Design and Implementation of an OSD System and Prefetching Models

by

Riyaz Shiraguppi

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

June 2008

Design and Implementation of an OSD System and Prefetching Models

A Thesis

Submitted to the Department of Computer Science & Engineering

of Indian Institute of Technology, Kanpur

in Partial Fulfillment of the Requirements for the degree of

Master of Technology

by

Riyaz Shiraguppi

Certificate

It is certified that the work contained in the thesis titled ***"Design and Implementation of an OSD System and Prefetching Models"*** by ***Riyaz Shiraguppi*** has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

June 2008

Dr. Rajat Moona

Department of Computer Science & Engineering,

Indian Institute of Technology,

Kanpur-208016.

Abstract

Object based device (OSD) technology [2], built on top of SAN [6], provides solution to problems of security and scalability associated with the block level storage. An intelligent OSD device exports object interface and provides access to object data only if the requester possesses the required security token. Security tokens are issued by the OSD manager based on security and access control policies. In this thesis, our aim had been to build a high performance OSD system using prefetching techniques. We have implemented a skeleton of OSD system consisting of target and initiator side components. Our OSD target implementation is independent of underlying iSCSI driver software. At the initiator side, we have implemented an OSD manager and an OSD client. We have designed Prefetching Models to hide the costs of communication between OSD client and manager, and to achieve high performance. We introduced graph based model for access analysis at client side and use this information for prefetching. Our server side model is based on prediction of objects required in future by grouping related objects. We have tested the efficiency of this model on file-based as well as on block-based object controllers. The prefetching model is effective even if only half of the prefetched data is actually used. In order to build the server assisted prefetching, we built support of collection objects in IBM Controller Project [22] where the target is a block based object store. Our own implementation of the target which is file-based object store is also T-10 standard [17] compliant and has the support for collection objects.

Acknowledgements

It is a pleasure to thank the many people who helped me throughout.

I would like to express my gratitude and thank my supervisor, Dr. Rajat Moona. This work would not have been possible without his support and guidance. Despite his busy schedule, he used to make out time for the weekly meetings, which helped in continuous progress of the work done. His encouragement, perception, advice, and lots of good ideas have helped me throughout.

I would like to thanks Mr. Hemant Khachane, my colleague from Calsoft who always helped me in various difficulties and encouraged me. I am also thankful to all my MTech friends for constantly encouraging and supporting me throughout my stay here at IIT Kanpur.

I cannot end without thanking my parents, Mr. Moulasab Shiraguppi and Mrs. Ruksana Shiraguppi, and my sister Asama. Their encouragement and love have always helped me recover from difficult phases during my stay and study here at IIT Kanpur. I wish to thank my entire extended family for providing a loving environment for me. Their constant affection and encouragement has helped me achieve my goals.

Contents

List of Figures

Chapter 1

Introduction

Storage technologies can be classified into there major categories as Direct Attached storage (DAS) [5], Network Attached storage (NAS) [25], Storage Area networks (SAN) [3].

DAS is computer storage directly attached to server. NAS is file-based storage architecture for hosts connected to an IP network or a LAN. The typical functionality of a NAS requires file servers to manage metadata, which describes how files are stored on the devices. NAS advantages are in terms of data sharing and secure access through server.

SAN technology is intended to replace bus-based architecture with a high-speed switched fabric. SAN is block based architecture where client can have direct access to disks on switch fabric network.

The object based device (OSD) system [1], as a SAN technology, combines the advantages of high-speed, direct-access SANs, and the data sharing and secure access capabilities of NAS.

1.1 Motivation

SAN is a dedicated, high performance storage network that transfers data between servers and storage devices, usually separate from the local area network. DAS and NAS suffer from the performance issues such as difficulty of data management in a network environment, scalability and performance bottlenecks of a single fileserver etc. The SANs provide a promising solution to these problems.

Main advantage of a SAN is in its ability to transfer large data in units of blocks, which is important for bandwidth intensive applications. In a SAN, it is possible to have parallelization of data and control paths where in multiple machines can access and process data and metadata simultaneously. This imposes the need of security and access control to the device. SAN security relies on coarse grain techniques involving LUN masking [26] and zoning [27].

Object based storage disk (OSD) [1] provides an added abstraction of objects which relates to the application data. OSD design is intended to embed storage intelligence into the disk. OSD does not expose low level information such as disk geometry, and implements all layout optimizations internally utilizing higher-level information that can be provided through the object interface. A security domain is associated with each object. Therefore a client trying to access object from the OSD device needs to acquire a security token. The security token also includes a list of permitted operations allowed for the requester on the given object. A security token check is performed by the OSD for each request. It therefore prevents both the accidental access for undesirable requests and the malicious access for unauthorized requests.

Security token handling is facilitated by an OSD manager. OSD manager is responsible for managing policy decisions for objects and granting security tokens. The security token also carries a message integrity code (MIC) to prevent client from modifying token. The security token along with the corresponding message integrity code is termed as Credentials.

OSD system introduces additional overhead of acquiring Credentials before actual access to data which can shadow its security advantages. This can be improved by implementation of intelligent mechanisms for prefetching of Credentials.

1.2 Scope of This Work

In this thesis, we describe design and implementation of our OSD system. This includes target side implementation consisting of OSD simulator, and initiator side implementation consisting of an OSD client and an OSD manager.

OSD Simulator is a prototype implementation of OSD device compliant with T10 standard [17] for iSCSI based OSD protocol. We built OSD client and OSD Manager side components on the top of IBM initiator [11].

We addressed issue of Credential traffic overheads in the OSD System. We have designed a graph based model for implementation of client initiated prefetching at OSD client.

We also designed and implemented server initiated prefetching model at OSD manager. We tested our server initiated prefetching model on our OSD simulator and a real block based object controller which is part of IBM ObjectStore project [22]. We had extended IBM Objectstore project for support of collection objects [1], a mechanism that may be used by a client to pass on the relationship of objects to the server.

1.3 Organization of Thesis

Rest of the thesis is organized as follows. In chapter 2, we survey OSD technologies and describe technical aspects of the OSD system. In chapter 3, we present design and implementation of our OSD target simulator. In chapter 4, we describe OSD initiator side components where we present design and implementation of the OSD client and the OSD manager. In chapter 5, we discuss prefetching techniques and need of credentials prefetching in the OSD system. In chapter 6, we discuss graph based model for client-initiated prefetching. We discuss OSD client design based on this model. In chapter 7, we discuss prefetching model for server-initiated prefetching. We present design and implementation of OSD manager based on this model. Results show that this prefetching model is efficient and improves the performance. In chapter 8, we discuss IBM object controller project and our contribution to this project for support of collection objects. Finally, we conclude in chapter 9.

Chapter 2

OSD Technology Survey

In this chapter, we provide an overview of the OSD System. We also describe OSD protocol and the extensions required in the traditional protocols to support the OSD protocols.

2.1 Overview of OSD System

2.1.1 OSD System Components

An OSD system (figure 2.1) typically consists of the following components.

2.1.1.1 OSD Client

An OSD client runs applications which access data from an OSD device. It places a request for the security token to the OSD manager. After acquiring the security token, it integrates the OSD service request with the security token and sends it to the OSD device server. The OSD request token is acquired much less frequently as compared to the data.

2.1.1.2 OSD Manager

An OSD manager is responsible for the management of policy of access to objects in an OSD and issuance of security tokens to the OSD client. It has two sub-components, namely, policy manager and security manager.

The policy manager maintains database of policy information (akin access control information in a regular file systems) about objects. A request to acquire the security token for access to an object is granted based on the policy information associated with the object. The security manager shares secret keys with the OSD device server which is used for computation and verification of the integrity of security tokens.

2.1.1.3 OSD Device Server

An OSD device server is responsible for processing the OSD commands after appropriate authentication of the security token supplied by the OSD client. It supplies object data and metadata to the client as a response to the request. It also manages the order of serving the requests for an efficient system.

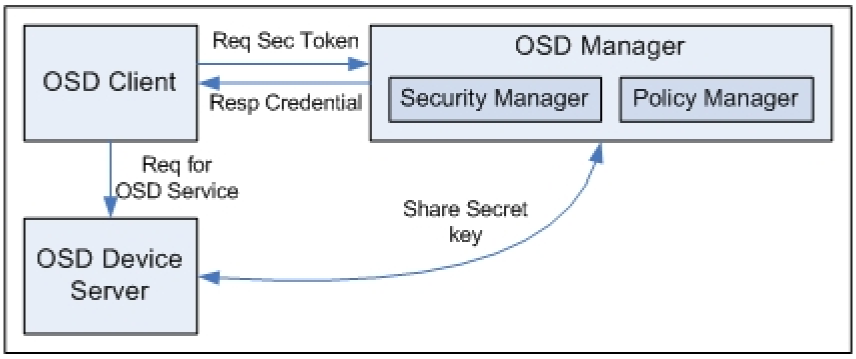


Figure 2.1: OSD System Components

2.1.2 Advantages of OSD Technology

An OSD system has certain advantages over traditional block based storage systems.

Improved Security

In OSD system, request authorization is performed directly by the storage device. This can prevent problems of accidental access and malicious access due to unauthorized/undesired requests faced in the block based SANs. Security mechanism in the OSD device allows the individual disc drive to validate each read and write request.

Improved Reliability

While existing block devices require external support for replication manager and reliability manager, OSD can have these functionalities embedded in the device. Depending on the type and importance of the object, storage component of OSD may implement replication, error check mechanism for the object and manage in-built reliability manager.

Improved storage management

Storage component of a file system is moved to the device server in an OSD based storage system. The OSD device is self managed independent unit and can manage data in the object for performance, reliability or other options in a dynamic and flexible manner.

Improved Scalability

Object based storage provides scalability in terms of storage space and metadata management. OSD device handles the management of metadata. All or part of metadata can be associated and stored directly with data objects and automatically carried between layers of the storage architecture and across devices. This allows metadata layers to be handled in a manner that simplifies the storage system and improves scalability.

2.1.3 OSD Device Components

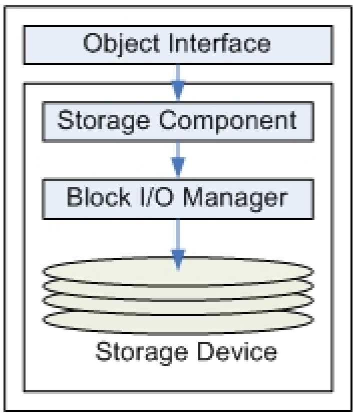


Figure 2.2: OSD Device

The OSD device consists of several components as shown in figure 2.2.

Object Interface

The Object interface is visible to the external environment. Data on the OSD device is accessed through commands that perform operations on particular OSD object. Command parameters specifies Object ID of the object, type of data access operation and range of data for performing data access operation.

Storage Component

This component is responsible for implementing object storage intelligence into the OSD disk. The storage management component is focused on mapping logical object constructs (e.g., files, database entries or any other kind of object data) to the physical organization of the storage media. The physical organization of data may be a block-oriented organization, file-system oriented organization or just a raw disk organization. As an example, our OSD simulator stores data in a file-system based organization while the IBM object-controller [40] stores data in a block oriented organization.

In addition to mapping data, the storage management component also maintains other information about the OSD objects (e.g., size, usage quotas, associated username) and stores in the form of attributes.

Block I/O Manager

This component is similar to the block I/O manager in the traditional disks. It is responsible for translating commands from the storage component to the block commands. For example, if the storage component architecture is that of a file system, this component performs the file system to disk block mapping.

Storage Device

This is the physical media where data is stored. As an example, the storage device can be a SCSI or an IDE hard-drive that talks in terms of LBA addressing.

2.1.4 Technology Issues

In an OSD system, device storage intelligence is pushed to the disk. Embedding this sort of intelligence makes OSD design costly. OSD technology needs implementation of security algorithms and management of object abstraction embedded on the device.

Currently, OSD devices with these embedded functionalities are not present in the market. OSD protocol is therefore implemented using a simulator for an OSD device. Simulator exports the object interface and maps object commands to the block commands which are implemented on the block devices. We implemented one such simulator that maps the objects to local files. We shall discuss design and implementation of our simulator in chapter 3.

Another issue with an OSD system is concerned with the transition from the block based system. Most applications today use the file system based abstraction and work on the block based storage. It would be a while before OSD based applications start appearing. Till that time, the OSD must integrate the block based architecture as well. SCSI based OSD model (described next) makes this transition easy.

2.2 OSD-SCSI Model

T10 community [17] in collaboration with industry groups is engaged in standardization of OSD protocol. Most standards are developed during 1999 through 2004. OSD standards are released as OSD specification documents. OSD version 1 had undergone 10 revisions. Latest draft is OSD version 2 revision 2. This standard defines a logical unit model for OSD logical units and SCSI commands for communicating with OSD.

Under T-10 standardization, an OSD is implemented as a SCSI device. T-10 defines additional commands to be carried over SCSI protocol for handling OSD devices. The OSD-SCSI command set is designed to provide efficient operations of object based input/output. These commands manage the allocation, placement, and access of variable-size data-storage containers, called objects. Objects can contain whole operating system or application constructs such as a database table, a picture, a video or similar constructs.

iSCSI protocol over TCP/IP acts as a carrier for SCSI commands. The industry contemplation to shift SAN architectures to the IP-SAN technologies has been the driving force to adopt iSCSI for implementing object based transport across the network. The OSD implementations are typically built over iSCSI protocols.

2.2.1 OSD Objects

As defined by the T10 standard, an OSD logical unit contains following types of objects.

2.2.1.1 Root Object

Each OSD logical unit contains one root object. The root object contains a list of Partition-IDs for the partitions in the logical unit. The root object is the starting point for navigation of the structure on an OSD logical unit. The root object does not contain a read/write data area.

2.2.1.2 Partition Object

A partition object contains a set of collections and user objects that share common security requirements and attributes. Each partition contains a list of User_Object_IDs and Collection_Object_IDs belonging to that partition. A partition object does not contain a read/write data area.

2.2.1.3 User Object

A user OSD object contains end-user data (e.g., file or database data).

2.2.1.4 Collection Object

A collection object is used for fast indexing of user objects. It acts as container of related objects. For example a container object may be used to contain a list of database tables belonging to a single database. A collection object does not contain a read/write data area.

2.2.2 OSD Meta-data

OSD object meta-data is stored in the form of Object attribute pages associated with each object. OSD object attributes allow the association of metadata with an OSD object. Attributes are used to describe specific characteristics of an OSD object and are organized in the form of pages. Within each attribute page, attributes are identified by an attribute number. A tuple containing <page no,attr no> is used to access particular metadata information for an object. There are a few fixed attribute pages defined by the standard and additionally vendor specific pages can be added.

2.3 Related Work

Several researchers from academic community as well as from industry have worked in OSD technology.

NASD [7, 28, 29, 30, 31] project at CMU provided initial drive for the OSD technology. Their implementation includes the NASD drive prototype, the NASD-NFS file-manager and client APIs for accessing NASD.

University of Santacruz is working on Ceph [20, 32, 33, 34, 35, 36] project. Ceph is a distributed network file system designed for petabyte-scale storage systems using OSD. Data is distributed using CRUSH, a hash-like distribution function that allows any party to calculate (instead of looking up) the location of data. This project is an OSD related project but is not compliant to T-10 standard.

University of Minnesota is working on an OSD project [24, 37, 38, 39] in its DISC Consortium [21]. They have a reference implementation suite that consists of a security manager and a target that builds object abstraction over a file system.

Panasas [18] is an active participant of SNIA OSD technical work group. The Panasas ActiveScale File System (PanFS) integrates an object-based clustered architecture to orchestrate file activity across the Storage Cluster and manage system performance. The file system virtualizes data across all StorageBlades, and presents a single, cache coherent unified namespace.

Cluster File Systems, Inc. [46], acquired by Sun Microsystems, Inc. [47], has used object based storage to build high performance storage systems. Their product Lustre [19], is implementation of OSD system with an installable Linux file system and a metadata server (MDS) to create a file system that can be used by multiple servers.

IBM Haifa Lab is working on Object Store [22] project. They have prototype of a T10-compliant object store target, T10 complaint OSD initiator for Linux (for iSCSI and FC) and iSCSI target in software.

