The ANSI T10 object-based storage standard and current implementations

D. Nagle
M. E. Factor
S. Iren
D. Naor
E. Riedel
O. Rodeh
J. Satran

Object-based storage is the natural evolution of the block storage interface, aimed at efficiently and effectively meeting the performance, reliability, security, and service requirements demanded by current and future applications. The object-based storage interface provides an organizational container, called an ***object****, into which higher-level software (e.g., file systems, databases, and user applications) can store both data and related attributes. In 2004, the ANSI (American National Standards Institute) T10 Standards body ratified an Object-based Storage Device (OSD) command set for SCSI (Small Computer System Interface) storage devices that implements the OSD interface. This paper describes the rationale for OSDs, highlights the ANSI T10 OSD V1.0 interface, and presents three OSD implementations: an OSD from Seagate, an IBM object-based storage prototype (ObjectStone), and the Panasas object-based distributed storage system.*

Introduction

Over the last 25 years, there has been a steady development in block storage interfaces: from the low-level, host-managed CHS (cylinder, head, sector) interface used in the original 5.25-inch disk drives (i.e., the ST-506 interface) to the LBA (logical block address) block interface common in current SCSI (Small Computer System Interface) and ATA (Advanced Technology Attachment) devices. However, inherent limitations in these block-based interfaces prevent them from meeting the performance, reliability, security, and service requirements demanded by applications.

The fundamental problem arises because block-based interfaces prevent storage devices from having information about any relationships among the data; essentially, a storage device considers each block as a unique container that may or may not be related to other blocks stored within the device. This makes it difficult for a storage device to optimize its internal resources. Moreover, the lack of access control of block-based devices renders them highly susceptible to security compromises, requiring other layers (e.g., networking) to provide security that, by its layered nature, is poorly matched to the needs of applications and users.

Object-based Storage Devices (OSDs) overcome these limitations by virtualizing physical storage into one or more objects. An object is a virtual container that higher-level software (e.g., file systems, databases, and user applications) can use in order to store both data and related attributes. This virtualization of storage allows OSDs to autonomously manage each object, using objects to abstract physical storage parameters while using attributes to interpret application-level requirements. As a result, OSDs optimize performance across storage resources and application parameters [1]. **Figure 1** compares traditional block-based storage architectures (left column) and the object-based storage architecture (right column).

In addition to virtualizing storage as objects, the OSD enables secure data access by incorporating a comprehensive access-control security protocol. The protocol requires that each command includes a cryptographically signed capability that specifies the object and allowable operations. A cryptographic

0018-8646/08/$5.00 © 2008 IBM

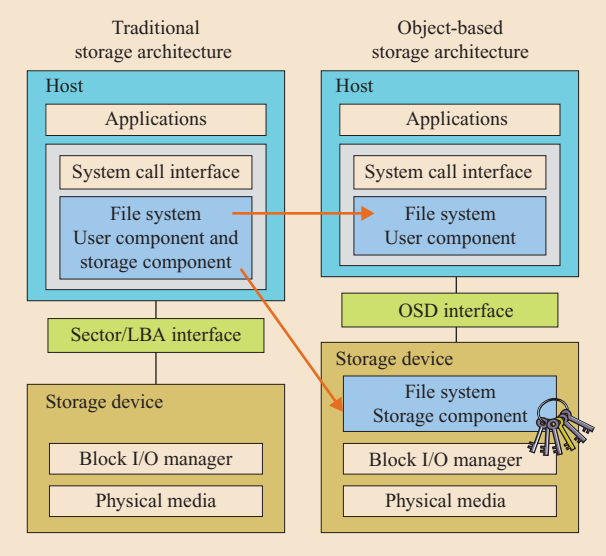

Figure 1

Evolving the storage interface. The figure indicates how the OSD migrates (lower arrow) a portion of the file system into the storage device while leaving the file system high-level policy functions (e.g., user authentication) to the server (upper arrow). A cryptographic mechanism (symbolized by the keys in the figure and generated using private keys stored in the OSD) is used to authorize client access. For more information, see Reference [2]. (OSD: Object-based Storage Device; LBA: logical block address.)

signature ensures that only authorized clients can access storage and only the object(s) to which the client has explicitly been granted access. This fine-grained security allows an OSD to become a first-class network citizen, that is, it is no longer restricted to isolated storage networks (e.g., SCSI or Fibre Channel) but able to sit directly on IP (Internet Protocol)-based networks (e.g., the Internet). The security also allows storage to be accessed by multiple independent and nontrusted clients.

This paper presents object-based storage, including the ANSI (American National Standards Institute) T10 OSD V1.0 standard and three implementations of OSD. We begin by making the case for object-based storage and then describe the basics of object-based storage, including its interface and command set. Next, we describe the OSD security model and three implementations of object-based storage from IBM, Seagate, and Panasas. Finally, we survey related work and present our conclusions.

Why evolve the storage interface

At the bottom of the storage protocol stack, commodity storage devices (i.e., SCSI and ATA devices) represent the device as a linear array of blocks. Each block maps to one or more physical sectors with device virtualization enabling automatic remapping of damaged sectors. Through higher-layer software abstractions (such as those associated with file systems and database systems), users perceive storage as a set of files in a file system or tables in a database. Each of these entities (e.g., files and tables) has an associated set of attributes (e.g., logical size, creation time, and access control) that must be managed, along with the data, by the underlying software structures.

Consider the typical file abstraction. To the user, a file is a linear set of bytes, associated attributes, and access controls as defined by the file system (e.g., Alice has read and write access to file "abc"). To store data, the file system maps the data and attributes of each file to one or more storage blocks. It then writes out the data, attributes, and the mapping (file to storage blocks) into a fairly sophisticated set of data structures that also reside on storage.

