. Files provide user applications with a high-level abstraction that enables secure data sharing across different operating systems, but often at the cost of limited performance due to bottleneck at file server. Blocks offer fast and scalable access, but this direct access comes at the cost of limited security and data sharing without a centralized server to authorize the I/O and maintain the metadata. Objects can provide the advantages of both files and blocks. Object is a basic access unit that can be directly addressed on a storage device without going through a server. This direct access offers performance advantages similar to blocks. In addition, objects are accessed using an interface similar to the file access interface, thus making the object easily accessible across different platforms. By providing direct, file-like access to storage devices, OSD enables both high performance and cross-platform sharing.

In OSD, part of today's normal file system functions can be moved into storage devices, as shown in Figure 1.6. file system includes two parts: user component and storage component. User component contains functions, such as hierarchy management, naming and user access control, while storage component is focused on mapping logical structures (e.g. files) to the physical structures of the storage media. By moving low-level storage functions into the storage device itself and

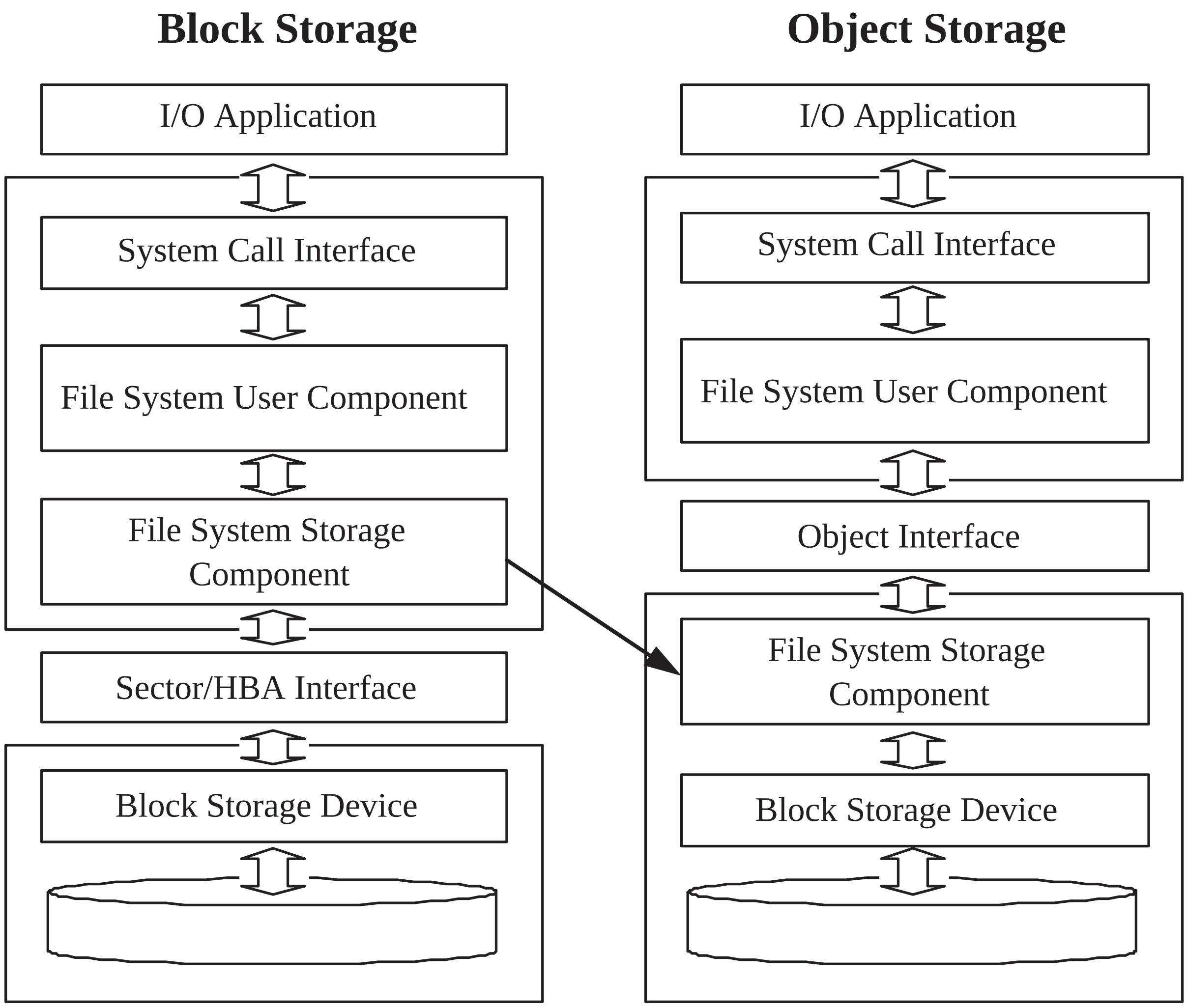


Figure 1.6: Comparison of Block Storage and Object Storage

accessing the storage at object level, the Object-based Storage Device enables:

- Intelligent space management in the storage layer
- Data-aware pre-fetching and caching
- Quality of Service (QoS) support
- Security in the storage layer

This movement is the continued trend of migrating the various functions into the storage devices. For example, the redundant check function has been moved into disk.

OSDs come in many forms, ranging from a single disk drive to a storage controller with an array of disks. OSDs are not limited to random access or even writable devices. Tape drives and optical media can also be used to store objects. The difference between an OSD and a block-based device is the interface, not the physical media [19].

1.2.2 Object Storage Architecture

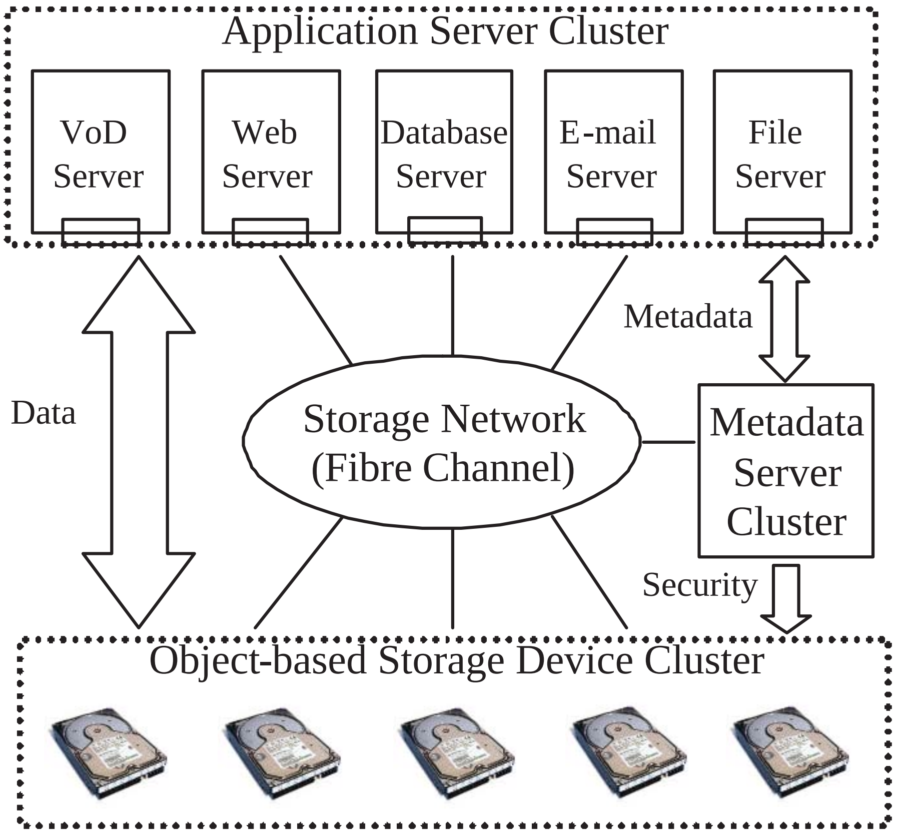


Figure 1.7: Object Storage Architecture

Based on object concept, the object storage architecture attempts to combine the advantages of both NAS and SAN. Figure 1.7 shows a typical setup of OSD. Unlike traditional file storage systems with metadata and data managed by the same machine and stored on the same device [20], a basic OSD architecture has the separate Metadata Server (MDS) from the storage. In a basic model, there are application servers, metadata server and object-based storage device. A separate cluster of metadata server manages metadata and file-to-object mapping, as shown in Figure 1.7. The metadata server is used as a global resource to find the location of objects, to support secure access to objects, and to assist in storage management functions. OSD cluster manages low-level storage tasks such as object-to-block mapping and request scheduling, and presents an object access interface instead of block-level interface [21].

The goal of such storage system with specialized metadata management is to efficiently manage metadata and improve the overall system performance. Based on this architecture, data path and metadata path are separated. Without the bottleneck of a file server, applications can directly access data stored in OSD. Moreover, object storage architecture is designed for parallel storage access and unlimited scalability. With all these benefits, object storage can assure high performance. In addition, metadata servers create a single namespace that is shared by all of the nodes in the cluster. Therefore, object storage architecture distributes the system metadata allowing shared file access without a central bottleneck. In short, OSD storage systems have the following characteristics:

- Cross-platform data sharing
- High performance via direct access and an offloaded data path
- Scalable performance and capacity
- Strong fine-grained security (storage level)
- Storage management
- Device functionality

These features are highly desirable across all kinds of typical storage applications. Particularly, they are valuable for scientific applications and databases, which generate high-level concurrent I/O demand for secure, shared files. The Object-based storage architecture is uniquely suited to meet the demands of these applications.

Besides its benefits, what kinds of challenges does OSD bring to us? OSD is a comparable new technology and has become a popular term among academic and industrial research communities. However, the new object concept can raise many new problems as well. For example, does today's storage infrastructure still fit OSD? Is there some new requirements for the metadata management? This study tries to identify those important challenges through prototyping and testing an OSD storage system.

1.3 Contributions and Organization of Thesis

1.3.1 Contributions

The study emphasizes the design of an OSD prototype, named BrainStor. The primary contributions of the thesis can be summarized as follows:

- A Fibre Channel OSD prototype is developed. The study also proposes a new OSD architecture with unique components, such as Object Cache Module and Object Bridge Module
- Based on the test results of the OSD prototype, the thesis demonstrates some key features of object storage, such as the scalability and virtualization, and further identifies some critical issues in the design of an object storage system, such as the frequent metadata access.
- Hashing Partition method is proposed to address the frequent metadata access issue. Based on this new method, the number of metadata access can be reduced. Moreover, the new methodology also simplifies the load balancing, scalability and failover design of the OMM cluster.
- Analysis results of the hashing method show that the Hashing Partition can reduce the number of metadata requests in both situations: with cache effects and without cache effects.

1.3.2 Organization of Thesis

The rest of the thesis is organized as follows. Chapter 2 discusses the other research projects related to object storage. Three important OSD related prototypes - NASD, Lustre and Intel OSD prototype - are discussed in detail. Chapter 3 is devoted to the BrainStor storage architecture, which enables cost-effective bandwidth and capacity scaling, compatibility, and centralized cache management. After that, the interfaces and communications between BrainStor nodes are detailed. Then the internal software architectures of BrainStor nodes are discussed. Chapter 4 presents the current BrainStor prototype running in the lab. Then test results from three benchmark tools: Iometer, IOzone and PostMark, are explained. Through these results, some critical issues in the BrainStor design are identified.

In order to address the metadata management issue identified in Chapter 4, Chapter 5 details a new metadata server cluster design, named Hashing Partition (HAP). HAP uses hashing method to reduce the number of metadata requests and adopts a common storage space to make the cluster more capable to handle metadata requests. Three key components of HAP are introduced. Then based on HAP design, an effective and low cost mechanism for load balancing, failover and scalability of metadata server cluster is presented in order to demonstrate the strengthes of HAP. Then metadata cluster rebuild is discussed. Next, the HAP and the directory metadata management is compared based on analysis results. Chapter 5 also describes some functional experiences of HAP. Finally, Chapter 6 summarizes the conclusions and future works of the study.

Chapter 2

Background

The concept of OSD has been around for the past 20 years. At the end of 70's, object-oriented operating systems raised the initial idea of object-based storage. Operating systems were designed to use objects to store files on disk. These systems include the Hydra from Carnegie Mellon University [24] and the iMAX-432 from Intel [25].

In the 80's, The SWALLOW project from Massachusetts Institute of Technology [38] implemented one of the first distributed object storages.

In the 90's, much of this work about OSD was conducted by Garth Gibson and his research team at the Parallel Data Lab at Carnegie Mellon University. Their work focused on developing the underlying concept of OSD with two closely related projects called Network Attached Secure Disks (NASD) [28] and Active Disks [23].

In 2002, an OSD Technical Working Group (TWG) has been formed as part of the Storage Networking Industry Association (SNIA). The charter of this group is to work on issues related to the OSD command subset of the SCSI command set and to enable the construction, demonstration, and evaluation of OSD prototypes. In 2004, OSD SCSI standard (Rev 10) from SNIA OSD TWG is approved by INCITS Technical Committee T10 as one of standard SCSI command sets.

While the standards are being developed, some similar technologies to OSD have been implemented in industry. The National Laboratories, Hewlett-Packard and Cluster File Systems company are building the highly scalable Lustre file system [32]. IBM is researching the object-based storage for their SAN file system, StorageTank [30]. Centera from EMC and Venti project from Bell implement the disk-based Write-Once-Read-Many (WORM) storage based on the concept of object access for content addressable storage (CAS).

In academic communities, a lot of researchers focus on OSD related topics, for example, Self-* project in CMU and Object Based Storage System (OBSS) project in University of California, Santa Cruz (UCSC). Researchers in the University of Wisconsin (Madison) explored a smart disk systems that attempt to learn file system structures behind existing block-based interfaces [37]. Some researchers in the Tsinghua University studied the cluster object storage from the application point of view [39].

Self-* project in CMU explores new storage solutions with automated management functions. Self-* storage systems are self-configuring, self-organizing, self-tuning, self-healing, self-managing systems. Self-* storage has the potential to reduce the human effort required for large-scale storage systems, which is critical as storage moves towards multi-petabyte data centers [33]. In this project, new interfaces between hosts and storage devices are studied [34, 35, 36].

UCSC OBSS project are investigating the construction of large-scale storage systems using object-based storage devices. On the side of object data management, researchers in UCSC are developing an Object-based File System (OFS), and allocates storage space from different regions according to the variable object sizes, rather than fixed-size blocks [40, 41]. On the side of object metadata management, they are working on experiments of metadata partitioning based on Lazy Hybrid Hashed Hierarchical (LH3) directory management [54]. They are doing research on replication algorithms and recovery under highly distributed systems [42].

In terms of available OSD related prototypes, NASD in CMU started the initial development work on OSD. Another development work is from Lustre project in Cluster File Systems, inc. Intel also provides a reference OSD implementation as part of its open source iSCSI project.

2.1 Network Attached Secure Disks (NASD)

Network Attached Secure Disks (NASD) project in CMU developed the basic idea of OSD. The aim of NASD is to enable commodity storage components to be the building blocks of high-bandwidth, low-latency, secure scalable storage systems [26, 27]. NASD explored adding processing power to individual disks, in order to process networking, security [46], and basic space management functions [29].

NASD sets up a standard for the OSD models. The major components in NASD prototype are NASD drive, file manager, and clients. In addition, storage manager is used to coordinate NASDs to build a parallel file system. Dr. Amiri detailed the design of NASD in his Ph.D. Dissertation [29]. And Dr. Gobioff proposed an object security architecture in NASD [46].

All the object data and metadata of NASD are persistently stored in its NASD drive. However, NASD has the separated access paths to data and metadata. File manager can handle all the metadata requests while NASD drive can respond to object data requests. There is also a metadata transition path between file manager and NASD drive. File manager can cache part of metadata in its local memory to accelerate the response of metadata requests to clients. In addition, NASD manages the object to block mapping by itself at NASD drive side.

2.2 Lustre

Lustre is the name of file system solution for high-end applications by Cluster File Systems, Inc. Lustre is a scalable cluster file system for very large clusters. Lustre focuses on solving scalability and management issues in large computer clusters [32]. Lustre runs over different networks, including Ethernet and Quadrics [31].

Lustre has separated data and metadata access paths as well as the separated persistent storage of data and metadata. Object Storage Target (OST) in Lustre stores the data objects and responds all the data requests, while Metadata Server (MDS) in Lustre stores the metadata and handles the metadata requests.

Another feature of Lustre is to adopt ext2, ext3 or other file systems to complete the object to block mapping. There is a filter layer implemented in Lustre, which converts the coming object requests to file requests that can be directly completed by local file systems, such as ext3.

2.3 Intel OSD Prototype

Intel provides an OSD implementation as part of Intel's iSCSI open source project to demonstrate the idea of OSD [22]. Intel OSD prototype includes two components: client and OSD. Client accesses OSD at object level by using the OSD SCSI commands defined in the SNIA OSD SCSI standard [16]. However, Intel OSD prototype does not have separated metadata and data paths.

Intel OSD prototype is a good platform to benchmark the SNIA OSD standard [16]. It provides a reference code of the standard. Adopting the similar object storage concept, NASD and Lustre are actually using self-defined interfaces.

Chapter 3

BrainStor

BrainStor aims at providing an intelligent storage solution based on OSD concept. BrainStor introduces new modules, such as a centralized Object Cache Module and Object Bridge Module, to the general OSD architecture. In BrainStor project, a Fibre Channel OSD prototype using the OSD SCSI command protocol [16] is developed. This protocol defined by SNIA OSD Technical Working Group (TWG) plays a critical role in the standardization process of OSD. In the following sections, the term “OSD protocol” is used with reference to the OSD SCSI command protocol [16].

3.1 BrainStor Architecture

In BrainStor, there are six main nodes, which are Object Storage Client (OSC), Object Storage Module (OSM), Object Cache Module (OCM), Object Bridge Module (OBM), Object Manager Module (OMM) and Security Manager Module (SMM). In addition, the OSC has two sub-modules: Object File-system Module (OFM) and Object Interface Module (OIM). All the nodes are scalable. There are the OSM Cluster and the OMM Cluster at the core of BrainStor, while other modules work as feature-enriched nodes. All these nodes are connected to storage network, as shown in Figure 3.1.

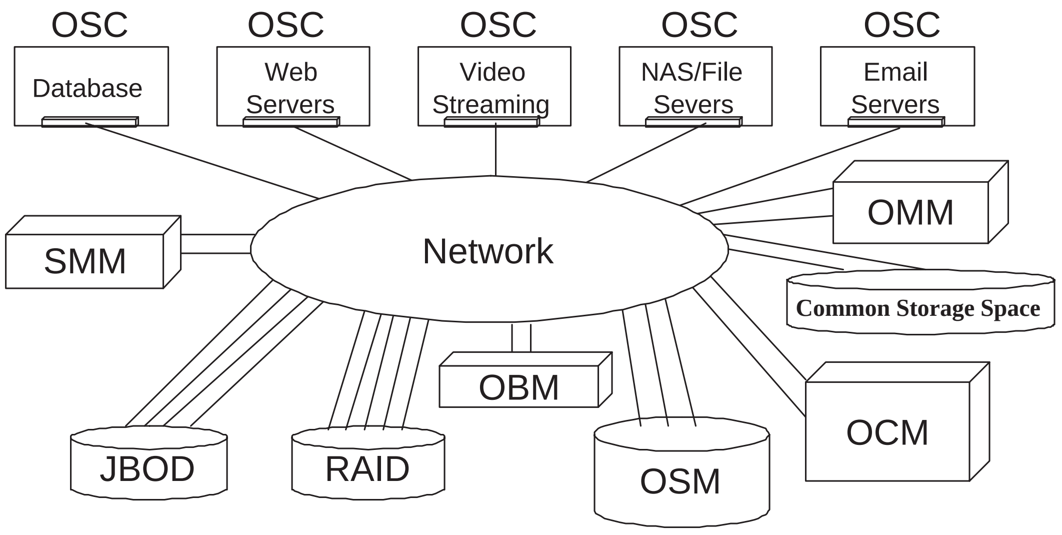


Figure 3.1: BrainStor Architecture

OSCs can be of all kinds of application servers, such as email servers and Video-on-Demand (VoD) servers. The OSM cluster is the storage place for raw data object. The OCM cluster is a cache cluster used to accelerate the access of storage. The OMM cluster manages the object metadata and file metadata. The OBM makes the BrainStor network compatible with the existing storage network and devices. As shown in Figure 3.1, OSCs can access the block storage device, such as JBOD and RAID system in SAN, through OBM. The SMM provides the security for BrainStor network. As an addition, a common storage space is used by the OMM cluster to faciliate the Hashing Partition implementation, which will be explained in Chapter 5.

BrainStor benefits the storage as follows:

Intelligence

Accessing at object level, BrainStor can learn important characteristics of data and its operating environments. In other words, storage can know what is the data stored in itself. Today's block-level storage devices are mainly unaware of the users and storage applications, which are using the storage. The only information that a block-based storage device knows about the data, is the Logical Block Address (LBA). Thus, there is entirely no difference to storage between the most important data and deleted files in recycle bin.

Object storage devices can understand the relationships between the blocks, and can use this information to better organize the data layout. In object storage, object attributes are associated with object. Object metadata includes static information about the object (e.g. creation time), dynamic information (e.g. last access time), and information specific to users (e.g. QoS agreement). Object metadata can also contain hints about the object's behavior such as the expected read/write ratio, the most likely patterns of access (e.g. sequential or random), or the expected lifetime of the object [19]. With knowledge of this kind of information, BrainStor can optimize storage management for applications.

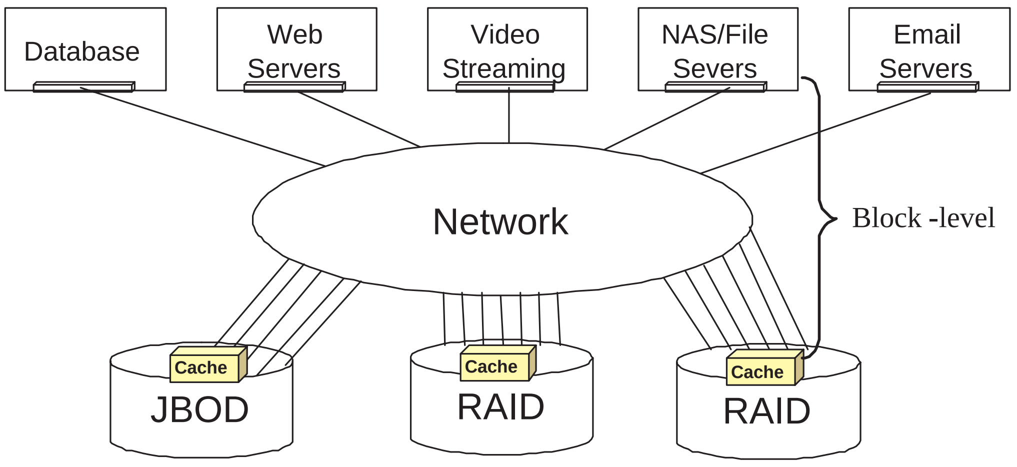


Figure 3.2: Cache in Current Storage Solution

Cache

BrainStor adopts a centralized cache module. Cache design is one of the most important issues in storage system design. As shown in Figure 3.2, the block-based storage adopts cache located at individual storage system and the cache is exclusively accessed by its host storage system. While in BrainStor, cache is centralized at the Object Cache Module for all storage modules. Furthermore the OCM is scalable and can be shared by all storage modules as shown in Figure 3.1. In addition, both OCM and OSM are directly accessed by OSCs. This design changes the role of cache from a storage device cache to a SAN cache.

High Performance

With a separated OMM cluster, dedicated for metadata management, BrainStor enables a direct high speed data path between OSCs and storage nodes, such as OSM, OBM or OCM. By removing metadata access out of data access path, there is no more additional queuing delay in the OMM. BrainStor can also adopt aggregation techniques, e.g. Redundant Array of Independent Nodes (RAIN) and RAID.

In addition, the OSC off-loads space management (e.g. allocation of free blocks and tracking of used blocks) to storage nodes. The OSC does not need to keep storage information (e.g. free block bitmap) in its local memory. This kind of information is maintained by the OSM in BrainStor. Thus OSCs have more resources to serve the applications.