Intel's Open Storage Toolkit [23] contains reference implementation of OSD. It incorporates an OSD file system, an OSD initiator and an OSD target simulator. University of Minnesota has built their implementation on top of the Intel's open Storage toolkit.

Chapter 3

Design and Implementation of OSD Target

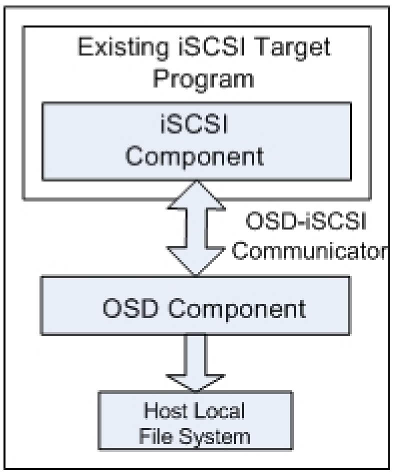


Figure 3.1: OSD Target Components

In this thesis, we designed and implemented an OSD Target. Our OSD Target consists of three major components: OSD component, iSCSI component, and an interface for communication between these components known as OSD-iSCSI communicator.

The OSD component implements the functionality of OSD device server and is independent of the underlying implementation of transport protocol such as iSCSI. As discussed in chapter 2, OSD component is implemented as a simulator for the OSD device. This is primarily because there is no hardware implementation of OSD device available in the market.

In our implementation of OSD component, we store data for the objects in files, a mechanism that greatly simplifies implementation since we used an underlying file system support. In our approach, objects are mapped onto hierarchical file namespace managed by the host file system. The OSD component can be implemented wholly as a kernel module, or as a user mode daemon. Our modular design permits the use of various iSCSI implementations by implementing OSD-iSCSI communicator module for each. The same idea can also be extended to support other IP storage technologies [49] in place of iSCSI.

The iSCSI component is implemented inside an existing iSCSI target program [48]. The iSCSI component provides support of OSD devices in iSCSI target program. The iSCSI component submits OSD commands to the OSD component using the OSD-iSCSI communicator.

The OSD-iSCSI communicator interfaces OSD and iSCSI components. Its implementation is specific to a chosen iSCSI component and the way the OSD component might be implemented (viz. user mode or kernel mode).

IITK OSD Simulator [16] is our implementation of OSD Target. We have implemented iSCSI component over the freeware iSCSI Enterprise Target (IET) [48]. We extended IET to support OSD devices. The OSD-iSCSI communicator is implemented using socket interface in our implementation.

3.1 OSD Component

The OSD component design in this thesis, is fairly generic. It has a modular structure as shown in figure 3.2.

The exporter module exports the OSD device as an iSCSI LUN. The OSD request server accepts connection requests from iSCSI component and passes them to the OSD task processor. The OSD task processor module implements objects as files on the local file system.

In our approach, there is a directory associated with each OSD LUN. This directory is part of local file system on the OSD target machine. Object data and meta-data information is stored in files created in the directory associated with the OSD LUN.

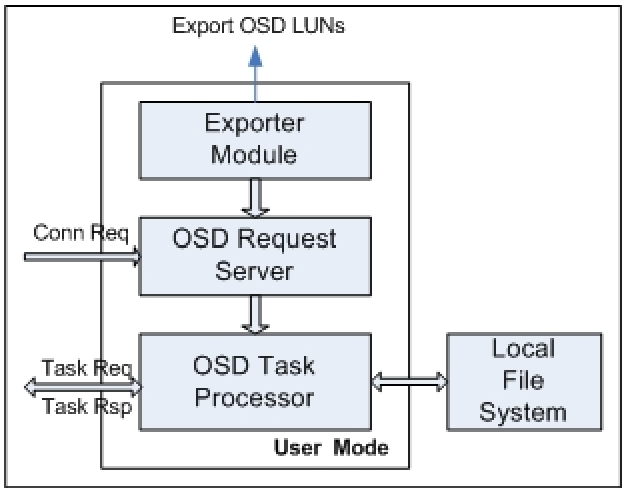


Figure 3.2: Modules of OSD Component

3.1.1 Exporter Module

OSD LUN to local directory name mapping is carried out through an OSD configuration file at the OSD target that includes the OSD LUN information. OSD LUN information includes OSD target name, LUN number, the local directory name for the OSD-LUN, the capacity (size) of the LUN and the OSD system ID. The Exporter module is responsible for associating a directory on the OSD target machine to the OSD LUN. It obtains this mapping by reading the configuration file. Exporter module is also responsible for communicating OSD LUN information to the iSCSI component.

Exporter Module scans entries in the configuration file to get information about the OSD LUN and updates the information maintained by the OSD request server.

3.1.2 OSD Request Server

OSD request server is responsible for initialization of OSD simulator module. It maintains a list of OSD devices available in the system. For each entry, the information maintained includes the local directory name, capacity and the OSD system ID for each OSD LUN as described in section 3.1.1. This information is used while processing an OSD command.

The OSD request server is responsible for initialization of the OSD configuration parameters such as maximum input/output data transfer length, get/set attribute buffer size, etc.

The OSD request server listens for OSD task processing requests from the OSD-iSCSI callback module (section 3.2.5). The iSCSI protocol defines SCSI command data blocks (CDB) which are used to carry OSD commands. Since the OSD requests are carried over the same iSCSI protocol that carries other SCSI requests, a task processing request can either be an OSD request or some other SCSI request. Depending upon the type of request, the OSD request server invokes either the OSD task processor or the task processing module that handles other SCSI requests.

3.1.3 OSD Task Processor

The OSD task processor is responsible for processing of OSD task processing requests that carry OSD commands. It accepts OSD commands from the iSCSI component through OSD request server and performs object operations which might translate to operations on the files of the local file system.

Various sub-units of OSD task processor are shown in figure 3.3.

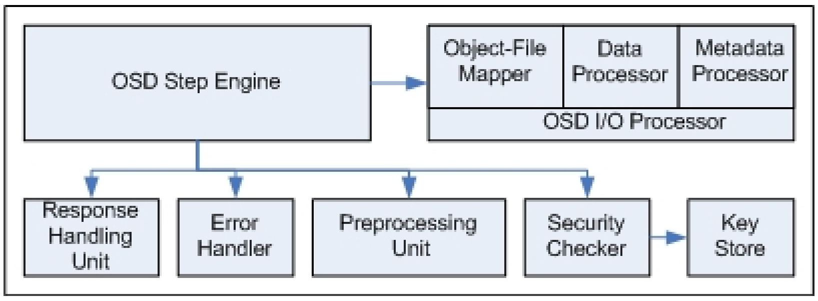


Figure 3.3: OSD Task Processor

3.1.3.1 OSD Step Engine

OSD step engine implements a state engine that is responsible for the management of various steps in processing of the OSD command. For each command, it manages a osd_req data structure. State field in this structure represents current step of processing of the corresponding command. For each step, it invokes corresponding module for processing of that step.

The steps of the OSD command processing are enumerated below.

Step 1: Security Handling

OSD step engine invokes security checker module for handling of security related processing of the given OSD command. The security related processing involves the checking of security tokens and validating the request.

Step 2: Preprocessing

After security handling step, OSD step engine invokes preprocessing module for preprocessing of the OSD command. The preprocessing involves extracting service action specific parameters and initializing fields of osd_req structure.

Step 3: Command Processing

OSD step engine invokes data processing unit of OSD I/O Processor (section 3.1.3.5) for processing of the OSD command.

Step 4: GET/SET Attributes Handling phase

OSD protocol allows the merging to GET/SET attribute operations along with any other command. In this step, parsing of the GET/SET command attributes and initialization of the corresponding data structures is performed. It invokes meta processing unit of the OSD I/O processor for processing of the GET/SET attributes.

Step 5: Command Response Phase

At the end of the command processing, the OSD engine invokes response unit to communicate results of the OSD command and the corresponding response to the iSCSI target for relaying it to the OSD client.

3.1.3.2 Security Checker

The security checker module is responsible for the implementation of security authentication at the OSD device. One of the fields in the OSD command header specifies security method to be used such as CAP_KEY (only request header is protected against tampering using message authentication code, MAC), ALL_DATA (in-out data along with request header is protected against tampering using message authentication code, MAC) etc. Security check is performed only when security method is a value other than NOSEC (to indicate no security).

OSD commands carry the object type and key version which relate to the security authentication. Depending on the object type and key version specified in the OSD command, the security checker extracts required key from the key store. A capability key is generated as per the OSD protocol using fields of the OSD command and this key. Integrity value (keyed hash or the MAC) is generated using capability key and compared with integrity value specified in the OSD CDB data structure.

3.1.3.3 Key Store

Key store is used to store various keys in the key hierarchy of OSD device. Key store implementation is done on the permanent device. Recently accessed keys are stored in the key cache. Key store implementation is described in section 3.4.3.

3.1.3.4 Pre-Processing Unit

OSD command needs to be pre-processed before actual execution to speedup the subsequent processing. In general, this unit identifies OSD device for which the given command is issued. It identifies service action and invokes corresponding parsing function to initialize command specific parameters.

As a part of pre-processing, various service action specific parameters from the OSD command descriptor block (CDB) are extracted and initialized to respective variables of osd_req structure.

3.1.3.5 OSD I/O Processor

This unit is responsible for the implementation of I/O operation for the OSD command. In our implementation, OSD objects are realized using the files on local file system. Therefore object operations are converted to the file operations and carried out on the file system local to the host. Our design provides compatibility to any host file system.

The OSD I/O processor carries out the object I/O using the following steps.

1. Object-File mapping

   This operation is responsible for identifying the file corresponding to the selected object. In general, object data and metadata information is stored in two separate files. Object to file mapping is further described in section 3.4.2.

2. Data Processing operation

   This operation implements OSD data handling. It involves getting data file corresponding to the given OSD object using the object to file mapping and implements data transfer operation by reading/writing the identified file.

3. Metadata Processing operation

   This operation implements OSD meta-data handling. The operation involves getting the filename corresponding to the metadata for the given OSD object using object to file mapping. As mentioned in the section 2.2.2, object metadata information is stored in terms of attribute pages. Object metadata operations are specified in terms of <attribute page, attribute no>. For example, "object creation time" attribute is stored as attribute #1 in page #3. Metadata operation is performed by calculating offset of the information corresponding to the attribute in the object metadata file.

3.1.3.6 Error Handler

Error handler is invoked whenever an error condition occurs during any step of the OSD command processing. It provides error description according to the SCSI error handling semantics.

3.1.3.7 Response Handling Unit

The response handling unit is responsible for sending response for the OSD command processing to the iSCSI module. Upon successful completion of the OSD command the response data (if any), along with the status to indicate success, is sent. Otherwise error handler is invoked to get error description and the status to indicate error is sent.

3.2 iSCSI Component

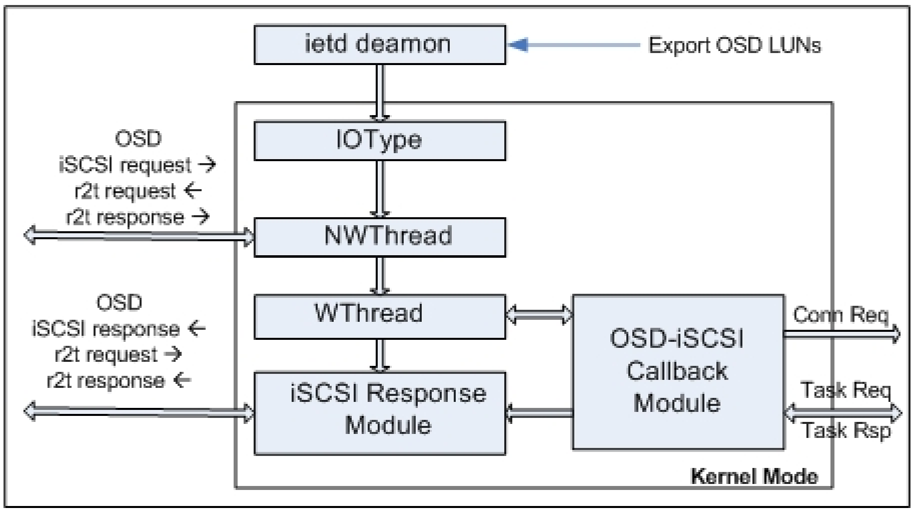


Figure 3.4: OSD-aware IET

We have implemented iSCSI component inside iSCSI Enterprise Target (IET)[48]. The IET is not OSD aware. We modified existing modules in IET and added new modules to make it OSD aware. We describe the implementation of IET in short and illustrate our enhancements to make it OSD aware.

Various modules of OSD aware IET are shown in figure 3.4. Out of these modules shown in figure 3.4, we modified existing IOType, WThread, NThread and iSCSI Response Modules. Additionally, we introduced OSD-iSCSI callback module which communicates with OSD component (section 3.1).

IET is implemented for GNU/Linux operating system and it consists of user mode and kernel Mode modules which communicate using netlink socket interface [52].

3.2.1 Ietd Daemon

The ietd daemon is iSCSI target program management utility. It is used to configure parameters such as iSCSI target port, uid, gid, etc. The ietd daemon takes entries from iSCSI configuration file "/etc/ietd.conf" and submits it to the kernel mode daemon. Exporter module of the OSD Component (section 3.1.1) makes entry for OSD LUN in the iSCSI configuration file and ietd daemon submits this entry to the IOType module.

3.2.2 IOType

This module is responsible for the management of iotype for the given device. The iotype for a device defines how operations are to be performed on the given device. For example, the most commonly used iotype in IET is “file-io”. The file-io is used to perform block operations on the device using generic file read/write functions.

Device iotypes can be registered with this module. Defining a new iotype requires one to implement and register new callback functions for that iotype. These functions are invoked while performing operations on the device of the new iotype.

The ietd daemon communicates with IOType module and passes configuration parameters read from the configuration file. For each entry in the configuration file, ietd daemon submits information containing target name, LUN, device iotype and path. IOType modules creates iSCSI LUN for each entry and initialize its access methods depending on its iotype. This iSCSI LUN is then exported to the initiator.

In our enhancements, we defined a new iotype as “osd-io” in this module. The corresponding callback functions for the osd-io device type are implemented by OSD-iSCSI callback module.

3.2.3 NWThread

This module listens for iSCSI request from the iSCSI initiator. It accepts iSCSI request, initializes iSCSI command structure and the corresponding LUN parameters, and allocates data buffers.

In our enhancements, we extended this module to support OSD commands. This enhancement includes accepting OSD commands and the corresponding OSD SCSI additional header segment (AHS) data.

3.2.4 WThread

This module is responsible for processing of the iSCSI Command. Commands are classified into two classes – iotype specific and iotype independent.

The iotype specific commands are implemented by invoking respective callback function registered for the given iotype. The iotype independent are implemented through generic functions.

We enhanced this module to support OSD commands. Our enhancements include passing control to the callback function for processing OSD commands.

3.2.5 OSD- iSCSI Callback Module

This module implements callback functions which are responsible for processing OSD commands. It communicates with the OSD component using OSD-iSCSI communicator. It first contacts OSD request server (section 3.1.2) for processing the given OSD request and then sends OSD task request to the OSD task processor. The task request comprises of OSD SCSI command header fields, OSD LUN information and data if any. In response, it receives OSD response status, OSD response data and OSD sense data in case of an error.

3.2.6 iSCSI Response Module

After processing of the iSCSI command, WThread module passes the output, or the error, to the iSCSI response module. Depending on the status of the command, it generates the iSCSI response. In case of an error while processing, the iSCSI response module sends the SCSI sense data depending on the type of error. We enhanced this module to support OSD commands. Our enhancements include sending appropriate sense data for OSD errors.

3.3 Design Features

Exporting a directory as OSD device

As mentioned earlier, we introduced “osd-io” as a new device type in addition to already supported other device types. A directory at the OSD target stores data corresponding to an OSD LUN. Files in this directory represent the object data and metadata as explained later.

