

Implementing and Evaluating Security Controls for an Object-Based Storage System

Zhongying Niu[†] Ke Zhou^{†*} Dan Feng[†] Hong Jiang[‡]
niel@smail.hust.edu.cn k.zhou@hust.edu.cn dfeng@hust.edu.cn jiang@cse.unl.edu
Frank Wang[§] Hua Chai[†] Wei Xiao[†] Chunhua Li[†]
f.wang@cranfield.ac.uk chsal@163.com showin@163.com li.chunhua@163.com

[†] Computer College, Huazhong University of Science & Technology, Wuhan, China
Wuhan National Laboratory for Optoelectronics, China

[‡] Department of Computer Science and Engineering, University of Nebraska-Lincoln

[§] Center for Grid Computing, Cambridge-Cranfield High Performance Computing Facility, U.K.

Abstract

This paper presents the implementation and performance evaluation of a real, secure object-based storage system compliant to the T10 OSD standard. In contrast to previous work, our system implements the entire three security methods of the OSD security protocol defined in the standard, namely CAPKEY, CMDRSP and ALLDATA, and an Oakley-based authentication protocol by which the Metadata Server (MDS) and client can be sure of each other's identities. Moreover, our system supports concurrent operations from multiple clients to multiple OSDs. The MDS, a combination of security manager and storage/policy manager, performs access control, global namespace management, and concurrency control.

We also evaluate the performance and scalability of our implementation and compare it with iSCSI, NFS and Lustre storage configurations. The overhead of access control is small: compared with the same system without any security mechanism, bandwidth for reads and writes with the CAPKEY and CMDRSP methods decreases by less than 5%, while latency for metadata operations with any of the security methods increases by less than 0.3 ms (5%). The system with the ALLDATA method suffers a higher performance penalty: large sequential accesses run at 46% and 52% of the maximum bandwidth of unsecured storage for reads and writes respectively. The aggregate throughput scales with the number of OSDs (up to 8 in our experiments). The overhead of the SET KEY commands for partition and working keys refreshed frequently is less than 2 ms.

* Corresponding author.

1. Introduction

It is desirable for storage systems to have the following five features: security, cross-platform data sharing, high performance, scalability and easy management. Because of the various limitations of DAS (Direct Attached Storage), NAS (Network Attached Storage) and SAN (Storage Area Network), it is difficult, if not impossible, for a storage system to have these five features simultaneously. The file-based NAS utilizes the file interface to provide secure capabilities and cross-platform data sharing, while the block-based SAN provides high performance through high-speed data access. By combining the advantages of high-speed and direct-access of SANs and the data sharing and security capabilities of NAS, and moving low-level storage functions of the file system into the storage device itself and accessing the device through a standard object interface, Object-Based Storage (OBS) enables a scalable, high-performance, cross-platform and secure data sharing architecture.

Owing to the fact that objects have their own attributes, OBS can set up flexible security mechanisms on the basis of objects that constitute the primary storage units. OBS can assign different security attributes to the whole device, a group of objects, individual object, or even extensive data in an object, and implement authentication separately. Moreover, OBS can also divide the whole system into several partitions whose security attributes and rules of accessing are decided based on applications, and authorize every I/O operation. Thus OBS can offer a higher level of security than the NAS and SAN systems.

In the past few years, object stores have received more and more attention, and most of the work has focused on the problem of security for object stores. Meanwhile the T10 Technical Committee of INCITS has made continued

efforts for the standardization of the object interface and the first version of the object based storage interface standard (also referred to as T10 OSD standard) [12] was ratified by ANSI in January 2005. The T10 OSD standard presents the OSD security protocol and its three security methods, namely CAPKEY, CMDRSP and ALLDATA, but very few OSD implementations are compliant to the OSD standard. In a recent paper [2], Du, et al. presented their experiences with the implementation of the T10 OSD standard. However, their implementation was preliminary and further work needs to be undertaken to demonstrate the advantages of the object based technology. They implemented a preliminary security manager that can hand-out capabilities to users and perform some preliminary key management tasks while the security manager did not authenticate users; it assumed that users are already authenticated using any of the standard mechanisms such as Kerberos. The metadata server, that is essential in separating the data and control path, was not implemented in their system. They presented performance analysis of their implementation and compared it with iSCSI and NFS, while their experiments were based on one client and one OSD and the evaluation results failed to fully demonstrate the advantages of the object based technology.

In this paper, we present an implementation of a real, secure object-based storage system compliant to the T10 OSD standard. In contrast to the work done by Du et al., our system implements the entire three security methods of the OSD security protocol, and an Oakley-based authentication protocol by which the Metadata Server (MDS) and client can be sure of each other's identities. Moreover, our system supports concurrent operations from multiple clients to multiple OSDs. The MDS, a combination of security manager and storage/policy manager, performs access control, global namespace management, and concurrency control. We also evaluate the performance and scalability of our implementation and compare it with iSCSI, NFS and Lustre storage configurations. We believe that our work involved in the implementation and performance evaluation of the standard will be valuable to the OSD research and development community at large.

The rest of the paper is organized as follows. Section 2 presents an overview of the OSD security protocol. Section 3 introduces the authentication protocol. Section 4 describes the system design and implementation. Section 5 presents the performance methodology and discusses the evaluation results. Section 6 covers related work. And, finally, Section 7 concludes the paper and points out directions for future work.

2. Overview of the OSD security protocol

2.1. Security model

The OSD security model consists of four components: (a) application client, (b) security manager, (c) policy/storage Manager, and (d) Object-Based Storage Device (OBSD).

An application client wishing to access a file must request a capability from the security manager. After receiving the capability request, the security manager contacts the policy/storage manager to get a capability including permission. If the operation is permitted, the security manager generates a credential including the requested capability and a capability key, which is returned to the application client. The capability key has been created with a key shared between the security manager and OBSD. When the application client gets the credential, it sends the capability as part of its request and a request integrity check value generated with the received capability key to the OBSD. The OBSD validates the request, ensuring that the capability has not been tampered with, was rightfully obtained by the client, and that the requested operation is permissible by the capability.

The shared secret key between the security manager and OBSD for the authentication of the OSD commands is refreshed periodically. The key exchange protocol is accomplished via two commands, SET KEY and SET MASTER KEY.

2.2. Security method

To improve storage system performance over different network environments, the OSD security protocol defines four different security methods: NOSEC¹, CAPKEY, CMDRSP, and ALLDATA. The decision of which security methods to employ is left to the network environments over which the storage system runs. The CAPKEY method is used in a secure network environment (e.g., IPsec) to provide access control security, while the CMDRSP and ALLDATA methods are used in an insecure network environment to provide network security as well as access control.

In the three security methods of the OSD security protocol, the high security method includes the functionality of the low security method, e.g., the ALLDATA security method includes the functionality of CAPKEY and CMDRSP, and the CMDRSP security method includes the functionality of CAPKEY.

2.3. Credential revocation

The OSD security protocol enables two mechanisms for invalidating a credential: key exchange and policy access tag [4].

¹ We do not consider NOSEC a security method, since by definition it does not provide security.