With all of the information stored within the device, one might expect that storage devices could leverage the information. After all, current storage devices already perform extensive processing, virtualizing block storage addresses to remap unusable sectors and automatically recover from I/O errors. However, block-based storage devices have no information about the relationship between blocks, making it impossible for the device to intelligently allocate its internal resources (i.e., computation, memory, network interface, and capacity). Instead, high-level software must crudely attempt to optimize physical characteristics and resource management policies of a device (e.g., read-ahead, caching, and disk queue scheduling) by guessing at the physical layout of disk blocks, the current head position, and internal queue lengths. For a few access patterns, these guesses work well. However, far too often, a large percentage of storage device performance or capacity is lost.

For example, commodity ATA disk drives often prefetch and cache entire disk tracks. This works well for sequential access workloads that read large chunks of data. However, for small, random I/O (e.g., one block) workloads, the extra time to read an entire track on each access can reduce performance by more than 50%. It is also common for disk drives to retry a failed read. However, for some deadline-driven applications, such as video, the loss of a data block is preferred to the delay incurred during a read retry. Block interfaces provide only limited means for applications to convey a deadline, forcing vendors to completely disable features that interfere with their applications requirements.

These difficulties exist since the current block storage device interface cannot convey semantic information between the application and the storage device. Of course, higher-level interfaces such as NFS (Network File

Table 1 Object-based Storage Device basic command set. The entire command set is documented in Reference [2].

Basic commands	*Security*	*Groups*	*Specialized commands*	*Management*
READ	SET KEY	CREATE COLLECTION	APPEND	FORMAT OSD
WRITE	SET MASTER KEY	REMOVE COLLECTION	FLUSH	CREATE PARTITION
CREATE†		LIST COLLECTION	LIST	REMOVE PARTITION
REMOVE†				
GET_ATTR‡				
SET_ATTR‡				

†Space management.
‡Attributes.

System), CIFS (Common Internet File System), and Centera** [3] interfaces are much richer, allowing custom storage solutions to optimize for application behavior. However, these optimizations are performed at a level above the actual storage devices, requiring additional hardware that must map to and incur the limitations of block-based storage. Moreover, while higher-level interfaces provide security, most have disparate security models that cannot coexist. OSDs provide a basic security model upon which a wide range of security architectures can be built, enabling flexible cross-platform sharing without additional resources.

Object-based storage

An object is a logical entity (e.g., a sequence of bytes) of storage with well-known, file-like access methods (e.g., read and write), object attributes describing the characteristics of the object, and security policies that authorize access [4]. An object is a variable-sized entity that can store a wide variety of data including text, images, audio/video, and database tables. An object can grow or shrink dynamically and is completely contained and managed within a single OSD. Objects are grouped into partitions that enable security, space management, and quota management. Each partition represents a security domain with its own set of keys (see the section on security).

Objects are identified using an object identifier (OID) that consists of two parts: a 64-bit *Partition_ID* to identify the partition the object belongs to and a 64-bit *User_Object_ID* that identifies the object inside the partition. The {Partition_ID, User_Object_ID} tuple uniquely identifies any object inside an OSD.

Along with storing data, each object includes a set of object attributes that associate metadata with the object. Object attributes describe specific object characteristics such as the total amount of physical bytes required for the object, the object logical size, or the object creation time. Some attributes are defined, interpreted, and maintained by the OSD, including timestamps (e.g., create time and last access time) and physical capacity. Other attributes such as quality-of-service (QoS) hints are defined by the higher-level software.

Within the OSD, each object is mapped to a set of underlying physical sectors that the OSD is responsible for maintaining by means of an OSD-based internal storage management system that functions much like a low-level file system, mapping disk blocks to objects and providing the semantic information necessary for the OSD to make sophisticated layout, caching, and prefetching decisions.

The OSD object model is flexible, allowing higher-level software to map each of its entities (e.g., file and table) to a single object, multiple objects, or a partial object. For example, a high-level file system might map a file to a single object. For large files, the file system might stripe the file across multiple objects to improve performance and reliability, similar to the ways in which RAID (Redundant Arrays of Independent Disks) systems map volumes across multiple storage devices. For small files, where metadata overhead can have a negative impact on capacity, the file system might map multiple files onto a single object, encoding each file size and position within the object attributes

Core OSD command set

Objects are accessed through a small set of common operations. **Table 1** highlights the most common operations as defined in the ANSI T10 OSD standard (see Reference [2] for a complete list).

Almost all OSD commands act on a specific object. For example, the READ and WRITE commands perform I/O for a specific object and include an offset parameter to specify where within the linear address space of an object the operation should begin. APPEND does not require an offset since it always appends data to the end of an object. Unlike block-based protocols, for which storage blocks always exist, OSD requires a CREATE command to

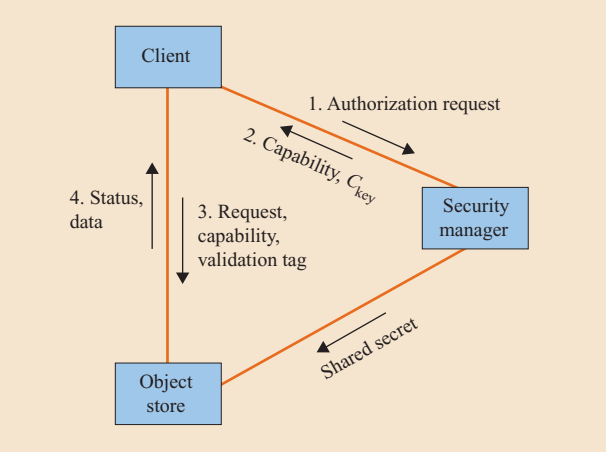

Figure 2

The OSD security protocol is based on a secret key shared between the security manager and the OSD (object store). The capability defines which operations are permitted on the object, whereas C_{key} results from applying a one-way function on the capability with the shared key. The validation tag depends on the chosen security method and is a function of C_{key}. For the CAPKEY method, the validation tag depends on the capability and the Channel_ID. The one-way function that is used is HMAC-SHA1, a hash-based message authentication code using the SHA1 hash function.

explicitly create an object before it can be accessed. Likewise, REMOVE is used to free all of the storage capacity used by an object, along with the associated OID.