Separation of iSCSI-OSD module and OSD simulator Module

Unlike other open source implementations, where the OSD target implementation is a mix of OSD code and iSCSI target code, we have maintained a clear separation between OSD and iSCSI components. Two components communicate with each other using the OSD-iSCSI communicator.

Compatibility

Since a clean separation is maintained between OSD and transport communicator, OSD component is compatible with any IP SAN technology used as the transport component. It can be integrated with such other technologies by implementing a communicator similar to the OSD-iSCSI communicator for the corresponding technology.

Support for Multiple OSD devices

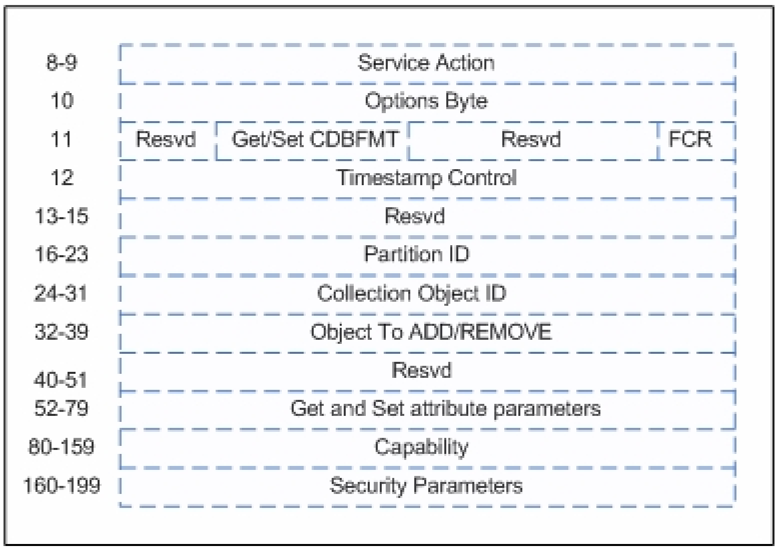
In several other implementations of the OSD targets [22,23], it is required to start new instance of iSCSI target for each OSD device. In our implementation, a single instance of the IET iSCSI target allows exporting any number of OSD devices. This is achieved by making several entries in the configuration file.

T10 Standard Compliance

The OSD component of our implementation accepts OSD commands over the socket interface and implements them. Our implementation is compliant to T10 OSD-2 specification rev0 [1].

Support for Collection Objects

Our implementation of the OSD target provides support for collection objects, as defined by the T10 standard. We support the OSD collection commands LIST COLLECTION, CREATE COLLECTION and REMOVE COLLECTION. In addition to these, we introduced two more commands for ease in accessing of the collection objects (These commands are known as ADD_TO_COLLECTION and REMOVE_FROM_COLLECTION). Format of these commands is similar to that of REMOVE COLLECTION as defined by the OSD-2 T10 standard with an additional field OBJECT_TO_ADD/REMO at offset 32 as shown in figure 3.5.



8-9	Service Action			
10	Options Byte			
11	Resvd	Get/Set CDBFMT	Resvd	FCR
12	Timestamp Control			
13-15	Resvd			
16-23	Partition ID			
24-31	Collection Object ID			
32-39	Object To ADD/REMOVE			
40-51	Resvd			
52-79	Get and Set attribute parameters			
80-159	Capability			
160-199	Security Parameters			

Figure 3.5: Format of ADD/REMOVE from Collection Command

3.4 Implementation

3.4.1 Open Source Website

Our OSD target is available with open-source licensed under GNU GPL [53]. Currently, version 1.0 is available for download through sourceforge site.

Project Home page: *http://iitk-osd-sim.sourceforge.net/*

Project Open-Soure Site: *http://sourceforge.net/projects/iitk-osd-sim/*

3.4.2 Object Storage Hierarchy

In our implementation, OSD object operations are implemented on the files of the file system at the OSD simulator. The modified IET exports a directory (osd_base) as OSD device.

There are two files associated with each object,

- _data : stores data information for the object.
- _attr : stores metadata information for the object as defined by the T10 standard.

OSD object files hierarchy is shown in figure 3.6. There are three kinds of objects for which the filenames are described below. In these names, the osd_base is the directory exported as an OSD LUN by the IET.

The files for root object data and metadata are stored under the osd_base directory itself. Data file is named as root_data while the metadata file is named as root_attr.

The partition objects are stored under a sub-directory identified by the partition id. In addition to this, we also store partition data and attributes in separate files. The partition zero data and metadata are stored under the osd_base directory as 0x0_data and 0x0_attr. Other partitions are stored as osd_base/part_id_attr and osd_base/part_id_data. For example, the partition 0x10001 information will be under osd_base/0x10001_data and osd_base/0x10001_attr.

The user and collection objects are stored under sub-directory of partition id. Data file is stored as osd-base/part-id/object-id_data while metadata file is stored as osd_base/part-id/object-id_attr.

According to the T10 standard, there is no data information associated with ROOT, PARTITION or COLLECTION objects. We however implement data files for ROOT, PARTITION, COLLECTION objects only for the internal purposes, to speed up processing of OSD LIST commands.

Root data file stores list of all partitions, partition data file stores list of all user/collection objects and collection data file stores list of all member objects. This data is not be accessible through any OSD command and is used only for the internal purposes. OSD commands with data operations on these objects will fail with error.

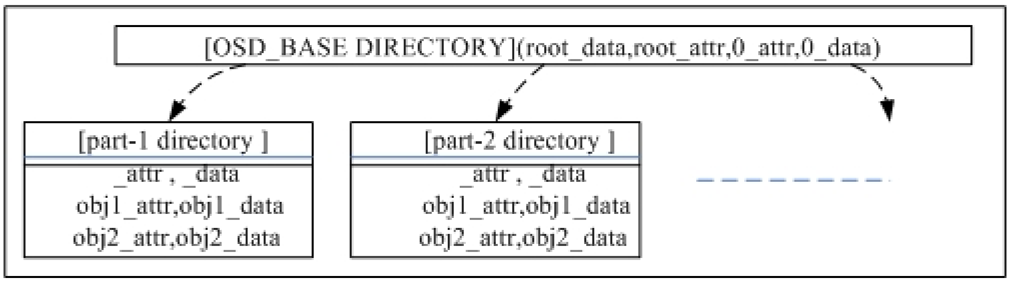


Figure 3.6: OSD Object File Hierarchy

3.4.3 Storage of Key Store

Cryptographic keys are associated with root object and partition objects only, storage of these keys is done in flat hierarchy. Key Store directory (osd_key_base) is a sub-directory of OSD base directory (osd_base/keys).

Keys are stored as described below,

- For root keys: osd_key_base/root_keys.
- For partition keys: osd_key_base/part_(part_no)_keys.
- For Working Keys : osd_key_base/working_(part_no)_(key_ver)_keys.

3.4.4 OSD Target Configuration File

Default configuration file for the OSD target is "/etc/osd_export.conf". Before starting OSD simulator program, one needs to export at least one OSD LUN. A directory can be exported as OSD LUN by making entry in the "/etc/osd_export.conf" file.

Format of an entry is given below,

```
$osd_target_name $lun_no_per_target $osd_lun_base_directory $capacity_in_Hex
$osd_system_id $username $master_key_digest $channel_token_buffer
```

An example of an entry is as following.

```
iqn.2006-09.com.ac.iitk:guest-osd.lvm 0 /tmp/osd/ FFFF iitk_kanpur iitk_user
ab34f2q123 e34f287da
```

This entry exports OSD target with name "iqn.2006-09.com.ac.iitk:guest-osd.lvm" LUN 0 with OSD system ID "iitk_kanpur" . This OSD LUN has OSD base directory as "/tmp/osd". Capacity of OSD LUN is 0xFFFF bytes. LUN owner name is "iitk_user". It specifies digest of master key and channel token buffer used for authentication.

Exporter module of OSD simulator takes each entry, and makes entries for osd-io devices in the iSCSI configuration file (i.e. /etc/ietd.conf). The entries in ietd.conf are used by the IET daemon to configure and export the iSCSI LUNs and include information such as the OSD target name, LUN number, OSD directory name.

Entry in the /etc/ietd.conf corresponding to the above example given as below,

```
Target iqn.2006-09.com.ac.iitk:guest-osd.lvm

Lun 0 Path=/tmp/osd/,Type=osdio
```

3.5 Putting All Together

The OSD simulator interacts with the IET daemon and other local file system activities as explained earlier. We shall see the flow of entire OSD simulator in this section with example of two actions. We shall see the flow on how to export an OSD LUN and the flow on how to perform write operation on this OSD.

3.5.1 Exporting OSD LUN

The OSD LUN export is explained in the sequential steps as given below.

Step 1: Adding LUN information

Entry need to be added in the OSD configuration file (/etc/osd_export.conf) for the OSD device to be exported. This is done through a text editor. OSD simulator program is started. This program interacts with the IET daemon.

Step 2: Flow of Exporter module

Exporter module scans OSD entries in the OSD configuration file and adds entry for each OSD LUN in the iSCSI configuration file (/etc/ietd.conf). It adds entry to the OSD LUN list maintained by the OSD request server as mentioned in the section 3.3.1. After this, iSCSI target program (IET) is restarted.

Step 3 : Creating OSD LUN

The ietd daemon of IET (section 3.2) scans entries in the configuration file (/etc/ietd.conf) and submits each entry one by one to the kernel mode daemon through netlink socket. IOType module (section 3.2.2) checks the iotype of this entry and associates osd-io functions for processing of the OSD LUN corresponding to the OSD LUN entries (the type being osd-io).

IOType module then adds OSD LUN in the iSCSI LUN list maintained by IET.

Step 4: Inquiry command from iSCSI Initiator

iSCSI initiator asks for available SCSI devices to the iSCSI target. iSCSI target sends a list of iSCSI LUNs to the iSCSI initiator. For each LUN, iSCSI initiator sends an INQUIRY (INQUIRY 83) [50] command, which requests information about the device owner, vendor, device type, etc.

NWThread module (section 3.2.3) gets INQUIRY command and invokes SCSI inquiry call handler of the WThread module. WThread module (section 3.2.4) identifies that command is intended for the OSD LUN, it invokes inquiry command handler of the OSD-iSCSI callback module (section 3.2.5). Inquiry command handler of OSD-iSCSI module sends task request to the OSD request server (section 3.1.1) of the OSD simulator through socket communication.

OSD request server preprocesses the request and passes it on to the OSD task processor (section 3.1.2). OSD task processor identifies the command as SCSI INQUIRY (non-osd command), invokes SCSI command handler and sends INQUIRY response to the iSCSI Target.

iSCSI response unit of iSCSI target sends INQUIRY response to the iSCSI initiator. iSCSI initiator accepts inquiry response, submits to the SCSI mid layer [51] which ultimately forwards it to SCSI upper layer driver for OSD (SO driver) of SCSI upper layer of linux SCSI sub-system [51]. SO layer at the initiator side creates a device node for this OSD LUN.

3.5.2 Writing Object Data

The OSD write operation is explained with an example of OSD WRITE command for writing 100 bytes. Since all OSD commands can also request for attributes, we also request GET attribute request for object data length along with the OSD write request in this example. The OSD write request is executed in the following steps.

Step 1: Initiator sends request

OSD initiator prepares an OSD command within a SCSI CDB. The service action argument of this command is chosen as OSD WRITE to indicate OSD write. The initiator also adds data to write as part of the SCSI command.

In OSD framework, each command may have a piggybacked GET/SET attribute command. We request for the object length as an attribute. For this request, the initiator adds entry in SCSI CDB to request for the GET ATTR of object data length (object attribute page #3 attribute #1).

Initiator sends the SCSI CDB to the given OSD LUN using iSCSI protocol.

Step 2: NWThread receives the OSD request

NWThread module of IET receives the SCSI request, parses it and identifies it as an OSD command. It then transfers control to WThread module of IET.

Step 3: Communication with OSD Simulator

WThread module of IET identifies the OSD request. It invokes "osd-io" corresponding to the OSD LUN which results in the activation of the OSD-iSCSI module.

OSD-iSCSI module passes the OSD command request along with data to the OSD request server at the OSD Simulator. OSD request server forwards the request to the OSD I/O processor. OSD step engine (section 3.1.3.1) of OSD I/O processor creates an osd_req structure for the command and initializes the OSD CDB part.

Step 4: Processing at security checker and preprocessing Unit

OSD step engine controls the invocation of various units. It first invokes the security checker for access control related processing. Security checker gets the security method, the object type and the key version from the OSD CDB. It the extracts the corresponding core key from the key store. Security checker now generates the capability key using the core key and capability fields of the OSD command. Capability key is used to calculate the integrity value of the OSD command frame.

The integrity value is compared with the value available in the OSD CDB. Assuming that the match is successful, OSD step engine now invokes the preprocessing unit. Preprocessing unit parses OSD_WRITE command and initializes the command specific parameters of the osd_req structure.

Step 5: Processing at OSD I/O Processor

After the preprocessing, the OSD step engine invokes the data processing unit of the OSD I/O processor. Data processing unit maps the object to the local file name and performs a file write operation on the file at the given offset for the object. OSD step engine now invokes the metadata processing unit of the OSD I/O processor to process GET ATTR request.

Metadata processing unit finds the name of the file containing metadata for the object, calculates offset of the given attribute from the file and reads the attribute value. This value is copied to the output data buffer. OSD step engine now invokes the response unit to send the command response and output data to the iSCSI-OSD module of iSCSI target.

Step 6: Processing at iSCSI response module

The OSD-iSCSI module forwards the command response and output data to the iSCSI response module. iSCSI response module sends response to the iSCSI initiator thus completing the execution of the WRITE command.

3.6 Running OSD Simulator

Our OSD simulator can be run as a service daemon or can be started manually.

In order to run it as a service daemon, the “service” command may be used as given below.

```
> service osd-target start
```

In response to this command, the output on terminal appears as,

```
IITK OSD Simulator Program Started
Exporting target = ..............
```

This action creates a daemon process "osd-sim" which starts the OSD simulator with the default parameters as given below.

- OSD port : 11026
- Debug Mode : -1
- iSCSI Configuration file : /etc/iscsi.conf
- OSD Configuration file : /etc/osd_export.conf

2. Starting Manually

The OSD simulator can also be started manually through “osd-sim” command with several optional command line arguments as given below.

- -p <port> : TCP port for the OSD service. Default value is 11026.
- -d <debug_level> : Debug level for OSD simulator
  - -1 : Run OSD simulator as a background daemon (also the default)
  - 0 : Prints error messages if any
  - 1 : Debug output from upper level functions
  - 2 : Detailed debug output from all functions
- -o <filename> : Name of the OSD configuration file to export OSD LUNs from. Default file name is taken as /etc/osd_export.conf
- -i <filename> : Name of the iSCSI configuration file to export iSCSI LUNs to. Default file name is taken as /etc/ietd.conf.

In addition to these parameters, the “-h” parameter may be used to get the information on the current configuration. This option does not start the daemon.

Chapter 4

Design and Implementation of OSD Client and Manager

In this chapter, we describe design of OSD client and OSD manager. As mentioned in chapter 2, an OSD client runs applications which access data from an OSD device. An OSD manager is responsible for management of policy decisions for access to objects in an OSD and issuance of security tokens to the OSD client.

4.1 OSD Client Design

An OSD client interacts with the OSD server to acquire data for the objects. For this operation, it needs to present its security authorization in form of the security tokens obtained from the policy manager. Our OSD client consists of various modules organized in the Application layer, Object layer and SCSI upper layer as shown in figure 4.1.

4.1.1 Application Layer

Application layer is a representation of programs and utilities that operate on OSD data. As objects are moved between layers, all layers except the application layer process only the control information while the data is processed by the application layer.