The first mechanism, key exchange, is a coarse-grained approach that exchanges the key between the security manager and the OS. The security manager may invalidate credentials for an entire partition by using the SET KEY command to update the working keys used to compute the credential integrity check value in those credentials.

The second mechanism is fine-grained and invalidates all outstanding credentials for a given object by utilizing the object policy access tag. The object policy access tag is maintained both at the security manager and OS, and it is a settable object attribute. The policy access tag allows the coordinated actions of both the security manager and OS to prevent unsafe or temporarily undesirable utilization of OS. By modifying the value of an object policy access tag, the security manager or OS can invalidate all outstanding credentials for the object.

2.4. Key management

The OSD standard defines a hierarchy of four types of keys: master key, root key, partition key, and working key. Except the working key, each master, root, and partition key represents two secret key values: an authentication key and a generation key. The authentication key is used to compute the credential integrity check values; and the generation key is used by future SET KEY commands and SET MASTER KEY commands to compute the updated generation key and new authentication key values.

The key exchange protocol is accomplished via two commands, SET KEY and SET MASTER KEY, which are carried under the OSD security mechanism. For the SET MASTER KEY command, the Diffie-Hellman key exchange protocol [13] is implemented, thus achieving Perfect Forward Secrecy (PFS). PFS ensures that a given Master Key is not derived from any other secret. In other words, if someone breaks a key, PFS ensures that the attacker is not able to derive any other key.

3. Authentication protocol

The standard defines the security protocol between: 1) the OS and the client and 2) the OS and the security manager (primarily for key management). Communications between the client and the security manager is outside of the scope of the standard. Based on the Oakley key determination protocol [10], we implement an authentication protocol by which the security manager and the client can be sure of each other's identities without exchanging the share key and agree on a secret key that is immediately available for use in encrypting subsequent conversations. What's more, by the enhanced cookie algorithm the authentication protocol can reduce the winning probability of cookie attacks

effectively and by the clock synchronization information exchanged in the protocol the authentication protocol can implement the clock synchronization between a client and the security manager.

3.1. Cookie algorithm

The basic mechanism of the Oakley key determination protocol is the Diffie-Hellman key exchange algorithm [3] which allows two parties to agree on a shared value without requiring encryption. But some deficiencies exist in the Diffie-Hellman key exchange algorithm, e.g., having offered no information about identities of both sides, it is easy to be attacked by the go-between; the complexity of the algorithm makes it vulnerable to be attacked by inundation. The Oakley protocol retains the advantage of Diffie-Hellman algorithm while overcoming its shortcoming, preventing clogging through the Cookie exchange mechanism and preventing the go-between from attacking by authenticating the Diffie-Hellman exchange. The Oakley key determination protocol includes the following three steps: 1) Cookie exchange, 2) Diffie-Hellman half-key exchange, and 3) Authentication.

Because the Oakley protocol does not define the method by which cookies are produced, identical cookies can easily be forged by attackers. When the go-between captures the first cookie from the initiator, it can forge the response cookie. The attacker is closer to the initiator than the responder, so the initiator will receive the forged cookie first and refuse the legal cookie that arrives later. As a result, the Oakley protocol cannot establish connection between the initiator and the responder. The enhanced cookie algorithm is given below [18]:

- 1) The initiator gets local host IP and the shared key².
- 2) The initiator uses the first 64 bits of a hash which is generated over the local host IP and the shared key as the initiator cookie.
- 3) The initiator sends the cookie generated in step 2 to the responder.
- 4) The responder gets the shared key.
- 5) The responder computes the hash over the initiator IP and the share key, and then compares the first 64 bits with the initiator cookie. If the result is identical, the next step will be executed, otherwise a condition of being attacked will be notified.
- 6) The responder computes the hash over the local host IP and the shared key, and then sends the first 64 bits of the hash, i.e., responder cookie, to the initiator.
- 7) The initiator computes the responder cookie and compares the result with the responder cookie that has been received from the responder. If the result is

² The shared key is also used for the following authentication; it is dependent on the passphrases of the users.

identical, the next step will be executed, otherwise a condition of being attacked will be notified.

8) Enter the Diffie-hellman exchange stage.

Because the hash function is one-way function and the enhanced cookie algorithm relies on the shared key and the IP addresses of the initiator and responder, the go-between cannot forge the cookie. The most a malicious host could accomplish would be to mount a denial-of-service attack against a valid host, as opposed to successfully authenticating as a valid host. Also, the CPU cost of the hash function is small relative to the network speed, so the hosts would not be clogged even they have been attacked by inundation.

3.2. Authentication flow

Notations used for the protocol description are as follows:

1) Cookie-C and Cookie-M (or CKY-C and CKY-M) are 64-bit pseudo-random numbers. The generation method is defined by the cookie algorithm.

2) UID is the user identity to be used in authenticating the user.

3) ID(C) and ID(M) are the identities to be used in authenticating the client and MDS respectively.

4) GRP that can be pre-defined or user-defined is a name (32-bit value) for the Diffie-Hellman group used for the exchange.

5) g^x and g^y are encodings of the Diffie-Hellman group elements, where g is a special group element indicated in the group description and g^x indicates that element raised to the x'th power. g^{xy} is the eventually obtained Diffie-Hellman key.

6) N_c and N_m are nonces selected by the client and MDS.

7) KEYID is the concatenation of the client and MDS cookies; it is the name of keying material.

8) sKEYID is used to denote the keying material named by KEYID. It is the final secret key generated in the OAKLEY determination protocol and depends on g^{xy} , KEYID, N_c and N_m obtained from the Diffie-Hellman algorithm.

9) EHAO is a list of encryption/hash/authentication options. Each item is a pair of values: a class name (i.e., encryption, hash and authentication) and an algorithm name.

10) EHAS is a set of three items selected from the EHAO list, one from each of the classes for encryption, hash and authentication.

11) $S\{x\}k_i$ indicates the signature over x using k_i . Signing is done using the algorithm associated with an

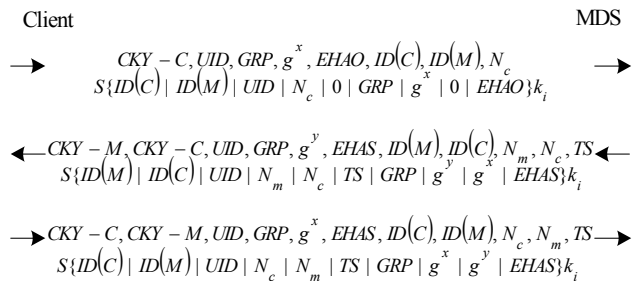


Figure 1. Authentication flow. authentication method; usually this will be RSA or DSS. The signature key k_i is the result of applying a one-way function to data $N_c | k$. Where k is the shared key between the client and MDS.

12) TS is the clock synchronization information sent back to the client by MDS.

13) $prf(a, b)$ denotes the result of applying pseudo-random function with key "a" to data "b".

