Experiences Building an Object-Based Storage System based on the OSD T-10 Standard

David Du, Dingshan He, Changjin Hong, Jaehoon Jeong, Vishal Kher,
Yongdae Kim, Yingping Lu, Aravindan Raghuveer, Sarah Sharafkandi
DTC Intelligent Storage Consortium
University of Minnesota
{*du, he, hong, jjeong, vkher, kyd, lu, aravind, ssharaf*}*@cs.umn.edu*

Abstract

With ever increasing storage demands and management costs, object based storage is on the verge of becoming the next standard storage interface. The American National Standards Institute (ANSI) ratified the object based storage interface standard (also referred to as OSD T-10) in January 2005. In this paper we present our experiences building a reference implementation of the T10 standard based on an initial implementation done at Intel Corporation. Our implementation consists of a file system, object based target and a security manager. To the best of our knowledge, there is no reference implementation suite that is as complete as ours. Efforts are underway to open source our implementation very soon. We also present performance analysis of our implementation and compare it with an iSCSI based SAN and NFS storage configurations. In future, we intend to use this implementation as a platform to explore different forms of storage intelligence.

1. Introduction

Recent studies show that the storage demands are growing rapidly and if this trend continues, storage administration costs will be higher than the cost of the storage systems themselves. Therefore intelligent, self managing and application aware storage systems are required to handle this unprecedented increase in the storage demands. To be self managing, the storage device needs to be more aware of the data it is storing. But the current block interface to storage systems is very narrow and cannot convey any such additional semantics to the storage. This forms the fundamental motivation behind revamping the storage interface from a narrow, rigid interface to a more "expressive" and extensible interface. This new storage interface is termed as the object based storage interface.

Storage devices that are based on this object based interface (referred to as Object Based storage devices) will store and manage data containers called objects which can be viewed as a convergence of two technologies: files and blocks [17]. Files have associated attributes which convey some information about the data that is stored within. Blocks, on the other hand, enable fast, scalable and direct access to shared data. Objects can provide both the above advantages. The NASD project at CMU [12] provided the initial thrust for the case of object based storage devices. Recently, Lustre [4] and Panasas [6] have used object based storage to build high performance storage systems. But both these implementations use proprietary interfaces and hence limit interoperability.

Standardization of the object interface is essential to enable early adoption of object based storage devices and to further increase its market potential. To address this concern, an object based storage interface standard (OSD T10) was ratified by ANSI in January 2005 and the first version released [8]. An implementation of the standard along with a filesystem, would help quicker adoption of the OSD standard by providing an opportunity for the interested vendors/researchers to obtain a hands-on experience of what OSD can provide. Another important advantage of a open source reference implementation is that it can serve as a conformance point to test for interoperability[1] when multiple OSD products arrive in the market. As explained earlier, the OSD interface is just a means to provide more knowledge about the data (through attributes of the object) to the storage device. Mechanisms that use this knowledge to improve performance are called *Storage Intelligence*. Researchers can build "layers" over a reference implementation to investigate into various techniques to provide storage intelligence.

[1] All member companies of DISC have expressed strong interest in establishing such an interoperability test lab.

Based on the above motivations, we have implemented a complete object based storage system compliant to the OSD T-10 standard. In this paper, we present our experiences building this reference implementation. Our work is based on an initial implementation done by Mike Mesnier from Intel Corporation (now at Carnegie Mellon University). Our implementation consists of a file system, an object based target and a security manager, all compliant with the T-10 spec. To the best of our knowledge, there is no open source reference implementation suite that is as complete as ours. Efforts are currently underway to open source our implementation soon and we believe that, once available, such a implementation can hasten the adoption of the OSD T-10 standard in the storage community.

The aim of this work was to develop a quick yet complete prototype of an OSD based storage system that is based on industry standards. We then want to use this implementation to explore the new functionalities that OSD based systems can provide to current and future applications. More specfically, we want to investigate on how applications can convey semantics to storage and how the storage system, in turn, can use these to improve some system parameters like performance, scalability etc.

The remainder of the paper is organized as follows. In Section 2 we first briefly present an overview of the T10 standard. Section 3 discusses the various design and implementation issues that we handled during implementing the standard. In Section 4 we discuss the performance evaluation methodology used and present results. Some relevant related work is presented in Section 5 . Section 6 concludes the paper and discusses avenues for future work.

2. Overview of the T10 SCSI OSD Standard

The OSD specification [8] defines a new device-type specific command set in the SCSI standards family. The Object-based Storage device model is defined by this specification. It specifies the required commands and behavior that is specific to the OSD device type.

Figure 1 depicts the abstract model of OSD in comparison to traditional block-based device model for a file system. The traditional functionality of file systems is repartitioned primarily to take advantage of the increased intelligence that is available in storage devices. Object-based Storage devices are capable of managing their capacity and presenting file-like *storage objects* to their hosts. These storage objects are like files in that they are byte vectors that can be created and destroyed and can grow and shrink their size during their lifetimes. Like a file, a single command can be used to read or write any consecutive stream of the bytes constituting a storage object. In addition to mapping data to storage objects, the OSD storage management component maintains other information about the storage