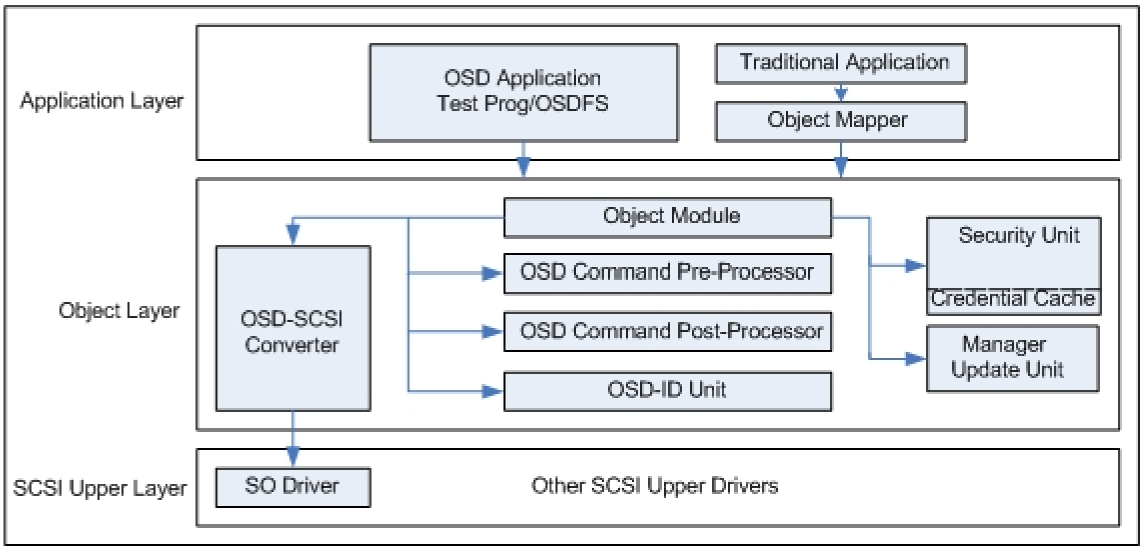


Figure 4.1: OSD Client Modules

4.1.1.1 OSD Application

An OSD object aware application is known as an OSD application. An OSD application manipulates its data in terms of objects. For example, an OSD application can implement a full-fledged file system based on OSD (OSDFS) or may be just a test application to test functionality of other layers.

Instead of using traditional read/write system calls, an OSD application calls object read/write functions of object layer to operate on the OSD data. Intelligent OSD application can also provide feedback regarding the object working set to the object layer (section 4.1.2).

4.1.1.2 Traditional Application

An application that is unaware of OSD system is a traditional application. It operates on the data using read/write system calls and assumes traditional block based architecture of underlying system. An object mapper is required which maps block based I/O to object based I/O in order to port such applications to the OSD system.

4.1.1.3 Object Mapper

An object mapper allows block based application to operate on OSD device. It is responsible for converting file system based block read/write calls to object read/write functions that are implemented by the object layer. This module therefore maps file and block level abstraction to the objects.

Object mapping can be either at the block level where data blocks are mapped to the objects, or at file level where files are mapped to the objects. Object mapper can also provide feedback regarding object working set to the object layer.

4.1.2 Object Layer

The object layer in an OSD client is responsible for implementation of security features, and communication with security manager and SCSI upper layer. OSD commands are submitted to the SCSI stack as SCSI commands after the formulation of SCSI CDBs. Object read/write calls made by the application layer are therefore processed to create SCSI CDBs and then passed on over iSCSI protocol.

This processing of OSD commands is done in the object layer. Besides this, the OSD request needs additional processing before sending to the lower levels and similar processing also needs to be done for the command response before passing it to the application layer. This processing is carried out in the object layer. Various subsystems of the object layer are the following.

4.1.2.1 Object Module

This module exports object read/write functions which are called by the application layer modules. It acts as a communicator between application layer and object layer, and is responsible for sequencing of operations in the object layer by invoking other modules.

Object module also sends feedback on object access to the analyzer module which is used later to make operating more efficient. This feedback includes the access information specifying object information, type of access and name of the application accessing this object.

4.1.2.2 OSD ID Unit

In our implementation, OSDs are implemented as SCSI devices. We have used SCSI device id to identify the OSD. OSD ID unit is responsible for the management of SCSI device id for

OSD devices in the system. SCSI device ID is used as identifier for OSD device for internal communication between various modules at client and communication with OSD manager.

SCSI architecture defines attribute pages associates with each SCSI device. SCSI device ID is stored in INQUIRY page. OSD ID unit obtains this ID by sending INQUIRY command requesting this attribute,

4.1.2.3 Object Command Preprocessor

The OSD requests from the application layer are preprocessed before they move on.

Preprocessing includes the following

- Invoking the security unit for obtaining security token.
- Placing the request for required metadata information of the object to the OSD device. This information may be modified as a side effect of the command execution. For example, the access time of an object is modified as a side effect of read command. Such modifications are updated to the OSD manager.
- Computing Message Integrity Code (MIC) for OSD request and attaching it with OSD request.

4.1.2.4 Object Command Postprocessor

This module is responsible for processing of metadata information requested in the pre-processing unit. Post processing is carried out after command execution. It includes updating the credential cache (section 4.1.2.6) and updating security manager.

For ”remove object” commands, the post processor removes corresponding entries from credential cache. For example, in case of REMOVE PARTITION command for an OSD LUN, the post processor removes all entries corresponding to the partition object from credential cache.

The post processor invokes manager update unit for the modification of required metadata information as a side effect of OSD command execution.

4.1.2.5 Security Unit

Security unit is responsible for the implementation of security features at the client side which includes getting credentials for the object. For an OSD request to the given OSD device, SCSI device ID is obtained from the OSD ID unit. Credentials for the object are first searched in the credential cache. If credentials are found but permission rights do not allow performing the given operation, requested OSD command is rejected with an error. Credential request contains OSD-SCSI ID, object information, user information.

4.1.2.6 Credential Cache

Credential cache is a local store used to store recently accessed credentials. Entries are added to the credential cache using the response of the security manager. Entries are removed after the execution of remove commands or when credentials expire or whenever the policy changes.

An entry in the credential cache consists of object information (OSD-SCSI ID, object type, part-id, object-id), user information (uid, gid), credentials and priority of object. Priority information of objects is obtained from the analyzer module. If the priority information is not found, given object is assigned medium priority. We use hash queues for management of the credential cache entries. Priority based replacement policy is used.

4.1.2.7 OSD-SCSI Converter

This module is responsible for encapsulation and decapsulation of OSD request into SCSI control descriptor block (CDB). A normal SCSI CDB size is 8 bytes while OSD CDB size is 200 bytes. Remaining 192 bytes of OSD CDB are send as additional header segment (AHS) [10] within the SCSI CDB.

4.1.2.8 Manager Update Unit

OSD manager maintains a database containing access control information and metadata for objects. This database needs to be updated after execution of CREATE/REMOVE commands at the OSD client. This task is done by the manager update unit. A request from the OSD client to the

OSD manager is sent by the manager update unit for updating this database as a side effect of command execution.

Manager update unit is invoked for the following OSD commands.

FORMAT, CREATE PARTITION, CREATE OBJECT, CREATE COLLECTION, REMOVE OBJECT, REMOVE COLLECTION, REMOVE PARTITION.

4.1.3 SCSI Upper Layer

SCSI upper layer driver for OSD (i.e. SO driver) is implemented as a part of the Linux SCSI stack [51]. Like other SCSI upper layer drivers such as those for SCSI disk (sd) and SCSI tape (st) devices, it is responsible for detecting and managing OSD devices in the host system. OSD command is submitted to this layer module. OSD ID unit queries the SO driver using ioctl system call to get a list of all OSD devices.

4.2 Communication Between OSD Client and OSD Manager

OSD client sends two types of requests to the OSD manager, namely, credential request and database update request.

4.2.1 Credential Request

This request is sent when credentials for an object is not found in the credential cache. OSD client prepares credential request containing access right information, user information and object information.

Access right information is given as an array of service actions for which access rights are required. User information specifies user id and group id of the user who wants to access the object. Object Information specifies OSD-iSCSI ID obtained from OSD-ID unit (section 4.1.2.2), partition ID, object ID and the type of the object.

Credential response from the OSD manager contains credentials for the given object.

4.2.2 Database Update Request

Database update requests are sent to update database at the OSD manager side as a side effect of the command execution. OSD client prepares the database update request containing OSD service action, user and object information.

Response to this request provides the status of the processing at the OSD manager side.

4.3 Working of the OSD Client

We illustrate the working of the OSD client with the process of identifying the OSD devices and corresponding initialization at the OSD clients, and an example of object create command.

4.3.1 Detecting OSD LUN

In order to detect the OSD LUN, iSCSI initiator with the OSD support queries for the list of OSD devices to the iSCSI target. In response to this, iSCSI target program sends a list of available SCSI devices to the iSCSI initiator. For each SCSI device iSCSI initiator then prepares an INQUIRY command and invokes the object layer of the OSD client. The object layer identifies the request for the OSD device and forwards it to corresponding SO driver of the SCSI upper layer. SO driver then creates a /dev entry for OSD device. All commands for the OSD LUN are then operated with this /dev entry.

4.3.2 OSD Client Object Layer Initialization

At the start, OSD client establishes connection with the OSD manager. For each OSD device LUN identified by the system, OSD ID unit sends an INQUIRY command to request for the SCSI device ID.

4.3.3 Working of the Object Create Command

The command to create an object is handled by the OSD client and then sent to the OSD device. We see the client side processing here.

An OSD aware application at the client invokes a function Create_object exported by the object module of the object layer (figure 4.1). It is then preprocessed by the pre-processing unit which identifies the command and additionally requests for the attribute object create time along with this command.

The security unit searches for the object credentials of parent partition (since this is an object create command) in the credential cache. If the credentials are not found in the credential cache, the security unit requests the OSD manager for the same. Upon response from OSD manager, it adds that entry in the credential cache. Integrity value of the OSD command is computed using capability key field of credentials.

OSD command with credentials and this integrity value is then given to the OSD-SCSI module. OSD-SCSI module encapsulates it into SCSI CDB and invokes the SO driver. SO driver submits the given OSD command to the SCSI middle layer which then sends it to the iSCSI initiator program.

iSCSI initiator program sends OSD request to the OSD target using iSCSI protocol and receives the response. The initiator then submits the OSD response back to the SCSI mid layer. SCSI mid layer forwards this response to the SO layer which is returned to the OSD-SCSI module. OSD-SCSI module de-capsulates the SCSI command and extracts the OSD response.

As a part of command post processing, post-processing unit extracts the object creation time and invokes the manager update unit to update the policy database related to the object.

Finally, object module returns from the Create_object function call and result is sent back to the application layer.

4.4 OSD Manager Design

The general architecture of the OSD manager is shown in figure 4.2.

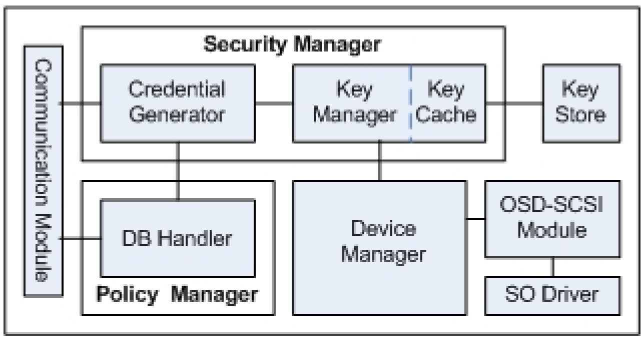


Figure 4.2: OSD Manager Architecture

4.4.1 Communication Module

The communication module in an OSD manager implements handling of communication between an OSD client and the OSD manager. There are two types of requests sent by OSD client namely – credential requests and database update requests.

For credential requests, it invokes the security manager to generate credentials for the given object. For database update requests, it invokes the policy manager to update the object database. It also maintains the history of accesses to the objects that can be useful for access analysis.

4.4.2 Security Manager

Security manager module is responsible for issuing credentials for the objects as per the request from the OSD client. It comprises of the key manager and the credential generator.

4.4.2.1 Key Manager

This module is responsible for the management of keys associated with objects. Keys are stored in the permanent key store to prevent loss from power failures or system reboots. Recently accessed keys are copied to the key cache in volatile memory.

Keys are stored as explained in section 3.4.3. Keys are associated with the root object and partition objects only. The keys of these objects are stored in the key store directory (osd_key_base) in a non-hierarchical manner. This directory is a subdirectory of the OSD base directory.

4.4.2.2 Credential Generator

Credential generator module is responsible for the generation of credentials using security algorithms. Credentials for an object are generated using policy information of that object. Credentials comprise of a capability key and the policy information for the user making the access to the object.

4.4.3 Policy Manager

Policy manager is responsible for storing policy information for each object and relations among objects. It maintains object database which contains information about object identification, object owner, ACL list and policy related information. Object entries are added/removed from the database as per the database update requests from the OSD client.

4.4.3.1 Database Handler

The database handler unit maintains and handles operations on database to store information about objects.

A database entry for an object includes the following.

- Expanded object ID including the OSD device ID, partition ID within the device, object ID within the partition and the object type
- User information including UID and GID of the user and access permissions
- Object creation time and other object related policy informations

Access permissions for the users are maintained in the form of a bit array with each bit specifying permission for specific OSD operations. For example, bit 0 specifies permission to create object, bit 1 specifies permission to read object etc.

There are at least three entries for each object. These include the entry for the owner of the object specifying owner uid, gid and the owner's permission rights; entry for the users belonging to same group as the owner with uid being NULL, gid field containing the gid of the owner and permission rights for these users; and an entry for users belonging to other groups with uid and gid being NULL, and permission rights for these users.

System administrators can add more entries for any other user or group of users as per the policy.

4.4.4 OSD Device Manager

This module is responsible for the management of the OSD devices in the system. This module interacts with the OSD device for initialization of security parameters and communication of the shared keys. Besides this, it also performs job of the maintenance of the OSD device.

OSD-SCSI converter module and SO Driver module in the OSD manager are similar to the ones for the OSD clients as discussed earlier and implant the same functionality.

4.5 Our Implementation

Our current implementation includes the basic skeleton of the OSD system. We have built initiator side modules on top of the IBM OSD initiator [11]. The client side implementation includes the object layer and upper layer modules as described earlier. We have also implemented a user mode OSD manager as described earlier. Our implementation supports the collection objects.

Chapter 5

Prefetching Techniques

Prefetching is useful for improving response time in client-server systems. Client-server systems are based on the two-tier or three-tier architectures where the server provides services to the client upon service requests. Major performance factors in these systems are service response time and network traffic between server and client.

Prefetching is useful for improving response time in client-server systems. Prefetching techniques bring data a priori in anticipation to the client side based on data access analysis. Prefetching can also be helpful in reducing network traffic. In an OSD system, prefetching of credentials of objects reduces the transactions between the OSD client and security manager and improves the system performance. We discuss this mechanism in this chapter.

5.1 Overview of Prefetching

Prefetching is a mechanism to bring data before it is actually processed. In a distributed environment, data needs to be fetched from the remote server before actual processing at the client. Since data is not local to the client, time to fetch the data impacts the overall the response time. Prefetching techniques are aimed to improve the time to fetch the remote data by bringing it a-priori and thus improving overall response time.

5.1.1 Prefetching Techniques

5.1.1.1 Aggressive (Sequential) Prefetching

Aggressive prefetching is a technique that brings data surrounding to the data currently being accessed without consideration to the history of past accesses. Aggressive prefetching assumes spatial locality of access and brings all data near to the data currently being accessed. This is quiet a naive approach where aim is to bring as much data as possible. This method is also called as sequential prefetching since it is useful when data access pattern is sequential in nature. However, if it is not the case, this may lead to the wastage of network traffic as it ignores user access behavior. This method is implemented in various file-systems for prefetching of file data.

5.1.1.2 Selective Prefetching

Selective prefetching is a technique where candidates suitable for prefetching are chosen by considering history of past accesses. This is typically achieved by analyzing the access pattern based on the past accesses and selecting the ones with high probability of future accesses. Selective prefetching requires smart applications which can determine suitable candidates. Suitable candidates are decided using access analysis techniques, study of access patterns and prediction of future data from the past history. Selective prefetching can be assisted by predictions based techniques or by application hints.

Prediction based techniques are based on the study of data access logs. Access logs are searched for certain access patterns. Dependency relations are defined among data. In this case, user application is unaware of the prefetching mechanism.