Attributes are accessed using the GET_ATTR (get attributes) and SET_ATTR (set attributes) commands, which can access lists (i.e., groups) of attributes within a single command. To help organize attributes, the per-object attribute namespace is divided into pages (i.e., groups) of similar attributes (e.g., all of the timestamp-related attributes will be placed on the same page). This allows for easy access to related attributes (e.g., all timestamps) by requesting all attributes on a specific attribute page. Attributes are interpreted and maintained by the OSD (e.g., timestamps) and are addressed at well-defined locations within the attribute namespace. To accommodate attributes created and managed by the higher-level software, much of the attribute namespace is undefined by the OSD and may be managed outside of the OSD using the same GET/SET_ATTR commands.

Finally, it is important to ensure that data and attributes are committed to stable storage. The FLUSH command controls when and which objects are flushed. FLUSH can be issued for a single object or a set of objects (called a *collection* in the OSD standard), allowing applications to efficiently commit a large number of objects to stable storage with just one command.

OSD optimizations

One of the common usage patterns of an OSD involves an operation (e.g., WRITE) on the data segment of an object and then accessing some attributes (e.g., a file system-maintained last-access-time attribute). To support this common access pattern, the OSD standard allows all commands to include a SET_ATTR and/or GET_ATTR operation. We call this *piggybacking* an attribute access on a command. Piggybacking attribute updates can significantly reduce the number of messages and hence the message processing on each OSD. Piggybacking can also improve client latency by removing the additional network roundtrip that a separate GET_ATTR or SET_ATTR command would create. Finally, by binding the attribute access with the command, piggybacking avoids the need to impose a locking layer to guard concurrent access to an object.

Other optimizations were added to overcome concurrency issues. For example, after an APPEND command completes, the higher-level software may need information about the offset within the object where the data write occurred (e.g., to enable recovery). The standard way to obtain this information is with a GET_ATTR request attached to the APPEND command. However, if two APPEND commands occur simultaneously, the state of such an attribute would be undefined. In order to avoid this type of semantic confusion, OSD includes a set of attributes (i.e., CURRENT_COMMAND_ATTRIBUTES) that records information specific to the command currently being executed. For an APPEND command, this information includes the starting byte of an append and any integrity check value (see the section on security).

The standard also includes a special collection object that stores lists of objects. The collection object is essentially a list of OIDs, similar to a file system directory. Initially, collections were proposed for temporal logging of OIDs. For example, a list maintained by the file system that stores all objects involved in an ongoing high-level file system operation (e.g., move) can be implemented with collections, avoiding the overhead of an external logging system. However, as collections were developed, we found that more advanced multi-object operations that operate on all objects within a collection can greatly reduce command messaging costs, allowing the OSD to create optimized resource schedules for large amounts of work. These multi-object operations are currently proposed for Version 2 of the ANSI T10 OSD standard.

Security

One of the main features of OSD is its security model (**Figure 2**). Unlike traditional block storage, OSD enforces secure access to the device by implementing access-control security [5, 6] *within* the storage device. This is particularly valuable for distributed applications

Table 2 Object-based Storage Device security methods. (CAP: capability; CAPKEY: capability key.) In the table, *no* indicates that a particular threat is not protected against. The entire security protocol is documented in Reference [2]. The entry *depends* indicates that the protection is contingent on whether the channel security protects against this threat.

Security method	*Threats*							
	Credential forgery	*Changes to capability*	*Rogue client use of CAP*	*Command or status replay*	*Changes to command or status*	*Data replay*	*Changes to data*	*Data, status, or command inspection*
NOSEC (no security)	No	No	No	No	No	No	No	No
CAPKEY without channel security	Yes	Yes	Yes	No	No	No	No	No
CAPKEY with channel security	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Depends
CMDRSP (command response)	Yes	Yes	Yes	Yes	Yes	No	No	No
ALLDATA	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No

[e.g., file systems based on storage area networks (SANs)], where device-enforced access control efficiently supports distributed environments with a mix of trusted and untrusted clients. The OSD security model separates policy from enforcement; it keeps security policies outside of the storage device while providing secure enforcement mechanisms within the OSD itself. Enforcement is implemented via a capability-based mechanism, enabling fine-grained access control that protects both the storage device and individual objects or partitions from unauthorized access. In order to ensure secure access to storage, every command must be accompanied by a cryptographically secure capability (CAP) that identifies a specific object and the list of operations that may be performed against the specified object. A detailed description of the capability and the corresponding signature is provided in Reference [6].

The security system is composed of three entities: a client, the object store, and a security manager, as shown in Figure 2. Before accessing an object, the client requests a capability from the security manager for the desired operation. On the basis of the security policy, the security manager returns an appropriate credential that includes 1) the capability that defines the access rights and 2) the capability key that is based on a secret symmetrical key shared between the security manager and the OSD. The client sends the capability with every command as part of its request and cryptographically secures the request using the capability key. The OSD validates the request, ensuring that the capability has not been tampered with and was rightfully obtained by the client and that the requested operation is permitted by the capability.