Data Sharing

The higher-level interface and the attributes about the stored data enable data sharing of objects. The interface to BrainStor is very similar to that of a file system. Objects can be created or deleted, read or written, and even requested for certain attributes. File level protocols, such as CIFS and NFS, have proven their strength to the cross-platform data sharing. Similarly BrainStor can also be shared between different platforms. Standardized object attributes improve data sharing by allowing different platforms to share a common set of information describing the data. Object attributes defined in the OSD protocol, contain information analogous to that contained in an inode. The inode is the data structure used in many UNIX and Linux file systems to describe the file [45]. Therefore, many technologies used in the file level cross-platform sharing, can be integrated with BrainStor easily.

Security

Security is another important feature of object-based storage that distinguishes it from block-based storage. There are many similarities between the BrainStor architecture shown in Figure 3.1 and the SAN file system architecture shown in Figure 1.4. Both of them have storage and application servers connected to the network; both of them have separated servers from storage. In this type of architecture, security is an important issue. Neither clients nor the network is trusted, since clients and storage devices can be anywhere on the network. Therefore, there exists the risk of unauthorized clients accessing the storage, or authorized clients accessing the storage in an unauthorized manner.

In block-based storage, although the security does exist at the device and fabric level (e.g. devices may require a secure login and switches may implement zoning), an attacker can easily use its controlled legitimate client to access blocks that should not be accessed by the client (e.g. modify its own commands to access the blocks belonging to others). Although zoning technology can help to certain extent, attacker can at least access all the storage in zones, which are open to its controlled clients. This situation becomes worse in a SAN file system environment, where all the storage is open to all clients in order to achieve the parallel access performance. In addition, storage cannot tell whether the coming requests are modified by attacker. Hence the entire storage network is also vulnerable to man-in-middle attack.

BrainStor adopts a credential-based access control system. The SMM generates credentials at the request of an authorized OSC. The credential gives the OSC access to specific object storage components. In BrainStor, every access is authorized according to the SCSI OSD protocol, while it is impossible to provide such security mechanism in a SAN file system deployment due to the limited block interface.

3.2 BrainStor Interfaces

BrainStor interfaces to clients are defined in OSD protocol. This SCSI command set is designed to provide efficient communication operations to OSD, which manage the allocation, placement, and accessing of variable-size data-storage containers, called objects [16]. By using this command set, OSC accesses BrainStor at object level.

3.2.1 Object Types and Commands

BrainStor system can contain the following object types according to OSD protocol [16].

- a) Root object: Each BrainStor system contains only one root object. The data of root object contains the list of Partition_IDs. And the attributes of root object contain global characteristics for the BrainStor system (e.g. the total capacity and number of partitions that it contains).

- b) Partition object: This kind of object is created by specific commands from an OSC. A partition contains a set of collections and user objects that share common security requirements and attributes. Some default values of partition attributes are copied from specified attributes in the root object. The data component of a partition is the list of User_Object_IDs.

- c) Collection object: This object is created by commands from OSCs. Support for collections is optional. It is used for fast indexing of user objects and operations involving multiple user objects. A collection is built within one partition. A partition may contain zero or more collections. A user object may be a member of many collections concurrently, or does not belong to any collections at all. Some default values of collection attributes are copied from specified attributes of the partition in which it is listed. The data component of a partition is the list of User_Object_IDs.

- d) User object: This object contains end-user data (e.g. file or database data). Its attributes include the logical size of the user data and time stamps for creation, access, and modification of the end user data. Some default values of user object attributes are copied from specified attributes of the partition in which it is listed.

Currently, BrainStor supports ten OSD SCSI commands:

- CREATE PARTITION (Service Action: 0x880Bh): to allocate and initialize a new partition, and to establish a new partition object as well.
- REMOVE PARTITION (Service Action: 0x880Ch): to delete a partition.
- CREATE (Service Action: 0x8802h): to allocate and initialize a user object.
- REMOVE (Service Action: 0x880Ah): to delete a user object.
- SET ATTRIBUTES (Service Action: 0x880Fh): to set attributes for a specified root, partition, or user object.
- GET ATTRIBUTES (Service Action: 0x880Eh): to get attributes for a specified object.
- WRITE (Service Action: 0x8806h): to write the specified number of bytes to the specified user object at the specified relative location.
- READ (Service Action: 0x8805h): to request storage modules to return data to the application client from a specified user object.
- OPEN (Service Action: 0x8804h): to communicate to BrainStor that a user object is to be accessed.
- CLOSE (Service Action: 0x8809h): to cause the specified user object to be identified as no longer in use.

In BrainStor, file metadata and some object metadata are centralized in the OMM and the object data is stored in the OSM. The FC communication between OSC and OMM is dedicated to metadata transition, which is named the Metadata Stream, as shown in Figure 3.3. The FC communication between OSCs and storage

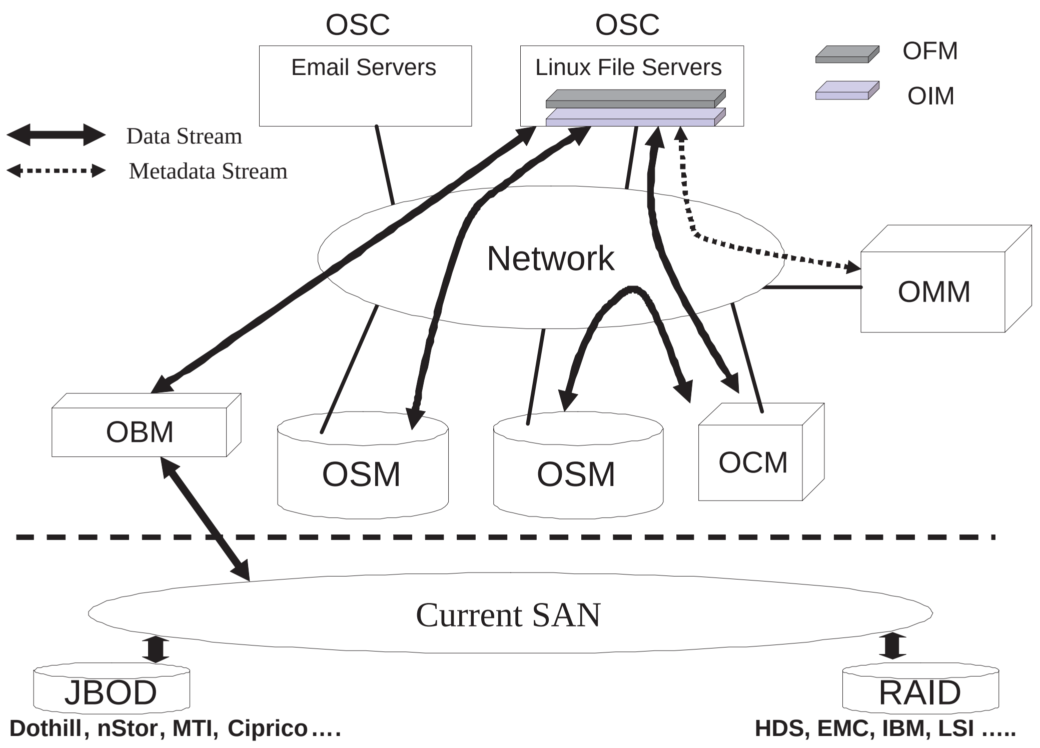


Figure 3.3: Data Access in BrainStor

nodes, such as OCM, OBM or OSM, is named the Data Stream. As can be seen in Figure 3.3, BrainStor have three different Data Streams. OSCs can access objects by directly requesting to OSMs. They can also request to OCMs for small objects and access object stored in general block SAN through an OBM.

3.2.2 Create and Write a New Object

Before an OSC accesses any data, it needs to create object partition by using CREATE PARTITION (0x880Bh) command. If the resources (e.g. free space) allow, the OMM creates a new object partition and return a unique partition ID to the OSC. The partition ID is then used in all the following access to the partition.

After the object partition is created, the OSC can create and access objects in that partition. Whenever the OSC wants to store data, if this is a new object, firstly the OSC sends OSD CREATE command to the OMM. Then the OMM creates an object ID (unique identity within BrainStor site) for this command and also generates a record to keep the metadata of this object. The object metadata includes the object ID and the OSM ID, which indicates the ID of OSM to store data of the object. The file-to-object mapping information and other security and QoS information are also stored in the metadata. Then, the OMM sends the response, which informs the OSC the new object ID and OSM ID. Finally, through the direct Data Stream, the OSC can store raw data of that object to the specified OSM. This procedure can be completed through OSD WRITE commands.

3.2.3 Read an Existing Object

Whenever an OSC wants to retrieve an object, firstly, through Metadata Stream, the OSC uses OSD SCSI commands (e.g. SET ATTRIBUTES and GET ATTRIBUTES) to access objects metadata in the OMM. If the requested object does not exist or the OSC does not have the access permission to that object, the OMM can reject OSC's requests. Otherwise, the requested metadata is sent to the OSC. Then after knowing the object metadata such as the object ID and ID of OSM storing the object, the OSC can initiate an OSD READ command to fetch the object from the OSM indicated by the OSM ID.

3.2.4 Access through OCM

When an OSC initiates random small I/O requests or requests to small objects, these requests go to the OCM instead of OSM. Then if other OSCs want to access the same data, they can directly fetch the data from the OCM. Moreover, the OCM can also merge random small requests into larger sequential requests. Small random requests can seriously degrade the performance of hard disk based storage, while larger sequential requests lead to high performance. Thus merging the small random I/O requests to large sequential I/O requests improves BrainStor small I/O throughput.

3.2.5 Access Example

In this example, the details of writing a new file to BrainStor system are shown. Suppose that a single file in a single subdirectory is copied to the BrainStor:

cp /dir1/file1 /mnt/BrainStor/

Where "dir1" is the name of the directory to be created and "file1" is the file to be written in that directory. It is assumed that root object of BrainStor and the partition object (*n, 0*) are known. The object partition of BrainStor has been mounted on the mount point, "/mnt/BrainStor/". It is also supposed that the OSC holds a valid capability for the following operations.

- Step 1: READ (Partition_ID: *N*, User_Object_ID: root directory ID): to read the content of root directory and check whether the "dir1" directory is already existed.
- Step 2: CREATE(Partition_ID: N): the OMM creates a new object in partition *N*, and return the User_Object_ID(f) to hold file "file1".
- Step 3: CREATE(Partition_ID: N): the OMM creates another new object in partition *N*, and return the User_Object_ID(d) to hold the content of directory "dir1".
- Step 4: WRITE(Partition_ID: *N*, User_Object_ID: f): to write contents of file1. If one WRITE cannot store all the data, there may be more than one WRITE commands needed.
- Step 5: WRITE(Partition_ID: *N*, User_Object_ID: d): to write contents of directory "dir1".
- Step 6: WRITE(Partition_ID: *N*, User_Object_ID: root directory ID): to update the content of root directory to contain directory "dir1".

3.3 BrainStor Nodes

3.3.1 Object Storage Client (OSC)

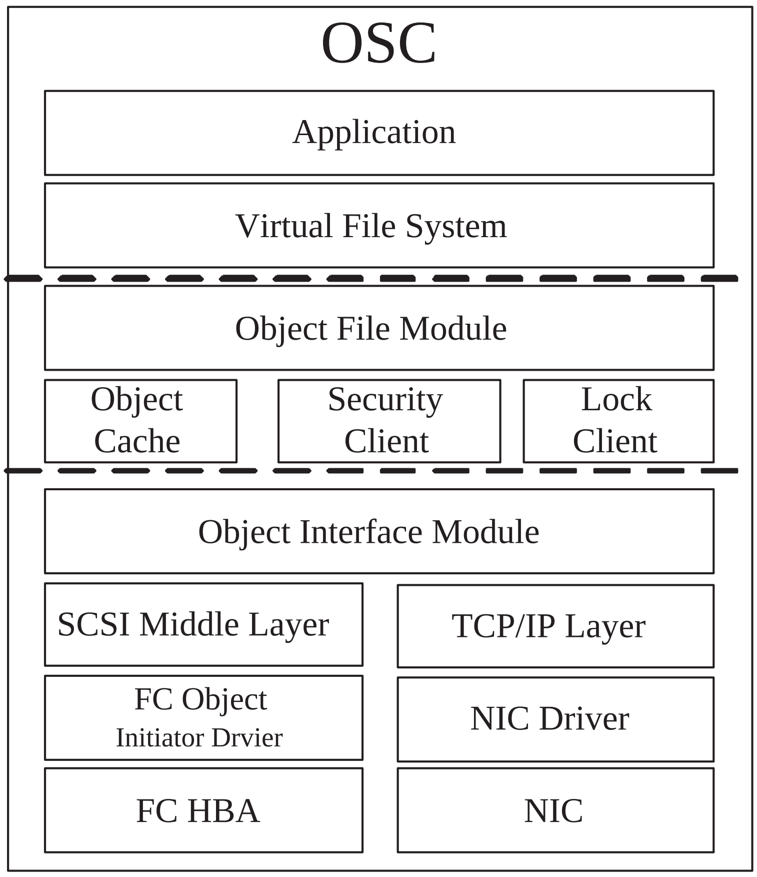


Figure 3.4: Object Storage Client (OSC) Architecture

An OSC is a server to outside network and a storage client to BrainStor. For example, it could be a Samba sever that provides file storing and sharing services to outside clients through Internet or Intranet. As a storage client, the OSC needs to request data for its application from other nodes in BrainStor. The aim of OSC's modules is to provide a set of interfaces to all kinds of server applications, and then these applications can freely access a virtual storage pool made up by all the other nodes within BrainStor.

The OSC is implemented in Linux and its internal software architecture is shown in Figure 3.4. Application module represents all kinds of applications, such as VoD server, email server, web server, database server and file server. If the applications are built up based on file access, BrainStor system can always support them.

```
static struct super_operations ofm_ops = {
        read_inode:             ofm_read_inode,
        dirty_inode:            ofm_dirty_inode,
        write_inode:            ofm_write_inode,
        put_inode:              ofm_put_inode,
        delete_inode:           ofm_delete_inode,
        put_super:              ofm_put_super,
        write_super:            ofm_write_super,
        write_super_lockfs:     ofm_write_super_lockfs,
        unlockfs:               ofm_unlockfs,
        statfs:                 ofm_statfs,
        remount_fs:             ofm_remount_fs,
        clear_inode:            ofm_clear_inode,
        umount_begin:           ofm_umount_begin
};
```

Figure 3.5: Super Operation APIs

```
static struct file_operations ofm_dir_operations = {
           read:           generic_read_dir,
           readdir:        ofm_readdir,
           fsync:          ofm_sync_file,
};

static struct file_operations ofm_file_operations = {
           read:           ofm_file_read,
           write:          ofm_file_write,
           mmap:           generic_file_mmap,
           open:           ofm_file_open,
           release:        ofm_release_file,
           fsync:          ofm_sync_file,
};
```

Figure 3.6: File Operation APIs

```
static struct inode_operations ofm_dir_inode_operations = {
        create:                 ofm_create,
        lookup:                 ofm_lookup,
        link:                   ofm_link,
        unlink:                 ofm_unlink,
        symlink:                ofm_symlink,
        mkdir:                  ofm_mkdir,
        rmdir:                  ofm_unlink,
        mknod:                  ofm_mknod,
        rename:                 ofm_rename,
};
```

Figure 3.7: Inode Operation APIs

```
static struct address_space_operations ofm_aops = {
        readpage:               ofm_readpage,
        writepage:              NULL,
        prepare_write:          ofm_prepare_write,
        commit_write:           ofm_commit_write
};
```

Figure 3.8: Address Space Operation APIs

The OSC contains two sub-modules: Object File-system Module (OFM) and Object Interface Module (OIM). All the modules at the OSC are Linux kernel-level modules. The OFM is a file system to Linux. The OFM registers its APIs with VFS, and VFS can pass application's data requests to the OFM through those standard file system APIs. Figure 3.5, 3.6, 3.7 and 3.8 show the primary APIs that the OFM supports. Figure 3.5 describes the operations that the super block of OFM supports. This set of APIs is mainly used to support file system metadata and inode access. Figure 3.6 gives the file level operations of both file and directory. Figure 3.7 shows the directory's inode APIs, which are used to manage the inodes in directory. Figure 3.8 defines the address space related operations, which are used to complete all the data transitions. They are called after it is confirmed that the system cannot find the requested data in its local memory. The OFM needs to register to VFS during its initialization phase. The following code is called in the function *init_module()* of kernel module *ofm*:

return register_file system(&ofm_fs_type);

After this, VFS can pass the application's data requests to the OFM by using those APIs defined in Figure 3.5 to Figure 3.8.

The primary jobs of OFM include hierarchy management, naming and user access control. The OFM performs metadata access from the OMM cluster and the mapping from the file requests to object I/O requests. The File Hashing Manager and Mapping Manager of HAP (introduced in Chapter 5) are function modules in OFM.

The OFM further contains Object Cache, Lock Client and Security Client sub-modules. The Object Cache provides object-level local cache to the OFM, while the Lock Client and Security Client play as the clients to support lock and security functions.

The primary work of OIM is to generate OSD SCSI commands from the object I/O requests and complete the object data access. The OIM needs to register to the SCSI mid-layer in Linux. The OIM controls two kinds of interfaces: Fibre Channel and Ethernet. Object data and metadata access use Fibre Channel while lock and security functions utilize Ethernet. It is because Fibre Channel is a storage protocol that is designed based on initiator and target modes, while TCP/IP is a communication protocol that is more suitable for the implementation of lock and security mechanisms.

3.3.2 Object Storage Module (OSM)

The OSM is the module that stores raw data objects. In data storage systems, there are two kinds of data: raw data and metadata. For example, when you save a movie file on disk, the data that represents the contents of movie is called raw data. In this case, the information describing this movie file, such as when the movie file is created, and where it is located, is called metadata. The OSM holds the raw data of that movie as user object.

Due to the unique characters of object, the OSM has more intelligence to

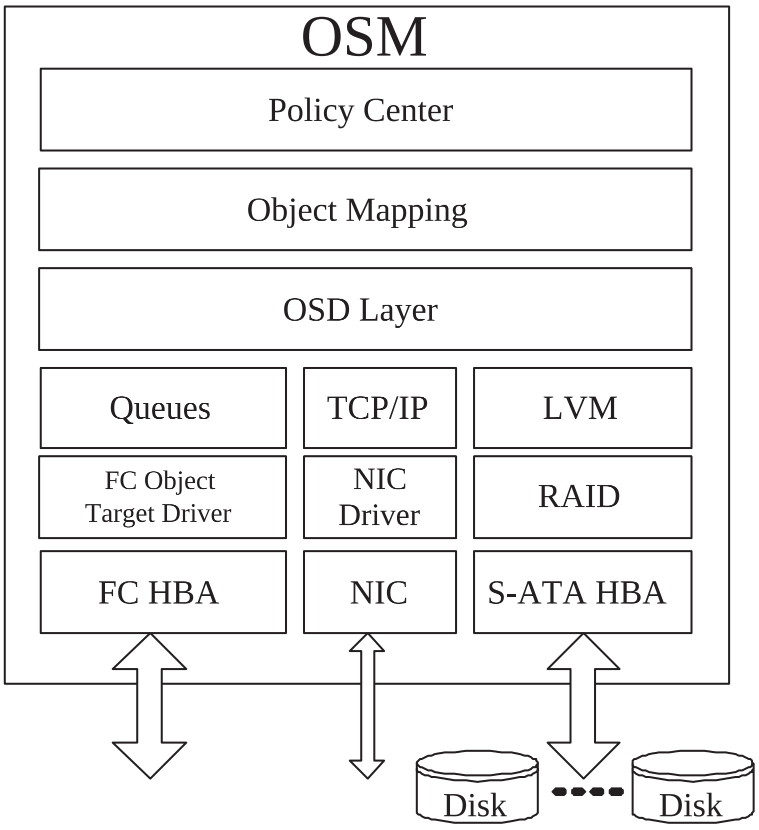


Figure 3.9: Object Storage Module (OSM) Architecture

manage its own storage. A block level storage (e.g. RAID system in SAN) can just store data according to LBA set by clients, while the OSM can decide the location of certain object according to metadata of that object. For example, the OSM can decide the allocation of an object based on its size and QoS attributes. In the prototype, one OSM holds sixteen 250G serial-ATA hard disks, and of course, there may be several RAID systems within one normal OSM. In case of locating that movie file, the OSM has the intelligence to put movie file in either a RAID 0 subsystem or a RAID 5 subsystem depending on the corresponding QoS attributes.

The OSM module is implemented in Linux and its internal software architecture is shown in Figure 3.9. The OSM has two kinds of interfaces: Ethernet and Fibre Channel. The Ethernet interface is used to communicate some management commands, e.g. OSM login command. Fibre Channel is used for data object access. The received OSD SCSI commands are put into several independent command queues. There is several independent kernel threads serving each queue, so that the OSM can achieve the parallel access at this level. This multi-thread parallel access mechanism can improve the disk access performance, and is especially helpful to small I/O requests. OSD Layer in Figure 3.9 provides the object interface and handle the incoming OSD SCSI commands. Object Mapping performs the mapping from object to block. For example, in order to access object 0x*abc*, object mapping module may indicate that logical block 0x*123* to 0x*321* store the data of this object. Then based on this mapping information, the OSM can complete the disk access for the object. Policy Center module performs all kinds of intelligent functions, such as QoS based allocation policy. At the low level, the OSM uses Logical Volume Manager [43] and RAID controller to manage the low-level block access. In BrainStor prototype, each OSM has two 8-channel 3ware SATA RAID adapters to access sixteen 250G SATA hard disk. Therefore, each OSM module has up to 4000G storage capacity. All storage resources are further integrated in RAID subsystems. The OSM supports the hardware RAID (including RAID 0, 1, 5, 10) by each 8 channel 3ware SATA RAID controller and the software RAID that can be used between two independent RAID controllers, such as RAID 50.

3.3.3 Object Cache Module (OCM)

The OCM is the centralized cache for all the other storage modules in BrainStor. The small random access is the performance killer to disk-based storage, because those accesses need frequent physical movements of magnetic heads, which introduces additional seek overhead compared to sequential access. On the other hand, data access in memory is based on electronic access, therefore the small random access can reach performance as good as the large sequential access. The introduction of OCM greatly improves the system's capability of handling small and random requests.

In BrainStor prototype, a single OCM is with 8G memory capacity. Only I/O requests to small objects are forwarded to the OCM. The definition of small

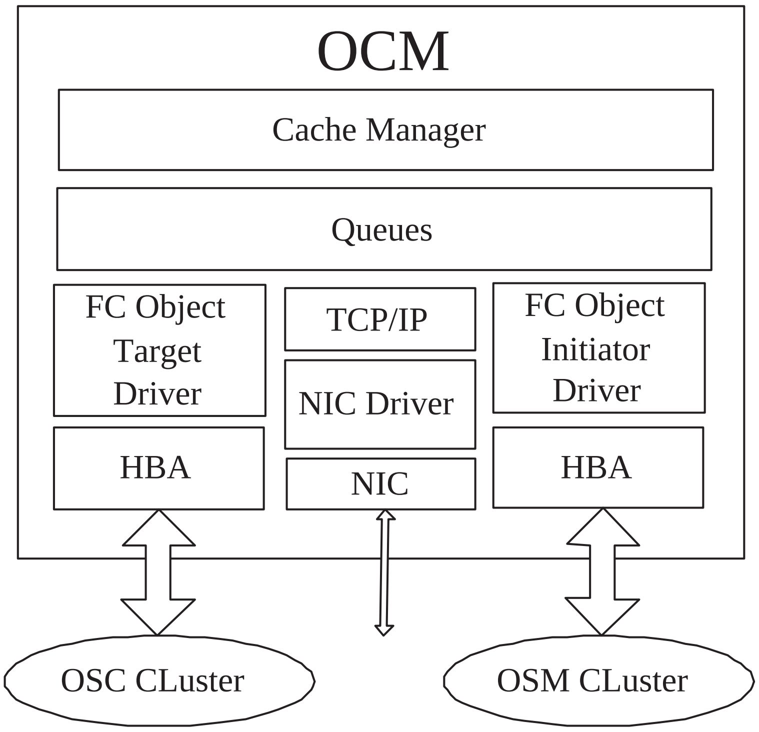


Figure 3.10: Object Cache Module (OCM) Architecture

object is that object with the size that is less than a preset value, such as 16KB. On the other hand, all I/O requests to non-small objects go to the OSM directly.