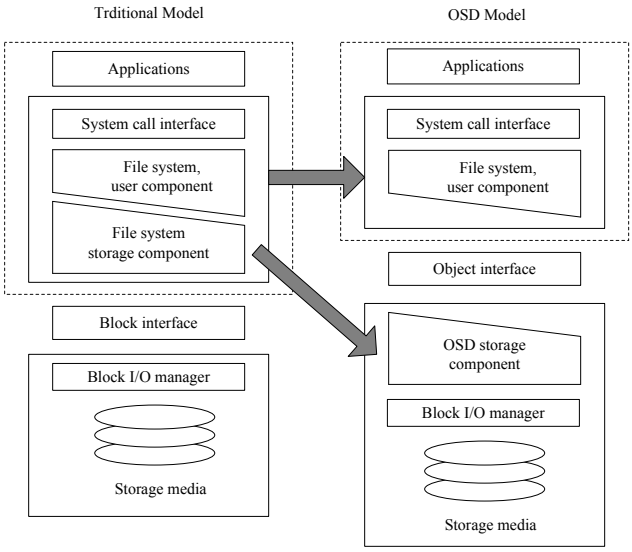

Figure 1. Comparison of traditional and OSD storage models

objects in attributes, e.g., size, usage quotas and associated user name.

2.1. OSD Objects

In the OSD specification, the storage objects that are used to store regular data are called *user objects*. In addition, the specification defines three other kinds of objects to assist navigating user objects, i.e., *root object*, *partition objects* and *collection objects*. There is one *root object* for each OSD logical unit [7]. It is the starting point for navigation of the structure on an OSD logical unit analogous to a partition table for a logical unit of block devices. *User objects* are collected into partitions that are represented by *partition objects*. There may be any number of partitions within a logical unit up to a specific quota defined in the *root object*. Every *user object* belongs to one and only one partition. The collection represented by a *collection object* is another more flexible way to organize *user objects* for navigation. Each *collection object* belongs to one and only one partition and may contain zero or more *user objects* belonging to the same partition. Different from *user objects*, all three kinds of aforementioned navigating objects do not contain a read/write data area. All relationships between objects are represented by object attributes discussed in the next section.

Various storage objects are uniquely identified within an OSD logical unit by the combination of two identification numbers: the Partition_ID and the User_Object_ID as illustrated in Table 1. The ranges not specified in the table are reserved.

Partition_ID	User_Object_ID	Object type
0	0	root object
2^{20} to $2^{64}-1$	0	partition object
2^{20} to $2^{64}-1$	2^{20} to $2^{64}-1$	collection/user object

Table 1. Object identification numbers

2.2. Object Attributes

Object attributes are used to associate meta data with any OSD object, i.e., root, partition, collection or user. Attributes are organized in pages for identification and reference. Attribute pages associated with an object is uniquely identified by their attribute page numbers ranging from 0 to $2^{32}-1$. This page number space is divided into several segments so that page numbers in one segment can only be associated with certain type of object. For instance, the first segment from $0x0$ to $0x2FFFFFFF$ can only be associated with *user objects*.

Attributes within an attribute page have similar sources or uses. Each of them has an attribute number between $0x0$ and $0xFFFFFFFE$ that is unique within the attribute page. The last attribute number, i.e., $0xFFFFFFFF$ is used to represent all attributes within the page when retrieving attributes.

The OSD specification defines a set of standard attribute pages and attributes that can be found in [8]. Certain range of attribute pages and attribute numbers are reserved for other standards, manufacturer specific or vendor specific ones. By this way, new attributes can be defined to allow OSD to perform specific management functions. In [15], a new attribute page containing QoS related attributes is defined to enable OSD to enforce QoS.

2.3. Commands

The OSD commands are executed following a request-response model as defined in SCSI Architecture Model (SAM-3) [7]. This model can be represented as a procedure call as following:

Service response = Execute Command(IN(I_T_L_x Nexus, CDB, Task Attribute, [Data-In Buffer Size], [Data-Out Buffer], [Data-Out Buffer Size], [Command Reference Number), OUT([Data-In Buffer], [Sense Data], [Sense Data Length], Status))

The meaning of all inputs and outs are defined in SAM-3 [7]. The OSD specification additional defined the contents and formats of CDB, Data-Out Buffer, Data-Out Buffer Size, Data-in Buffer, Data-in Buffer Size and sense Data.

The OSD commands use the variable length CDB format defined in SPC-3 but has a fixed length of 200 bytes. Each OSD command has an opcode $0x7F$ in CDB to differentiate it from commands of other command sets. In the same CDB, a two-byte service action field specifies one

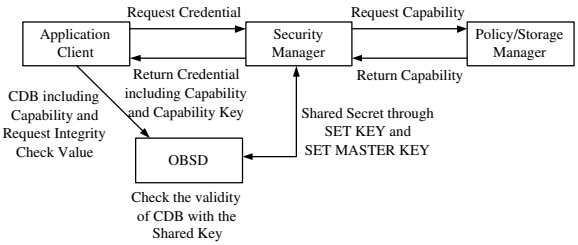

Figure 2. OSD Security Model

of the twenty-three OSD service requests defined in the OSD specification. Some of the CDB fields are specific to service actions and others are common for all commands. Every CDB has a Partition_ID and a User_Object_ID, the combination of which uniquely identifies the requested object in a logical unit. Any OSD command may retrieve attributes and any OSD command may store attributes. Twenty-eight bytes in CDB are used to define the attributes to be set and retrieved. Two other common fields in CDB are capability and security parameters that will be explained later.

Both Data-In Buffer and Data-Out Buffer contains multiple segments, including command data segments, parameter data segments, set/get attribute segments and integrity check value segments. Each segment is identified by the offset of its first byte from the first byte of the buffer. Such offsets are referenced in CDB to indicate where to get data and where to store data.