Selective prefetching can be based on the user application hints where smart user application can provide hints regarding its future accesses and access behavior. This is often more accurate than prediction based techniques. It however requires changes in the application which must provide hints.

5.1.2 Location of Prefetching

Depending upon the location where prefetching mechanism is implemented, prefetching can be server initiated where prefetching mechanism is implemented at the server side, or client-initiated where prefetching mechanism is implemented at the client side, or proxy-initiated where a proxy residing between server and client implements prefetching mechanism.

In server-initiated prefetching, server keeps a log of accesses of the clients. Data required by the client in future is predicted by studying accesses made in the past. For a given data request from the client, response includes requested data and additional prefetch data. It has advantage of having global analysis among clients, i.e., access analysis made for one client can be used for serving requests from other clients. Prefetching implementation at server adds extra load to server for processing required to generate prefetch data in addition to serving client requests.

In client-initiated prefetching, the client analyzes its own accesses by studying data access logs of applications running at client side. Prefetch data is requested along with the request of data currently being accessed. This is more accurate than the server-initiated prefetching but requires smart applications at client side. However, it does not have advantage of global analysis as in server initiated prefetching.

In a proxy-initiated case, a proxy forwards the client requests to the server. This is suitable for proxy environment such as in HTTP protocol. It combines advantages of server-based and client-based prefetching. Proxy, residing between client and server, keeps a log of accesses by the client and performs an access analysis on the log. Prefetch data request is sent along with data request received by proxy from the client. Proxy maintains a local cache which stores data responses.

5.2 Related Work

Prefetching is a common technique used for improving efficiency. Instruction and data prefetch is a common technique in modern processors. Prefetching of file data in the form of READ AHEAD is common in most file systems.

NFS version 3 [8] which is an extension of NFS version 2 [9] added a complex operation NFS READ DIRPLUS to save many lookups. This operation is used to get file handles and attributes along with file names which are member of the given directory. Prefetching of metadata for directory members is done with assumption that it will be required in future.

FileWall [12] is a file access control framework that allows file system administrators to enforce file access policies based on dynamic access context such as access history. It interposes on a client-server path and operates on network file system messages to enforce file access policies and uses prefetching for minimizing request to server.

SEER [15] is a predictive caching system for disconnected mobile operations. It provides a way for mobile computers to access resources in disconnected state by predicting information required a priori and cache the appropriate data while still connected. Files that are needed during disconnected state are predicted by studying user's file-access patterns. This information is used to infer relationships between various pairs of files and define semantic distance that quantifies how closely related they are. Files are automatically prefetched based on the semantic distance.

Prefetching in object oriented database is discussed in a technical report by E. E. Chang and R. H. Katz [13]. It discusses how prefetch algorithms can use structural relationships and inheritance semantics to improve DBMS response time in object oriented databases. User can provide the buffer manager with access hints which influences the buffer manager's prefetch strategy.

Based on this, Jung-Ho Ahn and Hyoung-Joo Kim [14] proposed an adaptable object prefetch policy known as SEOF. SEOF prefetches objects based on correlations and the frequency of requests.

5.3 Prefetching in OSD System

OSD system consists of three components as OSD client, OSD manager and OSD target device. For any request, OSD client gets credentials of given object from the OSD manager and then gets data of that object from the OSD target.

If we consider OSD system in terms of client-server architecture, for any request OSD client interacts with two servers, with OSD manager for service of object credentials and then with OSD target for service of object data information. These are two traffic points where prefetching can be implemented in the OSD system. We are particularly interested in how prefetching techniques can be useful in minimizing the network traffic between OSD client and OSD manager, related to the credentials requests and acquisitions. These techniques can also be used to reduce traffic between OSD client and OSD target by prefetching of object data.

OSD system emerged to provide solution to the security related problems in SAN network. However, for security, it introduces extra overhead of acquiring credentials before access to actual data. As part of a fault tolerant policy, object data in an OSD system may be stored across multiple OSDs. Therefore a data operation may require access to objects from multiple OSDs. It is possible to store credentials in the local cache after first access to an object thereby avoiding repeated requests for credentials. However, there is still a significant overhead of obtaining credentials in terms of time and network traffic even for the first access.

There is a significant difference between credentials traffic and regular data traffic. Credentials are fixed and small size units associated with each object while regular data is of variable size. Due to small size, it is possible to have caching of large number of credentials of objects. In that way the credential cache may be built over time using the first access of objects.

Chapter 6

Prefetching at OSD Client Side

The prefetching can be implemented on the OSD client as well as on the OSD server side with each having its own merits and demerits. We implemented client-initiated prefetching where the OSD client maintains a log of accesses. Access analysis is carried out on this log to identify frequently accessed objects and ordered sequences of objects. We used a graph based mechanism for access analysis.

An ordered sequence of objects is also termed as a pattern. These patterns are used for prefetching the objects credentials by the client.

We discuss client side components of the OSD system and integration of graph based model for prefetching of object credentials from the OSD manager. This model can also be applied for selective prefetching of objects from the OSD device.

Graph based techniques are used by several researchers for various applications. James Griffioen and Randy Appleton discussed graph techniques for prefetching to reduce file system latency [42]. In their approach, they create a graph with nodes as files in the file system and edges to represent the access sequence between nodes. Nexus [43] is a weighted graph based prefetching algorithm for metadata servers in petabyte-scale storage systems. This is designed for metadata servers. In this approach, relationship graphs are constructed dynamically by defining successor relationships. George Pallis, Athena Vakali and Jaroslav Pokorny discussed a clustering-based prefetching scheme [44] on a web cache environment. This is a proxy-initiated prefetching scheme where a graph-based clustering algorithm identifies clusters of correlated web pages based on the user's access patterns. Carl Tait and Dan Duchamp discussed detection and exploitation of file working sets [45] for effective prefetching using graph techniques.

6.1 Access Graph Model

We used access graph model for the object pattern analysis. The access graph model uses weighted graphs to create an ordered set of objects. This set is used for the selective prefetch.

The time duration for which accesses of the objects in the system are observed is termed as "observation window". This factor is dependent on the frequency of objects accesses and changes in the access patterns. Access analysis is carried out at the end of the observation period as a background process.

A log of object accesses is maintained per observation window. Past log files are stored for a limited number of observation windows. The log contains information specifying each accessed object, its access time, the type of access made on that object, etc. In particular, we consider the log for the prefetch of object credentials. In our case, information about type of access made on the object is not very useful but it might be useful for prefetching of the object data from the OSD target.

While doing the access analysis, we compute the object access count and inter-reference intervals to determine frequency of object access in the observation window. We use this information to prioritize objects with reference to their access frequency. In our model, we define four priority levels for objects - very high, high, medium, low.

Prioritization is useful for selective prefetching. Objects with higher priority can be brought a priori. Prioritization can also used in the replacement policy. Objects with low priority are chosen as candidates suitable for replacement.

6.1.1 Object Access Graphs

We build an object access graph by identifying pattern of objects accesses. The logs are searched for an access pattern, i.e., repeated sequence of objects.

If access patterns do not overlap then the task is simple. For a particular access on an object which falls in a pattern, all other objects in the pattern that are accessed after the given object are returned as candidates for a prefetching.

It is possible that two or more access patterns overlap. In this case, selection of a pattern can be done by determining the pattern that is closest to the current access behavior. This solution is expensive and may cause wastage of prefetch data when accesses of the objects are not according to the determined pattern. Alternately, objects from all patterns can be selected. However, there is a limit on the amount of data that can be prefetched. This implies a need to selectively prefetch objects from all patterns of which the given object is a member.

We propose a graph based solution for selective prefetching. This works by maintaining a weighted directed graph for each access pattern. There is a weight associated with each pattern which is a factor of the number of times the given access pattern is repeated. The factor value is decided depending upon the number of patterns and repetitions of the patterns in the observation window.

6.1.1.1 Construction of Access Graphs

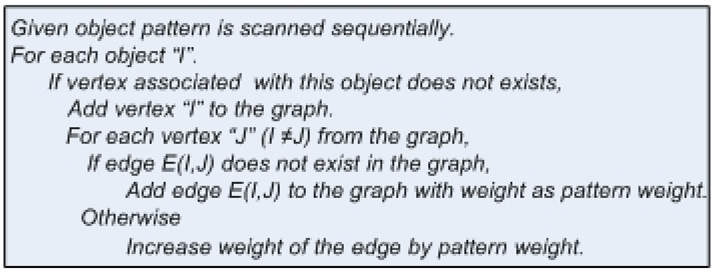
We construct a graph for each pattern with a vertex for each object in the graph. The graph is constructed using method in figure 6.1.

Figure 6.2 shows access graphs for various patterns with pattern weight as 10% of repetition count.

It is also possible for the application to give feedback regarding its working set. Objects in the working set are considered as a pattern and the highest pattern weight value is assigned to this pattern to ensure that this pattern has the highest priority.

Construction of directed graphs is useful for defining successor relations among objects. The list of objects obtained by the traversal of directed access graphs gives the set of objects having a high probability of access after the access of the given object.

The access graphs can also be constructed as undirected graphs. In this case, the weight of a resultant undirected edge will be the sum of weights of the directed edges between two vertices. An undirected graph is useful for defining an object working set, i.e. a collection of objects with high probability of getting accessed together.



```
Given object pattern is scanned sequentially.
For each object "I".
    If vertex associated with this object does not exists,
        Add vertex "I" to the graph.
        For each vertex "J" (I ≠J) from the graph,
            If edge E(I,J) does not exist in the graph,
                Add edge E(I,J) to the graph with weight as pattern weight.
            Otherwise
                Increase weight of the edge by pattern weight.
```

Figure 6.1: Algorithm for Constructing Access Graph

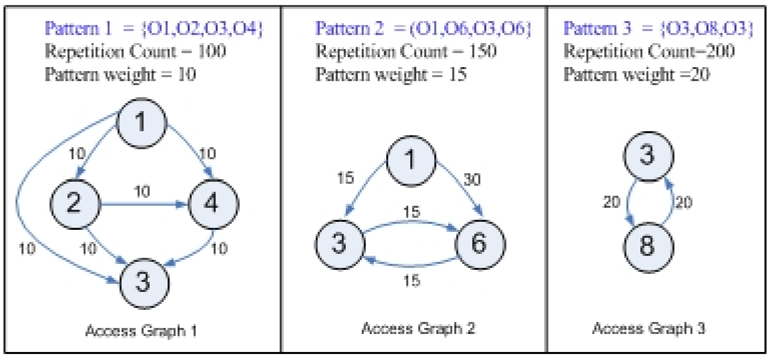


Figure 6.2: Access Graphs

6.1.1.2 Merging of Graphs

Two access graphs can be merged if their associated patterns contain common objects. The resultant graph contains the union of vertices from both graphs.

The weight for edges of the new graph is assigned as follows. If the edge is common to both graphs, the weight of the edge is the sum of the weights of the given edge from each graph. Otherwise, it is equal to the weight of the edge in the graph where it is present.

Figure 6.3 shows merging of graphs shown in figure 6.2.

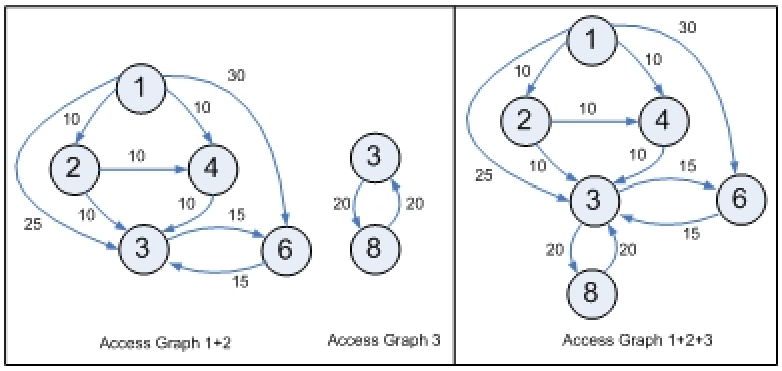


Figure 6.3: Merging of Access Graphs

6.1.1.3 Management of Access Graphs

Access graphs are stored in the database at the client side. For each access graph, there is an entry for every edge specifying its two vertices and the weight of the edge. There is also a unique identifier associated with each access graph.

There is another database maintained which stores the age value for each access graph. The age value is increased if the access graph is repeated in the current observation window. Age values are assigned in such a way that the values for the access graphs found in the recent observation windows are more than those of access graphs found in the past.

Age values are useful for determining replacement policy. It is not possible to store all access graphs from all past observation windows. Access graphs with lower age values are chosen as candidates suitable for replacement.

6.1.1.4 Permanent Access Graph

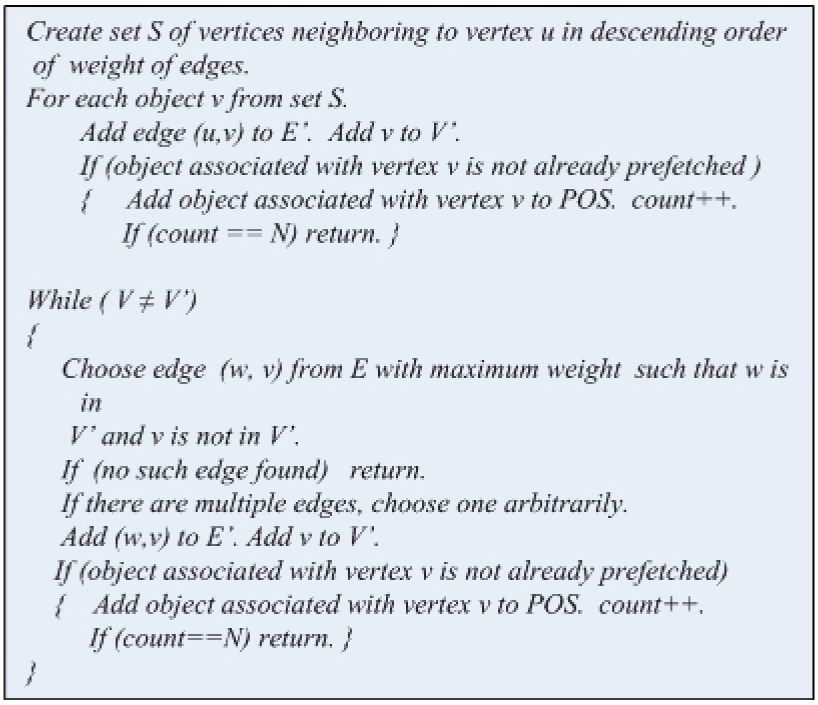
We call an access graph as a permanent access graph if it is always found in the past observation windows. Age values are useful in deciding permanent access graphs. Permanent access graphs have very high age values. The use of permanent access graphs is discussed in section 6.2.2.

6.2 Selective Prefetch using Access Graphs

As mentioned earlier, the size of data that can be prefetched is usually limited. Selective prefetching is done on the basis of weights of edges. An ordered Prefetch Object Set (POS) with objects in descending order of prefetch preference is constructed using an access graph. Depending upon the size of the prefetch buffer, prefetch data for the objects from this set is requested.

When an object is accessed, the access graph containing the given object is searched. If it is not found, prefetching is not done.

The POS is constructed using the algorithm shown in figure 6.4. This is a modified form of Prim's minimum spanning tree algorithm [54]. Initially, all immediate neighbors of the given object are



```
Create set S of vertices neighboring to vertex u in descending order
 of weight of edges.
For each object v from set S.
    Add edge (u,v) to E'. Add v to V'.
    If (object associated with vertex v is not already prefetched )
    {    Add object associated with vertex v to POS. count++.
         If (count == N) return. }

While ( V ≠ V')
{
   Choose edge (w, v) from E with maximum weight such that w is
     in
    V' and v is not in V'.
   If (no such edge found) return.
   If there are multiple edges, choose one arbitrarily.
   Add (w,v) to E'. Add v to V'.
  If (object associated with vertex v is not already prefetched)
  {   Add object associated with vertex v to POS. count++.
      If (count==N) return. }
}
```

Figure 6.4: POS Construction Algorithm

considered. Subsequently, a modified Prim's algorithm is used where the next vertex is chosen having the edge with the maximum weight.