The OCM also has a destage mechanism. Because the OCM does not provide persistent storage of objects and has limited cache capacity, the OCM finally needs to destage all data to the OSM. When the OCM is vacant, the OCM may write data to the OSM in order to synchronize objects. In this case, the written objects are still kept in the OCM for future access. When the OCM does not have enough memory for new object requests, the OCM need perform object replacement. Some objects will be written to the OSM and removed from the OCM. The replacement algorithm can utilize certain memory replacement policies, such as LRU, FIFO, and LFU [44]. The OCM can choose an algorithm for small objects based on different application workload characters.

Because the OCM needs to destage data to the OSM, there may be a "page fault" when the OSC wants to access a small object that has already been destaged to the OSM. Therefore, in order to handle the "page fault", the OCM also needs to retrieve object back from the OSM and cache it for the potential future access.

The internal software architecture of OCM is shown in Figure 3.10. The OCM has two Fibre Channel interfaces. One is used to communicate with OSCs and a Fibre Channel object target mode device driver controls it, and the other is used to communicate with the OSM cluster and a Fibre Channel object initiator mode device driver controls it. Queues in Figure 3.10 are used for the received object SCSI commands from clients. Cache Manager module handles all the requests. In addition, cache manager also needs to implement the cache destage and replace mechanisms. During the destage phase, cache manager can generate OSD WRITE commands to store data to the OSM cluster. In case of “page fault”, cache manager also needs to use OSD READ command to read data from the OSM.

As discussed above, the OCM is a centralized cache for the entire BrainStor system. The OCM is managed by the OMM and accessible to all the OSCs. In addition, the OCM is scalable. The OMM can easily make OSCs access a new OCM without any downtime. The feature is very important because the unpredictable and increasing need of handling small random data requests. The scalability of OCM cluster is a unique feature of BrainStor.

3.3.4 Object Bridge Module (OBM)

The OBM makes BrainStor network compatible with existing block network and hardware. The OBM can map the object requests to block access commands that can be completed by normal block storage device, e.g. RAID system, in current SAN. After a data center adopts BrainStor system, it can still use their exiting SAN infrastructure and storage hardware.

The software architecture of OBM is shown in Figure 3.11. The OBM also has two Fibre Channel interfaces. One is connected to BrainStor network, and a Fibre Channel object target mode device driver controls the interface. Object SCSI requests are coming through this interface and queued. The other Fibre Channel

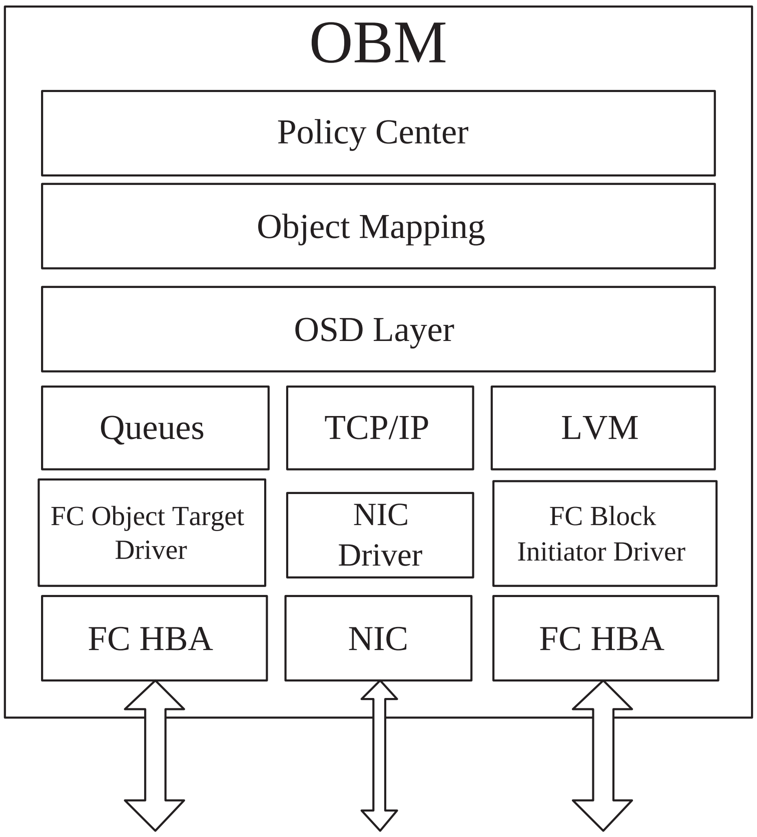


Figure 3.11: Object Bridge Module (OBM) Architecture

interface is connected to a SAN and a Fibre Channel initiator mode device driver controls it. This interface can pass normal block SCSI command to access block storage. The OSD layer, Object Mapping and Policy Center are similar to modules in the OSM.

3.3.5 Object Manager Module (OMM)

The OMM is the management center of BrainStor. First of all, the OMM holds metadata, which is used by the OSC to access objects. For instance, the OSC must know information like which OSM in the OSM cluster contains the needed object before it initiates any requests. Secondly, the OMM holds global information about BrainStor, such as the number of available OSMs, the access modes of each OSM and access priority of each OSC. Every device needs to login OMM when it boots up. In addition, the OMM has intelligent functions such as storage virtualization.

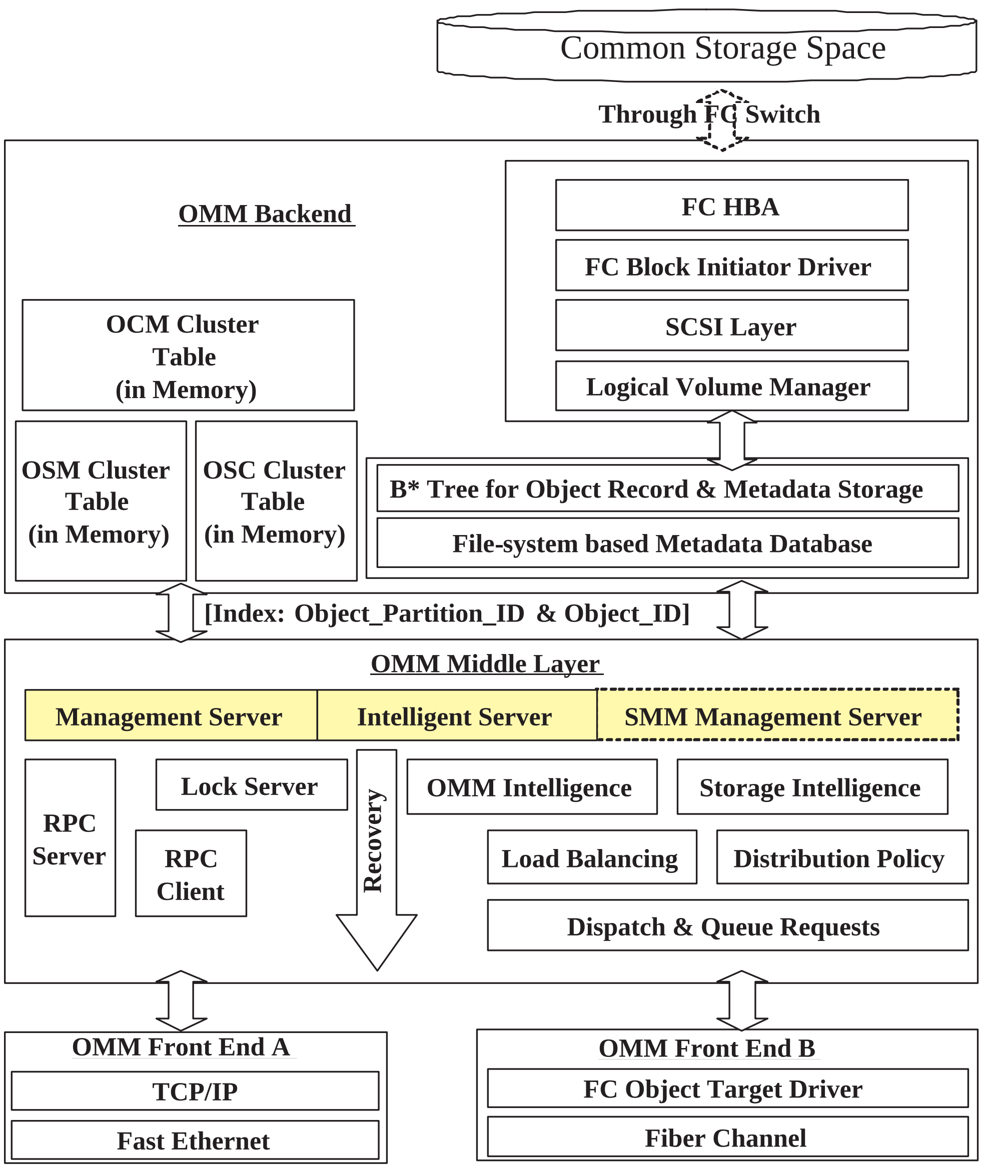


Figure 3.12: Object Manager Module (OMM) Architecture

In BrainStor prototype, the OMM provides different virtual disk images to different OSCs. That means, although all OSCs share the same storage pool, different OSCs have different views and different access rights to the same storage pool. For example, OSC1 may view the BrainStor as 2T virtual disk and OSC2 regards the same BrainStor as 3T virtual disk. Actually both storage space are allocated across all the OSMs, thus both OSCs can achieve the best parallel access performance.

The internal software architecture of OMM is shown in Figure 3.12. The OMM consists of three parts: the OMM front-end, the OMM middle layer and the OMM back-end.

The OMM front-end includes low level drivers to control the hardware interfaces. One Fibre Channel interface is used to communicate with OSCs, while the Ethernet interface can help to set up a management channel with all the other nodes. As shown in other software architecture figures, every node has an Ethernet interface, and there is a message passing channel by using socket communication. Every node has two threads serving the Ethernet interface. One is used to receive messages from other nodes, and the other is used to send messages to certain node indicated by the IP address. Thus all the nodes can communicate with each other through Ethernet.

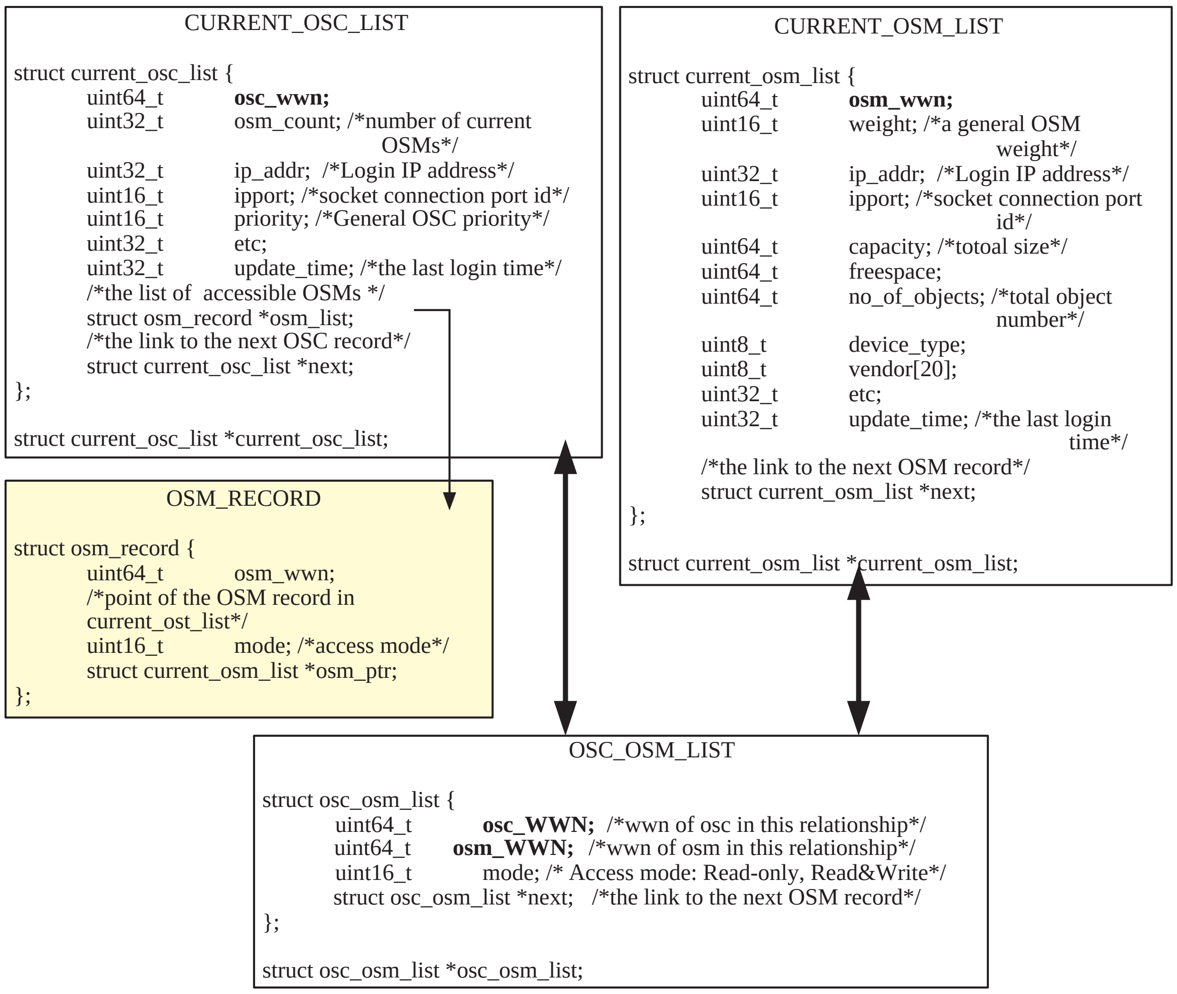
The OMM middle layer performs three works: responding to metadata requests, intelligent functions and management work. The basic function of OMM is to handle all coming metadata requests from Fibre Channel interface. In addition, the OMM needs to provide an object-level lock mechanism because there is an OSC cluster accessing BrainStor instead of one OSC. A lock server in the OMM performs the task.

Intelligent functions of OMM middle layer support services, such as load balancing, storage virtualization, OMM cluster load balancing, OMM cluster failover and scalability. The Intelligent Server module administrates intelligent functions. Load balancing means that the data object can be evenly distributed to the OSM cluster by the OMM, therefore the OSC can access objects from different OSMs in parallel. Storage virtualization is another key feature of BrainStor. All the OMM cluster and other nodes are transparent to OSC applications, which simply regard BrainStor as a virtual storage with very huge storage capacity. Moreover, the scalability of OSM cluster is transparent to OSCs. Even if the huge virtual storage cannot satisfy the storage requirement of application servers, it can dynamically grow to provide unlimited storage without any downtime of OSCs' applications. OMM load balancing, OMM cluster failover and scalability functions will be discussed in Chapter 5. The Logical Partition Manager of HAP (in Chapter 5) is actually one function module of Intelligent Functions in OMM middle layer.

The Management Server module in OMM middle layer manages all the information about BrainStor. All other nodes need to login or report their own information or status to the OMM by using some RPC commands. For example, a logout command is used to report the departure of a node to the OMM. Thus there is a RPC server that processes all the coming management commands from other nodes. Each OMM also has a RPC client in order to communicate with other OMMs. In addition, the OMM needs to detect the removal and addition of all the other nodes. Although nodes can report their removal to the OMM, some node failures, for example power failure, leave no time to logout. In short, the OMM needs to maintain all the information needed by the intelligent functions.

The OMM back-end performs the real metadata access. The back-end is self-developed database with outstanding cache ability. The cache performance is even comparable with that of file systems. The back-end of each OMM can exclusively control and access one or several logical partitions in the common storage space. Hence the OMM back-end can provide great performance without the concurrence control problem. In addition, because the OMM back-end accesses the logical partitions through a small SAN at block level, another Fibre Channel interface is used to complete this access. The logical partitions and the common storage space are discussed in detail in Chapter 5.

The OMM back-end maintains tables about current BrainStor setup. There are the OSM cluster table, the OSC cluster table and the OCM cluster table as shown in Figure 3.12. Because the OBM is treated as OSM in the OMM, the OBM cluster information is also kept in the OSM cluster table. Figure 3.13 describes the data structure *current_osc_list*, *current_osm_list* and *osc_osm_list*. *current_osc_list* and *current_osm_list* record information of the active OSCs and OSMs in BrainStor, respectively. The *osc_osm_list* records the relationship between OSCs and OSMs. There are three modes of relationship: *no-read-no-write*, *read-only*, and *read-write*. Based on the relationship list, *current_osc_list* structure also maintains a local *osm_list* that provides a quick index of the accessible OSMs for an OSC. In order



```
CURRENT_OSC_LIST

struct current_osc_list {
        uint64_t        osc_wwn;
        uint32_t        osm_count; /*number of current
                                   OSMs*/
        uint32_t        ip_addr;  /*Login IP address*/
        uint16_t        ipport; /*socket connection port id*/
        uint16_t        priority; /*General OSC priority*/
        uint32_t        etc;
        uint32_t        update_time; /*the last login time*/
        /*the list of  accessible OSMs */
        struct osm_record *osm_list;
        /*the link to the next OSC record*/
        struct current_osc_list *next;
};

struct current_osc_list *current_osc_list;
```

```
CURRENT_OSM_LIST

struct current_osm_list {
        uint64_t        osm_wwn;
        uint16_t        weight; /*a general OSM
                                weight*/
        uint32_t        ip_addr;  /*Login IP address*/
        uint16_t        ipport; /*socket connection port
                                id*/
        uint64_t        capacity; /*totoal size*/
        uint64_t        freespace;
        uint64_t        no_of_objects; /*total object
                                number*/
        uint8_t         device_type;
        uint8_t         vendor[20];
        uint32_t        etc;
        uint32_t        update_time; /*the last login
                                     time*/
        /*the link to the next OSM record*/
        struct current_osm_list *next;
};

struct current_osm_list *current_osm_list;
```

```
OSM_RECORD

struct osm_record {
        uint64_t        osm_wwn;
        /*point of the OSM record in
        current_ost_list*/
        uint16_t        mode; /*access mode*/
        struct current_osm_list *osm_ptr;
};
```

```
OSC_OSM_LIST

struct osc_osm_list {
        uint64_t        osc_WWN;  /*wwn of osc in this relationship*/
        uint64_t        osm_WWN;  /*wwn of osm in this relationship*/
        uint16_t        mode; /* Access mode: Read-only, Read&Write*/
        struct osc_osm_list *next;   /*the link to the next OSM record*/
};

struct osc_osm_list *osc_osm_list;
```

Figure 3.13: Data Structure of OMM Tables

to handle a coming request from an OSC, such as OSD Create command, the OMM can find out the available OSMs directly from the *current_osc_list* of the OSC.

3.3.6 Security Manager Module (SMM)

The SMM performs the security manager functions defined in OSD protocol. The SMM generates credentials at the request of an authorized OSC, and also returns a capability key with each credential. The credential gives the OSC access to specific object storage components. The capability key allows the OSC and storage nodes to authenticate the commands and data they exchange.

BrainStor adopts the OSD security model, defined in the OSD protocol, which is a credential-based access control system. The fundamental element in an object based security system is a cryptographically capability that encloses a tamper-proof description of the rights of a client. Out of the main data path, the SMM can create this capability that represents the security policy. With possession of this capability, the client can access the storage nodes, and it is the job of the storage nodes to validate the integrity of the capability to ensure that neither it nor the request has been modified. Particularly, without maintaining client-specific authentication information on storage nodes, BrainStor can scale independently from the number and types of clients in the system. Moreover, the capability is created out-of-band, thus it is not a bottleneck. The credential gives the application client access to specific objects. Clients present these capabilities to OSM on every I/O request. The request sent to an OSM includes the command, the OSC capability, and a digest (integrity check value of the entire request with the capability key).

The OSM needs to validate the integrity of the capability to ensure that neither it nor the request has been modified. Secrets shared ONLY between SMM and the storage nodes are used to generate a keyed hash of the capability, which is just the capability key. Therefore, it can protect both the capability and the whole client request from modification by the client itself or by the man-in-the-middle. Upon receipt of a new request, the OSM firstly computes its own key based on the capability presented in request and the secrets shared between it and the SMM. If the capability is correct, OSM's own key should be equal to the capability key. Then the OSM can validate the client's digest by matching it with its own keyed hash of the request (using OSM's own key). If they match, there is no modification of the client's capability and the request. Then, the OSM can process the request. An application client that only has the capability (e.g. obtained by monitoring CDBs sent to the OSM) but not the capability key is unable to generate commands with valid integrity check value. Then the OSM can deny the unauthenticated access of OSCs. Therefore, every request is authorized by the SMM and validated by storage nodes. In addition, the man-in-middle attacks can also be detected.

As defined in OSD protocol, the SMM may reside in the OMM, in the OSM, in the OSC, or as a separate entity, however the security requirements on the communications mechanism shall not change based on the location of the SMM [16]. In BrainStor design, the SMM is designed as an installable software module. It can be integrated with the OMM as shown in Figure 3.12 or just works as an independent server. Wherever the SMM is, it should be out-of-band and support clustering of all the other nodes in BrainStor.

3.4 BrainStor Virtualization

In an ideal storage, users treat storage devices as a virtual storage space with unlimited capacity. All the internal scalability and errors are transparent to users. Storage virtualization gathers all physical storage resources into a single pool. From a central and simple interface, network administrators can administrate common policies and services across the entire storage pool. This is independent of the vendor brand, type, and protocol represented by each physically attached storage system.

In a SAN, there are in-band or out-of-band solutions to the virtualization design. The virtualization control function of in-band solution resides on a dedicated appliance within the data-path, as shown in Figure 3.14. Application servers need to send their data access commands to a virtualization server through fibre, and the server can further complete the requests by accessing storage systems connected to the SAN. Data transitions between virtualization server and storage components can be parallel access. For example, the IPStor from FalconStor company is a typical in-band storage virtualization product. One drawback of this solution is that the virtualization server becomes an obvious bottleneck because all the I/O requests need to go through it.

On the other hand, the out-of-band virtulization solution removes the virtualizaton server from data path, as shown in Figure 3.15. Application servers

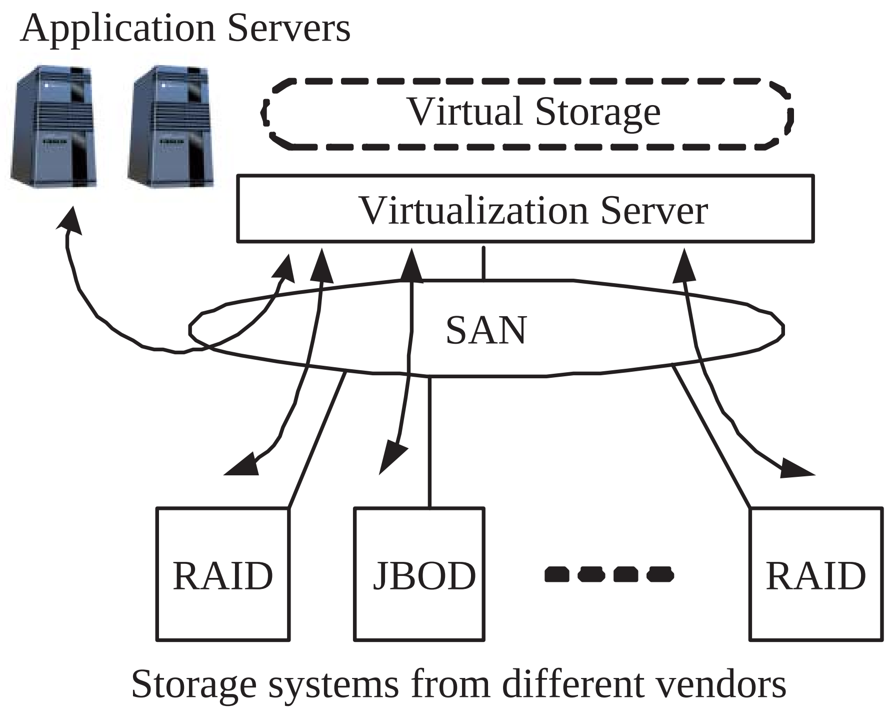


Figure 3.14: In-band Storage Virtualization

can get the necessary virtualization information (e.g. the WWNs of available storage components) from an out-of-band virtualizaton server, which maintains the information about all the storage and their configuration.