Authentication Flow. Figure 1 shows the authentication flow. The client generates a unique cookie and associates it with the expected IP address of MDS, and its chosen state information: the group identifier GRP, a pseudo-randomly selected exponent x , EHAO list, nonce, identities. The first authentication choice in the EHAO list is an algorithm that supports digital signatures, and this is used to sign the message with the signature key k_i . The client further sets a timer for possible retransmission and/or termination of the request.

When MDS receives the message, it may choose to ignore all the information and treat it as merely a request for a cookie, and continue to exchange the key using the conservative mode of the Oakley protocol. In this paper we assume that MDS is more aggressive and accepts all the information offered by the client, i.e., group with identifier GRP, first authentication choice which must be the digital signature method used to sign the initiator message. Then MDS validates the signature with k_i generated by MDS with the shared secret k , and associates the pair (CKY-C, CKY-M) with the following state information:

- 1) User identity from the message.
- 2) The source and destination network addresses of the message.
- 3) The key state of "unauthenticated".
- 4) The first algorithm from the authentication offer.
- 5) Group GRP, a "Y" exponent value in group GRP, and g^x from the message.
- 6) The nonce N_c and a pseudo-randomly selected value N_m .
- 7) A timer for possible destruction of the state.

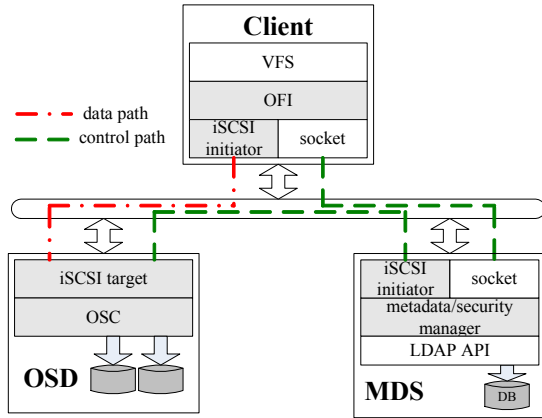


Figure 2. System model.

MDS computes g^y , forms TS with the concurrent system clock, forms the reply message, and then signs the ID , TS and nonce information with k_i and sends it to the client.

When the client receives the MDS message, and if the signature is valid, it will synchronize with MDS and send the confirmation message signed with the signature key k_i . In this step, the client and MDS compute $z = (g^y)^x = (g^x)^y = g^{xy}$ and associate it with the $KEYID = CKY - C | CKY - M$. The final result of this exchange is a key with $KEYID = CKY - C | CKY - M$ and value $sKEYID = \text{prf}(N_c | N_m, g^{xy} | CKY - C | CKY - M)$.

4. System design and implementation

Our object-based storage system consists of three components: OSD (Object-based Store Device), MDS (Meta Data Server) and clients. The three components are interconnected by IP network. As shown in figure 2, the grayed blocks are our reference implementation.

The client implements two kernel modules: the Object Filesystem Interface (OFI) and the iSCSI initiator driver. The OFI is implemented compliant with VFS like any other file systems on Linux. It receives the file operation commands from VFS and translates the file operations to object operations. The iSCSI initiator driver provides iSCSI transport to access remote iSCSI targets over IP networks.

OSD implements the SCSI OSD command sets compliant to the T10-OSD standard specification. It includes two kernel modules: the Object Storage Controller (OSC) and the iSCSI target driver. The iSCSI target driver accepts and decapsulates iSCSI PDUs from the iSCSI initiator driver and presents the decapsulated SCSI OSD commands to OSC. OSC processes the SCSI OSD commands and manages the physical storage media.

MDS executes as a user level process that implements

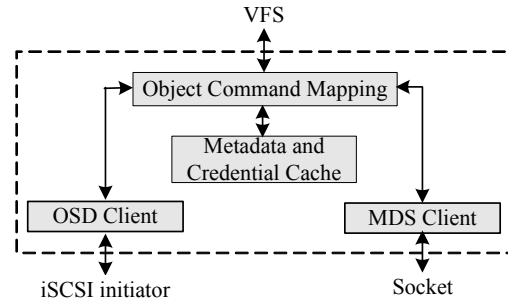


Figure 3. Object filesystem interface.

the functionalities of the Security Manager and Policy/Storage Manager. It includes the metadata/security manager and iSCSI initiator driver. The metadata/security manager is responsible for authenticating users, authorization, metadata maintenance, global namespace management, key management and workload balance; it presents the SET KEY and SET MASTER KEY commands to OSD through the iSCSI initiator driver.

4.1. Object filesystem interface

As depicted in Figure 3, OFI consists of four modules, namely, the Object Command Mapping (OCM), the OSD Client (OC), the MDS Client (MC) and the Metadata and Credential Cache (MCC). OCM translates file operations to object operations, gets metadata and credentials via MC and accesses OSD by OC. OC presents object commands to OSD through the iSCSI channel. MC accesses MDS through the TCP/IP protocol and implements an MDS command collection that we have defined in our system. MCC caches credentials and metadata in the client to improve the system performance, as evidenced in our experiments.

User processes access OFI through the Virtual Filesystem Switch (VFS), which offers user processes a unified, abstract and virtual file interface. After receiving the file operation commands from VFS, OCM searches the metadata and credentials related to the requested file in MCC. If MCC has cached the metadata and credentials, then OCM translates file operations to object operations and sends object commands to OC, or else MCC first requests the metadata and credentials through MC. Except for sending object commands, OC is responsible for transferring data to and from OSD. A data buffer in OC enables multiple small object writes to be “batched” into a single large network write.

4.2. Object storage controller

The OSD object abstraction is designed to repartition the responsibility for managing the access to data on a storage device by assigning the storage device additional responsibilities in the area of space management [12]. The

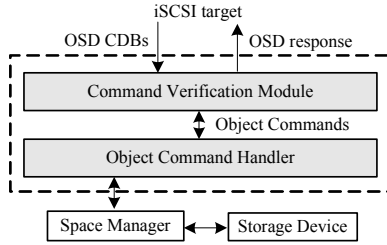


Figure 4. Object storage controller.

task of the traditional file system was divided into two parts: the file system user component and the file system storage component. The user component contains the functions of hierarchy management, naming and user access control. The storage component is focused on mapping file system logical view to the physical storage media.

In our case, the user component is assigned to MDS, and OSD takes the task of the storage management. MDS decides how files are mapped to objects and which OSDs serve to store the objects. There is one root object for each OSD logical unit and it is the starting point for the navigation of the structure on the OSD logical unit. There are any number of partition objects and user objects for an OSD logical unit, while collection objects are not supported in our current implementation. All partition objects and user objects are managed by the local filesystem; the partition objects are mapped onto directories and the user objects are mapped to files. As the functions of hierarchy management, naming, and user access control are managed by MDS, all user objects that belong to the same partition are on an equal footing with each others, and there is no hierarchical relationship (like traditional file systems) for objects. So OSD can only maintain a platform namespace for every partition, all user objects belonging to the same partition are stored in one directory.

As depicted in Figure 4, the Object Storage Controller (OSC) consists of two modules: the Command Verification Module (CVM) and the Object Command Handler (OCH). The CVM first checks the validity of the OSD CDBs from the iSCSI target and then presents the legal object commands to OCH. OCH translates object operations (e.g., read objects) to filesystem calls to finish object operations.