If the return status of an OSD command is CHECK CONDITION, sense data are also returned to report errors generated in OSD logical units. The sense data contain information that allows initiators to identify the OSD object in which the reported error was detected. If possible, a specific byte or range of bytes within a *user object* is identified as being associated with an error. Any applicable errors can be reported by include the appropriate sense key and additional sense code to identify the condition. The OSD specification chooses descriptor format sense data to report all errors so several sense data descriptors can be returned together.

2.4. Security Model

Figure. 2 shows the OSD security model consisting of four components [8, 11]: (a) Application Client, (b) Security Manager, (c) Policy/Storage Manager, and (d) Object-based Storage Device (OBSD). Whenever an application client performs an OSD operation, it contacts the security manager in order to get a capability including the operation permission and capability key to generate an integrity check value with OSD Command Description Block (CDB). When the security manager receives the capability request from the application client, it contacts the pol-

icy/storage manager to get a capability including permission. After obtaining the capability, the security manager creates a capability key with a key shared between the security manager and OBSD and makes the credential consisting of the capability and capability key, which is returned to the application client. Now the application client copies the capability included in the credential to the capability portion of the CDB and generates an integrity check value of the CDB with the received capability key. The CDB with the digested hash value called the request integrity check value is sent to the OBSD. When the OBSD receives the CDB, it checks the validity of the CDB with the request integrity check value. The shared secret between the security manager and OBSD for the authentication of the CDB is maintained by SET KEY and SET MASTER KEY commands [8].

2.4.1. OSD Security Methods There are four kinds of security methods in OSD [8, 11]: (a) NOSEC, (b) CAPKEY, (c) CMDRSP, and (d) ALLDATA.

In NOSEC, since the validity of the CDB is not verified in CDB, the requested integrity check value is not generated, but the capability of the CDB is obtained from the security manager and policy/storage manager.

In CAPKEY, the integrity of the capability included in each CDB is validated. The requested integrity check value is computed by the application client using the algorithm specified in the capability's integrity check value algorithm field, the security token returned in the security token VPD page [8], and the capability key included in credential. The OBSD validates the CDB sent by the application client with the request integrity check value included in the CDB and the newly computed request integrity check value from the CDB where the request integrity check value field is initialized into zero.

In CMDRSP, the integrity of the CDB (including capability), status, and sense data for each command is validated. The application client computes the request integrity check value of the CDB using the algorithm specified in the capability's integrity check value algorithm field, all the bytes in the CDB with the request integrity check value field set to zero, and the capability key included in credential. The OBSD validates the CDB sent by the application client by comparing the received request integrity check value with the newly computed request integrity check value.

In ALLDATA, the integrity of all data between an application client and an OBSD in transit is validated. The application client computes the request integrity check value in the CDB using the same algorithm specified for the CMDRSP security method, which is validated in the OBSD. Also, for checking the integrity of the data, the application client computes the data-out integrity check value using the algorithm specified in the capability's integrity check value algorithm field, the used bytes in the Data-Out Buffer segments [8], and the capability key included in credential.

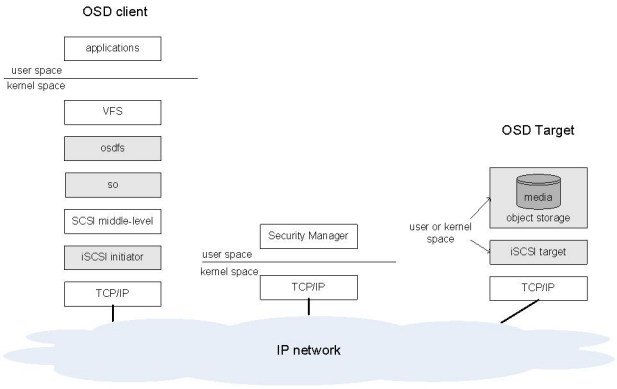

Figure 3. Overview of reference implementation

3. System Design and Implementation

The reference implementation consists of client components and server components shown in Figure 3 as grayed blocks. The client components include three kernel modules - the osd file system (*osdfs*), the scsi object device driver (*so*) and the iSCSI initiator host driver. The osd file system is a simple file system using object devices instead of block devices as its storage. The *so* driver is a SCSI upper-level driver and it exports an object device interface to applications like *osdfs*. The iSCSI initiator driver is a SCSI low-level driver providing iSCSI transport to access remote iSCSI targets over IP networks. The server components include the iSCSI target server and the object storage server. The iSCSI target driver implements the target side of the iSCSI transport protocol. The object target server module manages the physical storage media and processes SCSI object commands. The functions and internal architectures of these components are elaborated in following sections.