The symmetrical secret keys shared between the OSD and the security manager are arranged hierarchically and updated periodically. Every credential is based on one of the keys from this hierarchy, chosen according to the command being issued. Within an OSD, each partition is essentially a separate security domain and carries its own keys—the PARTITION keys. Each OSD also has a ROOT key that is used to create and operate on partitions. Finally, the OSD MASTER key is used to initialize the device. A key update renders all credentials that are based on that key invalid.

Keys are updated periodically, typically by a higher-level key in the hierarchy. The update of a key at a given level invalidates all keys at lower levels. The update of the topmost MASTER key is done via the Diffie–Hellman key exchange protocol [7], thus achieving forward secrecy. For all other keys, an existing key is used to generate the newly computed key.

To support various application requirements and security environments, the standard supports four different security modes (**Table 2**) that provide various degrees of security under various security assumptions. The modes are NOSEC, CAPKEY, CMDRSP, and ALLDATA. NOSEC is used in an environment where the network is trusted or secured and the client is trusted not to forge the capability, thus protecting against unintentional errors. CAPKEY is most useful when the channel between the OSD and the client is externally secured (e.g., by an authenticated IPSec, or Internet Protocol security, channel) but the client is untrusted. CMDRSP and ALLDATA address environments in which the channel between the OSD and the client is not externally secured, and the client is untrusted. CMDRSP ensures the integrity of commands and arguments, while ALLDATA ensures the integrity of the data as well.

Current implementations

In this section, we describe three different OSD implementations: a prototype Seagate OSD, an IBM

Research OSD implementation, and the Panasas OSD used in its distributed file system. In each section, we discuss the considerations and advantages that arise in implementing object-based storage.

Seagate object drive

Seagate designed and made a protype of an object disk that could be a continuation of its existing line of high-end hard disks. This imposed several important engineering constraints on the implementation. Since disk drives are commodity devices produced in large quantities, cost optimization is a criterion. Drives are also high-performance devices, handling interface speeds up to 3 Gb/s and sustained media speeds of more than 170 MB/s. This means that high performance must be achieved with the limited CPU resources available in a drive.

To give a precise example of the type of resources available in a disk drive, the prototype Seagate OSD drive demonstrated at Storage Networking World (SNW) in April 2005 was a Cheetah** 15K.3 Fibre Channel drive. This product line contains two 32-bit embedded processors and an 8-MB buffer. The OSD code ran on one of the processors alongside other portions of the drive firmware. The OSD code was optimized for size, so that it required tens of kilobytes of on-drive code, and specialized for the drive hardware.

Every major change in drive interfaces over the last several decades has moved intelligence from the host to the drive. For hard disks, moving to an object-based interface is the next logical step in this evolution. OSD extends earlier interfaces by providing two crucial pieces of information to drives. First, since space management is delegated to drives, drives now have basic information such as which blocks are free and which are in use. This allows drives to optimize reliability and service quality.

The second piece of information gained from the OSD relates to information about application requirements. Through shared attributes, applications can relay preferences for QoS and reliability. The drive can use this information to optimize storage and retrieval, matching application needs to detailed drive capabilities. The prototype drive defines two shared attributes to enable this communication: a QoS attribute to locate objects in different performance zones and a reliability attribute to enable mirroring or more advanced internal parity schemes. The phrase "performance zones" refers to the fact that different parts of the disk drive have different performance characteristics.

With traditional sector-based drives, disk access is based on fixed-length blocks accessed by block addresses. As a result, any change in the sector size requires widespread software modifications. With an OSD, the drive manages block size and space management. Users specify the OID, byte offset, and number of bytes to transfer, allowing drives to use any convenient underlying block size (the prototype Seagate OSD uses 4 KB).

The addition of space management is a natural extension of the functionality provided by the drive. Current drives already contain relocation maps for defect management, zoning, and other performance and reliability tracking. Allocation maps extend this functionality by providing richer semantics to help minimize seeks and to optimize read/write caching. When the drive becomes aware of which data belongs to which object, it can attempt to deduce access patterns and improve prefetching and scheduling decisions.

Another consideration for space management is flexibility to support various applications. For example, a streaming server might make use of larger allocation units, which would have an impact on the performance of desktop applications. Drives can deduce workload environments (e.g., access patterns and object sizes) and automatically optimize their space management strategies. Shared attributes can be used to fine-tune optimizations on an object-by-object basis. The prototype Seagate OSD implementation takes advantage of both approaches. Seagate defines attributes that clients can use to control how space is allocated and how fast objects will grow. Additional drive parameters can be configured statically using format-time options.

Exception handling and recovery are critical for the proper functioning of the drive. As with a host-based file system, each drive must protect metadata. This can be done by internally mirroring the metadata or by protecting it with more powerful error-correction codes. When dealing with partial-drive failures, OSD provides fine-grain recovery. Repairing a single object is much simpler than rebuilding an entire volume. The OSD can further improve rebuild times because it eliminates the need to rebuild free space and allows for critical objects to be rebuilt on a priority basis. OSD drives can unobtrusively perform repairs or fence objects if they are not repairable. Errors in free space no longer need to be recovered; thus, they can simply be remapped.

Achieving high data-transfer rates to a hard disk is difficult because of its limited computational power. In order to overcome this limitation, specialized hardware was used wherever possible. The OSD protocol has more complex commands than those used in standard SCSIs. Additional processing is required to handle the verification of access rights and command integrity. To minimize overhead, the prototype Seagate OSD implementation splits per-command verification into two phases. First, basic checks that do not require extensive processing or disk access are performed. Those requests that pass initial verification are placed in a queue for further processing. The second phase of verification

occurs when the time comes to process the command and all information about the object is available in memory. Note also that there is a fixed overhead whether the request is for a single byte or several megabytes of data, making overhead more noticeable with smaller requests. When multiple requests are queued, this overhead can be amortized over the data access time.