4.3. Metadata/Security manager

As shown in Figure 5, the metadata/security manager consists of five modules: the Authentication Module (AM), the File Command Handler (FCH), the Buffer Manager (BM), the Credential Generator (CG) and the Key Manager (KM). AM is responsible for authenticating users and implements the authentication protocol. An application client wishing to access the storage system must login first and negotiate a secret key with MDS for

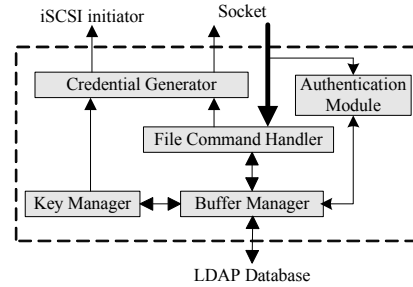


Figure 5. Metadata/Security manager.

use in encrypting subsequent conversations. After login success, the client sends the request to MDS, including the requested filename, operation code. FCH is responsible for mapping the file requests onto corresponding objects, and returns the object metadata, including the object identifier, operation rights, security policy etc. to CG. CG acquires the appropriate key from KM to generate a cryptographically secure credential for that object. BM calls the LDAP (Lightweight Directory Access Protocol) API to get the metadata from the LDAP Database and caches the recent metadata that is used frequently. A hash algorithm is implemented for fast retrieving the cached information.

KM is responsible for key management and refresh. When an OSD is added to the system the key hierarchy between OSD and MDS is built via the SET MASTER KEY and SET KEY commands committed by KM. During the use of OSD, the master key does not change unless the system administrator is changed. The root key refresh command is a privilege assigned to the administrator that manually refreshes the root key in case it is comprised. The partition key and working key are refreshed periodically according to the key refreshing policy. All the SET MASTER KEY and SET KEY commands are performed under the protection of the CMDRSP method.

The Hybrid Scheme for Object Allocation. In an object-based storage system, files are mapped into one or more data objects stored on OSDs. The policy for object allocation is a critical aspect affecting the overall systems performance. At present, most object allocation schemes employ one of two techniques [14]. The first one allocates a file to one device by using hashing functions that map file IDs to OSD IDs. This approach converts a file to one object and sends it to only one device. The second object allocation technique, called fragment-stripping or fragment-mapping, uses equal-sized fragments of each file to widely distribute the file among the OSDs. Both techniques have their advantages and disadvantages: the hashing approach achieves good load balance and provides rather high effectiveness in data allocation, but it can not provide readily available parallelism for large files; the fragment-stripping approach takes full advan-

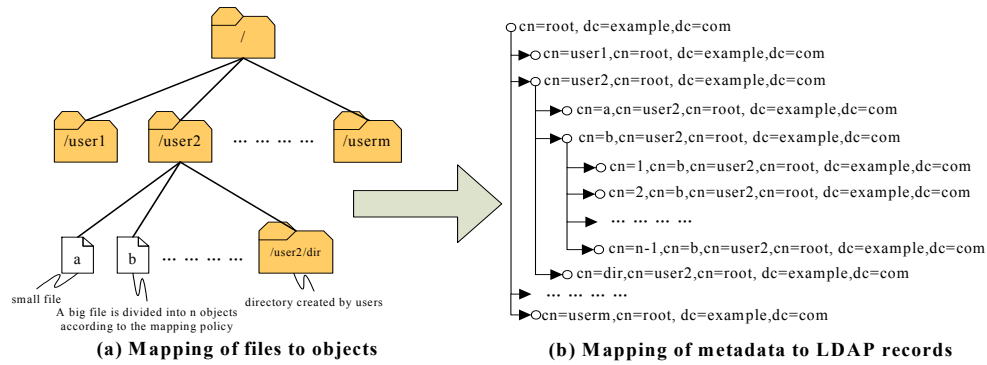


Figure 6. Mapping of files to objects and metadata to LDAP records.

tage of device parallelism, simplifies the clients' operations, but this policy is not fit for small files.

In our case, we employ a hybrid scheme of hashing and fragment-stripping for object allocation [14], which combines the best aspects of these two techniques while avoiding their disadvantages. The scheme converts a small file to a single object and a large file to multiple objects and each object will be distributed to an OSD (see Figure 6(a)). According to OBFS [15], the boundary of small and large files should be around 512KB. How many objects a file is divided into and which OSDs serve to store the objects are based on the size of the file, the total number of OSDs, the speed and free-capability of devices, as well as other parameters. The algorithm makes sure that an optimal number of objects is mapped from a file and the best-conditioned devices are used first. Figure 6 depicts the mapping of files to objects and metadata to the LDAP records. The /user_i is the work directory of the user(i) (see Figure 6(a)). The directory created by users and the object metadata are stored as records in the LDAP database (see Figure 6(b)).

4.4. Timing

In order to detect replays and expiration on capabilities, the clocks of OSD, MDS and client must keep some degree of synchronization. But the OSD protocol does not define the clock synchronization protocol.

By the Simple Network Time Protocol [9], we achieve clock synchronization between MDS and OSD. We run a SNTP sever at MDS and keep clock synchronization with the SNTP host located in the Internet. On each OSD a SNTP client synchronizes the OSD with the MDS.

When a client communicates with an OSD or MDS, the client must know the proper (secure) time value for generating request nonces. But not every remote client prefers to run the SNTP client in its own host. The client only needs to know the current value of the OSD clock but needs not to adjust the system clock to the network time. We maintain a secure clock value in the client by the authentication protocol. At the Diffie-Hellman half-key exchange stage MDS sends the current value of the MDS

clock, then at the client login period the time difference between the client system clock and the MDS system clock is used to adjust the request nonce timestamps. When OSD verifies the request nonce timestamps invalidation, a new time difference will be obtained according to the value of the OSD clock returned by OSD.

5. Performance evaluation

We ran experiments to evaluate the following: (1) the performance of our OBS system with the three security methods against an unsecured storage system; (2) system throughput and scalability under a bandwidth-intensive workload; and (3) the overhead of key refreshes.

Our experiments were conducted on 3 to 17 Super Micro SUPER X6DH8-XB nodes, each with one Intel 604-pin EM64T (NoconaTM) Xeon 3.0 GHz processor, 2MB L2 cache, PCI-E x8 slot (Physical x4), PCI-X-133 MHz slots, 800MHz Front Bus (FSB) and a total of 512MB PC2700 DDR-SDRAM physical memory. The machines are connected by a 1000 Mbit/s Ethernet switched through a Cisco Catalyst 3750-24PS switch. In addition, each node is connected to a Highpoint Rocket 2240 Raid controller attached to 15 SATA disks (300GB each). All machines run RedHat AS3 Linux kernel 2.4.21.