BrainStor adopts the out-of-band solution, because it already has a centralized metadata center, the OMM cluster. All metadata requests of client are sent to the OMM cluster. With knowledge of the overall setup of BrainStor, the OMM is able to allocate OSMs to the OSC, based on its own policy. For example, different weight and priority can be assigned to all the OSCs and OSMs, as shown in Figure 3.13. OSMs with certain weight can be reserved for OSCs with the high priority. The OMM can send this storage virtualization information (e.g. a list of OSM_ID) as part of the metadata. Object file system can get the OSM_ID list from returned metadata. Then the coming raw data requests can be directly sent to the specified OSMs. Therefore, in order to access its own data, one OSC may even utilize the entire OSM cluster, which is completely transparent to its applications.

Another feature of storage virtualization is its support to storage scalability. The OMM cluster has all the nodes status information, and it can also periodically

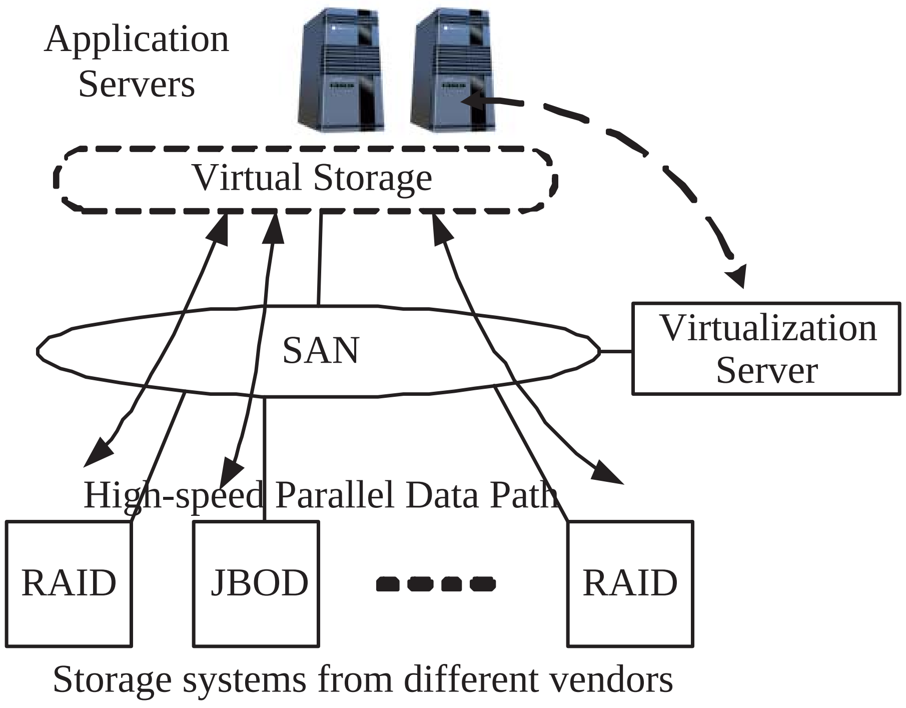


Figure 3.15: Out-of-band Storage Virtualization

check the BrainStor network. Hence any changes, e.g. the addition of new OSM, can be dynamically detected and responded. During the runtime, BrainStor virtualization supports the addition and removal of OSCs, OCMs, OBMs and OSMs. All the changes and corresponding process are transparent to clients' application. After the OSC boots up, it can treat the BrainStor as a huge virtual storage pool with almost unlimited capacity. During the runtime, if BrainStor detects that there is potential possibility of out-of-space, it can inform the storage administrator to add more storage dynamically. All these operations do not affect OSC's applications at all. Hence there is no downtime due to the scaling of storage capacity.

3.5 Summary

BrainStor is an object storage, which aims at providing an intelligent storage solution. BrainStor introduces new modules, such as a centralized Object Cache Module and Object Bridge Module. There are six nodes in BrainStor. OSCs can be all kinds of application servers, such as email servers and Video-on-Demand

(VoD) server. The OSM cluster is the storage place for raw data object. The OCM cluster is a centralized cache cluster used to accelerate the access of storage. The OMM cluster manages all the object metadata and file metadata. The OBM can make the BrainStor network compatible with the existing storage network and devices. The SMM provides the security for BrainStor network.

In BrainStor, the OSC can contact the OMM cluster for metadata and access object through OCM, OSM or OBM. The internal software models of OSC, OSM, OCM, OBM and OMM are discussed in detail. The SMM in BrainStor follows the security model defined by OSD protocol. BrainStor adopts the out-of-band storage virtualization solution. The OMM cluster works as the out-of-band virtualization server in BrainStor.

Chapter 4

Experiment and Result Discussion

4.1 BrainStor Prototype

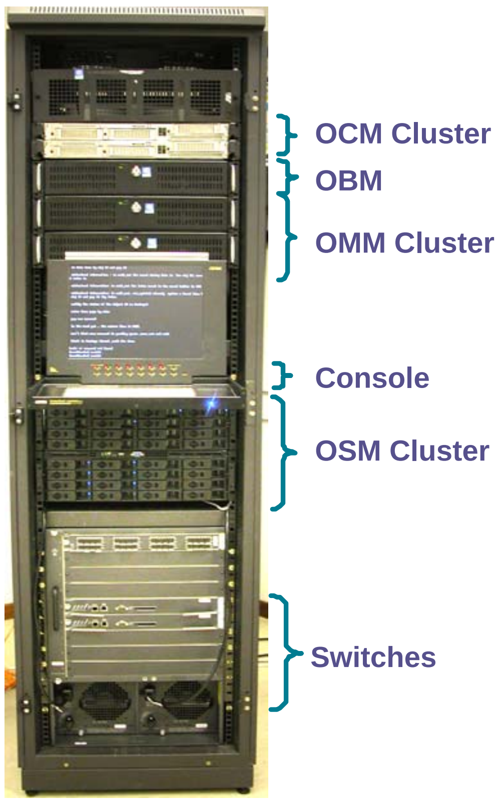


Figure 4.1: Current BrainStor Prototype

Figure 4.1 shows a picture of BrainStor prototype in the lab. Core modules include OSC, OMM, OCM, OBM and OSM. This BrainStor prototype is an OSD prototype over Fibre Channel network. Main features of BrainStor prototype are

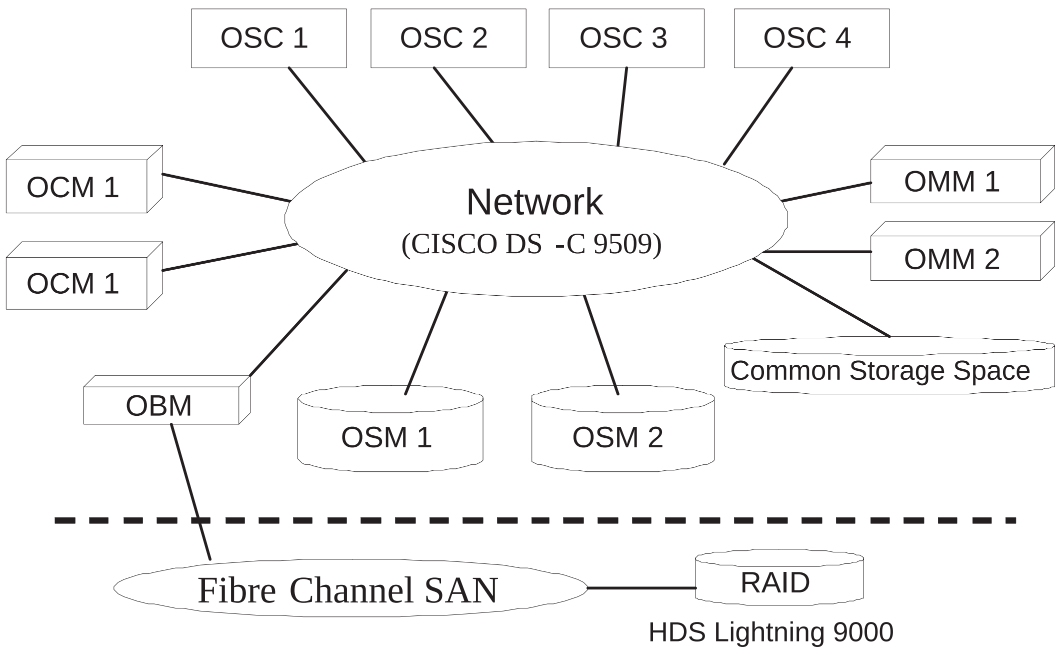


Figure 4.2: BrainStor Prototype Logical Connection

summarized as follows:

- Develop an OSD prototype over Fibre Channel network
- Define and develop a centralized Object Cache Module
- Define and develop an Object Bridge Module
- Preliminary results: 145MB/s for single OSC and 190MB/s for single OSM over 2G FC
- OSM cluster storage virtualization
- OMM cluster dynamic load balancing, scalability and failover support
- Integrate OSD storage with email server (Sendmail)

Figure 4.2 presents the corresponding internal logical connection of modules in the current BrainStor prototype shown in Figure 4.1. All the nodes are connected to FC switch, CISCO DS-C 9509 director, and Ethernet switch, Compex DSR2216. The current version of BrainStor prototype already supports the clustering of all the nodes. A RAIDTec JBOD (not shown in Figure 4.1) is used as common storage space. The OBM connects to a block-based SAN and accesses LUNs in the HDS Lightning 9000 storage system, located in DSI's Network Storage Lab. The hardware configurations of each modules are shown in Table 4.1.

Table 4.1: Hardware Configuration of BrainStor Nodes in Experiments

Node Type	OMM	OCM	OBM	OSM
CPU	Intel Xeon 2.4GHz	Intel Xeon 2.4GHz	Intel Xeon 2.4GHz	Intel Xeon 2.4GHz
Memory	1G DDR266 ECC memory	8G DDR266 ECC memory	512M DDR266 ECC memory	512M DDR266 ECC memory
Storage	The common storage space (RAIDTec JBOD)	Use memory as storage	SAN storage	16x250G SATA WD HDD (7200RPM)
Fibre Channel	2xQlogic FC adapter	2xQlogic FC adapter	2xQlogic FC adapter	1xQlogic FC adapter
Ethernet	Onboard NIC (1000Mbps)	Onboard NIC (1000Mbps)	Onboard NIC (1000Mbps)	Onboard NIC (1000Mbps)
Linux Kernel	2.4.20	2.4.20	2.4.20	2.4.20
RAID controller	No	No	No	Yes (3ware 8500 8xSATA RAID Controller)
Motherboard	TYAN s2722	SuperMicro X5DPI-G2	TYAN s2722	TYAN s2712

4.2 BrainStor Experiments

BrainStor experiments are designed to benchmark the BrainStor architecture and further identify the key issues in real OSD system developments. Because the object is a concept between file and block, test tools include block-level benchmark tool, Iometer [47] as well as file system benchmark tool, IOzone [48]. PostMark test tool[49] is also used to evaluate the access performance to small files. In addition, the Finisar Fibre Channel analyzer is used to verify the throughput at physical level.

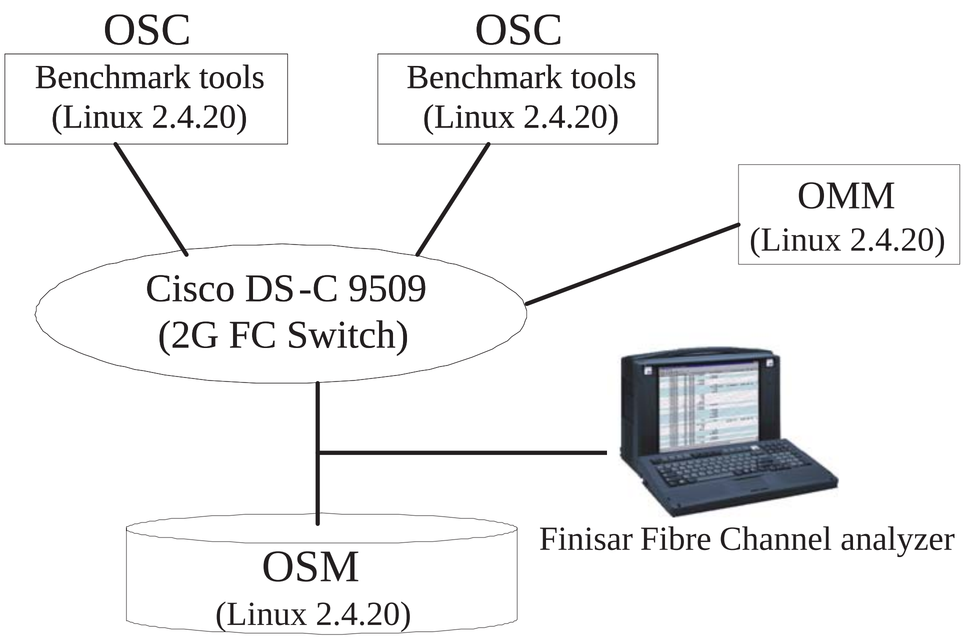


Figure 4.3: Typical Test Setup

A typical test setup is shown in Figure 4.3. The hardware configurations of all the nodes are shown in Table 4.1. The OSM adopts a 16-HDD RAID 0 for the best performance. All the nodes used in test are connected through a Fibre Channel Director (CISCO DS-C 9509). The Finisar Fibre Channel analyzer can be used to monitor the physical transition on the fibre. Normally, it is connected between the Fibre Channel switch and the OSM module in order to verify the real data transition performance. In the following tests, the effects of OSCs' local cache are minimized. The test results are the physical data transition results, which are already verified by the analyzer.

4.2.1 Iometer Test

The purpose of Iometer test is to benchmark the BrainStor prototype from block device test point of view. Iometer is an industry standard benchmark tool to block devices, such as disk and RAID systems [47]. Following experiments use the Linux version Iometer (2003.12.16) in the OSC system.

The explanations of different configuration symbols used in Iometer tests are as follows:

- 1-OSC-r/1-OSC-w: BrainStor uses the basic setup, which includes one OMM and one OSM. There is one OSC running the Iometer read or write test.
- 2-OSC-r/2-OSC-w: BrainStor uses the basic setup, which includes one OMM and one OSM. There are two OSCs running the Iometer read or write test simultaneously.
- 3-OSC-r/3-OSC-w: BrainStor uses the basic setup, which includes one OMM and one OSM. There are three OSCs running the Iometer read or write test simultaneously.
- 4-OSC-r/4-OSC-w: BrainStor uses the basic setup, which includes one OMM and one OSM. There are four OSCs running the Iometer read or write test simultaneously.
- 4C-2OSM-r/4C-2OSM-w: BrainStor prototype includes one OMM and two OSMs. There are 4 independent OSCs running the Iometer read or write test simultaneously.
- 4C-2OSM-r1(2)/4C-2OSM-w1(2): BrainStor prototype includes one OMM and two OSMs. There are 4 independent OSCs doing the Iometer read or write test simultaneously. 1 and 2 indicate the result at OSM1 or OSM2 respectively.

The above naming method is used in all the Iometer test results. Table 4.2 details the primary Iometer settings used in the tests [47]. The benchmark criterias used in Iometer tests include performance, I/O per second, average response time and OSM CPU utilization.

Table 4.2: Iometer Configuration in Experiments

Configuration Name	Setting
Number of Mangers	One manager per OSC
Number of Workers per Managers	3
Outstanding I/Os	4
Test Connection Rate	4
Read/Write	100% read or 100% write
Access Pattern	100% Sequence
Request Size	16KB - 512 KB
Ranp Up Time	30 seconds
runtime	300 seconds

4.2.1.1 Iometer Read Test

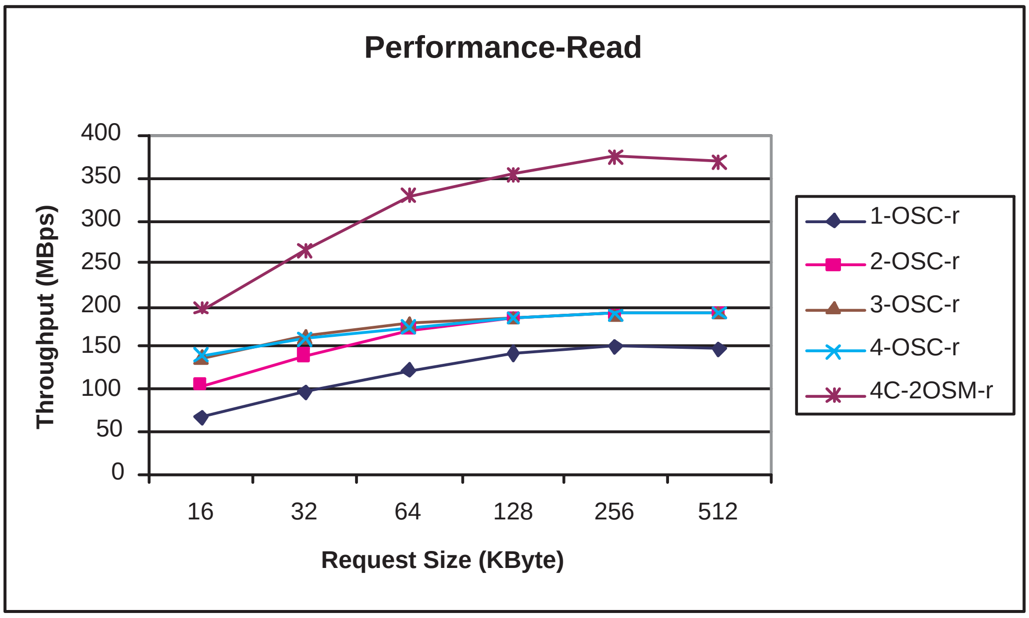


Figure 4.4: Performance in Iometer Read Test

Figure 4.4 shows the read performance. The x axis indicates the size of read requests (*KByte*) and the y axis shows the read performance (*MBps*, Mega-Byte per second). As shown in results, when the number of OSC is increased from 1 to 2, the performance increases sharply. The best performance of BrainStor is increased from 145.8MBps to 191MBps at 512KB request size when the number of OSCs increases from 1 to 2.

When the third OSC is added, the performance with request size larger than 64KB cannot achieve obvious improvement (less than 5%). When the fourth OSC is added, the performance is similar as that of 3 OSCs (less than 2% variation).

This scenario is because the OSM already has a heavy load when there are two OSCs runing the read test. The performance bottleneck is at the OSM side when there are 4 OSCs running. The OMM is not a bottleneck because Iometer test does not involve many metadata access. In addition, when the throughput approaching the maximum, the increasing request size cannot improve the performance significantly. So the throughput of requests larger than 128KB tends to stabilize, as shown in Figure 4.4.

Therefore, similar to the practical situation, another OSM can be dynamically added in order to overcome the bottleneck. As can be seen in Figure 4.4, the performance increases sharply again when 4 OSCs do Iometer read test based on BrainStor prototype with 2 OSMs. The best performance can reach 374MBps when 4 OSCs test the prototype with read requests at the size of 256KB. Thus, the tests show that BrainStor supports storage scalability, which can improve not only the capacity but also the performance.

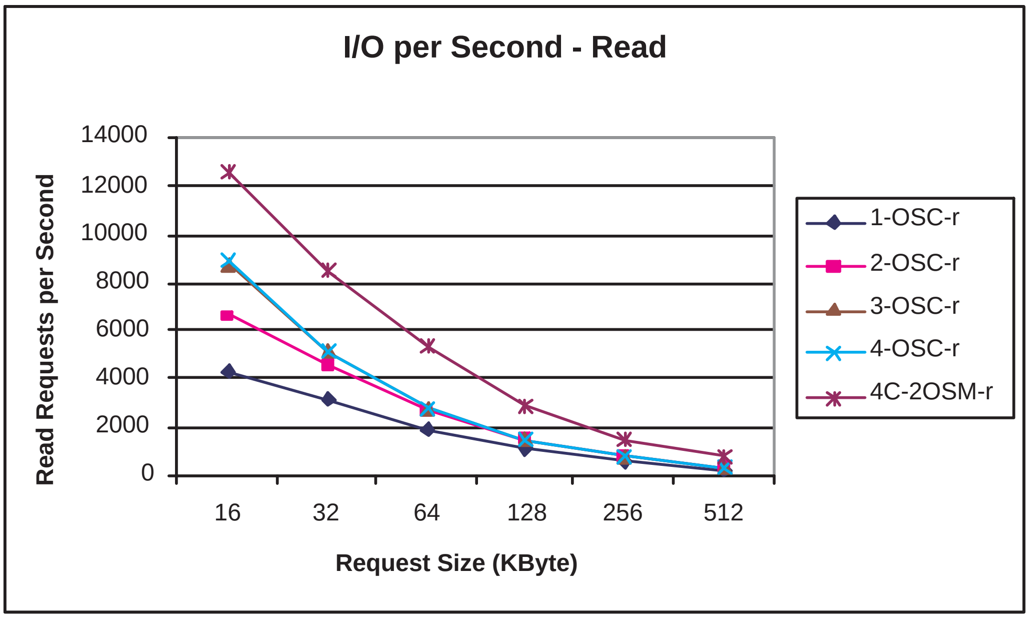


Figure 4.5: IOps in Iometer Read Test

Figure 4.5 shows the I/O number per second (IOps) during the read tests. The x axis represents the size of read requests and the y axis indicates the IOps.

Normally, the larger the request size, the lower IOps, because more time are needed to process a large I/O request than a small I/O request. In case of the large I/O request, less commands are needed to be transmitted through fibre and processed by both initiator and target. As a result, large request size always leads to better performance.

The changes under different setup are similar to the changes in the read performance as shown in Figure 4.4. The IOps with 3 OSCs and 4 OSCs are similar due to the load limitation of one OSM. Therefore after another OSM is added, there is obviously increasing in terms of IOps. Thus, the OSM scalability can also improve the I/O processing ability of BrainStor.

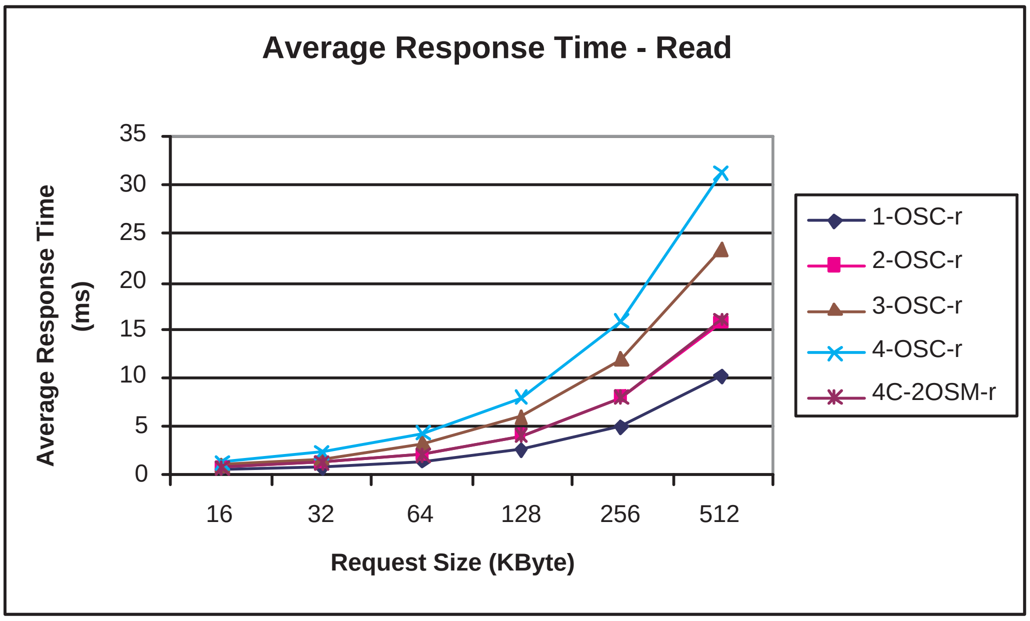


Figure 4.6: Average Response Time in Iometer Read Test

Figure 4.6 shows the average response time in the read tests. The x axis indicates the size of read requests and the y axis represents average response time.

Normally, the larger the request size, the longer the response time. Figure 4.6 shows that the average response time keeps increasing with the number of OSCs. For example, although the performance of 3-OSC-r and 4-OSC-r are similar, the response time of 4-OSC-r is much higher than 3-OSC-r. This is also due to the limitation of OSM processing capability.