IBM object-based storage

The IBM Haifa Research Laboratory has implemented two generations of object disks. The first generation, called Antara [8], was followed by ObjectStone [4]. ObjectStone was built to validate the concept of an object-based storage controller. As such, it was designed to work on the entire range of hardware storage platforms developed at IBM. The minimal hardware configuration was set to 128 MB of nonvolatile RAM (NVRAM), one back-end disk, 1-Gb/s Ethernet, and 512 MB of RAM. The prototype was built on Linux** 2.6 and worked with regular disks and RAID arrays as a back end.

ObjectStone supports V1.0 of the OSD standard, including the NOSEC and CAPKEY security modes, and some additional enhancements. For example, it implements a CLEAR operation that creates a hole (i.e., a portion of a file that contains all zeros) in the middle of the object; this operation was needed by the object-based SAN.FS file system. The CLEAR command has since been incorporated into V2.0 of the standard.

The transport protocol to ObjectStone was iSCSI (Internet SCSI) [9, 10]. This enabled building and testing the prototype on a standalone Linux x86 machine connected to Ethernet. ObjectStone has some basic support for Fibre Channel. This enables it to work under certain configurations as part of existing IBM storage controllers.

OSD commands use two rarely supported SCSI features: extended CDBs (command descriptor blocks) and bidirectional commands. Support for these features had to be added to the SCSI initiator and target. The changes that were required to get the Linux SCSI stack to support OSD were made by IBM and contributed to the open-source community [11]. The hope is that the Linux community will adopt these modifications and merge them into the Linux kernel. ObjectStone implemented an iSCSI target with support for extended CDBs and bidirectional commands.

ObjectStone was built in close cooperation with the SAN file system team at the IBM Almaden Research Center. An object-based SAN file system is complicated; it contains a cluster of metadata servers, object disks, and client machines. It was clear that such a complex system would be exceedingly difficult to debug. As a design guideline, it was decided that the OSD was to be debugged in isolation and brought together with the rest of the system only when it was of the highest possible quality. To this end, two debugging and test tools were built: an OSD test suite and an OSD simulator.

The OSD test suite [12] was developed as a black-box testing tool. The basic idea was to create a testing tool that can read a script file and send OSD commands to an OSD target. Various scripts were written to stress different aspects of the implementation. Because of the black-box approach, the tool could be used with other implementations.

It was impossible to build SAN.FS without an initial OSD to work with. Therefore, an additional tool was built, an OSD reference implementation that was called the *OSD simulator* [13]. The simulator runs on a standard Linux operating system-based machine as a user-space process. It uses the local file system; an object is implemented by a file in */tmp*. The simulator was later used for comparison-based validation; a set of commands were executed against ObjectStone and the simulator, and the results were compared. This is a general technique and was later applied to the Seagate OSD while setting up the SNW demonstration. The simulator is available on the IBM alphaWorks* online delivery system.

The internal OSD file system was implemented using copy-on-write (or *shadowing*). This means that the back-end disk is split into blocks, and live blocks (those currently in use) are never overwritten. The file system is, essentially, a tree of blocks and modifications are made by modifying the leaves and propagating the changes up the tree. The OSD objects and catalogs were implemented with b-trees, which were an obvious choice because they are a classic external-memory data structure and many file systems use them [14, 15]. However, using traditional b-trees posed some unexpected difficulties. The first difficulty involves multithreading in which the root node of an object or catalog experiences numerous concurrent modifications, and with shadowing, every leaf modification requires updating the root. Second, the leaf-chaining that occurs means that b-tree leaves are chained together from left to right; in this case, modifying a single leaf also requires shadowing all of the nodes to its left. In order to solve these difficulties, top-down b-trees were used, and leaf-chaining was removed (see [16] for details).

Panasas

Panasas has built both an OSD and a distributed file system that provides the standard POSIX** interface (Portable Operating System Interface), extensions for concurrent write applications, and Network File System (NFS) and CIFS support. The object-based storage hardware consists of two Serial ATA disk drives, a 1.8 GHz IA-32-based processor, 512 MB to 1 GB of RAM, and a 1-Gb/s network interface controller (NIC).

The OSD software runs inside the FreeBSD** (Berkeley Software Distribution) operating system and mirrors data across the two disk drives. The higher-level Panasas File System operates above the OSD interface and writes the data using the RAID-5 algorithm across all of the OSDs in the system [17].

In addition to supporting the basic OSD interface, Panasas developed a number of enhancements. First, because Panasas performs RAID-5 operations above the OSD interface, efficient reconstruction required that the OSD provide additional information about an object, information that would normally be unavailable. Therefore, we added a READ MAP command that allowed the OSD to return a map of the damaged regions of an object. The reconstruction software, which is layered above the OSD interface, fetches the damaged-region map from the OSD to discover exactly which bytes within an object require reconstruction. For small objects, the reconstruction saving is nominal, but for large objects (e.g., terabyte), the saving is substantial.

There are two other types of maps that the Panasas OSD can return. First, READ MAP can return the regions of an object that are stored as space-efficient holes. The reconstruction engine uses this information to avoid writing all zeros to the OSD, instead using SET_ATTR on the logical length of the object to recreate the space-efficient storage of the hole.

Second, the command DIFF READ tells the OSD to compare two objects and return a map indicating which bytes within two objects have potentially different data. Normally, random objects do not share any storage, but Panasas implemented snapshots (i.e., copies at a point in time) in a space-efficient manner (i.e., copy-on-write), allowing a snapshot to share some or all of its storage with the original partition. Reconstruction uses the regions that have the same data to rebuild objects that have a snapshot chain (i.e., a set of related copies at particular times), rebuilding the common regions and then applying a snapshot command before writing the different data regions. Without the map of differing regions, snapshot reconstruction would be unable to recreate the data relationships, vastly increasing the capacity consumed in each snapshot and effectively making every reconstructed snapshot a full data image.