3.1. OSD Filesystem

The *osdfs* file system uses object devices as its storage. Regular files are not surprisingly stored as user objects. Directory files are also stored as user objects whose data contain mappings from sub-directory names to user object identifiers. The metadata of both regular files and directory files, i.e., information in VFS inodes, are stored as an attribute of their user objects. This mapping from traditional file system logical view to objects stored in object storages is illustrated in Figure 4 So far, there is no consideration

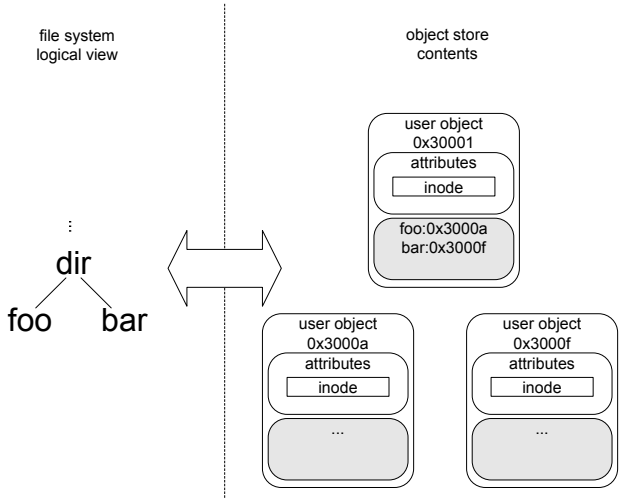

Figure 4. Mapping of files to objects

of special files like device files, pipe or FIFO. For each *osdfs*, a partition object is created to contain all user objects corresponding to regular files and directory files in the file system. Therefore, when mounting an existing *osdfs*, the partition object identifier and the user object identifier of the root directory of the file system need to be provided as mounting parameters.

The *osdfs* file system is implemented compliant with VFS like any other file systems on Linux. Therefore, it can also take advantage of the generic facilities provided by VFS including inode caches, dentry caches and file page caches. Different from other block-device file systems like ext3, *osdfs* can not use the buffer cache of Linux operating system since buffer cache is designed for block devices. In fact, buffer caches are not necessary for applications of object devices since the purpose of buffer caches is to access block disks in large contiguous chunks to achieve high disk throughput. In the object storage model, this storage management function is offloaded into object-based storage devices.

The *osdfs* file system currently is a non-shared file system since there is no mechanism in place to coordinate concurrent accesses from multiple hosts to the same objects. The OSD standard has not yet defined any concurrency control mechanism for the objects. In [13], an iSCSI-target-based concurrency control scheme has been proposed for iSCSI-based file systems. Similar mechanism is expected to be added in the future versions of the OSD standard.

3.2. SCSI Object Device Driver

The SCSI object device driver (*so*) is a new SCSI upper-level device driver in addition to SCSI disk (sd), SCSI tape (st), SCSI CDROM (sr) and SCSI generic (sg) drivers. Its main function is to manage all detected OSD type SCSI devices just like the sd driver manages all disk type SCSI devices and help the higher level applications to access these devices.

The *so* driver provides an well-defined object device interface for higher level application like *osdfs* to interact with the registered OSD devices. In this way, applications and device drivers can be modified without affecting each other. Currently, this object device interface is exactly the OSD commands interface define in T10 OSD standard [8].

Linux kernel currently only supports block devices, character devices and network devices [10]. Fortunately, the Linux block I/O subsystem was designed so generic that the object device driver can fit it easily. The *so* driver registers itself as a block device to Linux kernel. It implements the applicable block device methods defined by the *block_device_operations* structure including *open*, *release*, *ioctl*, *check_media_change* and *re-validate*. The Linux block I/O subsystem uses request queues to allow device drivers to make block I/O requests to devices. The request queue is a very complex data structure designed to optimize block IO access for disks including IO scheduling (like elevator, deadline or anticipatory scheduling) and IO coalescing. Once again, such storage management functions are offloaded into object storages in OSD model. The *so* driver bypasses the request queue and directly passes SCSI commands to SCSI middle-level driver, who will asks the appropriate SCSI low-level driver to further handle the commands.

3.3. iSCSI Transport

The iSCSI initiator driver and the iSCSI target server together implement the iSCSI protocol, which is a SCSI transport protocol over TCP/IP. It can transport both SCSI OSD commands and SCSI block commands.

The iSCSI initiator driver is implemented as a low-level SCSI driver. When the host starts or this driver is loaded as kernel module after the system starts, it tries to discover logical units (LUN) on pre-configured iSCSI targets, setup iSCSI sessions with accessible LUNs and negotiate session parameters with the targets. During the discovery process, the targets inform the initiator what type of SCSI device they are, either OSD or disk currently. The SCSI middle-level driver asks every known upper-level driver including *so* to check whether they are willing to manage the specific type of device. The *so* driver will register and manage OSD type devices and the *sd* driver will handle disk type devices. After the discovery phase and parameter negotiation phase, the sessions enter full feature phase and are ready to transfer iSCSI protocol data units (PDU).

As illustrated in Figure 5, the sending and receiving of iSCSI PDUs are handled by a pair of worker threads called

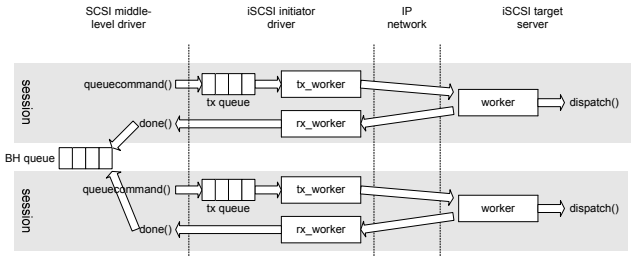

Figure 5. iSCSI implementation

tx_worker and *rx_worker* created for every active iSCSI session. Each session has a transmission queue (*tx_queue*) that the session's *tx_worker* thread can get the PDUs for sending. When there is no PDU to send in the queue, *tx_worker* threads are blocked. Any *rx_worker* thread is blocked until the *tx_worker* thread of its session has successfully sent out a PDU and unblocks it to receive responses or data.