As an example, consider the generation of the POS for vertex 2 of the graph constructed in step 2 of figure 6.3. Initially, immediate neighbors vertices 3 and 4 are chosen. This creates a partial tree containing vertices 2, 3, 4. The modified Prim's algorithm is applied on this partial tree which chooses vertex 8 (as edge 3-8 has the maximum weight) and then vertex 6. Thus, the POS generated for vertex 2 is {3, 4, 8, 6}.

It may be seen that the edges corresponding to the objects with repeated occurrence in a pattern have higher weight. This gives higher preference for prefetching such objects.

It is possible to save time for building a POS for frequently accessed objects by constructing it a-priori and storing it in the database. Such a POS is known as a Permanent POS. Permanent POS is typically used for patterns that occur always in observation windows.

6.3 OSD Client Design Integrated with Graph Model

We have seen various sub-modules of an OSD client in section 4.1. Certain modules are modified to support OSD client-based prefetching using Access Graph Model. Access graph Model adds a new "Analyzer Module" in the object layer.

6.3.1 Object Module

In the prefetching OSD-client with access graph model, the object module sends information about object access to the analyzer module. Access information specifies object information, type of access and name of the application accessing this object.

6.3.2 Analyzer Module

Figure 6.5 shows various units of the analyzer module its interface with other modules.

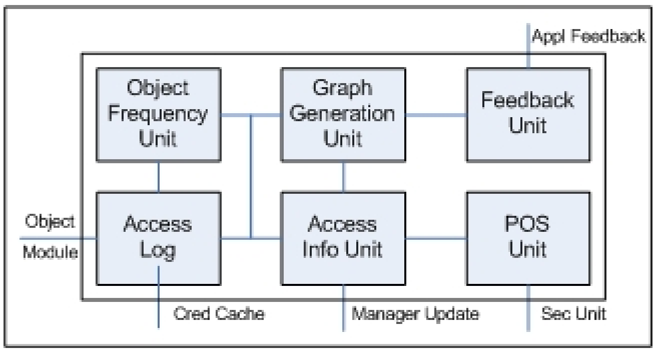


Figure 6.5: Analyzer Module

6.3.2.1 Access Log

Information about the current object access sent by object module is added to the access log. The access log is maintained in a very simple manner since it is updated for each access of every object. We implemented it as simple text file for each observation window with each line specifying time of access, information about application accessing it, object ID information and type of access.

6.3.2.2 Object Frequency Unit

This module is responsible for prioritization among objects. It works on the log file associated with the current observation window and assigns priorities to each object depending on the number of accesses and average interval time. Priority information is stored in the database maintained by access information unit.

Object frequency unit is activated once in every observation window, at the end of the observation period.

6.3.2.3 Graph Generator Unit

Graph Generator unit is responsible for the construction of access graphs. It works on the log file associated with the current observation window and searches for the access patterns. As there is a high probability of getting a pattern around frequently accessed objects, it takes feedback from the object frequency unit regarding accesses of high priority objects.

For each access pattern, the graph generator unit constructs a directed graph known as access graph. A pattern weight is also assigned to each access graph.

After construction of the access graphs, these graphs are merged if their associated access patterns contain common objects. Final access graphs are stored in the database at access information unit.

Graph generator unit is run once every observation window at the end of the observation period.

6.3.2.4 Feedback Unit

Application layer can also help in the process of pattern analysis by providing information regarding the object working set. Feedback unit exports object calls that allow application to provide object working set. It also submits each working set as a pattern with very high repetition count to the graph generator Unit.

6.3.2.5 Access Information Unit

Access information unit maintains databases for object priority, weighted access graphs, age of access graphs and permanent POS. These databases are updated once every observation window at the end of the observation period.

The "object priority" database contains an entry for each object found in the observation window with its priority. Entries are added by object frequency unit during analysis of the access Log. This database is used by several units, including the credential cache to query priority information of objects while adding them to the credential cache; by security unit which queries this database before sending prefetch requests for high priority objects to the OSD manager and by POS unit which queries this database for building permanent POS associated with the high priority objects.

The "weighted access graphs" database stores the access graphs. A unique identifier is assigned to each access graph. Access graph information is maintained in the form of edge list representation with a weight assigned to each edge. Entries are added by the graph generator unit at the end of the observation period. This database is queried by POS unit for construction of POS.

The "age of access graphs" database stores information about each access graph (with unique identifier) and its age. Age is computed using method described in section 6.1.4.3. Graph generator unit adds a new entry for every new access graph found in the current observation window. It also updates age values for old access graphs. This database is queried by the POS unit for construction of permanent POS. This is also queried by the manager update unit to update OSD manager with permanent access graphs.

The "permanent POS" database contains POS for high priority objects belonging to permanent access graphs. POS unit adds entries in this database at the end of the observation period. Manager update unit also queries this database to update OSD manager with permanent access graph.

6.3.2.6 POS Unit

POS unit is responsible for generation of prefetch object set. Request for selective POS of an object is sent to this unit by the security unit. First, this unit queries the "permanent POS" database to check whether POS for the given object exists. If it doesn't exists, it queries the "weighted access graphs" database for access graph containing this object. POS is generated using algorithm as described in section 6.2.1.

6.4 Features of Graph Access Model

Features of our graph model can be summarized as the following.

We have constructed weighted directed graph for each pattern which allows us to define successor relationship between various objects within the graph. Weight of each edge in the graph describes strength of this relation. Merging of these access graphs allows us to define strength of this relation across multiple patterns. Unlike other approaches [43,45] where prediction is done considering only single access pattern, merged access graphs makes it possible to consider multiple patterns for prediction. This can make prediction more accurate.

Our access algorithm to generate ordered prefetch object set is a modified form of Prim's minimum spanning tree algorithm [56]. Initially, prefetch object set is constructed using all immediate neighbors of the object giving them high priority. Later on using a modified Prim's algorithm for maximum weight we add objects having high priority of accesses after accessing a object from the partial prefetch set.

Chapter 7

Server Side Prefetching Model

We implemented an another prefetching model where the OSD manager can predict objects required by a client and send credentials for them as prefetch data. Prefetch mechanism is based on identifying relations among objects. Relations are defined among objects using access history, feedback from the client, common collection membership and other factors. We observed significant enhancement in performance of object command processing and network traffic.

7.1 Prefetching Policy at Sever Side

Prefetching policy is devised for minimizing overheads of the credential requests. This is also helpful in reducing number of credential requests from a client to the security manager.

In our approach, the OSD security manager predicts the objects that might be required by the client and provides the credentials for all such objects along with the credentials for the requested object. We refer to these additional security tokens as prefetch data. OSD client can help in this process by providing hints for the prefetch data. It is usually possible to group objects accessed by a client depending upon their access behavior.

We introduce a relationship among objects. Two objects are termed as RELATED if when an object is accessed, there is a high probability that other object will be accessed in near future. Group of such objects form an “object relation group”.

Prefetching mechanism is effective if it sends prefetch data for all objects in the group upon access to any object from the group. In our case, the prefetching mechanism should return credentials for all objects in the object relation group upon credential request for an object from the group. The object in the request is termed as an ORIGIN object. OSD manager generates prefetch data by making prediction about objects required by the client in the near future. This prediction will be based on identifying relations among objects. There are various factors that are used to define relations among objects. We provide details about prefetching mechanism in the next section.

7.2 Relations Among Objects

We group objects by defining relations among them. Both OSD client and OSD manager can participate in the process of defining relations. OSD client can have full participation or partial participation in the process of defining relations.

When the client participates fully, it uses collection object support and makes related objects member of a collection. Object belonging to same collection are considered as related and OSD manager does not require to perform additional processing to relate them. While in case of partial participation, client provides hints to the manager. OSD manager uses those hints while grouping objects.

However the access analysis done by the client is based on the access done by the client only. This mechanism therefore does not have advantage of global analysis which is possible only when the manager defines the relations. OSD manager can define relations by performing access analysis of objects. Relations are defined among objects by study of object access patterns and by considering various other parameters such as creation times.

7.2.1 Client-defined relations

In chapter 6, we discussed the implementation of client initiated prefetching mechanism. We describe the role of OSD manager in this chapter. As described earlier, OSD client can have either full participation using collection membership or partial participation by providing access hints in the process of defining relations.

7.2.1.1 Collection Membership

An OSD client can define relations among objects by making them member of a collection. This is the simplest scheme that can be implemented by an OSD client. Depending upon the type of application data it is accessing, OSD client can group objects that are likely to be accessed together and make all objects from a group as member of a collection.

From OSD manager's perspective, it does not need to analyze objects and their access patterns to create relations. If at some instance, the object whose credentials are requested by the client is part of a collection, OSD manager will send credentials of other objects in the same collection as prefetch data. An object can be part of multiple collections. In that case, either client provides hints about desired collection or the OSD manager can use other mechanisms to determine the required collection.

7.2.1.2 Hints from Client

An OSD client can also provide hints for the prefetch data. This is useful when collection objects are not supported in the system or where the RELATED objects do not share the membership of a collection object. There are various hints which can be provided by the clients while requesting credentials for an object. These include the following.

- objects created together along with the object in the OSD request.
- objects of specific object type, e.g., list of all partition objects required by utility such as df, fdisk etc.
- objects with specific permission masks, e.g., list of all objects with READ_ONLY or WRITE_ONLY permissions.
- objects with particular policy information.
- combinations of above

OSD manager looks up the object database depending upon the hints and send credentials for these objects as prefetch data.

7.2.2 Manager-defined relations

An OSD manager can use the object access history to establish access relations between objects. This approach can be used in addition to the OSD client-defined relations but is particularly useful when the client is not intelligent enough for defining relations among objects or for providing prefetch hints. In this approach, OSD manager maintain the object access history and analyzes this for prediction of objects that may be required in the near future. Access pattern analysis can be done in following ways.

7.2.2.1 Closely spaced in terms of creation time

This method is useful for applications where client data processing is done in steps. For example, the situations where required objects are created initially and then the operations are performed in subsequent steps. OSD manager can treat all objects created within a close interval of time as related.

7.2.2.2 Study of Object Access patterns

OSD manager can analyze object access history and object metadata stored in the object database. It can then identify access pattern and group objects as related when their accesses are made within small time duration. This however requires large processing at OSD manager side.

7.3 Optimizing Prefetch Data

Sending prefetch data is useful for reducing initial command processing delays for the prefetched objects. It also reduces number of credential requests. However, there will be no optimization with respect to the credential response traffic if complete credentials are sent for each prefetched object. This can lead to a wastage of network bandwidth especially when prediction is not correct.

By noting the fact that prefetch data is always passed with credentials of the ORIGIN object, it is possible to optimize prefetch data by sending just part of credentials for the prefetched objects. Mechanism can be built at client side to reconstruct credentials for the prefetched objects from partial credentials and the credentials of the ORIGIN object. There is a possibility of having identical values of various fields in credentials of two objects only when they belong to the same OSD LUN and partition, and credentials are created using same key specific data within close time interval.

Common fields are generally related to the security key specific information, security token validity period, parent partition information etc. While comparing credentials for the ORIGIN object and RELATED objects, it is observed that there are almost 50% identical fields. Therefore, instead of sending entire credentials for each prefetched object, it is sufficient to send just 50% of credentials unique for the prefetched object.

The OSD client gets a list of partial credentials as prefetch data along with the credentials for the ORIGIN object. For each prefetched object, credentials are reconstructed and added to the credential cache. This results in cutting down half the traffic between a client and manager in OSD system.

7.4 Design of OSD Manager Integrated with Prefetching Model

We have described various modules of the OSD manager earlier. We now describe modules related to the implementation of prefetching module. Figure 7.1 shows modules of OSD manager integrated with prefetching model.

Prefetching model is implemented by two new sub-modules in security manager and policy manager modules. The communication module is extended to maintain the history of objects that is used by the dependency engine (described in section 7.4.3.1) for predicting the prefetch data.

A new prefetch generator module is responsible for implementation of prefetching mechanism at OSD manager side. This module generates the prefetch data. For the given ORIGIN object in the credential request, it takes a list of RELATED objects from object dependency engine. Credentials for each of the RELATED objects is obtained from credential generator module. For each of these credentials, parts of credentials consisting of fields distinct from the credentials of the ORIGIN object are used to generate prefetch data.

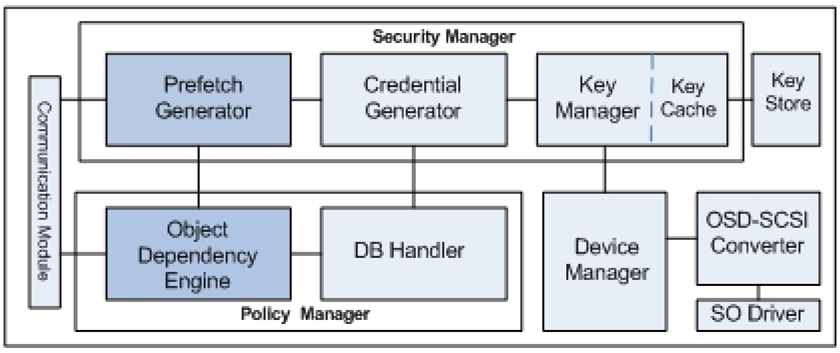


Figure 7.1: OSD Manager Integrated with Prefetch Model

7.4.1 Policy Manager

As described earlier, the policy manager maintains two databases, namely, object database and relation database. For each related datagroup, relation database contains the list of the objects which form related object group. There is also a unique identifier associated with each relation. Relation database is updated as per the relations defined by the object dependency engine (ODE).

The ODE module implements the core of the prefetching model. It is responsible for identifying or defining relations among objects and it outputs the list of related object for a given ORIGIN object.

As discussed earlier, there are various ways of defining dependency among objects. The easiest way is when client defines dependency using common collection membership. In this case, collection identifier is used as the relation identifier. Objects belonging to the collection of the ORIGIN object are considered as RELATED objects.

When dependency is specified using client hints, ODE requests database handler module for related objects using client provided hint information. Result of the queries provide the list of RELATED objects.

When client does not provide the hints, ODE uses intelligent techniques to define relations among objects. These include searching and analyzing access patterns in the access history maintained by the communication module. If an access pattern is found, objects in the group are considered to be related. The ODE assigns unique identifier for this relation and makes entry for each object in the relation database. If no access pattern is found, ODE tries to collect information about objects from the object database to find relations. This information includes object creation timings, ACLs, policy data, etc. Task of defining relations among object is done as a background process at the OSD manager. This, therefore, does not delay the processing of given credential request.

7.5 Performance Model

In this section, we look at the efficiency of the prefetching mechanism. In particular, we look at the impact on reduction of traffic and processing time.

For calculations, we assume that after the first request, credentials for an object are stored in the credential cache at OSD client side. Subsequent accesses for credentials of this object result in a hit in the credential cache. In our analysis, we assume that credentials never expire and size of credential cache is sufficiently large so that the entries once cached are never replaced. Therefore, credential cache hit is 100% for subsequent accesses on all objects.

The following symbols are used in the performance modeling

- T_{cred} = Time required to get credentials from credential cache.
- T_{mgr} = Time required to get credentials from OSD manager.
- T_{sec}^{1} = Total time required by the security unit in an OSD client for the first access of an object,
  $T_{sec}^{1} = T_{cred} + T_{mgr}$.
- a = Number of accesses to an object done by an OSD client.
- T_{sec}^{a} = Total time required by the security unit in an OSD client for "a" number of accesses to an object with no prefetch.
- T_{sec}^{pa} = Total time required by the security unit in an OSD client for "a" number of accesses to an object with prefetching mechanism.
- S_r = Number of objects maintained at OSD manager for object relation group.
- p_d = Probability of correctness of prediction of object relations by the OSD manager.
- N = Average number of objects accessed by a client within some time period.

- B_{sec} = Total network traffic for credentials for an object without any prefetch hints,

  $B_{sec} = C_{req} + C_{rsp}$.

- B^p_{sec} = Total network traffic for credentials for an object with prefetch hints,

  $B^p_{sec} = C^p_{req} + C_{rsp}$.

- B^n_{sec}= Credential network traffic for n objects with no prefetch.