Beneath the OSD interface, Panasas has designed a high-performance file system (OSDFS, or object-based storage device file system) that leverages the semantic information conveyed across the OSD interface. The file system is log based with writes cached in a battery-backed buffer that aggregates tens of megabytes of writes before committing them to disk. This creates large sequential writes to disk. It also allows out-of-order writes to sequential data ranges, often from different clients, to be reordered for optimal sequential layout.

To optimize reads, OSDFS performs aggressive prefetching and caching inside the RAM of the OSD. Every client stream (i.e., a set of commands issued from a single client) is tracked by OSDFS to determine whether a sequential access pattern can be detected. Once a pattern is discovered, spare cycles are used to prefetch up to 6 MB of data for each stream. Because OSDFS tracks each client stream independently, OSDFS can detect different patterns among different clients concurrently accessing the OSD. Each stream is assigned its own prefetch context that OSDFS uses to determine the maximum amount of prefetching that should be performed. Also, OSDFS tracks the number of active prefetch contexts, adjusting their individual and collective prefetch sizes to ensure fair (i.e., balanced) performance for each of a large number of concurrent client streams and to manage the amount of buffer cache dedicated to each stream.

Supporting a large number of concurrent client streams (hundreds per OSD) required several refinements to the disk scheduling and caching algorithms of the OSDFS. Once the number of demand requests exceeds the head bandwidth of the OSD, the OSDFS turns off prefetching to prevent it from competing with demand requests. Further, OSDFS uses client-supplied per-command sequence numbers to detect when a large application read() is broken into several OSD READ commands. This primarily happens when the RAID engine of the client must skip the RAID-5 parity blocks stored within each object. The client file system tags each group of sequential commands, so the OSD will wait until all of the commands within a group are received. This allows OSDFS to issue a single large read to its disks, maximizing read bandwidth and minimizing the number of seeks. Note that multiple sequential commands are not a problem for a lightly loaded OSD (i.e., fewer than 20 client streams), which relies on its prefetch engine to always perform large sequential reads.

OSDFS also defines different classes of service by marking each OSD command with a service-level tag. Service levels are set by the client and represent different application classes such as the applications of the client, the background scrubber (i.e., a process that continuously reads the disk to check for data errors), the reconstruction engine, backup applications, and the internal performance optimizer of the Panasas client file system. OSDFS maps these different service levels to soft real-time guarantees to ensure that each class of service will receive a specified amount of the CPU cycles of the OSD. The percentage of an OSD that each service level is guaranteed can be changed at any time by the management software, allowing a system administrator to dynamically change the priority of services such as for reconstruction or backup.

Related work

Object-based storage has its origins in the network-attached storage research of the 1990s. The government-funded Network-Attached Secure Disks (NASD) project [18] at Carnegie Mellon University focused on cost effectively adding processing power to individual disks for networking, security, and basic space management functionality. This turned into a larger, industry-sponsored project and eventually yielded the current T10 OSD standard.

Other institutions continue research on object-based storage. For example, Ceph is a petabyte-scale object-based storage system being developed at the University of California, Santa Cruz, for high-performance computing environments [19]. It was built using a large number of OSDs, each of which contains one or more disk drives managed by a local CPU. Ceph is a distributed storage system that provides load-balancing across all OSDs to eliminate hotspots (frequently accessed data blocks) and provide scalability.

The Digital Technology Center of the University of Minnesota has an open-source reference implementation [20] of the OSD standard and uses it for intelligent storage research for disks and tapes [21]. This implementation is an extension of an earlier reference implementation of Intel [22]. The Network Storage Technology Division of the Data Storage Institute [23] and the Ohio Supercomputer Center [24] are other academic institutions actively working on OSD-related projects.

On the industry side, Sun Microsystems recently added OSD support for Solaris** as part of the OpenSolaris** project [25]. Sun also acquired Cluster File Systems, Inc., and hence the Lustre** File System [26] that uses objects. XAM (Extensible Access Method) is a related standard being developed by the Storage Networking Industry Association (SNIA) that uses objects for fixed content storage [27]. Ongoing work also exists to map XAM to OSD so that OSDs can be used for fixed content storage.

Conclusions

The Object-based Storage Device interface provides significant semantic information (both temporal and spatial) that can be used to significantly improve storage along four dimensions: 1) performance, 2) security, 3) data sharing, and 4) manageability.

In terms of performance, data-specific information allows the OSD to optimize layout, minimizing the number of seeks. OSD can also use semantic information stored in object attributes to co-locate multiple objects accessed simultaneously (e.g., a UNIX** user's configuration files, or dot-files). OSDs can be much more resource efficient, with prefetching and caching strategies that match the demands of specific individual workloads by tracking the progress of requests made to individual objects and detecting access patterns in order to issue future prefetches and manage internal cache space.

OSD security prevents unauthorized access to individual objects, ensuring that data is not compromised, even if the physical storage device is stolen. OSD security also enables flexible data sharing models, allowing groups with disparate security needs and policies to share the same physical storage devices without compromising security. Even within the same organization, OSD security enables sharing by guaranteeing that individual file systems that may have correctness problems (e.g., errors) cannot cause corruption outside of the data that they individually manage.

OSD attributes can describe a range of QoS parameters including bit-rate, access pattern, error handling, and reliability parameters. For example, video streams that can tolerate missing data (e.g., video data) would set an object attribute to disable read retries. Fault-sensitive applications may specify high degrees of redundancy for OSD-based RAID arrays. The result of these improvements is significant increases in the performance of storage systems.