When applications request to access storage devices, the SCSI upper-level device drivers are asked to construct SCSI commands (either OSD commands by *so* or block commands by *sd*). The SCSI middle-level driver passes the SCSI commands to the iSCSI initiator driver by calling a low-level driver specific *queuecommand()* method. When iSCSI initiator driver's *queuecommand()* is call, it encapsulates the SCSI commands and any associated data into iSCSI PDUs and puts the PDUs on appropriate session transmission queues. Reversely, the iSCSI initiator driver decapsulates iSCSI PDUs received on the IP network and trigger the callback function *done()*. This callback function is actually an hardware interrupt handler that enqueues a delayed software interrupt into the Linux bottom-half (BH) queue. The application processes waiting for the response are waken up by the bottom-half handler.

The iSCSI target server is the peer component of the iSCSI initiator driver. It maintains active sessions with connected iSCSI initiators. There is one dedicated worker thread for every session to both receive and transmit iSCSI PDUs from and to the peer. Noting that there can be multiple sessions between an initiator and a target if the initiator is allowed to access more than one LUNs on the target. Received iSCSI PDUs are dispatched to appropriate processing functions.

3.4. Object Based Target

The primary function of the object based target is to expose the T-10 object interface to an initiator and abstract the details of the actual storage architecture behind this interface. The underlying storage architecture could, in turn, be based on existing storage technologies (like RAID, NAS, SAN) or object devices. An implementation of the target has to address the following key issues: interpret the OSD SCSI commands from the initiator to match the underlying storage device, manage free space in the storage architecture, maintain physical locations of data objects, provide concurrency control. In the next paragraphs, we first provide a broad overview of our target implementation and then elucidate few key implementation aspects in further detail.

Our target executes as a user level server process that implements an iSCSI target interface. Therefore an iSCSI initiator can establish a session with the target and execute OSD SCSI commands. A worker thread is spawned for each incoming connection and is responsible for decapsulating the iSCSI CDB and interpreting the commands. So the server acts as a command interpreter that affects the state of the storage based on the commands sent by the initiator. Our current implementation does not support concurrency control at the target to maintain consistency when multiple clients write to the same user object or make changes to the namespace. In the following paragraphs, we explain in further detail the two central functions of the object based target.

Storage and namespace Management: In order to store and retrieve user objects, the target should manage the free space and maintain data structures to locate objects on the storage device. These two functions form the core of any filesystem. We therefore offload these tasks to an ext3 filesystem. All user objects and partitions are mapped onto the hierarchical namespace that is managed by the filesystem. Other functionalities like the quota management, maintaining fine grained timestamps is done by our code, outside the scope of the filesystem. As a straightforward mapping, user objects are mapped to files and partition objects are mapped onto directories. We currently do not support collection objects as it is not part of the normative section of the standard. We also store the attributes of the root object, partition objects and user objects as files. We however do realize that this method of using the filesystem as a means to manage storage may have certain drawbacks. For example, the overhead of opening and reading a file for a GET ATTRIBUTE command can be prohibitively high. We have identified optimization of the storage management module as one of the key areas of future work.

Command Interpreter: The command interpreter is responsible for converting the object commands into a form that can be understood by the underlying storage system. In our case, since we use a file system to abstract the storage, the command interpreter translates the OSD SCSI commands to filesystem calls. For example, an OSD WRITE is converted to a *write()* call and so on. Every command goes through five distinct phases during its execution.

1. Capability Verification: In this step, the capability is extracted from the CDB and checked if the requested command can be executed on the specified object. The command is not executed if the client does not have the required permissions or the if the credibility of the CDB cannot be verified. The precise steps have been discussed in detail in Section-2.4

2. Attribute Pre-process: Every command can get and set attributes belonging to the object at which the command is targeted. If the command to be executed is one of REMOVE, REMOVE PARTITION, REMOVE COLLECTION, then the attributes should be set and got before the command is executed. The attribute preprocess stage checks if the current command belongs to this group and if so performs the get and set attribute operations.

3. Command Execution: During this stage, the command is actually executed at the target. Each command requires some set of mandatory parameters which either are embedded into the *service action specific field* of same CDB as the command (refer Table 40,41 [8]) or are sent as separate data PDUs. The command is translated into a file system equivalent and the corresponding system call is made with the required arguments.

4. Attribute Post-process: In this stage all the attributes that are affected by the execution of the command are updated. For example : an successful OSD WRITE operation should change all the attributes related to quota, timestamp etc. Another task that is performed in this phase is to process the set and get attribute portion of the CDB if the current command is not one of {REMOVE, REMOVE PARTITION, REMOVE COLLECTION}

5. Sense data collection: For each session, we maintain a *sense* data structure that tracks the execution status of the commands through the above stages. This data structure contains information on the partition ID, user object ID involved, function command bits (refer Table 34 in [8]), sense key and additional sense code (ASC) to track cause of error. Whenever an error occurs during any stage, we update this data structure to capture the cause of the error. In this final stage, we encapsulate the *sense* data structure into a PDU as defined in [8] and return it to the initiator. This additional information provides the initiator more knowledge to react to unforeseen circumstances.