The configuration for our OBS system includes one MDS, several OSDs and clients, each using separate machines. Also, we compare the performance of our OBS system with those of iSCSI, NFS and Lustre. For all the above storage configurations, the same client-server machine combination was used, same disk partitions were used at the target or OSD to ensure the disk performance remains constant across all configurations. For iSCSI, we use the open-source Source Forge Linux iSCSI implementation (version unh-iscsi-1.6.00) and SCSI Target Mid-level for Linux (version sct-0.9.3) to set up the iSCSI configuration. Loading the initiator driver creates a SCSI device on the client. iSCSI performance is measured on a ext2 filesystem constructed on this SCSI device. For NFS, we use the default Linux implementation of NFS version 3 for our experiments and ext2 as the default filesystem. The NFS daemon was set up on the target and exported a share

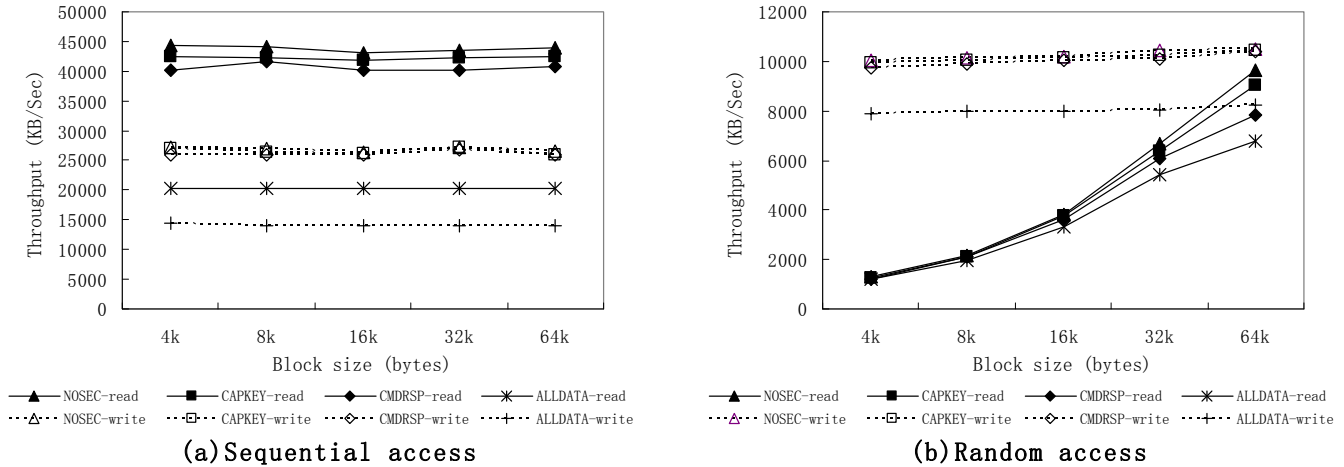


Figure 7. Performance of three methods and the baseline.

directory in the common test partition on the target. For Lustre, we employ the stable version 1.4.5 for our experiments and run the MDS, an OST and a client on the three nodes respectively.

5.1. Performance evaluation for the different security methods

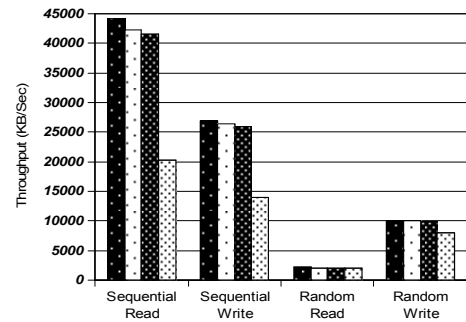
This section compares the performance of our OBS implementation with the three security methods against its unsecured counterpart. The motive of this experiment is to see how much performance degradation is incurred when security protocol overhead is added to an object-based storage system.

We ran the Iozone benchmark on the client. File size was set to 1Gbytes, using record sizes from 4Kbytes up to 64Kbytes. For both files and records, each size increases will simply be a doubling of the previous value. The metric for evaluating the Iozone performance is the average number of megabytes read and written per second with two access patterns: random and sequential.

Figure 7 shows the performance of a one client and one OSD system. As expected, the random accesses are slower than sequential accesses. For the sequential accesses there is only a marginal difference among different block sizes; for the random reads as the block size is increased, the performance is enhanced. But for the random writes as the block size is increased, the performance is few enhanced. We attribute the primary reason to the data buffer in the clients of our system, which enables multiple small objects writes to be “batched” into a single large network write.

In the next paragraphs, we used the performance measurements of the NOSEC method as a baseline for our other performance measurements, showing the effect of the OSD security protocol on object-based storage system performance for each security method.

Figure 8 shows the performance of all three security



	Sequential Read	Sequential Write	Random Read	Random Write
■ NOSEC	44198	26960	2184	10127
□ CAPKEY	42342	26364	2115	10019
■ CMDRSP	41601	25946	2110	9893
□ ALLDATA	20346	14042	1978	7962

Figure 8. Performance of three security methods and the baseline for 8KB blocks.

methods and the baseline (unsecured storage system) in a system using 8KB blocks. We chose 8KB blocks because, although many current UNIX systems use 4 KB blocks, we believe that 8KB (or even larger) is an appropriate size in future storage systems. As shown, the protocol suffers primarily because it was designed to confirm all data transmitted between the client and the OSD.

In a system dominated by small random reads, any of the security methods would be acceptable, and would not reduce performance significantly. In systems with many sequential operations or even a moderate number of writes, however, only the CAPKEY and CMDRSP methods maintain performance within 95% of unsecured storage. To achieve data integrity³, the system with the ALLDATA method suffers a higher performance penalty: large sequential accesses run at 46% and 52% of the maximum

³ With openssl, it takes 743 μ sec to perform a HMAC-SHA1 operation for a block size of 64 bytes.

Table 1. Completion times for sub-operations.

Sub-operation	Completion time (μsec)
Generating a capability key	37
Generating a credential	45
Read a LDAP record	640
Read 10 LDAP records each time	880
Write a LDAP record	815
Delete a LDAP record	2750

bandwidth of the baseline for reads and writes respectively, while random writes run at 79% of the maximum bandwidth.

5.2. Latency of metadata operations

We ran a set of micro benchmarks on our OBS system and measured the latency of each metadata operation in order to evaluate the performance of MDS. All the latency benchmarks were run on a 7-level directory tree and there were 10 files in every directory, each of size 4KB. For directory operations, each operation (e.g., `mkdir` or `readdir`) was performed 10 times on every level directory. The file system was mounted before each operation started and unmounted before each operation completed. For file operations, each operation (e.g., `create` or `rm`) was performed on each of the files in one directory. We instrumented the MDS Client to gather the latency results. All numbers reported are the average of the 10 runs. We also instrumented the metadata server to report the time spent in fine-grained sub-operations. While we gathered the raw performance of metadata operations, the completion times of sub-operations were not recorded (To prevent the monitoring overhead from influencing performance results).

Table 1 reports the completion times for some sub-operations. The overhead is only 45 μsec for generating a credential and 37 μsec for generating a capability key. The low latencies generally translate to cache hits at the MDS. In case of cache misses, MDS must get capabilities from the LDAP database, which usually incurs greater latencies. The LDAP access latency for delete operations is 2750 μsec , which is larger than that for read and write operations. The overhead is 640 μsec and 815 μsec for reading and writing a record respectively. The results indicate that the batch reads can decrease the total overhead for accessing LDAP database, which cost only an extra 240 μsec for reading 10 records each time in contrast with reading one record from the LDAP database.