The addition of one OSC running Iometer means that there are more I/O requests generated at the same time. However, because the OSM has already reached its maximum throughput, although 4 OSCs can generate more I/O requests in a unit time, the OSM cannot complete them all immediately. More I/O requests fail to increase the overall performance. Therefore many requests have to wait in the queue and the average response time increases.

From Figure 4.6, it can be seen that after another OSM is added, the average response time drops sharply and reaches the same level as 2-OSC-r. Thus, adding OSMs can also reduce the average response time.

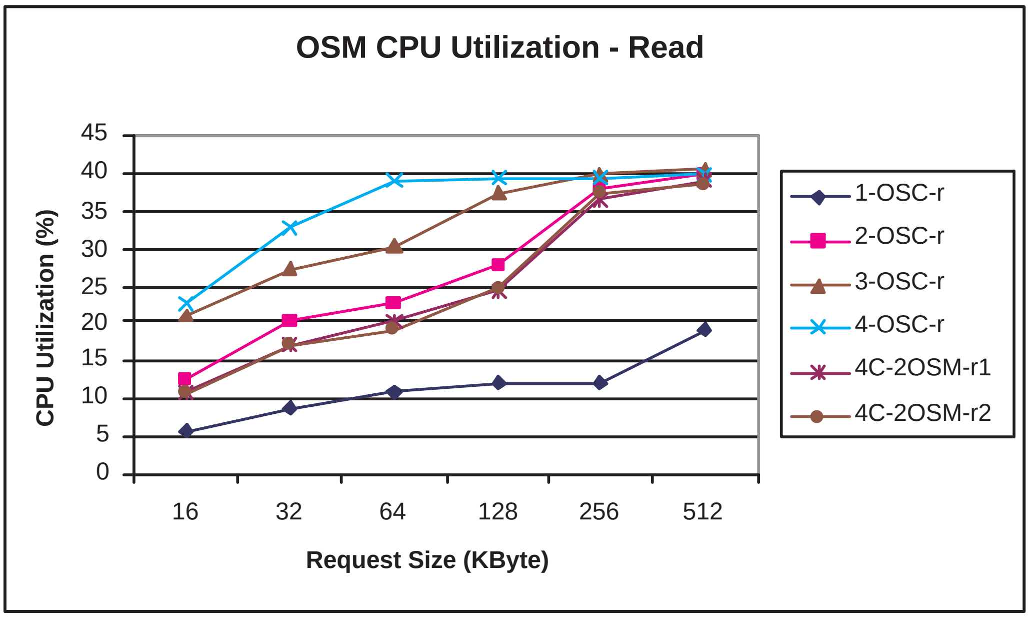


Figure 4.7: OSM CPU Utilization in Iometer Read Test

Figure 4.7 shows the OSM CPU utilization during the read tests. The x axis represents the size of read requests and the y axis indicates the CPU utilization.

As can be seen in Figure 4.7, the OSM CPU utilization is increasing with its performance in a certain test setup. The results from 4C-2OSM-r1 and 4C-2OSM-r2 are similar and a bit less than that of 2-OSC-r. Clearly, two OSMs can almost share all the read requests evenly, which means that BrainStor can load balance between the OSM cluster.

One interesting observation is that although the performance of 3-OSC-r and 4-OSC-r is similar, their respective CPU utilizations differ a lot. There are about 30% difference when the request size is 64KByte. It shares the same reason with the difference in the average response time. As discussed above, more OSCs means that there are more coming requests waiting in command queues. Hence extra CPU cycles are taken to process the incoming requests and maintain command queues.

Figure 4.7 shows that the highest OSM CPU utilization is 40.5% under the read test. Therefore based on the OSM hardware setup introduced above, the CPU power of OSM is far than enough to support the functions of OSM. The maximum throughput of OSM is limited not by its CPU, but by disks.

4.2.1.2 Iometer Write Test

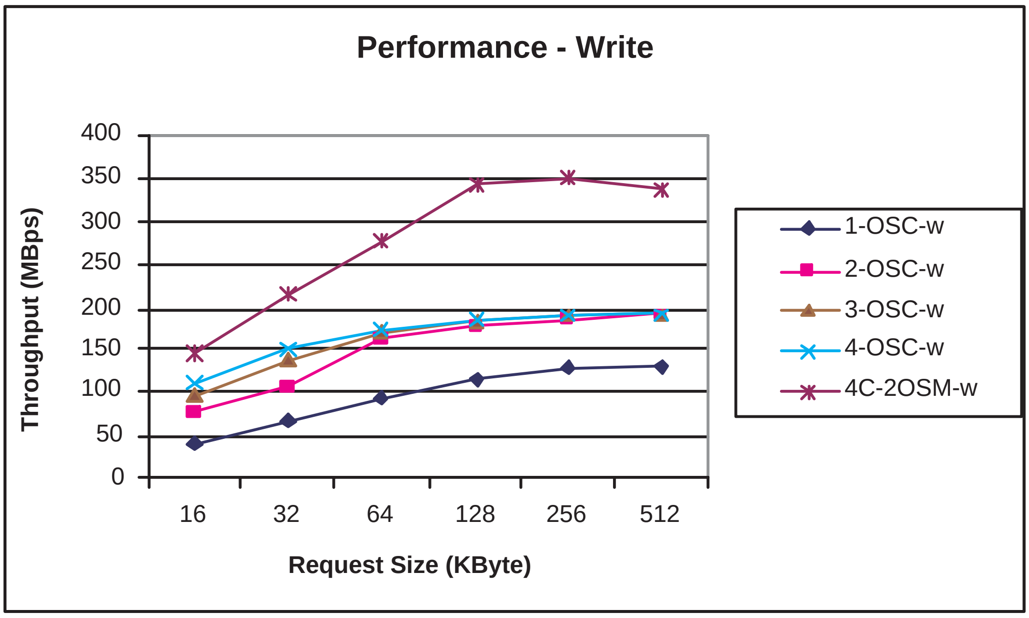


Figure 4.8: Performance in Iometer Write Test

Figure 4.8 shows the write performance during the Iometer write tests. The x axis stands for the size of write requests ($KByte$) and the y axis shows the write performance ($MBps$). The result shows that when the number of OSCs is increased from 1 to 2, the write performance increases sharply. The best write performance of BrainStor is increased from 129.4MBps to 191MBps at 512KB request size when the number of OSCs increases from 1 to 2.

However when the third and the fourth OSCs are added, the performance with request size larger than 64KB cannot achieve obvious improvement (less than 4%). And the improvement is less obvious when adding the fourth OSC than adding the third one, which means that the OSM has already reached its maximum write throughput.

Therefore, similar to read test, the dynamical addition of another OSM is a possible solution to remove the bottleneck. In Figure 4.8, the write performance increases sharply again when the second OSM is added. The best write performance can even reach 350MBps when 4 OSCs send write requests to BrainStor with 2 OSMs at the size of 256KB. Thus, the OSM scalability can significantly improve the capacity as well as the write performance.

Compared with Figure 4.4, the write performance is slight lower. This is due to the additional *buffer_ready* notification in write type transition through Fibre Channel [17]. During the write process, instead of transmitting data and write command together, the OSC can only send write command to the OSM. After the OSM have corresponding buffer ready for DMA transition of the write command, the OSM needs to send back a *buffer_ready* notification to the OSC. After the OSC receives the notification, data can be written to the OSM through FC network. Because every write command has one more transition than read command, the write performance is slightly lower than read performance.

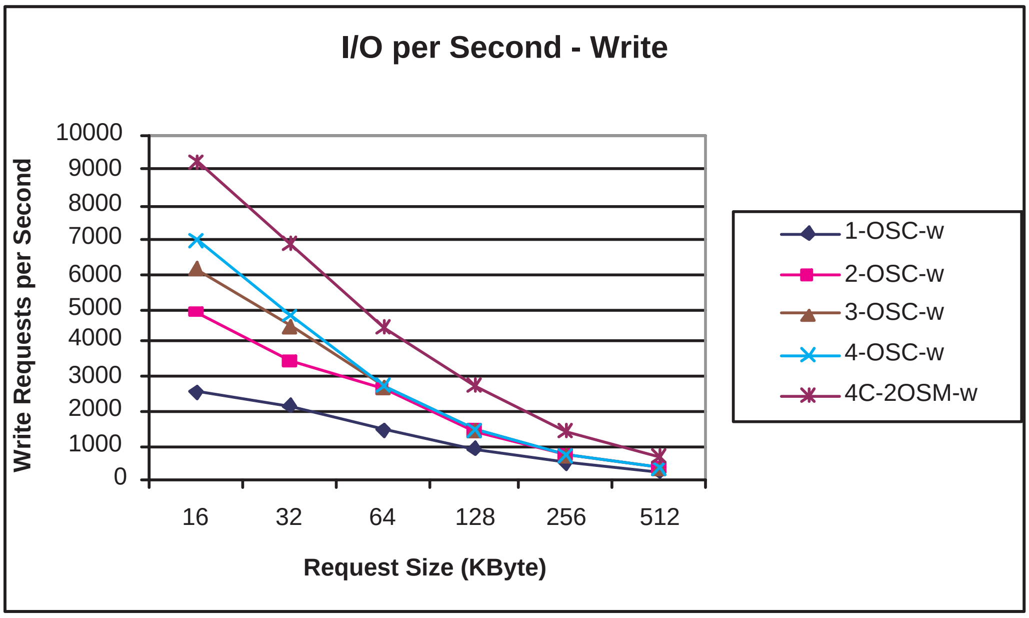


Figure 4.9: IOps in Iometer Write Test

Figure 4.9 shows the I/O per second (IOps) during the write test. The x axis represents the size of write requests and the y axis indicates the IOps.

Normally, the larger the request size, the lower the IOps, because more time are needed to process a large I/O request than a small I/O request. As discussed in Section 4.2.1.1, although the larger write request size leads to less IOps, it brings better write performance. The changes under different setups are accordant with the changes in Figure 4.8. Due to the limitation of one OSM, the IOps with 3 OSCs and 4 OSCs are just slightly different when the request size is larger than 64KB. After another OSM is added, there is obvious increase in terms of IOps. Thus, storage scalability improves the write IOps as well.

Compared with Figure 4.5, the IOps of write test is lower due to the additional *buffer_ready* notification.

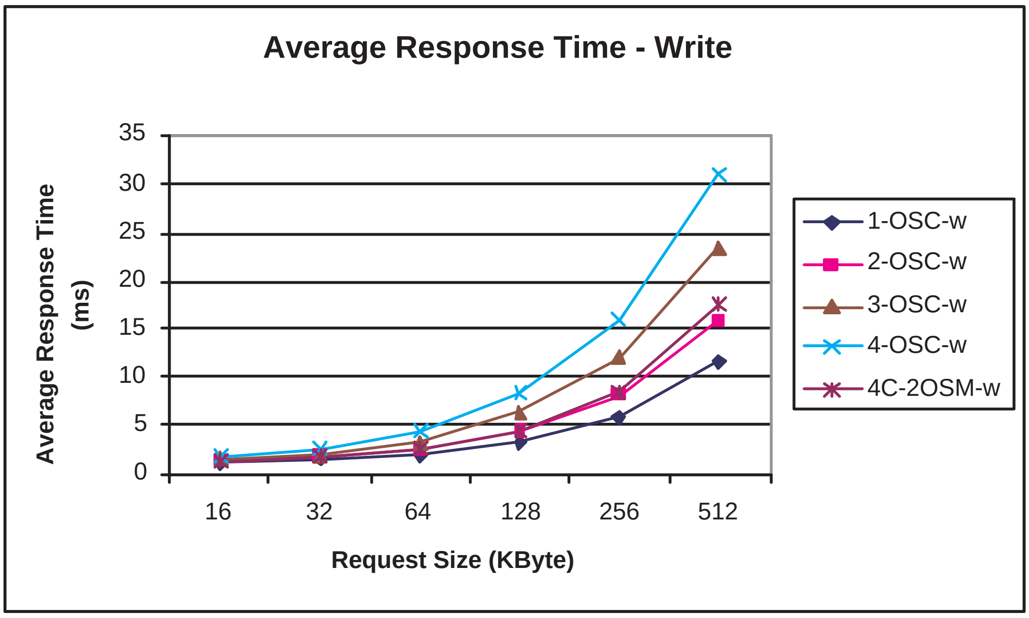


Figure 4.10: Average Response Time in Iometer Write Test

Figure 4.10 shows the average response time during the write test. The x axis represents the size of write requests and the y axis indicates average response time (ms).

Normally, the larger the size of write requests, the higher the response time. Figure 4.10 also shows that the average response time keeps increasing with the number of OSCs. This is also due to the limitation of OSM processing capability and more write requests have to wait in the command queues, as discussed in Section 4.2.1.1.

As can be seen in Figure 4.10, after another OSM is added, the average response time of write test drops sharply and reaches the same level as 2-OSC-w. Thus, storage scalability can help to reduce the average response time of writer as well. Compared with Figure 4.6, the average response time of writer is slightly longer than that of reader, which is also due to the additional *buffer_ready* notification.

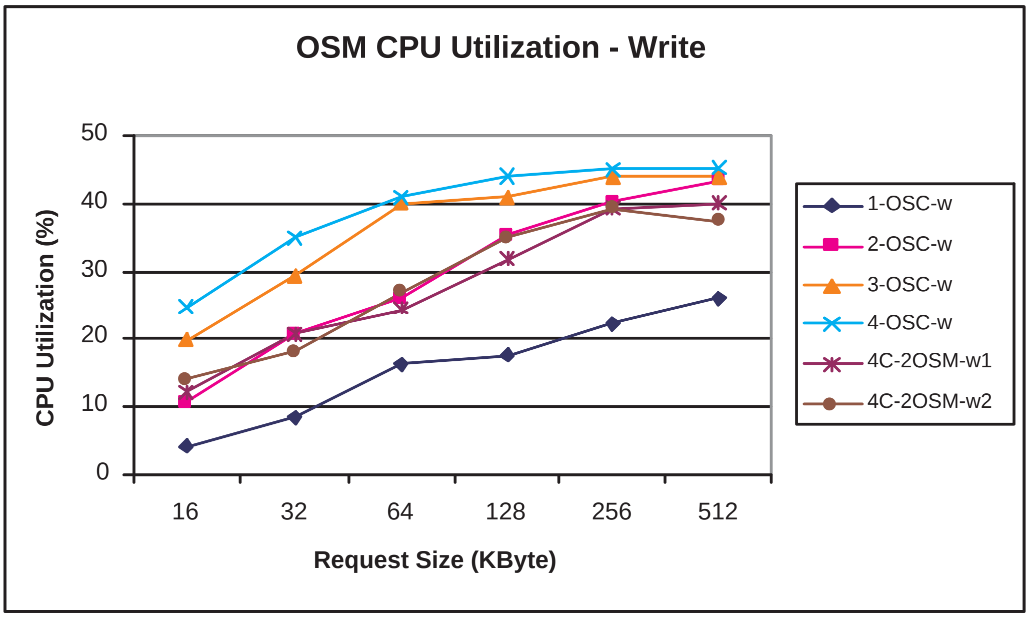


Figure 4.11: OSM CPU Utilization in Iometer Write Test

Figure 4.11 shows the OSM CPU utilization during the tests. The x axis represents the size of write requests and the y axis indicates the CPU utilization (%).

As can be seen in Figure 4.11, the OSM CPU utilization normally increases as the throughput increases under certain test setup, e.g. 1-OSC-w. The results from 4C-2OSM-w1 and 4C-2OSM-w2 are very close. Two OSMs can almost share the write requests evenly, and BrainStor can load balance between the OSM cluster.

One interesting observation is that although the performance of 3-OSC-w and 4-OSC-w is similar, their respective CPU utilizations show obvious difference. It is because extra CPU cycles are taken to process the incoming write requests and maintain the command queues, as discussed in Section 4.2.1.1.

Figure 4.11 shows that the highest OSM CPU utilization is 46.5% under the write test, which is higher than that of read testing (40.5%). The CPU power of OSM is also more than enough to support the functions of OSM. The maximum throughput of OSM is limited not by its CPU, but by disks.

4.2.2 IOzone Test

The purpose of IOzone test is to benchmark BrainStor prototype to file operations. IOzone is a standard file system benchmark tool. The benchmark generates and measures a variety of file operations [48].

In order to evaluate the BrainStor performance, IOzone setting "-*I*" is used to bypass the OSC's side cache effect. IOzone write test measures the performance of writing a new file, which includes writing both data and metadata of the file. IOzone read test measures the performance of reading an existing file. IOzone can conduct the read/write test based on different request size. The request size is less or equal to the size of file. The request record sizes vary from 4KByte to 512KByte in the IOzone test. The internal procedure of IOzone tests is to create the corresponding test files at specified size in the tested file system and conduct read or write test with one particular request size. Thus the performance results are measured according to the size of test file and the size of request size.

Figure 4.12 and 4.13 show IOzone test results of the BrainStor prototype with one OSC, one OMM and one OSM. The x axis represents the size of test file, the y axis indicates the size of request record, and the z axis represents the performance in Kilo-Byte per second (KBps).

As can be seen from Figure 4.12 and 4.13, the performance of both reader and writer is increasing sharply with the size of request size. This scenario can also be found in Iometer results. The best performance of reader and writer can be above 100MBps for single OSC access. The major read performance is around 80 - 100MBps and the major write performance is around 60 - 80MBps. IOzone results are lower than Iometer results because IOzone tests involve additional file operations, such as open, and metadata operations, such as updating the access time of file.

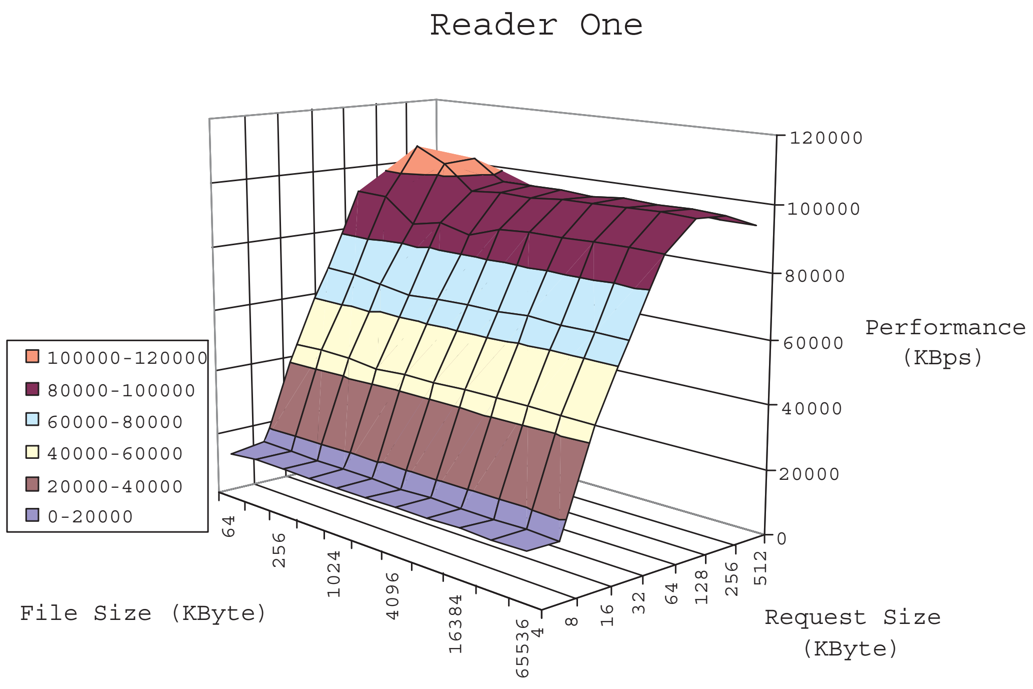


Figure 4.12: Performance in IOzone Read Test

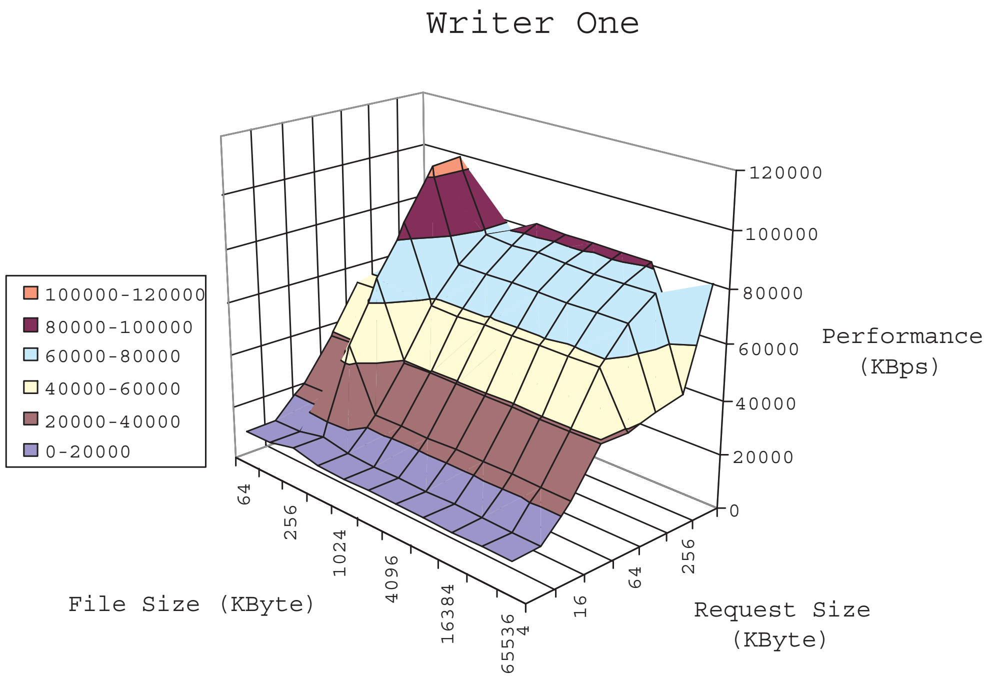


Figure 4.13: Performance in IOzone Write Test

Reader performance is better than writer performance. One reason for this difference between reader and writer is due to the *buffer_ready* notification of write commands, as discussed in Section 4.2.1.2. Another reason is that IOzone writer needs to store both data and file metadata, while IOzone reader only needs to read data.

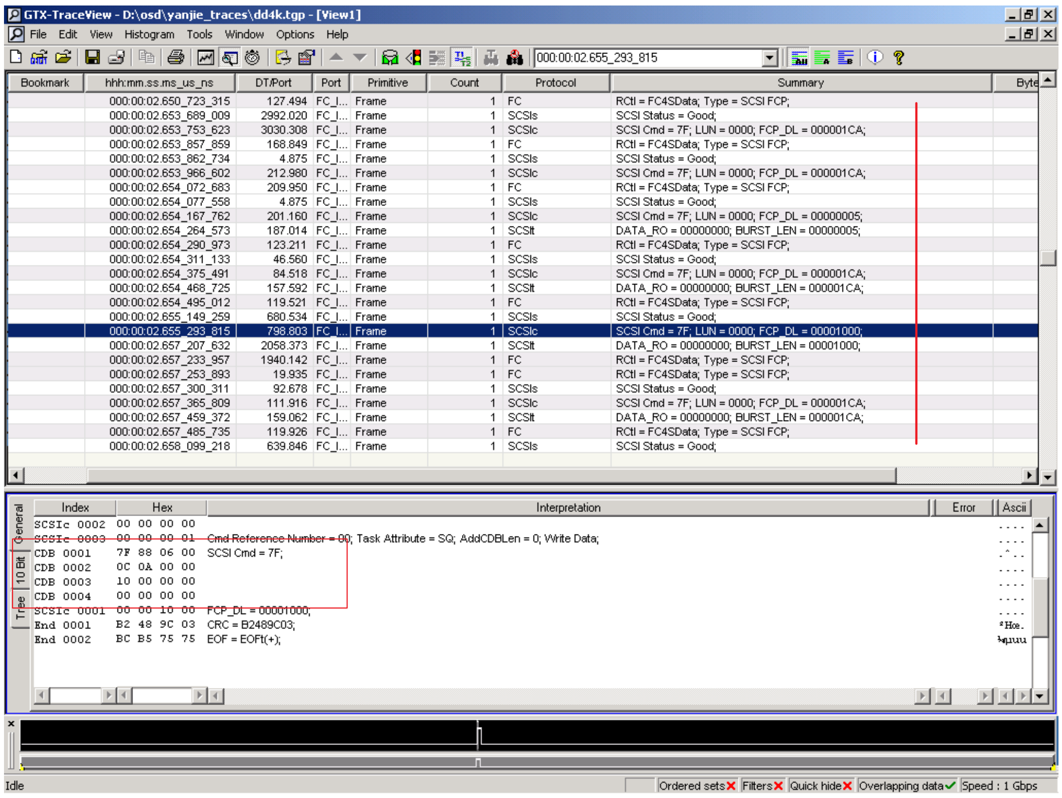
4.2.3 PostMark Test

The purpose of PostMark test is to learn the metadata request percent of the total requests when the OSC accesses thousands small files in BrainStor. PostMark is a benchmark to measure performance for the ephemeral small files used by Internet softwares, in particular: electronic mail, net news and web-based commerce.