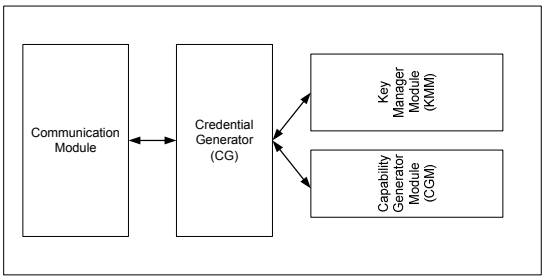

Figure 6. Security Manager

3.5. Security

Security is one of the fundamental features of OSD. In order to access an object, a user must acquire cryptographically secure credentials from the security manager. Each credential contains a capability that identifies a specific object, the list of operations that may be performed on that object, and a capability key that is used to securely communicate with the OBSD. Before granting access to any object, each OSD checks whether the requestor has the appropriate credential.

Our implementation contains a client and a server security module to implement the security mechanisms between the client and the OBSD as described by the standard. In addition, we have also implemented a preliminary security manager that can hand-out capabilities to users and perform some preliminary key management tasks. The current implementation assumes that the communication link between the user and the security manger is secure. The security manager does not authenticate users; it assumes that users are already authenticated using any of the standard mechanisms such as Kerberos [14].

The Security Manager As depicted in figure 6, the security manager consists of four modules, namely, the communication module, the credential generator (CG), the key manager module (KMM), and the capability generator module (CGM). The communication module is responsible to handle network communications. The CG is responsible to generate cryptographically secure credentials using the keys supplied by the KMM and the access control information (capabilities) supplied by the CGM.

In order to acquire a capability, a user should send a capability request to the security manager. The communications module transfers the request to CG. The CG queries the CGM to acquire capability for the requested object. The CGM maintains a MySQL [5] database that contains access control information per object. A client has to supply her UNIX UID and GID along with the requested OID to the CGM. Using this information, the CGM creates the capability for that object and returns it to the CG. Upon receipt of the capability from the CGM, the CG acquires appro-

CPU	Two Intel XEON 2.0GHz w/ HT
Memory	512MB DDR DIMM
SCSI interface	Ultra160 SCSI (160MBps)
HDD	Hitachi Ultrastar, 73.5 G, 10,000 RPM
Average seek time	4.7 ms
NIC	Intel Pro/1000MF

Table 2. Configuration of OSD Target and Client

priate key from the KMM to generate a cryptographically secure credential for that object.

The KMM is responsible to manipulate and generate appropriate keys. It maintains a repository of keys that are shared with the OBSDs. It determines the type of key to be used based on the command requested by the user. For example, if SET KEY command is desired to change a certain partition key, then that partition's root keys are acquired. The key manager then returns the appropriate keys to the CG. The CG then generates the credential and transfers it to the user.

The Client-Server Modules Whenever a user wants to access an object, the client side security module transparently contacts the security manager and obtains a credential for the requested object. After receiving the credential, the client cryptographically secures the commands and sends to the OBSD. According to the T10 standard the client can choose one of the following four security methods to securely communicate with the OBSD: NOSEC, CAPKEY, CMDRSP, or ALLDATA. Our current implementation supports NOSEC, CAPKEY, and CMDRSP methods.

Readers should recall that each OSD shares a set of keys with the security manager. The security manager is responsible to exchange these keys with each OBSD. The OSD standard mandates SET KEY and SET MASTER KEY commands for this purpose. Of these, SET KEY is currently supported in our implementation.

4. Performance Evaluation

In this section, we evaluate the performance of our OSD reference implementation. We perform experiments to evaluate the performance of each component in our implementation. First we describe the testbed that was used in our experiments and then explain each experiment in detail.

Table 2 shows the configuration of the machines that we used for the OSD target and client. The embedded gigabit ethernet NIC on the server and client connects them to a Cisco Catalyst 4000 gigabit ethernet switch. We believe that such a system makes fair emulation of future intelligent

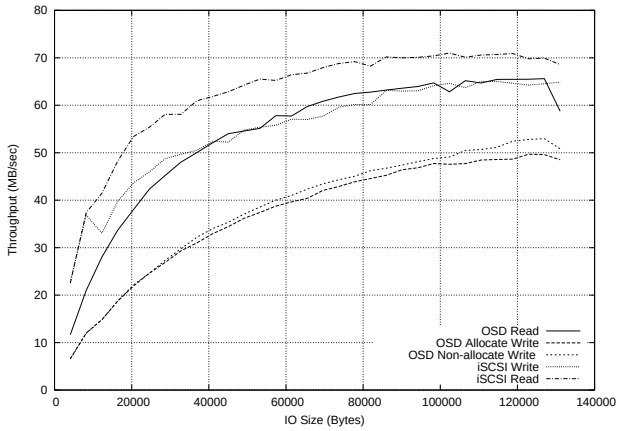

Figure 7. Raw performance comparison of OSD and iSCSI

storage devices. In each experiment, we compare the performance of the OSD client and target with those of a iSCSI based SAN storage system and a NFS based NAS device. For all the above storage configurations, the same client-server machine combination was used, same disk partitions were used at the target to ensure the disk performance remains constant across all configurations. We used the Intel iSCSI initiator and target to set up the iSCSI configuration. Loading the initator driver creates a SCSI device on the client. iSCSI performance is measured on a ext2 filesystem constructed on this SCSI device. For the NAS configuration, we set up the NFS daemon on the target and exported a directory in the common test partition on the target.