Table 2 depicts the measured latencies for the NOSEC, CAPKEY, CMDRSP and ALLDATA methods. The messages between MDS and clients are cryptographically hardened using the secret keys generated by the authentication protocol. The login operation involves both authentication and key negotiation, so it requires a significantly latency, roughly 20.5 ms. In contrast, the logout operation costs only 267 μsec .

Impact of Security Methods. First of all, we observe that the latencies for the directory operations (e.g., `mkdir`, `readdir` and `rmdir`) while using the CAPKEY, CMDRSP and ALLDATA methods are similar to the ones reported for the NOSEC method. This is because for the directory operations the similar access control is performed for any of the security methods. But the latencies for the file operations (e.g., `create`, `rm`, `write` and `read`) while using the CAPKEY, CMDRSP and ALLDATA methods increase by less than 0.3 ms (5%). This is because for the file operations the additional cryptographic overhead for generating capability keys incurred in the CAPKEY, CMDRSP and ALLDATA does not occur in the NOSEC. We also observe that the write and read commands have significantly lower latencies than other commands. This is because the write and read requests, after the first, will be serviced by the MDS cache. The `rm` and `rmdir` commands respectively incur a much larger latency than other file and directory operations due to the high overhead in deleting a LDAP record.

Impact of Directory Depth. The numbers shown in Table 2 gave a preliminary indication of the sensitivity of the latencies to the depth of the directory where the metadata operation was performed. For each operation except `readdir`, the latencies are comparable for directory depths of zero and three. By systematically varying the directory depth, we examine this sensitivity in detail. Figure 9 plots the directory operation latencies for varying directory depth. A directory depth of i implies that the operation is executed in `mnt_point:/dir1/.../diri`. For `readdir` commands, the latencies significantly increase as we increase the directory depth. This is due to the need to access the directory inode as well as the directory contents to construct the local directory tree. In contrast, it slightly increases with directory depth for `mkdir` and `rmdir` commands, since the directory content access and inode lookup is done by MDS.

5.3. Performance comparison of OBS, iSCSI, NFS and Lustre

In this section, we study the performance of our OBS system and compare it with those of iSCSI, NFS and Lustre. We ran the same benchmark that we had used in the first experiment to measure the performance of iSCSI, NFS and Lustre, while for our OBS system the performance measurements of the NOSEC method are employed. Figure 10 shows the throughput for sequential and random access. For sequential reads, our OBS system and NFS yield comparable performance and outperform Lustre and iSCSI, while for sequential writes, our system is outperformed by Lustre and NFS and the performance is better than that of iSCSI. For random accesses, our OBS system achieves a better performance for both reads and writes: the system outperforms Lustre for reads and yields comparable performance with iSCSI and NFS, while in

Table 2. Latency of metadata operations (μsec).

	Directory Depth 0				Directory Depth 3			
	NOSEC	CAPKEY	CMDRSP	ALLDATA	NOSEC	CAPKEY	CMDRSP	ALLDATA
mkdir	2952	3112	3031	3032	3052	3225	3235	3239
readdir	550	533	539	531	2182	2207	2160	2136
rmdir	4048	4291	4235	4236	4208	4327	4277	4270
create	4566	4651	4640	4611	4639	4693	4718	4756
rm	5303	5351	5459	5523	5346	5467	5470	5598
write	317	340	338	342	317	349	356	353
read	320	361	363	370	317	366	365	371
login	20512							
logout	267							

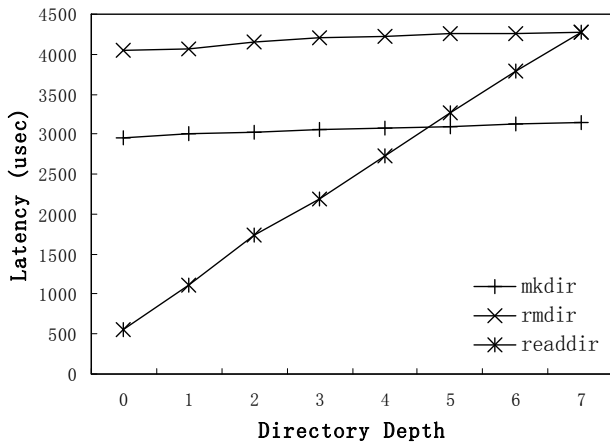


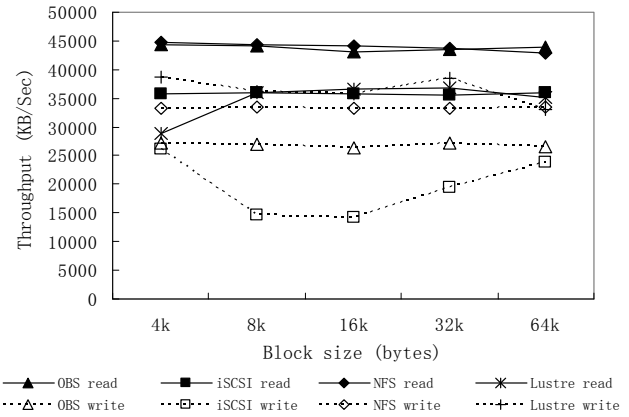
Figure 9. Effect of the directory depth on the operation latency.

write case, our system outperforms iSCSI and NFS and with the transfer size 16 KB or less, such performance is also better than Lustre.

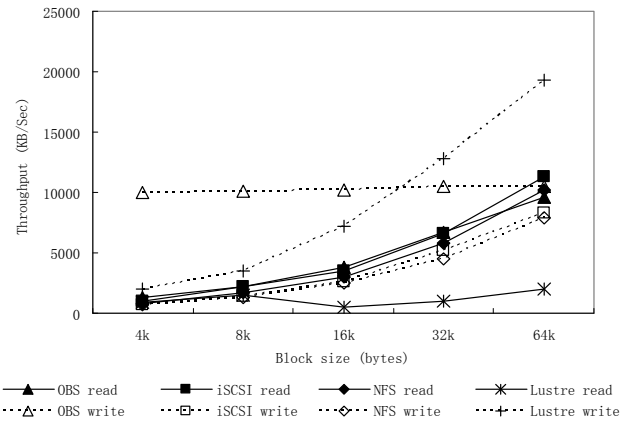
5.4. Aggregate throughput and scalability

In this section, we ran the Iozone benchmark to measure the throughput for various numbers of OSDs and clients, each running on a separate machine. The size of the test file was set to 1Gbytes and record size to 8Kbytes.