Finally, applications can benefit from the close association of data and attributes. While conventional file systems provide a simple data-access paradigm, OSDs have standardized an extensible system, including user-defined attributes as well as data aggregations. Library software that accesses OSD initiators could enable direct aggregated manipulation of data and attributes. Some attributes could be directly mapped into attributes required by applications concerned with information life-cycle management and content awareness. These attributes include retention time, deletion time, refresh interval, fixed content, and content authenticity or provenance certificates. Although these types of metadata and the appropriate actions related to them can be provided by an application that accesses data using a conventional file system, with OSDs the data–attribute relationship can be guaranteed at the OSD level (and be certified by compliance authorities); some attributes can even be standardized or registered as application-domain attributes to ensure interoperability between different vendors. In addition, their implementation in OSD implies better performance (a smaller number of I/O operations performed by servers, as the OSD has combined operations for data and attributes).

The advent of OSDs has spurred the development of better file systems. Some will perhaps have new access characteristics. This is evidenced by the standardization efforts around NFSv4 extensions for parallel (direct) data access and by the rapid spread of SAN-based file systems

that have OSD characteristics either already included or planned for future systems.

*Trademark, service mark, or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

**Trademark, service mark, or registered trademark of EMC Corporation, Seagate Technology, Linus Torvalds, the Institute of Electrical and Electronics Engineers, The FreeBSD Foundation, Sun Microsystems, Inc., or The Open Group in the United States, other countries, or both.

References

1. A. Raghuveer, S. W. Schlosser, and S. Iren, "Enabling Database-Aware Storage with OSD," *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, September 2007, pp. 129–142.
2. "Information Technology—SCSI Object-Based Storage Device Commands (OSD)," ANSI/INCITS 400-2004, the InterNational Committee for Information Technology Standards (formerly NCITS), December 15, 2004; see *http://www.techstreet.com/cgi-bin/detail?product_id=1204555*.
3. EMC Corporation, Centera Family; see *http://www.emc.com/products/family/emc-centera-family.htm*.
4. M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran, "Object Storage: The Future Building Block for Storage Systems: A Position Paper," *Proceedings of the Second International IEEE Symposium on Emergence of Globally Distributed Data*, Sardinia, Italy, June 20–24, 2005, pp. 119–123.
5. H. Gobioff, "Security for a High Performance Commodity Storage Subsystem," Ph.D. dissertation, CMU-CS-99-160, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, July 1999; see *http://www.pdl.cmu.edu/PDL-FTP/NASD/hbg_thesis.pdf*.
6. M. Factor, D. Nagle, D. Naor, E. Riedel, and J. Satran, "The OSD Security Protocol," *Proceedings of the Third IEEE International Security in Storage Workshop*, San Francisco, CA, December 2005, pp. 11–23.
7. M. E. Hellman, B. W. Diffie, and R. C. Merkle, "Cryptographic Apparatus and Method," U.S. Patent No. 4200770, 1980.
8. A. Azagury, V. Dreizin, M. Factor, E. Henis, D. Naor, N. Rinetzky, O. Rodeh, J. Satran, A. Tavory, and L. Yerushalmi, "Towards an Object Store," *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies*, April 2003, pp. 165–176.
9. "Information Technology—SCSI Architecture Model-3 (SAM-3)," ANSI/INCITS 402-2005, the InterNational Committee for Information Technology Standards (formerly NCITS), February 14, 2005; see *http://www.techstreet.com/cgi-bin/detail?product_id=1212605*.
10. IP Storage; see *http://www.ietf.org/html.charters/OLD/ips-charter.html*.
11. IBM OSD Initiator; see *http://sourceforge.net/projects/osd-initiator*.
12. D. Naor, P. Reshef, O. Rodeh, A. Shafrir, A. Wolman, and E. Yaffe, "Benchmarking and Testing OSD for Correctness and Compliance," *Proceedings of the IBM Verification Conference (Software Testing Track)*, November 2005; see *http://www.haifa.ibm.com/projects/storage/objectstore/papers/osd_test.pdf*.
13. IBM Corporation, IBM Object Storage Device Simulator for Linux; see *http://www.alphaworks.ibm.com/tech/osdsim*.
14. S. Best, "Journaling File Systems," *Linux Magazine* **4**, No. 10, 24–31 (2002).
15. A. Sweeny, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS File System," *Proceedings of the USENIX 1996 Annual Technical Conference*, San Diego, CA, January 1996; see *http://www.usenix.org/publications/library/proceedings/sd96/sweeney.html*.
16. O. Rodeh, "B-trees, Shadowing, and Clones," *ACM Trans. Storage* **3**, No. 4 (2008); see *http://www.usenix.org/event/lsf07/tech/rodeh.pdf*.
17. D. Nagle, D. Serenyi, and A. Matthews, "The Panasas ActiveScale Storage Cluster: Delivering Scalable High Bandwidth Storage," *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing,* 2004, p. 53.
18. 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," *ACM SIGOPS Operating Syst. Rev.* **32**, No. 5, 92–103 (1998).
19. Ceph: Petabyte Scale Storage; see *http://ceph.newdream.net/*.
20. University of Minnesota, DISC OSD T10 Implementation; see *http://www.sourceforge.net/projects/disc-osd*.
21. D. He, N. Mandagere, and D. Du, "Design and Implementation of a Network Aware Object-Based Tape Device," *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, 2007, pp. 143–156.
22. Intel's Open Storage Toolkit; see *http://www.sourceforge.net/projects/intel-iscsi*.
23. Data Storage Institute, Network Storage Technology Division; see *http://www.dsi.a-star.edu.sg/*.
24. P. Wyckoff, D. Dalessandro, A. Devulapalli, and N. Ali, "Applicability of Object-Based Storage Devices in Parallel File Systems," Ohio Supercomputer Center; see *http://www.osc.edu/research/network_file/projects/object/*.
25. Sun Microsystems, OpenSolaris Project: Object Storage Device (OSD) Support for Solaris; see *http://opensolaris.org/os/project/osd/*.
26. Sun Microsystems, Lustre Wiki; see *http://wiki.lustre.org/index.php?title=Main_Page*.
27. Storage Networking Industry Association, XAM Initiative: Information Independence for Applications and Storage; see *http://www.snia.org/forums/xam/*.