- B^{pn}_{sec}= Credential network traffic for n objects with prefetching mechanism.

- C_{rsp} = Size of the response to the credential request in bytes.

- C_{req} = Size of credential request with no prefetch hints, in bytes, $C_{req} = \delta C_{rsp}$.

- C^p_{req} = Size of credential request with prefetch hints in bytes. We assume $C^p_{req} \simeq \alpha C_{rsp}$.

- $C_{partial}$ = Size of optimized credentials of prefetched objects in bytes,

  $C_{partial} = \mu C_{rsp}$.

Considering our implementation,

$\delta = 0.42$, $\alpha = 0.6$ and $\mu = 0.45$.

Note that, p_d is 1 when relations are directly defined by the client.

7.5.1 Reduction of traffic between OSD Client and OSD Manager

With prefetch mechanism, for a credential request of an object, credentials for all objects from its relation group are returned. Therefore, the total number of credentials returned as prefetch data is S_r while the number of useful credentials of objects is expected to be $n = p_d S_r$.

Therefore,

$$\begin{aligned} & B^{pn}_{sec} = B^p_{sec} + (S_r - 1)C_{partial} \\ \Rightarrow \quad & B^{pn}_{sec} = (C_{rsp} + C^p_{req}) + (S_r - 1)C_{partial} \end{aligned}$$

In absence of a prefetching mechanism, the traffic for an object is B_{sec}. The traffic for all objects is similar in nature therefore the total traffic for credentials of "n" objects without prefetching is

$$B^n_{sec} = nB_{sec} = n(C_{rsp} + C_{req}).$$

The reduction in traffic due to prefetching for n objects can be written as $(B^n_{sec} - B^{pn}_{sec})$ which simplifies to

$$n.(C_{rsp}+C_{req})-(C_{rsp}+C^{p}_{req})-(S_r-1)C_{partial}$$

$$\text{or, } [n(1+\delta)-(1+\alpha)-(S_r-1)\mu]C_{rsp}$$

For an effective prefetching mechanism, this value should be positive, i.e.,

$$\begin{aligned}
& [n(1+\delta)-(1+\alpha)-(S_r-1)\mu]C_{rsp}\geq 0\\
\Rightarrow\quad & n(1+\delta)\geq[(1+\alpha-\mu)+\mu S_r]\\
\Rightarrow\quad & n\geq\frac{(1+\alpha-\mu)}{(1+\delta)}+\frac{\mu S_r}{(1+\delta)}\\
\Rightarrow\quad & p_d\geq\frac{(1+\alpha-\mu)}{(1+\delta)S_r}+\frac{\mu}{(1+\delta)} \qquad (\because n=p_dS_r)
\end{aligned}$$

Using $\mu \simeq 0.45$, $\delta \simeq 0.42$,$\alpha \simeq 0.6$, this inequality reduces to

$$p_d\geq\frac{1.15}{1.42S_r}+\frac{0.42}{1.45}$$

When p_d is taken as 0.5, S_r must be at least four.

This gives an approximation to effective size of a relation object group maintained at the OSD manager. Prefetching is useful even when the probability of prefetch data being useful is 0.5 provided that the size of relation group at the server side is least four.

This means, there is benefit of prefetching model when for a credentials request of object, along with Credentials response of given object, partial credentials of three objects are sent, out of which only one object is actually useful.

A better value of p_d can be achieved by making stronger relation discovery between objects and it will yield even better performance.

Our analysis is however based on a single object relation group of size S_r. In general, client fetches other objects too. Hence the credential requests are sent for the remaining $(N-n)$ objects which do not belong to this relation group. These objects form several other relation groups. If we take the size of relation group i as S_{r_i}, the expected number of useful objects in relation group i is $p_{d_i}S_{r_i}$ where p_{d_i} is probability of correctness of prediction.

Therefore, overall traffic for N objects is,

$$B^{pN}_{sec}=\sum_i B^{pn_i}_{sec}$$

Assuming all relation groups to be roughly equal in size and each containing n useful objects,

$$B^{pN}_{sec} = \frac{N}{n} B^{pn}_{sec}$$

Traffic for N objects without prefetching is,

$$B^{N}_{sec} = N(C_{rsp} + C_{req})$$

Therefore, reduction of overall traffic can be written as,

$$\left[1 - \frac{B^{pN}_{sec}}{B^{N}_{sec}}\right] \text{ which is equal to } \left[1 - \frac{\frac{N}{n} B^{pn}_{sec}}{N(C_{rsp}+C_{req})}\right]$$

Using $\mu \backsimeq 0.45$, $\delta \simeq 0.42$,$\alpha \simeq 0.6$ and all group similar with size S_r and probability of correctness as p_d, this reduces to

$$\left[1 - \frac{(1.15 + 0.45 * S_r)}{1.42 * p_d S_r}\right]$$

This shows that expected reduction in network traffic between OSD client and manager depends on two factors– correctness of prefetched data, and size of object relation group. These two factors are however related and increasing size of object relation group results in reduction in correctness of the prefetch data p_d especially when relations are not directly defined by the OSD client.

When relations are directly defined by the client using collection membership, $p_d = 1$ and $S_r = N$. In this case, reduction depends on N and it increases with N, saturating at around 66.7% for large values of N.

7.5.2 Reduction in Command Processing time at security Unit

After the first access, credentials will be accessed from the credential cache. Hence,

$$T^{m}_{sec} = (m-1)T_{cred} + T^{1}_{sec}$$

In absence of any prefetching mechanism, time for the first access to the object is T^{1}_{sec}, where

$$T^{1}_{sec} = T_{cred} + T_{mgr}$$

Therefore, the time for m accesses can be written as,

$$T_{sec}^{m} = mT_{cred} + T_{mgr}$$

With prefetching mechanism, credentials of the prefetched object are fetched along with the ORIGIN object earlier. Hence, the credentials are found in the Credential Cache. Therefore, the time for m accesses is,

$$T_{sec}^{pm} = mT_{cred}$$

The total gain for the prefetched object can be written as,

$$\left[1 - \frac{mT_{cred}}{mT_{cred} + T_{mgr}}\right]$$

T_{mgr} is usually a much larger value compared to T_{cred}. Therefore, for small values of m (around 5 to 10), the saving in the time is very high.

However, there are overheads associated with accessing the ORIGIN object. These overheads are for identifying RELATED objects, for generating credentials for each of RELATED objects, and for optimization and sending the prefetch data. If the correction of prediction is low, these overheads may be significant and even result in negative gains.

7.6 Our Implementation

We have integrated our OSD system with the prefetching model. Currently, we have implemented prefetching mechanism where prefetch hints are provided by the OSD client using collection membership.

7.7 Results

We have tested efficiency of the OSD system integrated with the prefetching model. Our experiment include the measurement of execution time of different modules.

7.7.1 Setup

We have used three machines, each for OSD client, manager and OSD target. Three machines have the similar configuration as Intel 2.8 GHz CPU, 256 MB RAM and 100 Mbps network. All machines run Linux kernel 2.6.

The following experiments were performed.

- Object Access Operation: OSD WRITE of 100 bytes. Write size is kept small so that the effects of metadata and credentials operation are prominent compared to time needed to transfer the actual data.
- Average number of distinct objects that are likely to be accessed together by the client at particular instance (n) is varied in various experiments. We have performed experiment, for different values of n. Values of n are 10, 20, 30, 40, 50. Experiment is repeated 10 times for each value of n.
- Total number of accesses per object is taken as 5.

As mentioned earlier, our current prefetching mechanism is based on hints from client using collection membership. This makes probability correct prediction at OSD manager as 1. To have different values of correctness of prediction for the experiment, we have modified OSD client for providing incorrect hints. Basically, OSD client provides over-estimation about objects as members of collection. We are maintaining object relation group at OSD manager whose size depends upon membership of collection objects as defined by the client. The results are shown in Table 7.1 for a value of n as 10. The values in the first column of this table are interpreted as follows.

- For no-prefetch, modified OSD client does not provide any prefetch hints. In this case, size of object relation group at OSD manager can be considered as 1 $(S_r = 1)$ because no prefetch data is sent.
- For $p_d = 0.5$, modified OSD client makes $2n$ objects as members of the collection group with only n objects are useful. This scenario therefore simulates a case with $S_r = 2n$ and $p_d = 0.5$.

- For $p_d = 0.66$, modified OSD client makes $1.5n$ objects as members of the collection group with only n objects are useful. In this case, $S_r = 1.5n$.
- For $p_d = 1$, n objects which are likely to be accessed together are made member of collection by modified OSD client $(S_r = n)$. This represents scenario where a real client would provide hints for the prefetch.
- Similarly, $p_d = 0.58$, and $p_d = 0.86$,

The second column provides the time (in microseconds) to acquire the credentials by the OSD client for the ORIGIN object. To ensure that the effects of the Credential Cache do not show up in this table, the time is shown only for the first access for the object.

The third column of the table provide the time (in microseconds) for the entire operation including the acquisition of token, generation of MIC for the OSD request, and the time needed to process the request at the OSD target side to provide response.

The fourth and the fifth columns provide the same values for the RELATED objects. Without prefetching (when $S_r = 1$), the RELATED objects are treated independently and hence the credentials for these objects are acquired independent of the ORIGIN object. The time for the acquisition of the token remains the same as that of the ORIGIN object. With prefetching, the token of RELATED objects are acquired along with the ORIGIN object and hence the values in the fourth column are not applicable. The values in the fifth column therefore include only the time needed to perform operation on the object.

Further, Table 7.2 describes the time for the first access of the ORIGIN object for various values of n with different correctness of prediction. Figure 7.6 shows the same values in the graph.

7.8 Analysis of Prefetching Model

We have observed the fact that efficiency of prefetching mechanism depends on two factors– correctness of dependency prediction and size of object relation group. We have analyzed efficiency in terms of reduction in processing time of command and traffic between OSD client and manager.

	Origin Object		Related Object	
	Credentials	Total	Credentials	Total
no prefetch	661.13	2341.38	661.13	2341.38
p=1	3821.94	5542	-	1700.2
p=0.85	4530.34	6352.81	-	1700.2
p=0.66	5287.64	7032.09	-	1700.2
p=0.58	6034.5	7780.23	-	1700.2
p=0.5	6680.71	8460.74	-	1700.2

Table 7.1: Time in Microseconds for OSD Accesses with and without Prefetching when number of useful object in the group is kept constant at 10.

	n=10	n=20	n=30	n=40	n=50
no prefetch	2341.38	2300.52	2266.72	2247.78	2350.81
p=1	5542	8361.03	11265.44	14060.66	16336.90
p=0.87	6352.81	9960.79	13421.05	17137.15	21385.39
p=0.66	7032.09	11343.72	15192.26	19595.56	23786.81
p=0.58	7780.23	12891.47	17936.38	24306.91	31254.94
p=0.5	8460.74	14051.01	19571.29	25964.86	32679.9

Table 7.2: Command Execution Time in Micro-seconds for Origin Object

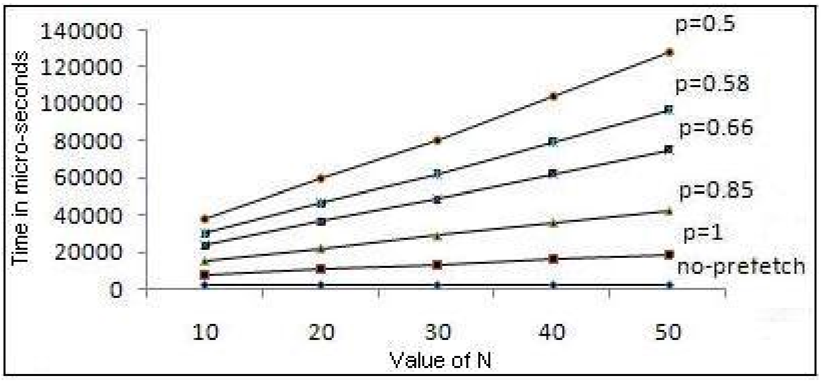


Figure 7.2: Overall Command Time for First Access of ORIGIN Object for *n*.

We notice that the prefetching reduces traffic between the OSD client and manager. In terms of time needed to process the OSD requests, the prefetching results in reduction of time because the credential requests are not sent to the security manager. However, there are additional overheads associated with processing of credential requests of the ORIGIN objects for generating prefetch data. Results can be improved dramatically when client participates in the process of defining relations. This imposes need of smart client for feedback of objects access.

Table 7.1 shows advantages of prefetching for access of RELATED objects. At the same time, Table 2 describes overheads of prefetching in acquisition of the ORIGIN object for the first time.

With larger values of n, these overheads increase. However, as indicated earlier, the overall time for all the objects is expected to be lower as can be seen in Table 7.1.

For example, with $p_d = 0.66$, total time for getting 50 objects (the number of useful objects in the relation group with 1 ORIGIN and 49 RELATED) is $23786.81 + 49 * 1700$ or $107086\mu s$. Without prefetching, this time becomes $50 * 2350.8$ or $117540\mu s$ with an overall effective gain of about 10%.

Chapter 8

IBM Object Controller (OC)

Object store [40] team at IBM Haifa Lab in Israel is working on the implementation of various components of OSD system.

IBM object based controller (OC) is a part of object store project. This was designed to provide object storage device capabilities on top of a block-based storage controller. OC works over the iSCSI protocol and runs on a standard Linux server.

OC originally did not support collection objects. We introduced this support to the OC project and tested efficiency of our server side prefetching mechanism as discussed earlier.

8.1 OC Architecture

OC is a user mode program consisting of sub-modules such as iSCSI target module, T10 front end, OSD module and OSD-target-FS as shown in figure 8.1.

OC has in-built implementation of iSCSI target module. The iSCSI target module provides support for exporting OSD LUNs as SCSI devices to the iSCSI initiator. T10 front end is a communication interface between iSCSI target module and OSD module and provides compliance with T10 standard. OSD module performs role of OSD command interpreter. It accepts OSD commands from iSCSI target and implements them on OSD-target-FS.

OSD-target-FS module implements a user mode file system. This file system accepts commands from OSD module and converts them into block operation. OC allows exporting of any file or device as OSD device. The block operations are implemented on the given file or device.

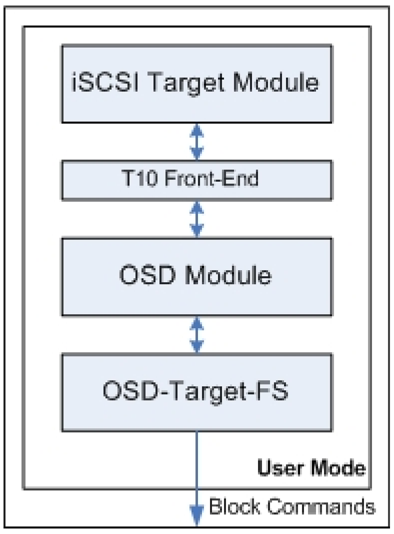


Figure 8.1: OC Modules

8.2 Differences between OC and IITK-OSD-SIM

OC implements block based storage controller while IITK-OSD-SIM implements file based storage controller. OSD module of OC passes OSD command to OSD-target-FS which maps them to block operations. In case of IITK-OSD-SIM, the OSD commands are mapped onto file operations which are performed on the files associated with the object. OC comes with built-in support of iSCSI module and OSD-target-FS. These modules communicate internally with the OSD module. This saves kernel-user context switches and makes OC faster than IITK-OSD-SIM. This fact is observed also in results as described later.

In OC, OSD target is a mix of iSCSI target module and OSD module while IITK-OSD-SIM provides a clear separation of these modules. This separation allows OSD simulator (section 3.1) module of IITK-OSD-SIM to work with any iSCSI target software. As compared to this the OC provides a tighter integration between OSD simulator and iSCSI target. To make OC work with new iSCSI target software requires major changes in the code.

8.3 Collection Support in OC

Collection support in OC required to implement support of collection commands and handling of operations to add/remove objects from collection as defined in T10 standards. In our implementation, we also provide additional support for collection hierarchy.

T10 standard supports three collection commands – CREATE COLLECTION, REMOVE COLLECTION and LIST COLLECTION. It however does not specify any command for addition or removal of objects from collection. In OC, we have implemented these operations using get/set attribute commands.