First, we varied the number of OSDs from 2 to 8 and measured the bandwidth of reads and writes by one client. The fragment-strip policy for object allocation was used in this experiment, i.e., the file was striped among all the OSDs. Figure 11 shows the system throughput as a function of the number of OSDs for the NOSEC, CAPKEY, CMDRSP, and ALLDATA methods. As expected, the throughput grows with the number of OSDs. The number of 4 delivers nearly the maximum performance permitted by a client for reads and the number of 8 for writes. We



(a) Sequential access



(b) Random access

Figure 10. Performance comparison of OBS, iSCSI, NFS and Lustre.

also observe that the throughput for the ALLDATA method grows slower than that for the NOSEC, CAPKEY and CMDRSP methods and the throughput surface is almost flat. This is because the overhead introduced by the ALLDATA method, which is dominated by MAC computation on data,

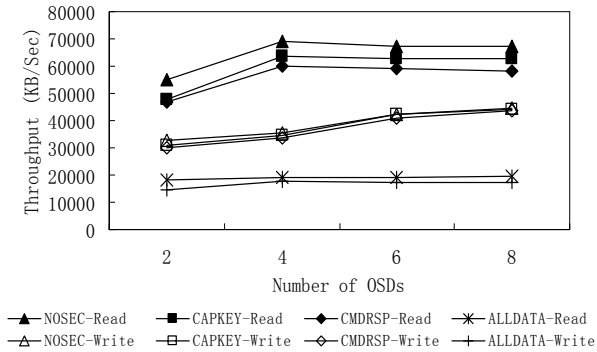


Figure 11. Performance of the single-client and multiple-OSD system.

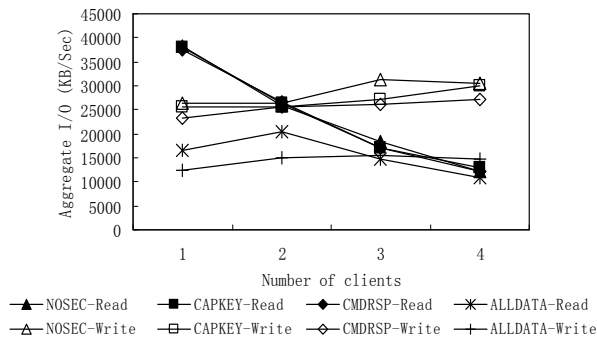


Figure 12. Aggregate bandwidth with 1 OSD.

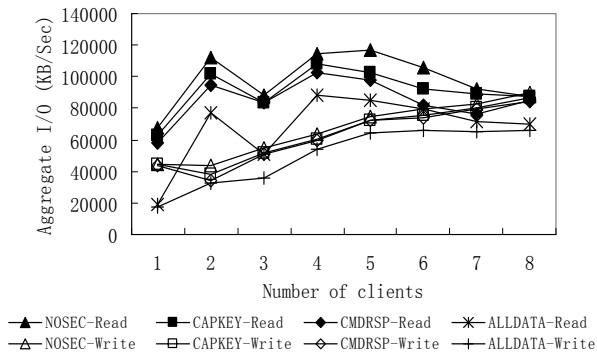


Figure 13. Aggregate bandwidth with 8 OSDs.

is high enough to limit the throughput of the client.

We also ran the Iozone benchmark in the throughput mode to measure the aggregate bandwidth by multiple clients for various numbers of OSDs. On each client, one Iozone process was active during the measurement and opened one file at a time for reads or writes. Figure 12 shows the aggregate bandwidth with 1 OSD. The aggregate write bandwidth increases with the number of clients, while the aggregate read bandwidth decreases with the number of clients. The decrease trend in the aggregate read bandwidth can be attributed to the overhead of concurrency control in OSD. Improving OSD is one of the main subjects that we identify as future work.

Table 3. Completion times for SET MASTER KEY and SET KEY commands.

Command		Completion time (μsec)
SET MASTER KEY		28725
SET KEY	Root	2962
	Partition	1922
	Working	1398

Figure 13 shows the aggregate bandwidth with 8 OSDs. According to the hybrid scheme for object allocation, each file was mapped to three objects and each object was distributed to an OSD. As shown in Figure 13, the aggregate write bandwidth increases with the number of clients, while the earlier trend that we observed in Figure 12, where the aggregate read bandwidth decreases with the number of clients, is weakened. One reason is that the system with a larger amount of OSDs can yield a higher aggregate bandwidth. Another primary reason is that the hybrid scheme ensures that one file is distributed to only several OSDs but not all, that is, one client is served by only several OSDs at a time. The hybrid scheme effectively balances the overhead among all of OSDs. We also observe that there is a bottom in the aggregate read bandwidth curve around the number of 3. This is because that multiple read requests to an OSD are emerging when the number of client is more than 2. As the number of clients scales up the aggregate bandwidth reaches the upper limit permitted by the OSDs and remains relatively constant. Although, the theoretical results in [14] show that for 1GB files the number of 40 is the optimal number of objects mapped from one file, Figure 11 shows that for reads the client yields the maximum performance when a file is distributed across 4 OSDs and above and for writes the number of OSDs for the maximum performance is 8 and above. Distributing a file across more OSDs can not improve the aggregate bandwidth; on the contrary, it brings extra overhead of concurrency operations to OSDs.

5.5. Overhead of key refreshes

In this section, we instrumented the metadata server to report the time spent for key refresh commands. There is one root object for each OSD logical unit, each with one master key and one root key. There is one partition object for an OSD logical unit, each with one partition key and 16 working keys.

Table 3 shows the completion times for SET MASTER KEY and SET KEY commands. All the commands are performed under the protection of the CMDRSP method. The SET MASTER KEY command involves the two steps, namely, SEED EXCHANGE and CHANGE MASTER KEY, as well as invalidating all the root, partition and working key, so it requires a significantly latency, roughly 28.7 ms. In contrast, the SET KEY command costs only 2962 μsec for a root key, 1922 μsec for a partition key and

1398 for a working key. The SET KEY command costs an extra 1564 μ sec for a root key than that for a working key due to an additional overhead for invalidating the partition key and the working key, which is dominated by accessing the LDAP database in our implementation. Similar observation is true for the SEY KEY command for a partition key. An additional overhead of 524 μ sec is introduced.

6. Related work

Object storage was first proposed in the Network-Attached Storage Devices project [5] at CMU and the two commercial products, Panasas [11] and Lustre [8], have been released in the commercial market. Also, the security of object stores has received significant attention in the past few years. Gobiuff, et al. [6] presented a cryptographic capability system for network-attached storage systems, built an NASD prototype implementation of a drive and adapted NFS and AFS to run with NASD drives.

The protocol presented by Azagury, et al. [1] defined a proprietary protocol for authorization on top of the secure communication layer while utilized a standard industry protocol for authentication, integrity and privacy on the communication channel (IPSec for IP networks).

Leung and Miller [7] proposed a coarse-grained capability security protocol for object-based storage systems. They have implemented the security mechanism for Ceph, UCSC's implementation of an OBS [16], and evaluated the performance and scalability of the protocol. Their numbers demonstrated that coarse-grained capabilities can effectively reduce the protocol overhead and client request latencies.

But these security solutions use proprietary protocols and hence limit interoperability. The OSD security protocol defined in the ANSI T10 OSD standard proposed three security methods for different application's requirements. Michael Factor et al. [4] further described the three methods of the OSD security protocol and analyzed their performance theoretically but they did so without performance testing. Du et al. [2] presented their experiences with the implementation of the T10 OSD standard; however, their implementation was preliminary and further work needs to be undertaken to demonstrate the advantages of the object based technology.