Received September 18, 2007; accepted for publication October 14, 2007; Internet publication June 24, 2008

David Nagle *Google, Inc., 111 8th Avenue, New York, New York, 10011 (dfnagle@gmail.com)*. From 1995 to 2001, Dr. Nagle co-led the Carnegie Mellon University (CMU) Network-Attached Secure Disk (NASD) project and served as an Assistant Professor and Director of the Parallel Data Lab at CMU. In 2001, he joined Panasas, designing their datapath architecture, including a two-level RAID layout that was standardized in the IETF NFSv4 pNFS standard. He has been a major contributor to all of the OSD standardization efforts, including ANSI T10 SCSI OSD V1.0 and V2.0.

Michael E. Factor *IBM Haifa Research Laboratory, Haifa University Campus, Mount Carmel, Haifa 31905, Israel (factor@il.ibm.com)*. Dr. Factor holds a B.S. degree in computer science from Union College, and M.S., M.Phil., and Ph.D. degrees in computer science, all from Yale University. He is an IBM Distinguished Engineer with a focus on storage and systems. He has been an architect of advanced copy functions for the IBM DS family of storage subsystems and takes a leading role in storage-related research, including such topics as long-term digital preservation, storage power, and object-based storage. Dr. Factor is chair of the security subgroup of the SNIA OSD standardization effort. In the past, he was the manager of Distributed and Clustered Systems at the Haifa Research Laboratory. Previous projects he was involved with include the Cluster Virtual Machine for Java**, the XML file system, the IBM iSeries* integrated file system, and the Web server for the 1996 Atlanta Olympics.

Sami Iren *Seagate Research, Pittsburgh, PA 15222 (sami.iren@seagate.com)*. Dr. Iren is a Research Staff Member at Seagate Research in Pittsburgh, Pennsylvania. He is the co-chair of SNIA OSD Technical Working Group and has been a major contributor to the standardization of the OSD (Object-based Storage Device) since 2002. His research interests include intelligent storage devices and interfaces, multimedia, and networking. Before joining Seagate, he worked in the telecommunications industry for 3 years as a researcher and developer. Dr. Iren has a B.S. degree in computer engineering from Middle East Technical University, Ankara, Turkey, and M.S. and Ph.D. degrees in computer science from the University of Delaware.

Dalit Naor *IBM Haifa Research Laboratory, Haifa University Campus, Mount Carmel, Haifa 31905, Israel (dalit@il.ibm.com)*. Dr. Naor holds M.S. and Ph.D. degrees from the University of California, Davis, and a B.S. degree from the Technion, Israel Institute of Technology, all in computer science. She is a Research Staff Member and a manager of the Storage Systems and Performance Management group. She is developing advanced functions for storage systems, including support for long-term digital preservation and secure access to storage. She contributed to the development of the object storage T10 OSD standard, and in particular to the security protocol. Dr. Naor's prior areas of interests include combinatorial optimization, bioinformatics, applied security, and content protection.

Erik Riedel *Seagate Research, Pittsburgh, PA 15222 (erik.riedel@seagate.com)*. Dr. Riedel leads the Interfaces and Architecture department at Seagate Research in Pittsburgh, Pennsylvania. His group focuses on novel storage devices and systems with increased intelligence to optimize performance, improve security and reliability, automate management, and enable smarter organization of data. He is a member of the SNIA Technical Council, helping to lead industry-wide education, technology promotion, and standardization efforts. He also serves on the technical advisory board for the Technology Collaborative supporting technology startups in Pennsylvania. Before joining Seagate, Dr. Riedel was a researcher in the storage program at Hewlett-Packard Labs in Palo Alto, California, working on networked storage, distributed storage, and security. He has authored and coauthored eight issued patents and has a number of pending patent applications, as well as numerous technical publications on a range of storage-related topics. Dr. Riedel holds B.S., M.S.E., and Ph.D. degrees from Carnegie Mellon University. His thesis work was on Active Disks as an extension to Network-Attached Secure Disks (NASDs).

Ohad Rodeh *IBM Haifa Research Laboratory, Haifa University Campus, Mount Carmel, Haifa 31905, Israel (orodeh@il.ibm.com)*. Dr. Rodeh received his B.Sc., M.Sc., and Ph.D. degrees in computer science from the Hebrew University in Jerusalem. He is a Research Staff Member at the IBM Research Laboratory in Haifa. He has worked on distributed systems, security, and storage. Currently, his main interest is machine learning. He also teaches a course on storage at the Tel Aviv University.

Julian Satran *IBM Haifa Research Laboratory, Haifa University Campus, Mount Carmel, Haifa 31905, Israel (julian_satran@il.ibm.com)*. Mr. Satran holds an M.S.E.E. degree from the Polytechnic Institute of Bucharest, Romania. He is an IBM Distinguished Engineer. His current areas of interest span system and subsystem architecture, networking, and development and operating environments. He led several pioneering research projects in clustering, file system structure (and object storage), I/O and networking convergence (iSCSI), and future, more rational and scalable I/O subsystems. He has driven an industry-wide effort to standardize iSCSI and is now leading an effort to standardize object storage. He is a member of IEEE and ACM.