In the first experiment, we measure the raw read, write performance of the OSD target and compare it with the iSCSI configuration. The motive of this experiment is to measure the performance of the storage target without the overhead of the filesystem and effects of client caching. In this experiment, we write/read a 4MB file with multiple transfer sizes and measure the throughput. Figure 7 shows the results of this experiment. The iSCSI write operation writes a series of blocks, each of size equal to the *transfer size* on the block device. For the OSD case, we have two variations of the write operation: Allocate Write and Non-Allocate Write. The allocate write creates a user object at the target and allocates space at the target (by appending to existing object) for every write operation. The Non-Allocate Write, on the other hand, just re-writes over the pre-allocated blocks reserved by the Allocate Write. So the allocate write has the extra overhead of finding unused blocks on disk and updating the filesystem data structures at the target. This overhead explains the slightly degraded performance in the allocate write case when compared to the non allocate write. The semantics of the iSCSI write operation is closest to that of the OSD Non-Allocate Write. In general, the performance of an OSD operation is

Table 3. Filesystem Throughput (MB/s)

Operation	OSDfs				NFS				iSCSI			
	Maximum	Minimum	Average	Std. Dev	Maximum	Minimum	Average	Std. Dev	Maximum	Minimum	Average	Std. Dev
READ	15.47	11.9	14.51	1.033	94.80	26.49	66.73	16.84	76.44	33.49	57.46	11.73
WRITE	7.51	6.822	7.34	0.087	20.43	2.716	16.41	4.42	43.112	4.97	27.895	12.065

Command	Latency (μsec)	
	CAPKEY	CMDRSP
CREATE PARTITION	15040	14797
CREATE	3745	4024
LIST	1928	1970
LIST ROOT	1713	1896
SET ATTRIBUTE	1689	1950
WRITE	2141	2306
APPEND	2085	2263
READ	1654	1863
GET ATTRIBUTE	1677	1902
REMOVE	8387	8616
REMOVE PARTITION	10046	10178

Table 4. Per operation Latency

lower than that of the corresponding iSCSI operation due to the overhead imposed by the security mechanisms, context switches and filesystem overhead at the target. Also it can be noted that, for both iSCSI and OSD, higher transfer sizes yield better throughput. This is because the overall overhead of constructing PDUs is lesser for higher transfer sizes when compared to lower transfer sizes. The throughput saturates before reaching the network bandwidth limit of 1Gbps, indicating performance bottlenecks in both the iSCSI driver and OSD target implementations.

In the second experiment, we measure the latency of some OSD commands as seen by the OSD client. We instrumented the raw performance measurement tool used in the first experiment to gather the latency results. Table 4 reports the measured latencies for the two implemented security methods: CAPKEY and CMDRSP. First of all, we observe that CREATE PARTITION and REMOVE PARTITION have latencies which are an order of magnitude higher than other commands that operate on partitions (like LIST, GET ATTRIBUTE). These high numbers can be explained by breaking up command execution into the various events that happen. For a CREATE PARTITION, the target first creates a directory in the filesystem namespace and then creates one file for each mandatory attribute for the partition. 42 files were created in all for this purpose. Similarly the DELETE PARTITION command first deletes all the files associated with the partition attributes and then deletes the directory itself. This also explains why the CREATE and REMOVE commands have high latencies when compared to the other commands that operate on user objects. For the WRITE, APPEND and READ commands, 64 bytes of data were either written or read. The latencies while using the NOSEC method were observed to be very similar to the ones reported for CMDRSP and CAPKEY. This is because the additional cryptographic overhead[2] incurred in CMDRSP and CAPKEY is negligible when compared to the network latency. In other words, the network latency is the dominant factor in the overall observed latency.

In the third experiment, we study the performance of OSD filesystem using the IOZone filesystem benchmark [3]. Table 3 shows the throughput for the READ and WRITE operations for osdfs, NFS and ext2 over iSCSI. This table shows that the performance of the OSDfs is significantly lower than that of NFS and iSCSI for both READ and WRITE operations. We also observe (not shown in the table) that the earlier trend that we observed in Figure 7, where throughput increases with the transfer size, is no longer seen and the throughput surface is almost flat. The only difference in setup between Experiments 1 and 3 is that osdfs was introduced in the third experiment. So we can deduce that the overhead introduced by the OSD filesystem is substantially high enough to mask the effect of transfer sizes. Improving osdfs is one of the main issues that we identify as future work.

5. Related Work

In this section, we present other efforts geared towards building the reference implementation for the OSD T-10 spec. In the *Object Store* project at IBM Haifa Labs, a T-10 compliant OSD initiator [2] and a OSD Simulator [1] have been developed. A recent paper [18], from the same group, discusses tools and methodologies to test OSDs for correctness and compliance with the T10 standard. A simple script language is defined which is used to construct both sequential and parallel workloads. A tester program reads the input script file and generates OSD commands to the target and verifies the correctness of the result. Our work can complement IBM's implementation in providing a more usable interface to applications through our file system: osdfs. Also our implementation provides complete reporting of sense data back to the initiator.

[2]With openssl, it takes 3.49 μsec to perform a HMAC operation for a block size of 256 bytes.

6. Conclusion and Future Work

In this paper we presented our experiences with the implementation of the SCSI OSD (T-10) standard. Design and implementation issues at the target, client file system and the security manager were discussed and performance analysis results also presented. The forte of our implementation does not lie in the performance but rather in the completeness of the implementation and the usability of the system as a whole.