7. Conclusions and future work

In this paper we have presented the implementation and performance evaluation of a real, secure object-based storage system compliant to the T10 OSD standard. In contrast to previous work, our system implements the entire three security methods of the OSD security protocol defined in the standard and an Oakley-based authentication protocol by which the Metadata Server (MDS) and client can be sure

of each other's identities. Moreover, our system supports concurrent operations from multiple clients to multiple OSDs. The MDS, a combination of security manager and storage/policy manager, performs access control, global namespace management, and concurrency control.

We also evaluated the performance and scalability of our implementation and compared it with iSCSI, NFS and Lustre storage configurations. The overhead of access control is small: compared with the same system without any security mechanism, bandwidth for reads and writes with the CAPKEY and CMDRSP methods decreases by less than 5%, while latency for metadata operations with any of the security methods increases by less than 0.3 ms (5%). The system with the ALLDATA method suffers a higher performance penalty: large sequential accesses run at 46% and 52% of the maximum bandwidth of unsecured storage for reads and writes respectively. The aggregate throughput scales with the number of OSDs (up to 8 in our experiments). The overhead of the SET KEY commands for partition and working keys refreshed frequently is less than 2 ms.

We have focused our minds on enabling our system to seamlessly work with most of the applications in the real world. HUST [19], a massive storage system that was built at the Wuhan National Laboratory for Optoelectronics in China, integrates the object-based storage technology to provide support for GIS Grid and its applications. Based on HUST, we are investigating the access mode of the metadata as well as the characteristics of the data for optimizing our design. Also, we would like to implement the remaining OSD commands and make the upper-level applications benefit from the intelligent interface of object stores.

In OBS, files are mapped into one or more data objects. The policy for object allocation is a critical aspect affecting the overall systems performance. We employ the hybrid scheme for object allocation, while making an efficient approach is quite complex and the result of experimentation is quite different from the theoretical. We would like to revise the parameters in our algorithm and improve the scheme. CRUSH [17], a pseudo-random data distribution function used in Ceph, maps data objects to storage devices without relying on a central directory. We would also like to compare the overhead of our approach with that of CRUSH.

At present, object-based storage devices are designed to passively provide services and the predominance of the object store with intelligent interface and extended attributes is far from being displayed. We would like to push more intelligence into the OSD and make it more active. An active OSD can utilize the extended attributes of the object and redundant computing resource to provide active storage and to achieve self-managing, self-diagnosing and self-healing, so as to achieve higher availability.

Privacy must be considered in a long term extension of the OSD security system. There is no encryption in the

current version of the OSD standard. As a result, users must instead rely on encryption at different layers of the protocol stack (e.g., IPsec, application). However, we strongly believe that privacy is a core requirement for future storage systems. Therefore we are working on leveraging the associated infrastructure for encryptions and privacy mechanisms, defined by other standards to provide encryption for data in transit and store.

Acknowledgements

We thank the anonymous reviewers for their helpful comments and insights. This work was supported by the Key Basic Research Project under Grant No.2004CCA07400, National Science Foundation of China under Grant No.60503059, National Basic Research Program of China (973 Program) under Grant No.2004CB318201, the Program for New Century Excellent Talents in University NCET-04-0693 and NCET-06-0650, and the US NSF under Grant No. CCF-0621526.

References

- [1] Azagury, A., Canetti, R., Factor, M., Halevi, S., Henis, E., Naor, D., Rinetzky, N., Rodeh, O., and Satran, J. A two layered approach for securing an object store network. In *Proceeding of the 1st International IEEE Security in Storage Workshop*, pp. 10–23, 2002.
- [2] David Du, Dingshan He, Changjin Hong, Jaehoon Jeong, Vishal Kher, Yongdae Kim, Yingping Lu, Aravindan Raghuvveer, and Sarah Sharafkandi. Experiences in building an object-based storage system based on the OSD T-10 standard. In *Proceedings of the 23rd IEEE / 14th NASA Goddard Conference on Mass Storage Systems and Technologies*, College Park, MD, May 2006.
- [3] Diffie, W. and Hellman, M. New Directions in Cryptography. *IEEE Transactions on Information Theory*, V. IT-22, n. 6, June 1977.
- [4] Factor, M., Nagle, D., Naor, D., Riedel, E., and Satran, J. The OSD security protocol. In *Proceedings of the 3rd International IEEE Security in Storage Workshop*, pp. 29–39, 2005.
- [5] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 92–103, San Jose, CA, Oct. 1998.
- [6] Howard Gobioff, Garth Gibson, and Doug Tygar. Security for Network Attached Storage Devices. *Technical Report CMU-CS-97-185, Carnegie Mellon University, Pittsburgh, PA, 15213*, October 1997. <http://www.cs.cmu.edu/Web/Groups/NASD>.
- [7] Andrew W. Leung and Ethan L. Miller. Scalable Security for Large, High Performance Storage Systems. In *Proceedings of the 2nd ACM Workshop on Storage Security and Survivability*, October 2006.
- [8] Lustre. <http://www.lustre.org>.
- [9] D. Mills. *Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI*. RFC 2030, October 1996.
- [10] H. Orman. *The OAKLEY Key Determination Protocol*. RFC 2412, November 1998.
- [11] Panasas. <http://www.panasas.com>.
- [12] *SCSI Object-Based Storage Device Commands -2 (OSD-2)*. Project T10/1721-D, Revision 0. T10 Technical Committee, NCITS, October 2004.
- [13] Michael C. StJohns. *Diffie-Helman USM Key Management Information Base and Textual Convention*. RFC 2786, March 2000.
- [14] Fang Wang, Shunda Zhang, Dan Feng, Hong Jiang, Lingfang Zeng, and Song Lv. A Hybrid Scheme for Object Allocation in a Distributed Object-Storage System. In *Proceeding of the 6th International Conference on Computational Science*, UK, May 28-31, 2006.
- [15] Feng Wang, Scott A. Brandt, Ethan L. Miller, and Darrell D. E. Long. OBFS: A File System for Object-based Storage Devices. In *Proceeding of the 21st IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies*, College Park, MD, April 2004.
- [16] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, November 2006.
- [17] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06)*, Tampa, FL, Nov. 2006. ACM.
- [18] Fangjun Xie, Chunli Xie, and Zhi Gao. Analysis and mend of IKE protocol. *Computer Applications and Software*, pp. 103-104, 117, No. 9, 2004. (in Chinese)
- [19] Lingfang Zeng, Ke Zhou, Zhan Shi, Dan Feng, Fang Wang, Changsheng Xie, Zhitang Li, Zhanwu Yu, Jianya Gong, Qiang Cao, Zhongying Niu, Lingjun Qin, Qun Liu, Yao Li, and Hong Jiang. HUST: A Heterogeneous Unified Storage System for GIS Grid. *HPC Storage Challenge of Supercomputing*. Tampa, Florida, November 2006.