PostMark is designed to measure the transaction rates for a workload similar to a large Internet electronic mail server [49]. During the PostMark testing, it firstly generates an initial pool of random text files ranging in size from a configurable low bound to a configurable high bound. This file pool is of configurable size and can be located on any accessible file system. In BrainStor PostMark tests, the pool is located in the OSC's *"/mnt/brainstor/"* directory, which is the mount point of *ofm* file system.

Then, a specified number of transactions occur according to the configuration. Each transaction consists of a pair of smaller transactions: create file, delete file, read file or append file. Each transaction type and its affected files are chosen randomly to minimize the influence of file system caching, file read ahead, and disk level caching and track buffering [49].

PostMark tests are still based on the basic BrainStor prototype, which includes one OSC, one OMM and one OSM. All the nodes are connected by 2G Fibre Channel switch. In order to capture all the commands from the OSC, the Finisar Fibre Channel analyzer is used between the OSC and FC switch this time.



GTX-TraceView - D:\osd\yanjie_traces\dd4k.tgp - [View1]

```
Bookmark  hhh:mm.ss.ms_us_ns     DT/Port   Port    Primitive  Count  Protocol  Summary
          000:00:02.650_723_315   127.494  FC_I... Frame      1      FC        RCtl = FC4SData; Type = SCSI FCP;
          000:00:02.653_689_009  2992.020  FC_I... Frame      1      SCSIs     SCSI Status = Good;
          000:00:02.653_753_623  3030.308  FC_I... Frame      1      SCSIc     SCSI Cmd = 7F; LUN = 0000; FCP_DL = 000001CA;
          000:00:02.653_857_859   168.849  FC_I... Frame      1      FC        RCtl = FC4SData; Type = SCSI FCP;
          000:00:02.653_862_734     4.875  FC_I... Frame      1      SCSIs     SCSI Status = Good;
          000:00:02.653_966_602   212.980  FC_I... Frame      1      SCSIc     SCSI Cmd = 7F; LUN = 0000; FCP_DL = 000001CA;
          000:00:02.654_072_683   209.950  FC_I... Frame      1      FC        RCtl = FC4SData; Type = SCSI FCP;
          000:00:02.654_077_558     4.875  FC_I... Frame      1      SCSIs     SCSI Status = Good;
          000:00:02.654_167_762   201.160  FC_I... Frame      1      SCSIc     SCSI Cmd = 7F; LUN = 0000; FCP_DL = 00000005;
          000:00:02.654_264_573   187.014  FC_I... Frame      1      SCSIt     DATA_RO = 00000000; BURST_LEN = 00000005;
          000:00:02.654_290_973   123.211  FC_I... Frame      1      FC        RCtl = FC4SData; Type = SCSI FCP;
          000:00:02.654_311_133    46.560  FC_I... Frame      1      SCSIs     SCSI Status = Good;
          000:00:02.654_375_491    84.518  FC_I... Frame      1      SCSIc     SCSI Cmd = 7F; LUN = 0000; FCP_DL = 000001CA;
          000:00:02.654_468_725   157.592  FC_I... Frame      1      SCSIt     DATA_RO = 00000000; BURST_LEN = 000001CA;
          000:00:02.654_495_012   119.521  FC_I... Frame      1      FC        RCtl = FC4SData; Type = SCSI FCP;
          000:00:02.655_149_259   680.534  FC_I... Frame      1      SCSIs     SCSI Status = Good;
          000:00:02.655_293_815   798.803  FC_I... Frame      1      SCSIc     SCSI Cmd = 7F; LUN = 0000; FCP_DL = 00001000;
          000:00:02.657_207_632  2058.373  FC_I... Frame      1      SCSIt     DATA_RO = 00000000; BURST_LEN = 00001000;
          000:00:02.657_233_957  1940.142  FC_I... Frame      1      FC        RCtl = FC4SData; Type = SCSI FCP;
          000:00:02.657_253_893    19.935  FC_I... Frame      1      FC        RCtl = FC4SData; Type = SCSI FCP;
          000:00:02.657_300_311    92.678  FC_I... Frame      1      SCSIs     SCSI Status = Good;
          000:00:02.657_365_809   111.916  FC_I... Frame      1      SCSIc     SCSI Cmd = 7F; LUN = 0000; FCP_DL = 000001CA;
          000:00:02.657_459_372   159.062  FC_I... Frame      1      SCSIt     DATA_RO = 00000000; BURST_LEN = 000001CA;
          000:00:02.657_485_735   119.926  FC_I... Frame      1      FC        RCtl = FC4SData; Type = SCSI FCP;
          000:00:02.658_099_218   639.846  FC_I... Frame      1      SCSIs     SCSI Status = Good;

Index        Hex          Interpretation
SCSIc 0002   00 00 00 00
SCSIc 0003   00 00 00 01  Cmd Reference Number = 00; Task Attribute = SQ; AddCDBLen = 0; Write Data;
CDB 0001     7F 88 06 00  SCSI Cmd = 7F;
CDB 0002     0C 0A 00 00
CDB 0003     10 00 00 00
CDB 0004     00 00 00 00
SCSIc 0001   00 00 10 00  FCP_DL = 00001000;
End 0001     B2 48 9C 03  CRC = B2489C03;
End 0002     BC B5 75 75  EOF = EOFt(+);

Idle    Ordered sets✗  Filters✗  Quick hide✗  Overlapping data✓  Speed : 1 Gbps
```

Figure 4.14: Data Captured by Fibre Channel Analyser

Table 4.3: PostMark Configuration in Experiments

Configuration Name	Setting
Files per directory	1000
Number of subdirectories	10
Transitions	500
Read/Write	50% Read and 50% Write
Access Pattern	Random
File Size	512Byte - 512K

The PostMark test setting is shown in Table 4.3. The setting means that PostMark creates 10 subdirectories, each of which contains 1000 files with preset size. After the pool is set up, PostMark will conduct 500 transactions that may be read or write transactions on files that are randomly picked up in the pool.

During the test, FC analyzer can capture all the OSD commands, as shown in Figure 4.14. The main window shows the concise description of all the captured commands, data and status, while the smaller window below shows the detail description of the command that is highlighted in the above main window. The commands indicated with the vertical line are all OSD SCSI commands, which are indicated by the OSD operation code (0x7F). The service code from the detail description of the command tells which OSD operation each SCSI command stands for. In the above example, the details of Command Descriptor Block (CDB) can be found in the highlighted SCSI command in the rectangle. The service code of the CDB is 0x8806, which represents an OSD WRITE command. From the CDB, it can also be seen that this command intends to write 4KB (0x1000) data to object C in object partition A (0x0C0A) starting from offset 0.

Figure 4.15 shows the data request percent (DataReq%) and the metadata request percent (MetadataReq%) of the total requests during the PostMark tests at different file size. The results are obtained by analyzing and counting all the data commands and metadata commands in the captured data by FC analyzer during Postmark tests.

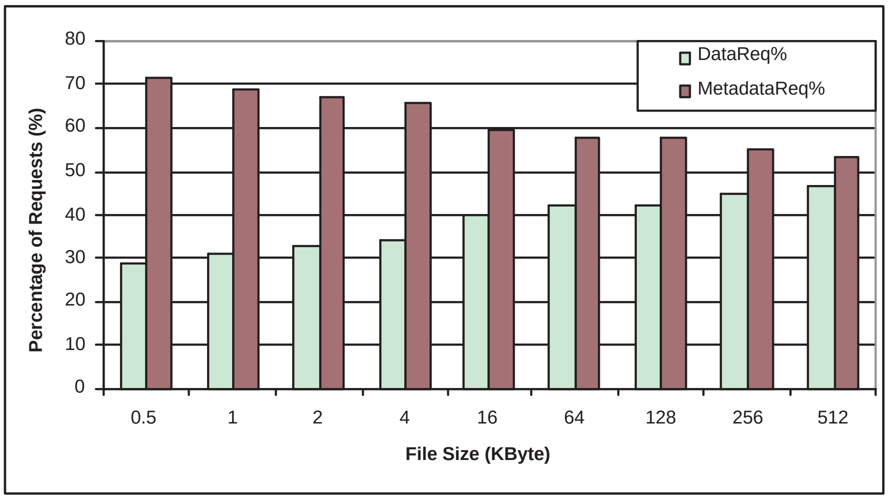


Figure 4.15: PostMark Test Results

As can be seen in Figure 4.15, there are too many metadata requests comparing to the data request in the BrainStor system. There is even more than 70 percent of all I/O requests are for metadata when using PostMark to randomly access ten thousands 0.5KB files, as shown in Figure 4.15. Numerous metadata requests queuing in the OMM can damage the overall system performance.

How BrainStor can better manage the metadata and reduce the number of metadata requests is an critical issue in OSD system design. As can be seen in

Figure 4.15, the first BrainStor prototype does not address the problem well. In order to solve this problem, we propose the Hashing Partition method, which will be discussed in the Chapter 5.

Because different tools has different focuses and vary test methods, PostMark can unveil the problem of BrainStor while other benchmark tools can achieve comparable good performance. PostMark concentrates on performance benchmark of the ephemeral small files, while Iometer cares the performance of raw devices and IOzone testes performance on the files. Iometer creates one very large test file, named *iobw.tst*, to simulate the entire raw disk and IOzone conducts tests by reading and writing within files. Both Iometer and IOzone do not involve many metadata accesses during their tests. On the other hand, PostMark creates a very large test pool with thousands of small files under different subdirectories, in order to simulate the storage of a large email server. It also randomly picks up small files to conduct test. Therefore there are a lot of metadata operations involved.

4.3 Summary

The current BrainStor prototype has the following features:

- An OSD prototype over Fibre Channel network
- A centralized Object Cache Module
- An Object Bridge Module
- Preliminary results: 145MB/s for single OSC and 190MB/s for single OSM over 2GFC
- OSM cluster storage virtualization
- OMM cluster dynamic load balancing, scalability and failover support
- Integrate OSD storage with email server (Sendmail) and others

Because the object is a concept between block and file, Iometer and IOzone are used to benchmark the BrainStor prototype from block and file perspectives respectively. Iometer results show that adding OSCs can improve the overall performance as long as the OSM cluster can support, and the OSM cluster scalability improves not only the storage capacity, but also the overall BrainStor performance, in terms of throughput, IOps, response time and OSM CPU utilization. Storage virtualization of BrainStor can eliminate the system downtime. IOzone results show that file-level performance is lower than block-level performance due to the additional file and metadata operations.

PostMark test unveils the metadata management challenges in the new OSD architecture. There are too many metadata requests to the OMM cluster, which damages the overall performance of BrainStor. Chapter 5 will address this problem in details.

Chapter 5

Hashing Partition (HAP)

In BrainStor, the performance, availability and scalability of the Object Manager Module (OMM) cluster are critical. Traditional metadata server cluster suffers from frequent metadata access and metadata movement within the cluster. In this thesis, a new method called Hashing Partition (HAP) is proposed for OMM cluster design [50]. Based on HAP, BrainStor can achieve good performance of OMM cluster load balancing, failover and scalability.

5.1 Problem

As discussed in Section 4.2.3, metadata management is a critical issue in BrainStor design. Figure 4.15 shows that too many metadata accesses make the OMM one potential bottleneck of BrainStor. In some cases, more than 70 percent of all I/O requests are for metadata during PostMark tests. A trace study of the Unix BSD file system also found that 50% to 80% of all file system accesses are to metadata [51]. Although the size of the metadata is small, the traffic volume of such metadata access degrades the OMM cluster performance and therefore damages the overall storage system performance.

The intensive metadata requests are attributed to the use of traditional directory metadata management in the preliminary BrainStor prototype. Although this method is widely used, the directory hierarchy must be traversed to get metadata information of each file. For example, in order to access the metadata of file: "/a/b/c/d", file system firstly needs to access the metadata and then the data of root directory "/", in order to know the metadata index of directory "a". Similarly, file system needs to access metadata and data of "a", "b" and "c". After file system knows metadata and data of all the nodes on the path of file "d" (including 4 metadata accesses and 4 data accesses), file system finally knows the metadata location of file "d" and access it. This problem of directory metadata management is often mitigated somewhat by OSC-side cache. However, cache does not help when large number of OSCs simultaneously access the same directory, which often happens in a clustering environment.

Besides the number of metadata requests, an extremely unbalanced load distribution among cluster lets several OMMs overload and most of others free. Therefore, although the cluster can support more load, the entire OMM Cluster becomes a bottleneck in BrainStor. For example, if most of "hot" metadata is located in the same OMM, this one will be "overheated". And moving all these data from its local disk to other OMMs introduces additional overheads.

Two different approaches are used to handle this metadata management problem. First of all, the direct response to this problem is to reduce the number of metadata requests, for example the hashing method. The second approach is to make the OMM cluster more capable to handle the increasing metadata requests. The OMM cluster should be able to perform load balancing during the runtime in order to avoid the uneven load distribution. In addition, in order to handle the growing metadata storage and provide reliable metadata storage, the scalability and failover capability of OMM cluster are critical to BrainStor system. However based on traditional cluster architecture, the performance of load balancing, failover and scalability is limited, because most of these operations lead to the inevitable massive metadata movement within cluster.

Some studies address the metadata management problem by using hashing method. The primitive forms of adopting hashing method in file system metadata management can be found in the Vesta parallel file system [53], which assigns metadata to OMMs based on a hash of the file identifier, file name, or other related values. The Lazy Hybrid metadata management method [54] presented a hashing metadata management with the hierarchical directory support, which dramatically reduced the total number of metadata requests, however Lazy Hybrid did not deal with reducing metadata movement between OMMs for load balancing, failover and scalability.

5.2 Solution - Hashing Partition (HAP)

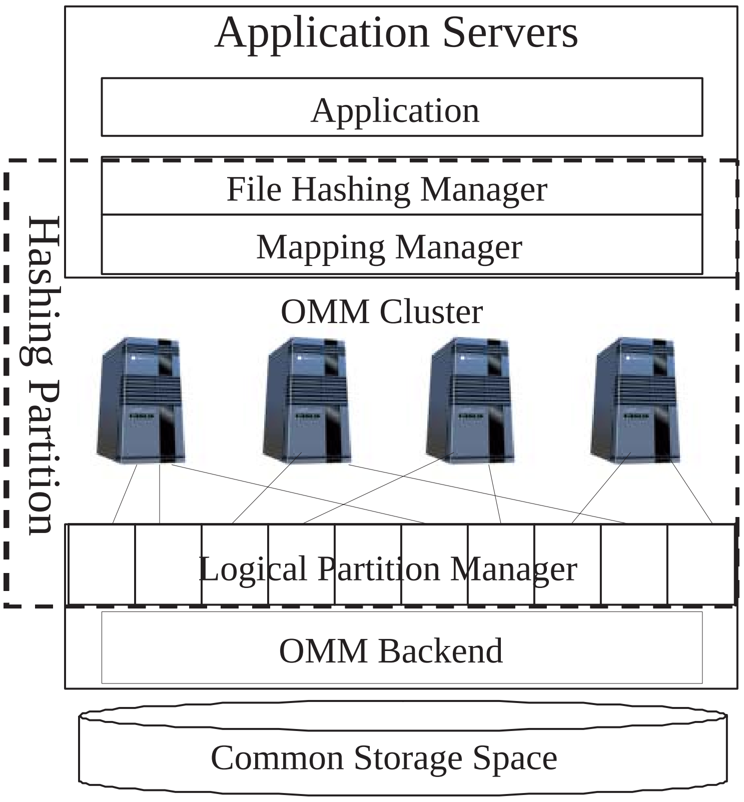


Figure 5.1: Hashing Partition (HAP)

Hashing Partition (HAP) is a new metadata management method, which provides a total solution for the file hashing, metadata partitioning, and metadata storage. HAP adopts the hashing method to reduce the number of metadata access, and focuses on reducing the cross-OMM metadata movement in a clustered design.

HAP also uses a common storage space in order to achieve high performance of load balancing, failover and scalability. There are three logical modules in the HAP: file hashing manager, mapping manager, and logical partition manager, as shown in Figure 5.1.

In addition, HAP employs an independent common storage space for all OMMs to store metadata, and this space is further divided into multiple logical partitions, as shown in Figure 5.1. Each logical partition contains part of global metadata table. Each OMM mounts and then exclusively accesses logical partitions allocated to it. Thus as a whole, the OMM cluster can access a unique global metadata table.

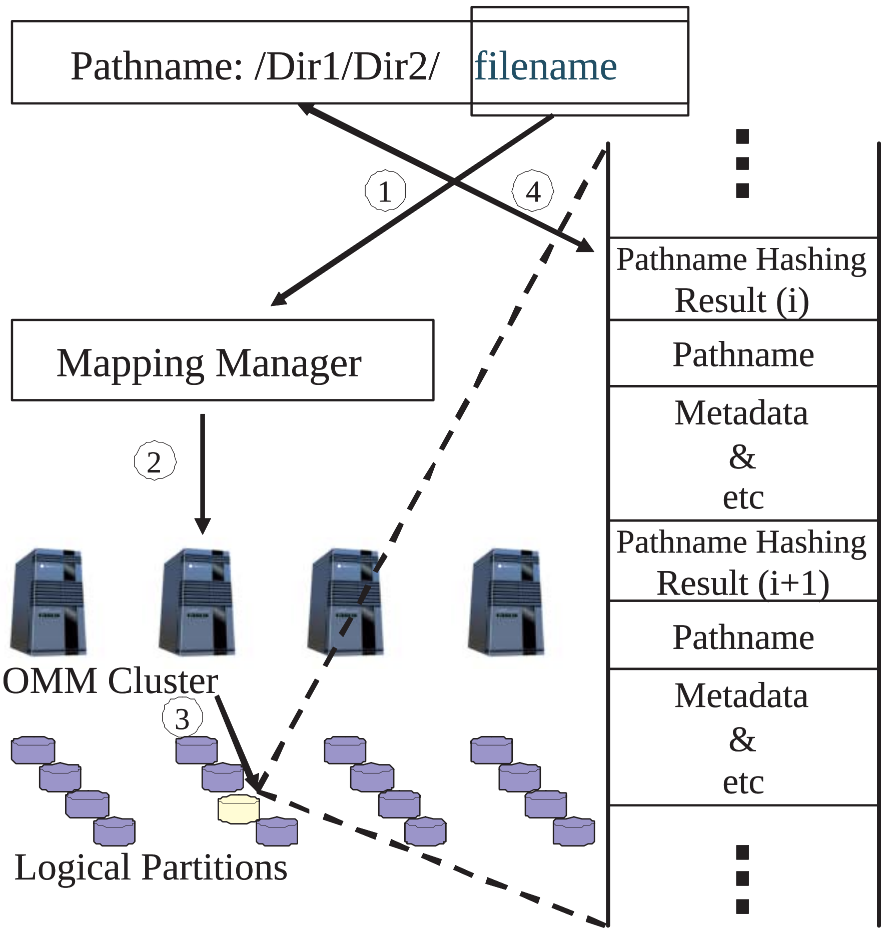


Figure 5.2: Metadata Access Pattern
1.Filename hashing, 2.Selecting OMM through Mapping Manager, 3.Accessing metadata by pathname hashing result, 4.Returning metadata to OSC.

The procedure of metadata access is described as follows. Firstly, file hashing manager hashes a filename to an integer, which can be mapped to the partition that stores the metadata of the file in the common storage space. Secondly, mapping manager figures out the id of OMM that currently mounts that partition. Then client sends a metadata request with the hashing value of pathname to the OMM. Finally, logical partition manager located in the OMM side accesses metadata on the logical partition in the common storage space. Figure 5.2 describes this efficient metadata access procedure. Normally, only a single message to a single OMM is required to access a file metadata.

5.2.1 File Hashing Manager

File Hashing Manager (FHM) performs all the hashing. It is part of the Object File-system Module in OSC architecture as shown in Figure 3.4. In the preliminary BrainStor prototype, the method of managing metadata is similar to that of traditional file systems, using directory metadata management. In this way, the metadata is managed by using a hierarchical directory structure. Whenever client wants to access metadata of a file, it needs to travel all the nodes on the file path, including access of metadata as well as content of directories in the path. Thus instead of using directory metadata management, HAP adopts hashing method that needs only one direct metadata access based on the hashing of the pathname.

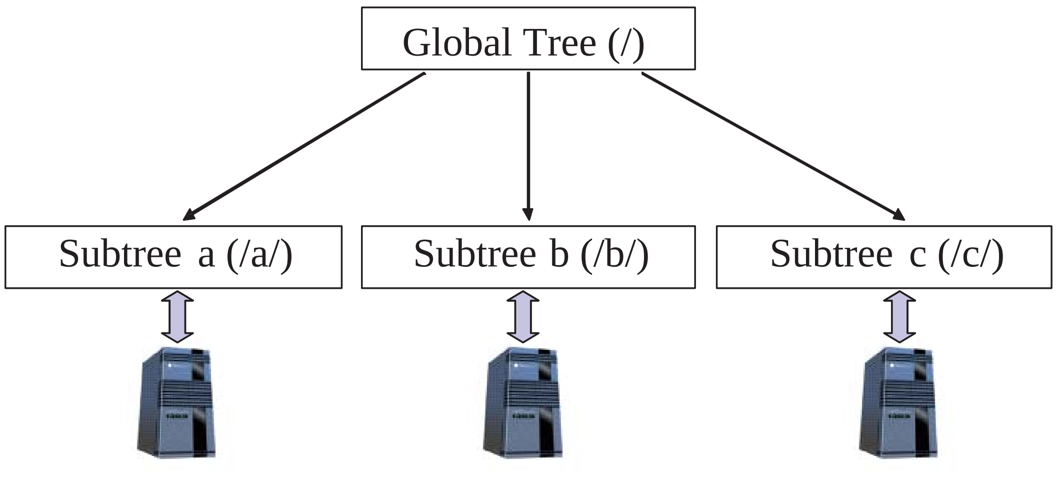


Figure 5.3: Directory Subtree Partitioning

In addition, the new design also adopts hashing partitioning that assigns metadata among OMMs based on a hash result, instead of directory subtree partitioning [52]. NFS [56], AFS [57], and Coda [58], LOCUS [59], and Sprite [60] adopt the directory subtree partitioning that partitions the namespace among servers according to directory subtrees. Given a simple example, there is a global directory tree that includes three subtrees. According to directory subtree partitioning, each OMM may just handle each subtree independently as shown in Figure 5.3. Comparing to the directory subtree partitioning, hashing partitioning avoids the severe bottleneck problems when a single file, directory, or directory subtree becomes popular. Based on a good hashing algorithm, hashing partitioning offers a more balanced distribution of metadata among OMMs.

There are two hashing partitioning methods:

I Pathname-hashing partitioning

II Filename-hashing partitioning

Pathname-hashing partitioning adopted in Lazy Hybrid [54] uses the full pathname (e.g. /a/b/filec) to hash, while HAP uses filename (e.g. filec) as the seed of hashing. Pathname-hashing introduces many metadata movements among OMMs when a rename operation on a directory happens. Based on method I, this operation will change the hashing results of most of the files in this directory subtree due to the changed pathname, then many metadata must be moved from one OMM to another one indicated by the new hashing results. This is terrible when renaming a subtree of more than 10,000 files. On the other hand, if the hashing only uses the filename, all the updates are completed within each OMM and there is no additional communication between OMMs.

However the filename hashing may introduce a potential bottleneck when a large parallel access to different files with the same name in different directories. Files with the same name have the same filename hashing result, therefore their metadata is mapped to the same OMM. Although it is possible that many parallel requests refer to some “hot” files with the same common names, such as readme and makefile, different “hot” filenames might not be hashed to the same result. Fortunately, the different hashing values of various popular filenames make all these “hot spots” distributed among the OMM cluster and reduce the possibility of the potential bottleneck. In addition, even if certain OMM is over-loaded, the dynamic load balancing policy (Section 5.3.1) can effectively handle this scenario and shift the “hot spots” from the overloaded OMM to the lightly loaded OMMs.