In our implementation, we allowed to create hierarchy among objects. Defining hierarchy allows to have better classification of objects. It is useful in many applications where we want intelligent OSD target managing low level storage and operations on the objects according to their quality of service and access behavior.

8.4 Results

We discussed the testing of server side prefetching model in section 7.8 where we used IITK-OSD-SIM as OSD target. We carried out same tests with OC as OSD target.

In our test procedure, the initiator side setup is same as mentioned in section 7.8. On the target side, we use OC with Linux kernel 2.6 running on Intel 2.8 GHz CPU, 256 MB RAM and 100 Mbps network.

The following experiments were performed which are the same as those performed on IITK-OSD-SIM based OSD target.

- Object Access Operation: OSD WRITE of 100 bytes. Write size is kept small so that the effects of metadata and credentials operation are prominent compared to time needed to transfer the actual data.
- Average number of distinct objects that are likely to be accessed together by the client at particular instance (n) is varied in various experiments. We have performed experiment, for

different values of n. Values of n are 10, 20, 30, 40, 50. Experiment is repeated 10 times for each value of n.

- Total number of accesses per object is taken as 5.

Experiments are performed for no-prefetch and different values of probability as 0.5, 0.58, 0.66, 0.86, and 1.

The results are presented in table 8.1 and 8.2. Table 8.1 shows time for OSD object accesses without prefetching and prefetching with different values of probability when number of useful object in the group are 10. The second column provides the time (in microseconds) to acquire the credentials by the OSD client for the ORIGIN object. The third column of the table provide the time (in microseconds) for the entire OSD operation for ORIGIN object. The fourth and the fifth columns provide the same values for the RELATED objects. This table is same as table 7.1.

As mentioned earlier, it is clear that without prefetching, the RELATED objects are treated independently and hence the credentials for these objects are acquired independent of the ORIGIN object. The time for the credential acquisitions remains the same as that of the ORIGIN object. While with prefetching, the token of RELATED objects are acquired along with the ORIGIN object and hence the values in the fourth column are not applicable. The values in the fifth column therefore include only the time needed to perform operation on the object.

Further, table 8.2 describes the time for the first access of the ORIGIN object for various values of *n* with different correctness of prediction. This table is same as table 7.2. Figure 8.2 shows the same values in the graph.

	Origin Object		Related Object	
	Credentials	Total	Credentials	Total
no prefetch	593.84	1985.36	593.84	1985.36
p=1	3402.1	4868.87	-	1410.22
p=0.85	4146.42	5524.31	-	1410.22
p=0.66	4602.727	5978.12	-	1401.22
p=0.58	5537.93	6920.72	-	1401.22
p=0.5	5936.571	7328.72	-	1401.22

Table 8.1: Time for OSD Accesses with and without Prefetching when number of useful object in the group is kept constant at 10.

	n=10	n=20	n=30	n=40	n=50
no prefetch	1985.36	1998.07	2019.71	2039.56	2023.21
p=1	4868.87	7424.55	9800.86	12479.62	15274.7
p=0.87	5524.31	8956.57	12223.13	15465.41	19186.44
p=0.66	5978.12	10161.46	13784.46	18549.18	22946.36
p=0.58	6920.72	11625.55	16634.33	21552.49	27119.591
p=0.5	7328.72	13272.1	18832.52	24335.136	30255.31

Table 8.2: Command Execution Time in Micro-seconds for Origin Object

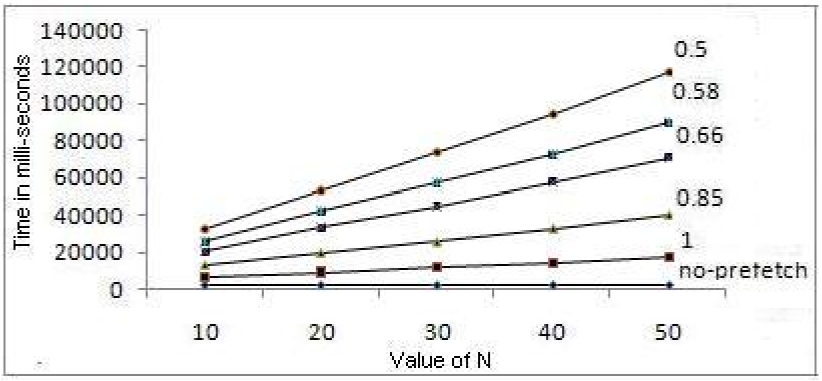


Figure 8.2: Overall Command Time for First Access of ORIGIN Object for *n*.

8.4.1 Analysis of results

Since experiments are carried out for testing of prefetching implementation at initiator side, behavior of tests on OC is almost to that of IITK-OSD-SIM as given in section 7.8.

Major difference is with respect to the time for processing of command after acquisition of credentials. It is clear that OC is about 20% faster than IITK-OSD-SIM. This, as explained earlier, is because OC has a built-in mapping between the objects and disk blocks while IITK-OSD-SIM implements objects as files local to the host.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

OSD enables creation of self-managed, heterogeneous, shared storage by moving low-level storage functions into the storage device itself. Major advantage of the OSD is that it combines direct access feature of SAN with security advantages of NAS. Object is exposed as a storage unit with abstraction for data and metadata. It provides security at object granularity.

We discussed the design and implementation of IITK OSD simulator with a few unique features such as support for exporting unlimited number of OSD LUNs with a single instance of iSCSI target, support for collection objects etc. We have presented design and implementation of an OSD client and manager.

We discussed prefetching techniques according to method of access analysis and location of implementation and addressed the need of prefetching of credentials in OSD system to minimize delay in the object access. We discussed the use of graph-based techniques for prediction of future accesses from the analysis of past accesses. Our algorithm works on object access graphs and generates sets for prefetch data. We discussed the integration of this model at OSD client for prefetching of the credentials from OSD manager.

We also implemented server side prefetching mechanism based on identifying relations among objects. We identified various ways for defining relations among objects and making better prediction of required data. We also implemented a mechanism for transferring partial credentials of prefetched objects for optimization of the response traffic. This makes prefetching model effective even when only 50% of prefetched data is useful.

We tested our server side prefetching model on IITK OSD simulator and IBM object controller (OC). For this we also extended OC to support collection objects mechanism.

9.2 Future Work

IITK-OSD-SIM is an implementation of file based object simulation where OSD commands are mapped to the file operations and performed on a file system local to the host. It is possible to implement an in-built block mapping from the objects in the IITK-OSD-SIM like OC which will make these object operations faster.

The OSD system make it feasible to use strategies of management and storage at OSD target side according to the object category. Objects can be categorized in terms of access type (random access/sequential access) and access frequency (more frequent/less frequent). OSD storage component can then do better storage management of objects according to the category. For example, frequently accessed objects can be stored in the faster media zones of hard-disk.

Server initiated and client initiated prefetching models can be implemented in the proxy environment. Further, prefetching models, described in the thesis, can be implemented for object prefetching to reduce data traffic between OSD client and OSD target.

Bibliography

[1] T10 Technical Committee. Information technology-SCSI Object-Based Storage Device Commands-2 (OSD-2). Working draft, project T10/1729-D, Revision 2., July 2007. *http://www.t10.org/ftp/t10/drafts/osd2/osd2r02.pdf*.

[2] Eric Riedel. OSD Architecture and Systems. SNIA Technical Tutorial. April 2006. *http://www.snia.org/education/tutorials/2006/spring/storage/Object-based_Storage-Devices-OSD_Architecture_and_Systems.pdf*.

[3] Riyaz Shiraguppi. OSD Technology survey paper. CS697 course report. *http://www.cse.iitk.ac.in/gsdl/collect/cse/index/assoc/HASHb573.dir/doc.pdf*.

[4] SNIA Object-Based Storage Device Technical Working Group, Home Page. *http://www.snia.org/tech_activities/workgroups/*.

[5] ITtoolbox Wiki. Direct Attached Storage. *http://wiki.ittoolbox.com/index.php/Direct_Attached_Storage*.

[6] Heng Liao. Storage Area Network Architectures, Technology White Paper. PMC-2022178. April, 2003. *http://www.pmc-sierra.com/cgi-bin/document.pl?docnum=2022178*.

[7] Gibson, et.al., A Case for Network-Attached Secure Disks. CMU SCS Technical Report CMU-CS-96-142, September 1996. *http://www.pdl.cmu.edu/PDL-FTP/NASD/TR96-142.pdf*.

[8] B. Callaghan, B. Pawlowski, P. Staubach. NFS Version 3 Protocol Specification RFC-1813. *http://www.ietf.org/rfc/rfc1813.txt*.

[9] Sun Microsystems, Inc. NFS: Network File System Protocol Specification RFC-1094. *http://www.ietf.org/rfc/rfc1094.txt*.

[10] SCSI Architecture Model-3 (SAM-3). Project T10/1561-D, Revision 14. T10 Technical Committee NCITS, September 2004. *http://www.t10.org/ftp/t10/drafts/sam3/sam3r14.pdf*.

[11] IBM Inc., IBM OSD Initiator. *https://sourceforge.net/projects/osd-initiator/*.

[12] Stephen Smaldone, Aniruddha Bohra, and Liviu Iftode. FileWall: Implementing File Access Policies Using Dynamic Access Context. Workshop on Spontaneous Networking. May 2006. *http://olab.rutgers.edu/workshops/2006/helsinki/slides/smaldone.pdf*.

[13] E. E. Chang, R. H. Katz. Exploiting inheritance and structure semantics for effective clustering and buffering in an object-oriented DBMS. Proceedings of the ACM SIGMOD international conference on Management of data. Portland, Oregon, United States. 1989. pp. 348 - 357.

[14] Jung-Ho Ahn, Hyoung-Joo Kim. SEOF: An Adaptable Object Prefetch Policy for Object-Oriented Database Systems. Proceedings of the Thirteenth International Conference on Data Engineering. IEEE Computer Society Washington, DC, USA, 1997, pp. 4 - 13.

[15] G. H. Kuenning. The Design of the SEER Predictive Caching System. Proceedings of Mobile Computing Systems and Applications, Santa Cruz, CA, December 1994.

[16] IITK OSD Simulator Project Home Page. *http://iitk-osd-sim.sourceforge.net/*.

[17] INCITS Technical committee T10 Home Page. *http://www.t10.org/*.

[18] Panasas Inc Home Page. *http://www.panasas.com/*.

[19] Lustre Home Page. *http://www.lustre.org/*.

[20] Sage A. Weil, et. al., Ceph: a scalable, high-performance distributed file system. Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation - Volume 7. Seattle, WA. 2006. pp. 307 - 320.

[21] The Intelligent Storage Consortium in the University of Minnesota's Digital Technology Center (DISC). *http://www.dtc.umn.edu/disc/*.

[22] Object store project at IBM Haifa research Labs.
http://www.haifa.il.ibm.com/projects/storage/objectstore/.

[23] Intel's Open Storage Toolkit. *http://sourceforge.net/projects/intel-iscsi/*.

[24] Kevin KleinOsowski, Tom Ruwart, and David J. Lilja, Communicating Quality of Service Requirements to an Object-Based Storage Device. Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'05), Monterey, California USA. pp. 224-231.

[25] Garth A. Gibson and Rodney Van Meter. Network Attached Storage Architecture. Communications of the ACM, Nov. 2000, vol. 43, issue 11. ACM Press New York, NY, USA. pp. 37-45.

[26] Bill King. LUN Masking in a SAN. QLogic White paper 1.952.932.4000.
http://www.virtual.com/whitepapers/QLogic_LUN_Masking_In_A_SAN_wp.pdf.

[27] Chris King. Zoning: The Key to Flexible SAN Management. QLogic White Paper 022202/Rev C.
www.qlogic.com/documents/datasheets/knowledge_data/whitepapers/whitepapers.zoning.pdf.

[28] Gibson, et. al., File Server Scaling with Network-Attached Secure Disks. Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (Sigmetrics '97), Seattle, Washington, June 15-18, 1997. pp. 272-284.

[29] Gobioff, et. al., Security for Network Attached Storage Devices. CMU SCS Technical Report CMU-CS-97-185, 1997.

[30] Gibson, et. al., Filesystems for Network-Attached Secure Disks. CMU SCS Technical Report CMU-CS-97-118, 1997.

[31] Gibson, et. al., NASD Scalable Storage Systems. USENIX99, Extreme Linux Workshop, Monterey, CA, June 1999.

[32] Andrew Leung, Ethan L. Miller, Stephanie Jones. Scalable Security for Petascale Parallel File Systems. Proceedings of SC '07, November 2007.

[33] Andrew Leung. Scalable Security for High Performance, Petascale Storage. Technical Report UCSC-SSRC-07-07, June 2007.

[34] Feng Wang. Storage Management in Large Distributed Object-Based Storage Systems. Ph.D. thesis, University of California, Santa Cruz, December 2006.

[35] Sage Weil, et. al., Ceph: A Scalable Object-Based Storage System. Technical Report UCSC-SSRC-06-01, March 2006.

[36] Christopher Olson, Ethan L. Miller. Secure Capabilities for a Petabyte-Scale Object-Based Distributed File System. Proceedings of the ACM Workshop on Storage Security and Survivability (StorageSS 2005), November 2005. pp. 64-73.

[37] Girish Moodalbail, et. al., Backup Aware Object based Storage. DTC Research Report 2007/25, June 2007.

[38] David Du, et. al., Experiences in Building an Object-Based Storage System based on the OSD T-10 Standard, Proceedings of the IEEE Conference on Mass Storage Systems and Technologies, MSST, 2006, College Park, Maryland, USA.

[39] Ying-ping Lu, et. al., QoS Provisioning Framework for OSD-based Storage, Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'05), 2005, Monterey, California USA. pp.28-35.

[40] Object Store Project at IBM Haifa research Lab Home Page.
http://www.haifa.ibm.com/projects/storage/objectstore/.

[41] Dave Hitz, et. al., File system design for an NFS file server appliance. Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference,1994, San Francisco, California USA. pp. 235 - 246.

[42] James Griffioen, KY Randy Appleton. Reducing file system latency using a predictive approach. Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1. Boston, Massachusetts. pp. 197 - 208.

[43] Peng Gu, et. al., Nexus: A Novel Weighted-Graph-Based Prefetching Algorithm for Metadata Servers in Petabyte-Scale Storage Systems. Proceedings of the Sixth IEEE International

Symposium on Cluster Computing and the Grid (CCGRID'06) - Volume 00. Singapore. 16-19 May 2006. pp. 409 - 416.

[44] Pallis G., et. al., A Clustering-based Prefetching Scheme on a Web Cache Environment. International Journal Computers & electrical engineering, Elsevier, 2007.

[45] Carl D. Tait et al., "Detection and Exploitation of File Working Sets," International Conference on Distributed Computing Systems, IEEE, Arlington, Texas, USA. May 20-24, 1991. pp. 2-9.

[46] Cluster File System Inc Home page. *http://en.wikipedia.org/wiki/Cluster_File_Systems*.

[47] Sun Microsystems Inc Home Page. *http://www.sun.com/*.

[48] The iSCSI Enterprise Target Project Home Page. *http://iscsitarget.sourceforge.net/*.

[49] SNIA Europe IP Storage Initiative. IP Storage Technology Paper.
http://www.snia-europe.org/downloads/ipstorage/IP_Stor_Tech_paper_Final.pdf.

[50] Information technology - SCSI Primary Commands - 4 (SPC-4). Project T10/1731-D, Revision 14. T10 Technical Committee NCITS, March 2008.
www.t10.org/ftp/t10/drafts/spc4/spc4r14.pdf.

[51] Douglas Gilbert. The Linux 2.4 SCSI subsystem HOWTO.
http://sg.torque.net/scsi/SCSI-2.4-HOWTO/.

[52] NetLink Sockets. Linux Man Pages. *http://linux.die.net/man/7/netlink*.

[53] GNU General Public License. *http://www.gnu.org/licenses/gpl.html*.

[54] R.C. Prim. Shortest connection networks and some generalizations. Bell System Technical Journal 36 (November 1957), pp 1389-1401.