We have identified three broad areas where substantial amount of work remains to be done. The first area, namely *feature additions*, focuses on adding some extra capabilities and functionalities to further demonstrate the advantages of the object based technology. First task in this area is implement the remaining OSD commands (PERFORM SCSI COMMAND, PERFORM TASK MANAGEMENT FUNCTION, SET MASTER KEY). The second task in this category is to design and build a metadata server (MDS). A dedicated metadata server is essential in separating the data and control path. The MDS will also perform global namespace management, concurrency control and object location tracking. [9] presents a relevant technique to map objects in a hierarchical namespace to a flat namespace. We also want to test interoperability of our implementation with the IBM initiator [2].

The second area of future work revolves around *performance improvement* of the current implementation. The performance of our target and the client implementation needs to be improved to fully realize the true benefits of object based storage systems. We plan to optimize the target in two distinct phases. In the first phase, the filesystem abstraction of storage will be replaced by a compact object-based, flat namespace storage manager. [19] presents a filesystem based on a flat, object based namespace. Techniques to efficiently store and retrieve extended attributes will be investigated and implemented. In the second phase, we plan to further optimize the target code to have it execute in minimal environments like RAID controller boxes.

Infusing Intelligence into the storage device is the third area that we have identified to channel our efforts into in the future. The object abstraction and extended attributes are excellent mechanisms to convey additional information to the storage device. One such example is providing QoS requirements of the objects [15]. How to use this additional information, to benefit the system, is termed as the storage intelligence. For example, [16] shows how QoS requirements, provided as service level agreements, can be used to schedule requests within the storage device. We want to investigate what knowledge can be provided to the storage and then design mechanisms that can exploit such additional knowledge to improve the performance of the storage device.

We also want to explore how applications in the real world, like data warehouses for Medical Information Systems, can benefit from intelligent storage. We are currently working with Mayo Clinic (Rochester) on building a system that can enable seamless data-mining across structured and unstructured data for medical research. We are investigating on building integrated indexing and search mechanisms at the storage device and layout optimizations to match the characteristics of the data. These algorithms would eventually be layered over our OSD implementation to demonstrate the capabilities of intelligent storage.

Acknowledgements

We would like to thank Mike Mesnier for providing us with the initial implementation of the reference model. We would also like to thank Nagapramod Mandagere and Biplob Debnath for testing our implementation for compliance with the standard. This work was supported by the following companies through DTC Intelligent Storage Consortium (DISC) : Sun Microsystems, Symantec, Engenio/LSI Logic, ETRI/Korea and ITRI/Taiwan. We would also like to thank the anonymous reviewers for their helpful comments.

References

[1] IBM object storage device simulator for linux. `http://www.alphaworks.ibm.com/tech/osdsim/`.

[2] IBM OSD initiator. `http://sourceforge.net/projects/osd-initiator`.

[3] Iozone filesystem benchmark. `http://www.iozone.org`.

[4] Lustre. `http://www.lustre.org`.

[5] MySQL Version 5.0. `http://dev.mysql.com/`.

[6] Panasas. `http://www.panasas.com`.

[7] *SCSI Architecture Model-3 (SAM-3)*. Project T10/1561-D, Revision 14. T10 Technical Committee NCITS, September 2004.

[8] *SCSI Object-Based Storage Device Commands -2 (OSD-2)*. Project T10/1721-D, Revision 0. T10 Technical Committee NCITS, October 2004.

[9] S. Brandt, L. Xue, E. Miller, and D. Long. Efficient metadata management in large distributed file systems. In *Twentieth IEEE/Eleventh NASA Goddard Conference on Mass Storage Systems and Technologies*, April 2003.

[10] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-hartman. *Linux Device Drivers*. O'Reilly, 3rd edition, Feburary 2005.

[11] Michael Factor, David Nagle, Dalit Naor, Eric Reidel, and Julian Satran. The OSD security protocol. In *Proceeding of 3rd International IEEE Security in Storage Workshop*, December 2005.

[12] Gibson G.A., Nagle D.F., Amiri K., Chang F.W., Feinberg E.M, Gobioff H., Lee C., Ozceri B., Riedel E., and Rochberg D. A case for network-attached secure disks. In *CMU SCS Technical Report CMU-CS-96-142*, September 1996.

[13] Dingshan He and David Du. An efficient data sharing scheme for iscsi-based file systems. In *Proceeding of 12th NASA Goddard, 21st IEEE Conference on Mass Storage Systems and Technologies*, April 2004.

[14] J. Linn. The kerberos version 5 GSS-API mechanism. RFC 1964, June 1996.

[15] Yingping Lu, David Du, and Tom Ruwart. Qos provisioning framework for an osd-based storage system. In *Proceeding of 13th NASA Goddard, 22nd IEEE Conference on Mass Storage Systems and Technologies*, April 2005.

[16] C. Lumb, A. Merchant, and G. Alvarez. Facade: Virtual storage devices with performance guarantees. In *Usenix conference on File and Storage Technologies (FAST)*, 2003.

[17] M. Mesnier, G. Ganger, and E. Riedel. Object-based storage. *IEEE Communications Magazine*, 41(8):84–90, August 2003.

[18] P. Reshef, O. Rodeh, A. Shafrir, A. Wolman, and E. Yaffe. Benchmarking and testing osd for correctness and compliance. In *In Proceedings of the IBM Verification Conference (Software Testing Track)*, November 2005.

[19] F. Wang, S. Brandt, E. Miller, and D. Long. OBFS: a file system for object-based storage devices. In *Proceeding of 12th NASA Goddard, 21st IEEE Conference on Mass Storage Systems and Technologies*, April 2004.