Therefore, in BrainStor, file hashing manager adopts method II using the filename hashing result to choose the OMM. File hashing manager performs two kinds of hashing: filename hashing for partitioning metadata in the OMM cluster, and pathname hashing for the metadata allocation and location in an OMM. To access metadata of a file in the OMM cluster, a client needs to know two facts: which OMM manages the metadata and where the metadata is located in the logical partition. Filename hashing answers the first question and pathname hashing solves the second one. For example, if the client needs to access the file, “/a/b/filec”, it uses the hashing result of “filec” to select the OMM that manages the metadata. Then instead of accessing directory “a” and “b” to know where is the metadata of “filec”, a hash result of “/a/b/filec”, directly indicates where to retrieve the metadata in the OMM.

Compared with directory metadata management, hashing method makes some operations expensive. For example, the directory rename operation affects all the hashing results of files or subdirectories within the renamed directory. As a result all the corresponding metadata records will be updated based on hashing method. On the other hand, the directory rename operation only needs to update the directory’s own data based on directory metadata management. Therefore, if the applications have a lot of such operations, hashing is not a good choice. Fortunately, a two years study of the Coda traces [61] for one machine in a general-purpose environment shows only 117 directory renames, 1851 directory symbolic links and less than 3000 directory permission and ownership changes. In addition, access control becomes a difficult issue in hashing method due to its different look-up method from traditional directory metadata management. Dr. Brandt has introduced a dual-entry access control list to address this problem [54].

5.2.2 Logical Partition Manager

Logical partition manager manages all logical partitions in the common storage space. It performs many logical partition management tasks, e.g. mount/umount, backup and journal recovery. For instance, logical partition manager can periodically backup logical partitions to a remote backup server. Logical partition manager is part of the Intelligent Server module in OMM middle layer as shown in Figure 3.12.

The location of metadata database is another important issue in the BrainStor design. In normal OMM cluster design, every OMM stores metadata on its local hard disk, and there are two metadata storing methods: one is that every OMM holds part of the global metadata table. The other is that every OMM holds a synchronized copy of global table. Method 1 faces difficulties for load balancing and failover design, such as additional metadata database movement or even metadata loss during the addition and removal of OMMs. Method 2 faces severe overhead to synchronize the global table for every metadata updating.

In addition, compared to the user data, metadata uses little storage, thus even in very large storage system, a central storage space for metadata is acceptable. Thus, in the new OMM Cluster design, HAP adopts a common storage space for metadata, which is further subdivided into Logical Partitions (LP). Each LP holds part of the global metadata table and is managed independently by the OMM backend software. During the runtime, OMMs mount all logical partitions and access metadata on them. Therefore, there is one copy of global metadata table that is accessible to all OMM. Dynamic load balancing and the addition and removal of OMM are easily achieved by switching the control of partitions without any metadata movement.

Because all OMMs access the common storage space at block level, the OMM cluster uses the iSCSI or Fibre Channel to build up a small Storage Area Network (SAN) for metadata storage. For example, in BrainStor system there is another independent and private zone on a CISCO DS-C 9509 director just for the OMM cluster, and a RaidTec JBOD is connected, as the common storage space for metadata. Based on this SAN structure, it is very easy to add more storage space to support the scalability of the common storage space.

The logical partitions can be managed by a local file system, by a cluster file system or by a database. A cluster file system,such as Global File System (GFS) [55] provides OMMs the ability to access all partitions synchronously at block level. However the synchronization overhead and cost of cluster file system is unacceptable in BrainStor. Database also offers each OMM the ability to simultaneously access a global metadata database. Due to the characters of metadata and its access pattern, high cache performance is needed at the OMM side. However, databases are good at atomic operations but bad at cache performance.

Thus HAP adopts a self-developed local file-system type database with very good cache performance. The specially designed database is based on a normal file system. Every OMM uses the database to manage its own logical partitions without considering synchronization with other OMMs. However, a strict requirement of this design is that one logical partition can be mounted and accessed by ONLY ONE OMM at a time, which is the basic principle of the following design.

This common storage space becomes a central node of the whole structure. The stability and availability affect the health of entire system. Redundancy technologies such as RAID, can reduce the damage of hard disk failure. Moreover, remote backup and replication technologies are also necessary to reduce the damage of the entire site failure. In short, this common storage space must be extremely reliable and recoverable even during some failures.

5.2.3 Mapping Manager

Mapping manager performs two kinds of mapping tasks: hashing result to logical partition mapping and logical partition to OMM mapping. Equation 5.1 describes these two mapping functions. Mapping manager is also part of the Object Filesystem Module in OSC architecture as shown in Figure 3.4.

$$\begin{aligned} Pi &= f(H(filename)) \\ OMMi &= ML(Pi, PWi, MWi) \qquad (5.1) \\ & Pi \in \{0, Pn\}; H(filename)) \in \{0, Hn\}; OMMi \in \{0, Mn\} \\ & (Hn \geq Pn \geq Mn > 0) \end{aligned}$$

Where, H represents a filename hashing function; f stands for the mapping function that transfers hashing result to partition number (*Pi*); *ML* represents the function that figures out OMM number (*OMMi*) from partition number and related parameters (PW and MW will be explained in Section 5.3.1); *Pn* is the total number of partitions; *Hn* is the maximum hashing value; *Mn* is the total number of OMMs.

Table 5.1: Example of MLT

Logical Partition Number	OMM ID	OMM Weight
$0 \sim 15$	0	300
$16 \sim 31$	1	300
$32 \sim 47$	2	300
$48 \sim 63$	3	300

When PW and MW are set, mapping manager simplifies the mapping function *ML* to a mapping table MLT, which describes the current mapping between OMMs and logical partitions. It is noted that one OMM can mount multiple partitions, however one partition can only be mounted to one OMM. To access metadata, mapping manager can indicate the logical partition that stores the metadata of a file based on the hash result of the filename. Then through MLT, mapping manager knows which OMM mounts that partition and manages the metadata of the file. Finally the client contacts the selected OMM to obtain the file metadata, file-to-object mapping and other information. Table 5.1 gives an example of MLT. Based on this table, in order to access metadata on logical partition 18, the client needs to send request to OMM1.

5.3 Load Balancing, Failover and Scalability

5.3.1 OMM Cluster Load Balancing Design

A good hash algorithm can make object metadata distributed evenly among all partitions, however it does not mean that every OMM works effectively. First of all, different OMM might have different hardware and even software capability. Secondly, the access frequencies of metadata are different and even dynamically change during the runtime. For example, some hot news, MP3 or even movies in a web server might be the extreme "hot spots" for only a short period of time, however after that period, their access frequency drops.

A Dynamic Weight algorithm is proposed to dynamically balance the load of OMMs. HAP assigns an OMM Weight (MW) to each OMM according to its CPU power, memory size and bandwidth, and uses a Partition Weight (PW) to reflect the access frequency of each partition. MW is a stable value if the hardware configuration of the OMM cluster does not change, and PW can be dynamically adjusted according to the access rate and pattern of partitions. In order to balance the load between OMMs, mapping manager allocates partitions to OMM based on Equation 5.2.

$$\frac{\sum PWi}{MWi} = \frac{\sum_{a=0}^{Pn} PWa}{\sum_{a=0}^{Mn} MWa} \tag{5.2}$$

Where, $\sum PWi$ represents the sum of PW of all partitions mounted by $OMMi$;

MWi stands for the *MW* of *OMMi*; *Pn* stands for the total number of partitions; *Mn* represents the total number of OMMs.

In addition, each OMM needs to maintain load information about itself and all partitions mounted on it, and periodically uses Equation 5.3 to calculate new values.

$$\begin{aligned} OMMLOAD(i+1) &= OMMLOAD(i) \times \alpha\% + OMMCURLOAD \times (1-\alpha\%) \\ PLOAD(i+1) &= PLOAD(i) \times \beta\% + PCURLOAD \times (1-\beta\%) \end{aligned} \tag{5.3}$$

Where, *OMMCURLOAD* is the current load of the OMM; *PCURLOAD* is the current load of the logical partition; *OMMLOAD(i)* represents the load status of an OMM at time i; *PLOAD(i)* stands for the load status of a logical partition at time i; α and β are constants used to balance the effects of old value and new value.

However, OMMs do not need to report their load information to the master node, e.g. one specified OMM, until an OMM alarms in its overloaded situation, such as the OMMLOAD exceeding the preset maximum load of the OMM. After receiving load information from all OMMs, the master node sets the *PW* of each partition using new PLOAD values. Then according to new *PW* and Equation 5.2, HAP shifts the control of certain partitions from the over-loaded OMMs to some lightly loaded OMMs and modifies MLT accordingly. This adjustment does not involve any physical metadata movement between OMMs.

5.3.2 OMM Cluster Failover Design

Typically, a conventional failover design adopts a standby server to take over all services of the failed server. In BrainStor design, the failover strategy relies on the clustered approach and supports the multi-OMM failures. In the case of an OMM failure, mapping manager assigns other OMMs to take over the work of the failed

OMM. Then the logical partition manager allocates the logical partitions managed by the failed OMM to its successors. Hence OSCs can still access metadata on the same logical partition in the common storage space through the successors.

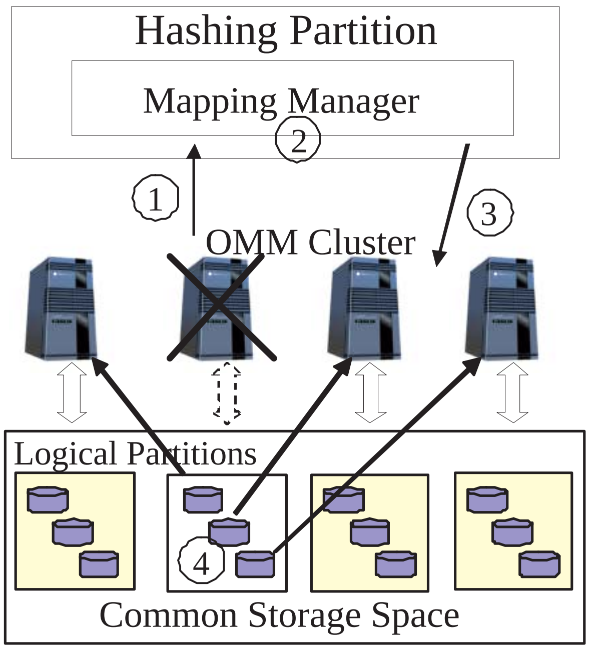


Figure 5.4: OMM Cluster Failover
1.Detecting the OMM failure, 2.Recalculating MW and adjusting MLT, 3.Other OMMs take over logical partitions of the failure one, 4.Journal recovery

Table 5.2: MLT after OMM1 Fails

Logical Partition Number	OMM ID	OMM Weight
$0 \sim 15$, $17 \sim 21$	0	400
X	X	X
$32 \sim 47$, $21 \sim 26$	2	400
$48 \sim 63$, $27 \sim 31$	3	400

First, let's consider a normal OMM removal. In this case, it may begin with a command from the system administrator, like "*rm OMM1*". Then HAP can work for this coming change and all MW will be updated. Then HAP generates a new MLT, based on Equation 5.2. For example, based on MLT shown in Table 1, if OMM1 is being removed by administrator or even suddenly crashes, Table 5.2 may be an example of the new MLT. Then logical partition manager can complete all adjustments based on the new MLT. Logical partition manager can umount logical partitions from the removing OMM and mount them to other OMMs according to the new MLT. During the process which may last for a very short time period, all coming requests are either queued or denied with a server busy message.

Then, let's look at the unpredictable OMM failures. During the disaster, one or more OMMs may fail and disappear from the OMM cluster without any aura, and of course logical partition manager has no time to perform any umount operations. HAP has certain monitor mechanism to detect the OMM failure in cluster. Then after the OMM cluster detects the node failure, the failover procedure is quit like the procedure of normal OMM removal. The difference is that HAP depends on the journal function to recovery the logical partitions. Therefore, the difference is that the mount process will invoke a recovery procedure based on journal information. Figure 5.4 shows this failover procedure.

5.3.3 OMM Cluster Scalability Design

In the OMM cluster, there are two kinds of scalability. The first one is the scalability of storage capacity for each partition in the common storage space. With the growth of metadata database, one day, the initial capacity of partitions may not be sufficient and new storage hardware should be plugged in. Second one is the scalability of OMM cluster. If the current OMM cluster cannot handle metadata requests effectively due to the heavy load, new OMMs will be set up to release the overhead of others. HAP supports both of these scalability requests dynamically and smoothly.

To consider the storage capacity scalability, the SAN structure provides great convenience to add more storage, by hot-plugging in hard disks to a RAID system or even connecting another new storage device to the switch. Besides the hardware support, HAP runs Logical Volume Manager (LVM) [43] at the OMM side to extend the storage of current partitions to the new hardware without downtime of the OMM. HAP is capable to smoothly increase size of logical partitions, therefore, HAP supports the storage capacity scalability.

Table 5.3: MLT after OMM4 is Added

Logical Partition Number	OMM ID	OMM Weight
$0 \sim 12$	0	240
$16 \sim 28$	1	240
$32 \sim 44$	2	240
$48 \sim 60$	3	240
$13 \sim 15$, $29 \sim 31$, $45 \sim 47$, $61 \sim 64$	4	240

HAP also significantly simplifies the procedure to scale the OMM cluster. If the current OMM cluster cannot handle metadata requests effectively due to the heavy load, new OMMs can be dynamically set up to release the overhead of others. To the addition of OMM, HAP adjusts MWs and thus generates a new MLT based on ML. For instance, still based on Table 5.1 in the above example, there is a new OMM, such as OMM4, added. Finally, the new MLT may be something like Table 5.3. This process does not touch the mapping relationship between filename and logical partition, because the number of logical partitions is unchanged. Following the new MLT, logical partition manager umounts certain partitions from existing OMMs and mounts them to the new OMM. This procedure introduces no physical metadata movement within the OMM cluster.

An important rule of adjusting MLT in all situations is to minimize the umount operations, because such operation means the loss of a warmed cache in one OMM, and a cache warm-up process in another OMM may degrade the system performance.

5.4 OMM Cluster Rebuild

Although HAP method dramatically simplifies the operation of OMM addition and removal, HAP actually has a scalability limitation, called Scalability Capability (SC). The preset number of logical partitions limits scalability capability, since one partition can only be mounted and accessed by one OMM at a time. For instance 64 logical partitions can support up to 64 OMMs. In order to improve scalability

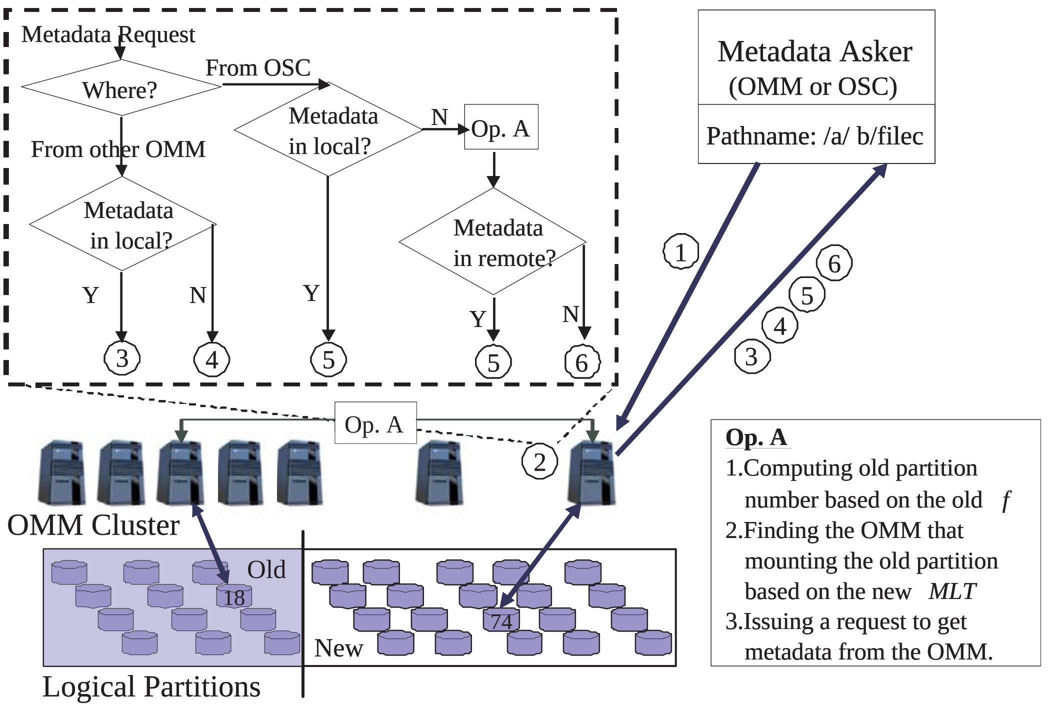


Figure 5.5: OMM Cluster Rebuild
1.Sending request to the OMM based on new mapping result, 2.Searching for metadata and making judgment (the rectangle on the left shows the internal logic and Op. A is explained in the bottom rectangle), 3.Returning metadata and deleting it in local, 4.Reporting Error, 5.Returning metadata, 6.Wrong filename

capability, BrainStor administrator can add storage hardware to create new logical partitions and redistribute metadata among the entire cluster. This metadata redistribution introduces multi-OMM communication because the change in the number of logical partitions requires a new mapping function f in Equation 5.1, and affects the metadata location of the existing files in logical partitions. For example, after scalability capability is improved from 64 to 256, the metadata of a file may need to move from logical partition 18 to logical partition 74. The procedure that redistributes all metadata based on new mapping policy and improves scalability capability, is called the OMM Cluster Rebuild.

In order to reduce the response time of the OMM cluster rebuild, HAP adopts Deferred Update algorithm, which defers metadata movement and distributes its overhead. After receiving the cluster rebuild request, HAP saves a copy of the mapping function f, creates a new f based on the new number of logical partitions, and generates a new MLT. Then logical partition manager mounts all logical partitions including both the old and new according to the new MLT. After that, HAP responds immediately to the rebuild request and changes the OMM cluster to a rebuild mode. Thus the initial operation for this entire process is very fast.

During the rebuild, the behavior of the system is as if all the metadata had been moved to the right logical partitions. And the difference is to deny another immediate change to improve the scalability capability of OMM cluster. For example, after system designers improve the SC from 64 to 256, BrainStor refuses another immediate change to improve SC from 256 to 1024 during the OMM cluster rebuild. Fortunately, the operation to improve the system scalability capability is very few, maybe once for several years, thus this effect is acceptable.

Based on Deferred Update algorithm HAP updates or moves the metadata upon the first access. If an OMM receives a metadata request, and the metadata has not been moved to the logical partition that is mounted by it, the OMM needs to use the old mapping function f to calculate the original logical partition number based on the filename. Then through the new MLT, the OMM can find the OMM that currently mounts the original logical partition and sends a metadata request to it. Finally the OMM retrieves the metadata from its original location and complete the client's metadata request as well as the metadata movement. Figure 5.5 describes this procedure.

In order to reduce the total time of the OMM cluster rebuild, besides the metadata movement upon first access, every OMM can have a thread to travel its metadata database and move the affected metadata to other OMMs. However the thread can only run in the background as system load permits. By setting the Maximal Permit Load (MPL) value, HAP can easily control the thread. Only the light-loaded OMMs whose loads are less than MPL, can run that thread to perform metadata movement. On the other hand, the heavy-loaded OMMs depend on the metadata access to move affected metadata to the new place. Thus, the overall performance is just slightly affected even during the OMM cluster rebuild. Furthermore, if a system simply requires that the time for the OMM cluster rebuild should be as little as possible, a large enough MPL lets these updating threads keep working and make the OMM cluster rebuild completed in the shortest time. Actually the longer the rebuild time, the less the effects to the system overall performance, because the rebuild process only costs spare system time and resources and does not compete with other critical metadata serving threads.

With all these algorithms, the OMM cluster rebuild can be completed effectively and BrainStor can support unlimited OMM cluster growth. However, in the actual application environment, according to the characters of the storage applications, a reasonable number of logical partitions can entirely avoid the OMM cluster rebuild.

5.5 Analysis and Experience

5.5.1 HAP Analysis

HAP uses hashing method to avoid the numerous metadata accesses, and uses filename hashing policy to remove the overhead of multi-OMM communication. However, most of current file systems are based on traditional directory metadata management. In Linux, Virtual File System (VFS) adopts file system interfaces based on the model of directory subtree structure. Therefore, one needs to redesign the Linux VFS in order to implement a hashing file system for HAP in Linux. Nevertheless the benefit of hashing method can still be demonstrated by some analysis.

Figure 5.6 compares the total number of accesses in order to get the metadata of files at different directory levels between HAP and a normal file system. The normal file system refers to a general file system adopting the directory metadata management, such as ext3. It is also supposed that there is no cache effect. The x axis stands for the depth of file in the directory tree, the y axis stands for the number of accesses in order to know the metadata of a file. Line 1 (L1) shows

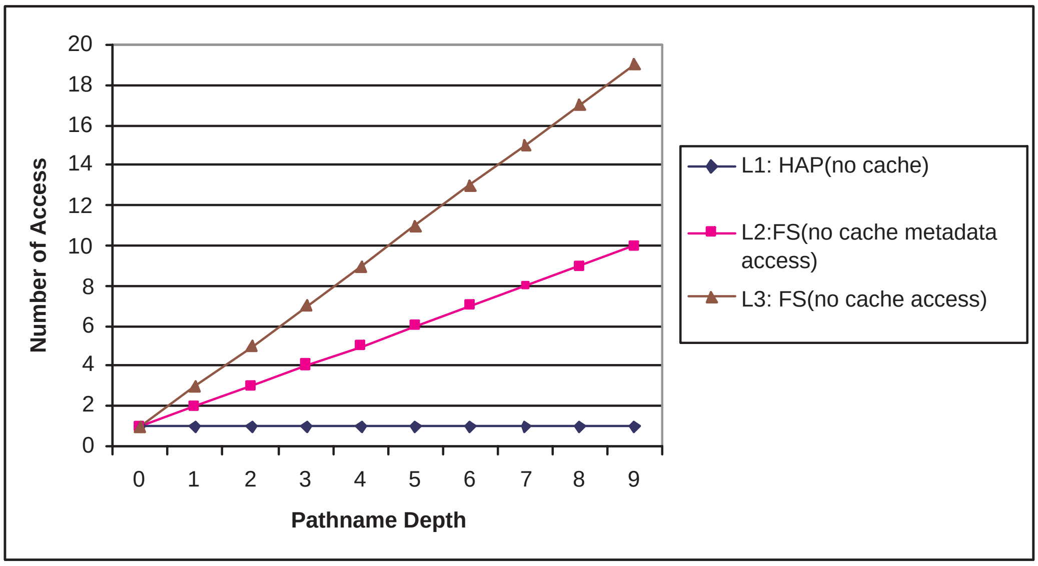


Figure 5.6: HAP Analysis Result without Cache Effects

the number of metadata access based on HAP without considering cache effect; Line 2 (L2) shows the number of metadata access based on normal file system without considering cache effect; Line 3 (L3) shows the number of metadata and data accesses based on normal file system without considering cache effect.

Because HAP adopts the hashing method, there is only one direct metadata access for each file, no matter what the depth of file pathname is. In order to simplify the analysis, the hashing collision is not considered in HAP analysis. Therefore, as the Line 1 shows that the number of accesses is always one. In order to access the metadata of a file, the traditional file system need go through all the nodes on the file path. Hence the number of metadata access is linearly increasing with the depth of pathname, as shown in Line 2.

In a normal file system, the sequence to access a file is to go through the metadata and data of nodes on the file path one by one. For example, in order to access the metadata of file: "/a/b/c", the metadata of "/" is accessed and then file system knows where the data of "/" is. After getting the data of "/", file system checks whether directory "a" is under the "/" directory by searching the content of "/". If "a" is found, the metadata location of "a" is known. Then similarly, file system needs to go through metadata of "a", data of "a", metadata of "b", data of “b” one by one. Finally, after knowing the metadata address of file “c” in the data of “b”, file system gets the metadata of file “c”. As a result, the number of accesses includes both metadata access and data access, and it is increasing even more sharply with the depth of pathname, as shown in Line 3. When the depth of pathname is 9, the number of metadata requests of normal file system is 10 times that of HAP and the number of all requests of normal file system is 19 times that of HAP.

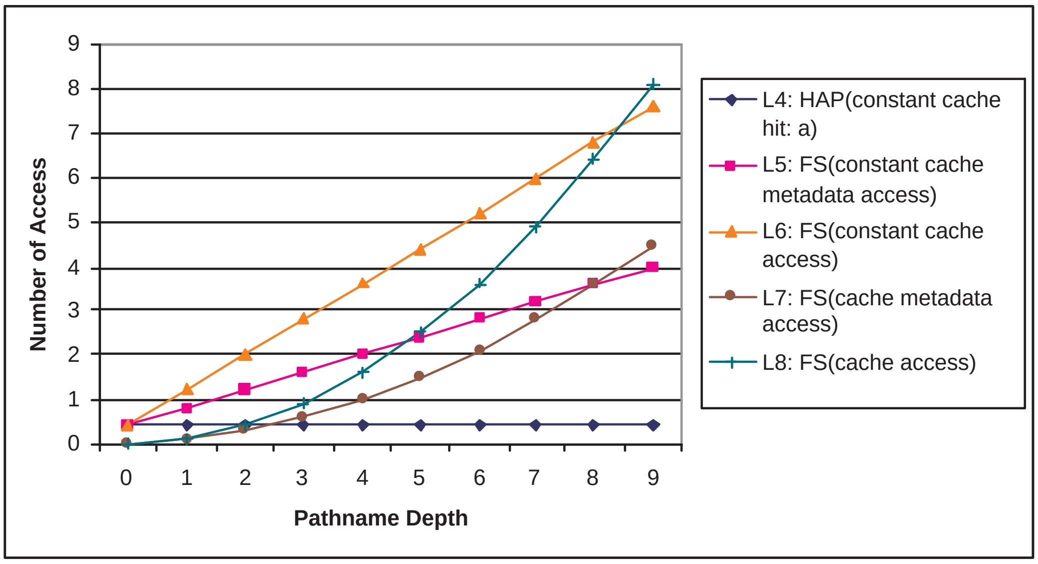


Figure 5.7: HAP Analysis Result with Cache Effects

Figure 5.7 shows the comparison result with the cache effect. Line 4 shows the HAP result when the cache hit rate is constant for files at all levels. Suppose that the cache hit rate is 60%, then number of accesses at all depth is 0.4 for HAP, as shown in Line 4 (L4). Line 5 (L5) shows the number of metadata access in normal file system where the cache hit rate is always 60% at all levels of directory subtree. Line 6 (L6) shows the number of accesses including both data access and metadata access in normal file system where the cache hit rate is always 60% at all levels of directory subtree. Line 7 (L7) shows the number of metadata access in normal file system where the cache hit rate is decreasing with the depth at the speed 10% per level. Suppose the cache hit rate is 1 at the root level (depth = 0). Then the cache hit of the first level is 90%. Line 8 (L8) shows the number of accesses including both data access and metadata access in normal file system under the same condition as indicated by the Line 7.

As shown in Figure 5.7, curves L5, L6, L7, and L8 all ascend quickly with the depth of directory. The variable cache hit rate leads to less metadata access than the constant cache hit rate when the pathname depth is less than 8. However, when the pathname depth is 9, the constant cache hit rate leads to less access. When the depth of pathname is 9, the number of metadata requests of normal file system in Line 5 is 10 times that of HAP and the number of all requests of normal file system in Line 6 is 19 times that of HAP; the number of metadata requests of normal file system in Line 7 is about 11 times that of HAP and the number of all requests of normal file system in Line 8 is 20 times that of HAP.

As discussed above, HAP has an obvious advantage to normal file system with either cache-enable or cache-disable, in terms of number of metadata access.

5.5.2 BrainStor Functional Experiments

HAP has been partially implemented in BrainStor prototype in order to support OMM cluster failover and scalability. In this section, some BrainStor functional experiments are explained in order to demonstrate HAP's strengths.

All the following experiment uses the standard BrainStor nodes introduced in Chapter 4. The following experiments follows the steps:

- Step 1: Add an OSM to a basic BrainStor setup in order to demonstrate the storage scalability.
- Step 2: Add an OMM to demonstrate the OMM cluster scalability.
- Step 3: Make one OMM failure to show the OMM failover ability, and finally recovery the failed OMM.

5.5.2.1 Storage Scalability Experiment

Experiment Setup

1. One basic BrainStor setup: one OSC, one OMM and one OSM connected by Fibre Channel Switch (CISCO DS-C 9509). All the nodes are also connected to a Compex DSR2216 Ethernet switch through their on-board NICs.
2. A standby OSM.
3. RaidTec JBOD as the common storage space.

Experiment Process

1. Power on the Fibre Channel Switch as well as Ethernet switch.
2. Set up the basic BrainStor prototype by simply powering on JBOD, OMM and OSM nodes sequentially. Then the basic BrainStor prototype is ready.
3. Power on the OSC that will automatically connect to the BrainStor prototype.
4. Run a test program in the OSC to keep creating files in the "/mnt/brainstor" directory, which is the mount point of BrainStor storage in Linux.
5. Connect the standby OSM to switches and power on it to dynamically scale the BrainStor storage while the test program in the OSC is running.
6. After the new OSM joins the BrainStor automatically, it received requests from the running program in OSC, and new objects are created accordingly in the OSM.

Observation Results

The experiment shows that BrainStor prototype supports storage scalability. Storage scalability is a key feature of advanced storage system. Without any downtime, BrainStor can scale the storage capacity and performance during the runtime of major applications. For example, during the BrainStor Iometer test, after the second OSM is dynamically added, the performance is almost doubled. Clearly, two OSMs can provide parallel access to the OSC.

5.5.2.2 OMM Cluster Scalability Experiment

Experiment Setup

1. One basic BrainStor setup: one OSC, one OMM and two OSMs connected by Fibre Channel Switch (CISCO DS-C 9509). All the nodes are also connected to a Compex DSR2216 Ethernet switch through the on-board NIC.
2. A standby OMM.
3. RaidTec JBOD as the common storage space.

(It is just the platform after storage scalability experiment.)

Experiment Process

1. This experiment can be directly conducted after storage scalability experiment, or it can be set up independently.
2. The test program used in storage scalability experiment is still running. The program keeps creating files in the "/mnt/brainstor" directory where Linux mounts the BrainStor storage. Because the OMM is in the debug mode, when the program is running, there are running messages on the screen of OMM due to the coming metadata requests.
3. Connect the standby OMM to switches and simply power on the OMM to dynamically scale the BrainStor metadata processing capability while the test

program is running. This new OMM is also with the debug message setting enabled.

4. After the new OMM joins the BrainStor automatically, there are running messages about the coming metadata requests on its screen as well. As a result, both OMMs have debug messages keep arriving on their screens.

Observation Results

The experiment shows that BrainStor prototype supports the OMM cluster scalability. The scalability of OMM cluster is crucial to handle the numerous metadata access requests. When the applications demands BrainStor to be more efficient in processing metadata requests, without any application downtime, a new OMM(s) can be dynamically added to solve the problem.

5.5.2.3 OMM Cluster Failover Experiment

Experiment Setup

1. One BrainStor setup: one OSC, two OMMs and two OSMs connected by Fibre Channel Switch (CISCO DS-C 9509). All the nodes are also connected to a Compex DSR2216 Ethernet switch through the on-board NIC.
2. RaidTec JBOD as the common storage space.

(It is just the platform after the OMM cluster scalability experiment.)

Experiment Process

1. This experiment can be directly conducted after the OMM cluster scalability experiment, or it can be set up independently.
2. The test program is still running. Just during the runtime, we can shutdown the power of one of OMMs directly in order to simulate an OMM failure.

3. The result of this OMM failure is that the program at the OSC stops for several seconds and goes on operation without any blocking. And the screen of the remaining OMM can still display continuously the debug messages of the arriving metadata access requests.

Observation Results

The experiment shows that the failover of OMM can be fully supported. After the OMM is removed without any notification, there are several seconds break of the running program. This is because BrainStor needs to ensure that the OMM is failed. After that, the existing OMM can take over the work of the failed one, as introduced in Section 5.3.2. Every OMM handles part of metadata storage in the common storage space (the JBOD). After the failure of one of the two OMMs, the remaining OMM can still support the metadata accesses that originally should go through the failed one.

After the OMM fails, a new OMM or the recovered OMM (the failed OMM) can be added to BrainStor. The procedure is still very simple and similar to the OMM cluster scalability experiment. For the recovered OMM, only the power-on is needed. For the new OMM, besides power-on, some simply commands to report its connection information such as WWN and IP address, are necessary. Then BrainStor can automatically detect it and allocate logical partitions accordingly. The entire recovery process does not introduce any downtime.

5.6 Summary

This chapter presents a new method of Hashing Partition to manage the OMM cluster in BrainStor system. HAP uses hashing method to avoid the numerous metadata accesses, and uses filename hashing policy to remove the overhead of multi-OMM communication. Furthermore, based on the concept of logical partitions in the common storage space, HAP method significantly simplifies the implementation of the OMM cluster and provides efficient solutions for load balancing, failover and scalability. A Dynamical Weight algorithm is also presented for OMM cluster load balancing.

Normally, the OMM cluster supports scalability without any metadata movement. However, if the OMM cluster scales to a number that is greater than the preset scalability capability, some metadata must be redistributed. This process is called the OMM cluster rebuild. In HAP, Deferred Update algorithm is proposed to improve the response time of the rebuild and minimize the cost of the OMM cluster rebuild.

Finally, HAP analysis results show that HAP can reduce the number of metadata access compared with that of the directory metadata management. The functional experiments show the BrainStor's storage scalability, OMM cluster scalability and OMM cluster failover functions. BrainStor can support these intelligent functions effectively with very simple design based on HAP.

Chapter 6

Conclusions and Future Works

6.1 Conclusions

This dissertation presents the design and implementation of BrainStor, a Fibre Channel OSD prototype. The primary motivation of the research is to provide an intelligent storage solution to feed the unlimited requirements of today's applications. Currently, the file-level NAS solution is good at cross-platform based on high-level abstract, but is poor in performance. And the block-level SAN solution benefiting from direct access, can achieve high performance, however it lacks effective means to provide cross-platform data sharing. Object is regarded as the convergence of file and block technologies and can also provide the advantages of both of them. BrainStor solution also aims at offering the strength of both NAS and SAN and overcoming their disadvantages. Based on object access, BrainStor system can achieve the high performance from direct access and the cross-platform data sharing from the high-level abstract object.

Another motivation of the study is to identify the key issues in OSD system design and implementation. OSD is a comparable new technology and has become a popular term in both academic and industrial research communities. However, the new object concept can raise many new challenges as well. In this study, an attempt is made to identify those important challenges through prototyping and testing an OSD storage system.

The main contributions of the thesis are summarized as follows:

1. BrainStor system, a Fibre Channel OSD prototype, is developed. BrainStor architecture presents an OSD architecture with unique Object Cache Module and Object Bridge Module.

There are six key components of BrainStor: Object Storage Client (OSC), Object Storage Module (OSM), Object Cache Module (OCM), Object Bridge Module (OBM), Object Manager Module (OMM) and Security Manager Module (SMM). In BrainStor, an independent OMM cluster is used to separate the metadata path and data path, thus that metadata server is removed from the data path and the OSC has the direct data access to storage. The OBM makes the BrainStor system compatible with the existing SAN components, such as RAID systems from different vendors. In addition, Brainstor also offers a scalable cache solution. OCM, as a centralized cache for the entire BrainStor system, can be scaled to meet the increasing and unlimited performance needs of storage applications. All the access to BrainStor is based on object, which enables the high performance and cross-platform data sharing at the same time.

2. Through analyzing BrainStor prototype test results, the dissertation shows some features of BrainStor, and further identifies some critical issues about OSD system design. Iometer and IOzone tests show that the storage scalability can greatly improve the capacity as well as overall performance of BrainStor, and storage virtualization of BrainStor can eliminate the system downtime. PostMark test unveils the metadata management challenges in the new OSD architecture.

3. In order to address the metadata management issue, the dissertation proposes a Hashing Partition (HAP) method in the OMM cluster design. HAP uses hashing method to avoid the numerous metadata accesses, and uses filename hashing policy to remove the overhead of multi-OMM communication. Furthermore, based on the concept of logical partitions in the common storage space,

HAP method significantly simplifies the implementation of the OMM cluster and provides efficient solutions for load balancing, failover and scalability. Dynamic Weight algorithm is also proposed for OMM cluster load balancing. HAP makes the once expensive operations in other systems simple and efficient. The massive metadata movement is replaced by some *mount/umount* operations, which can be completed instantly.

Normally, the OMM cluster supports scalability without any metadata movement. However, if the OMM cluster scales to a number that is greater than the preset scalability capability, some metadata must be redistributed. This process is called the OMM cluster rebuild. The Deferred Update algorithm is proposed to improve the response time of the rebuild and minimize its cost.

4. Analysis results of the hashing method show that the HAP can reduce the number of metadata requests compared with directory metadata management. The comparisons are conducted in two situations: considering cache effects and without considering cache effects. In both conditions, the result shows that HAP has obvious advantage to directory metadata management in terms of the number of metadata access.

In addition, BrainStor functional experiments demonstrate the storage scalability capability and OMM cluster's scalability and failover ability.

6.2 Future Works

With increasing interests on OSD technologies, more and more research works have been done to develop the promising technology. Object-based storage is the future trend of network storage.

For BrainStor project, the future works includes four aspects:

1. The distributed object file system

2. Management algorithms of the OMM cluster
3. Object management algorithms of OSM
4. The data security in BrainStor

In terms of the conjunction of BrainStor and other related technologies, to explore the application of BrainStor technologies in Grid storage is also an interesting topic.

Bibliography

[1] Technical Report of University of California, Berkeley, “How Much Information? 2003,” http://www.sims.berkeley.edu/research/projects/how-much-info-2003/printable_report.pdf, 2003.

[2] An internal working document of T10, “SCSI Architecture Model - 3 (SAM-3),” http://www.t10.org/ftp/t10/drafts/sam3/sam3r14.pdf, 2004.

[3] Leach, P. and D. Naik, “A Common Internet File System (CIFS/1.0) Protocol,” draft-leach-cifs-v1-spec-01.txt, December 1997.

[4] Marc Farley, “Building Storage Networks,” McGraw-Hill Companies, 2000.

[5] J. Tate, A. Bernasconi, P. Mescher and F Scholten, “Introduction to Storage Area Networks,” IBM Redbook, 2004.

[6] R. C. Burns, “Data Management in a Distributed File System for Storage Area Networks,” Ph.D. Dissertation, Univ. of California, Santa Cruz, March 2000.

[7] D. A. Patterson, G. Gibson and R. H. Katz, “A case for redundant arrays of inexpensive disks (RAID),” *ACM SIGMOD conference on management of data*, pp109-116, June 1988.

[8] G. A. Gibson and R. Van Meter, “Network Attached Storage Architecture,” *Communication of the ACM*, Volume: 43, Issue: 11, Pages: 37 - 45, 2000.

[9] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and Implementation of the Sun Network Filesystem," *In Proceedings of the Summer 1985 USENIX Conference*, Pages: 119 - 130, 1985.

[10] J. Lu, X. L. Lu, H. Han and Q. S. Wei, "A cooperative asynchronous write mechanism for NAS," *ACM SIGOPS Operating Systems Review*, Volume: 36, Issue: 3, 2002.

[11] I. Ari, M. Gottwals and D. Henze, "SANboost: automated SAN-level caching in storage area networks," *IEEE Proceedings of the International Conference on Autonomic Computing*, Pages: 164 - 171, 2004.

[12] B. Gordon, S. Oral, G. Li, H. Su and A. George, "Performance analysis of HP AlphaServer ES80 vs. SAN-based clusters," *IEEE Proceedings of the 2003 IEEE International conference on Performance, Computing and Communications*, Pages: 69 - 76, 2003.

[13] C. Y. Wang, F. Zhou, Y. L. Zhu, T. C. Chong, B. Hou and W. Y. Xi, "Simulation of fibre channel storage area network using SANSim," *The 11th IEEE International Conference on Networks*, Pages: 349 - 354, 2003.

[14] Y. Gao, Y.L. Zhu, H. Xiong, R. Kanagavelu, J. Yan, Z.J. Liu, "An iSCSI Design over Wireless Network," *IEEE International Conference On Networks (ICON 2004)*, November 2004.

[15] D. A. Patterson, G. Gibson and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," *ACM SIGMOD conference on management of data*, pp109-116, June 1988.

[16] R. O. Weber, "Information technology-SCSI object-based storage device commands (OSD)," Technical Council Proposal Document T10/1355-D, Technical Committee T10, August 2003.

[17] R. Snively and I. D. Allan, "dpANS Fibre Channel Protocol for SCSI," http://www.t10.org/ftp/t10/drafts/fcp/fcp-r12.pdf, Decemeber 1995.

[18] J. Satran et al., "iSCSI (Internet SCSI) Specification, Internet Draft," http://www.ietf.org/internet-drafts/draft-ietf-ips-iscsi-20.txt, 2003.

[19] Mike Mesnier, Gregory R. Ganger, and Erik Riedel, "object-based storage," *IEEE communications magazine*, August 2003.

[20] 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, March 1986.

[21] Thomas M. Ruwart, "OSD: A Tutorial on Object Storage Devices," *19th IEEE Symposium on Mass Storage Systems and Technologies*, April 2002.

[22] Intel, "Internet SCSI (iSCSI) Reference Implementation," http://www.intel.com/labs/storage.

[23] E. Riedel, G. Gibson, and C. Faloutsos, "Active Storage for Large-Scale Data Mining and Multimedia Applications," *Intl. Conf. Very Large DBs*, New York, NY, pp. 62-73. August 1998.

[24] G. Almes and G. Robertson, "An Extensible File System for HY-DRA," *3rd Intl. Conf. Software Eng.*, Atlanta, GA, May 1978.

[25] F. J. Pollack, K. C. Kahn, and R. M. Wilkinson, "The iMAX-432 Object Filing System," *ACM Symp. OS Principles*, Asilomar, CA, published in OS Rev., vol. 15, no. 5, December 1981, pp. 137-47.

[26] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, E. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, and D. Rochberg, "A Case for Network-Attached Secure Disks," CMU SCS Technical Report CMU-CS-96-142, September 1996.

[27] G. Gibson, D.F. Nagle, K. Amiri, F.W. Chang, H. Gobioff, E. Riedel, D. Rochberg and J. Zelenka, "Filesystems for Network-Attached Secure Disks," CMU SCS Technical Report CMU-CS-97-118, 1997.

[28] G. A. Gibson et al., "A Cost-effective, High-bandwidth Storage Architecture," *Architectural Support for Prog. Languages and OS*, San Jose, CA, October 1998.

[29] K. S. Amiri, "Scalable and Manageable Storage Systems," Ph.D. Dissertation, CMU-CS-00-178. Carnegie-Mellon Univ. December 2000.

[30] IBM, Storage Tank, http://www.almaden.ibm.com.

[31] R. Zahir, "Lustre Storage-Networking Transport Layer," Intel Document, September 2001.

[32] P. Braam, The Lustre Book, http://projects.clusterfs. com/lustre.

[33] G. G. R. Ganger, J. D. Strunk, A. J. Klosterman, "Self-* Storage: Brick-based storage with automated administration," Carnegie Mellon University Technical Report, CMU-CS-03-178, August 2003.

[34] G. R. Ganger, "Blurring the Line Between OSs and Storage Devices," Tech. rep. CMU-CS-01-166, Carnegie Mellon Univ., December 2001.

[35] J. D. Strunk and G. R. Ganger, "A Human Organization Analogy for Self-* Systems," *First Workshop on Algorithms and Architectures for Self-Managing Systems.* In conjunction with *Federated Computing Research Conference (FCRC).* San Diego, CA. June 2003.

[36] M. Mesnier, E. Thereska, D. Ellard, G. R. Ganger and M. Seltzer, "File Classification in Self-* Storage Systems," *Proceedings of the First International Conference on Autonomic Computing (ICAC-04)*, New York, NY. May 2004.

[37] M. Sivathanu, A. C. Arpaci-Dusseau, and R. H. Arpaci- Dusseau, "Evolving RPC for Active Storage," *Architectural Support for Prog. Languages and OS*, San Jose, CA, October 2002.

[38] D. P. Reed and L. Svobodova, “SWALLOW: A Distributed Data Storage System for a Local Network,” *Intl. Wksp. Local Networks*, Zurich, Switzerland, August 1980.

[39] F. Zhou, C. Jin, Y. Wu, W. Zheng “TODS: Cluster Object Storage Platform Designed for Scalable Services,” *Proceedings of the Fifth International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP.02)*, 2002.

[40] E. L. Miller, D. D. E. Long, W. Freeman, and B. Reed, “Strong security for distributed file systems,” *In Proceedings of the 20th IEEE International Performance, Computing and Communications Conference (IPCCC ’01)*, pages 34-40, Phoenix, April 2001.

[41] F. Wang, S. A. Brandt, E. L. Miller, and D. D. E. Long, “OBFS: A file system for object-based storage devices,” *In Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 283-300, College Park, MD, April 2004.

[42] Q. Xin, E. L. Miller, and T. J. E. Schwarz, “Evaluation of distributed recovery in large-scale storage systems,” *In Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 172-181, Honolulu, HI, June 2004.

[43] AJ Lewis, “LVM HOWTO,” http://www.tldp.org/HOWTO/LVM-HOWTO/, 2004.

[44] Kai Hwang, “Advanced Computer Architecture: Parallellism, Scalability, Programmability,” ISBN 0-07-031622-8, 1993.

[45] D. P. Bovet and M. Cesati, “Understanding the Linux Kernel, 2nd Edition,” ISBN: 0-596-00213-0, 2002.

[46] H. Gobioff, “Security for a High Performance Commodity Storage Subsystem,” Ph.D. Dissertation, TR CMU-CS-99- 160. Carnegie-Mellon Univ., July 1999.

[47] Intel, "Iometer Users Guide," http://www.iometer.org, July 2004.

[48] W. D. Norcott and D. Capps, "IOzone Filesystem Benchmark," http://www.iozone.org, 1998.

[49] KATCHER, J. "PostMark: A new file system benchmark," Tech. Rep. TR3022. Network Appliance, October 1997.

[50] J. Yan, Y.L. Zhu, H. Xiong, R. Kanagavelu, F. Zhou, S.L. Weon, "A Design of Metadata Server Cluster in Large Distributed Object-based Storage," *12th NASA Goddard, 21st IEEE Conference on Mass Storage Systems and Technologies (MSST 2004)*, April 2004.

[51] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson, "A trace-driven analysis of the Unix 4.2 BSD file system," *In Proceedings of the 10th ACM Symposium on Operating Systems Principles (SOSP 85)*, pages 15-24, December 1985.

[52] E. Levy and A. Silberschatz, "Distributed file systems: Concepts and examples," *ACM Computing Surveys*, 22(4), December 1990.

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

[54] Scott A. Brandt, Lan Xue, Ethan L. Miller, and Darrell D. E. Long, "Efficient metadata management in large distributed file systems," *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 290-298, April 2003.

[55] S. Soltis, T. M. Ruwart, and M. T. O'Keefe, "The global file system," In Proc. of *the Fifth NASA Goddard Conference on Mass Storage Systems and Technologies*, September 1996.

[56] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz, "NFS version 3: Design and implementation," *Proceedings of the Summer 1994 USENIX Technical Conference*, pages 137-151, 1994.

[57] 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, March 1986.

[58] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere, "Coda: A highly available file system for a distributed workstation environment," *IEEE Transactions on Computers*, 39(4):447-459, 1990.

[59] G. J. Popek and B. J. Walker, "The LOCUS distributed system architecture," Massachusetts Institute of Technology, 1986.

[60] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B.Welch, "The Sprite network operating system," *IEEE Computer*, 21(2):23-36, February 1988.

[61] L. Mummert and M. Satyanarayanan, "Long term distributed file reference tracing: Implementation and experience," *SoftwarePractice and Experience (SPE)*, 26(6):705-736, June 1996.