Towards an Object Store

Alain Azagury Vladimir Dreizin Michael Factor Ealan Henis Dalit Naor
Noam Rinetzky Ohad Rodeh Julian Satran Ami Tavory
Lena Yerushalmi
IBM Haifa Research Laboratories
{*azagury,dreizin,factor,ealan,dalit,noamr,orodeh,satran,tavory,ylena*}*@il.ibm.com*

Abstract

*Today's SAN architectures promise unmediated host access to storage (*i.e., *without going through a server). To achieve this promise, however, we must address several issues and opportunities raised by SANs, including security, scalability and management. Object storage, such as introduced by the NASD work [14], is a means of addressing these issues and opportunities. An object store raises the level of abstraction presented by a storage control unit from an array of 512 byte blocks to a collection of* objects. *The object store provides "fine-grain," object-level security, improved scalability by localizing space management, and improved management by allowing end-to-end management of semantically meaningful entities.*

This paper presents a detailed description of how an object store works and describes the design of Antara, our prototype object store. For a cache hit workload, our pure software prototype is able to service roughly 14000 4K I/O requests per second. We also present a layered security model for an object store which separates concerns of access security and network security, leveraging existing security infrastructure.

1. Introduction

Today's SAN architecture promises "democratization" of data access, *i.e.*, inexpensive, non-mediated, and shared access to centrally-managed storage. In existing SAN deployments, this promise is only partially met. Typically, each logical unit (LU) is used by only a single host; in other words, the storage is partitioned among the hosts, and the hosts treat the storage as if it were directly attached. Sharing is typically accomplished by having a file server mediate access to the underlying storage. Finally, today's SANs are almost 100% based upon Fibre Channel, an expensive technology.

SAN file systems, such as Storage Tank [6], Lustre [4], CXFS [26], *etc*., are, however, truly attempting to deliver on the promise of SANs by providing unmediated shared access to data. Further, IP-based SANs, *e.g.*, iSCSI [25], are expected to have a lower cost than Fibre Channel based SANs, due to the relatively lower cost of an IP infrastructures (*e.g.*, Ethernet) as compared to Fibre Channel. These systems, however, run up against several of the inherent issues raised by SANs: security and protection, end-to-end management at a meaningful semantic level, and scalability (in particular for allocation).

Object stores can address all three of these issues, although most of the work to date has focused on the problem of security for data requests from independent entities transferred over a shared network. An object store raises the level of abstraction presented by today's block devices. Instead of presenting the abstraction of a logical array of unrelated blocks, addressed by their index in the array (*i.e.*, the Logical Block Address or LBA), an object store appears as a collection of objects. An individual object is akin to a simple byte stream file, presenting the abstraction of a sparsely allocated array of bytes indexed from zero to infinity.[1]

In an object store environment, space is allocated by the storage controller (*i.e.*, the object store itself) and not by overlaying software such as a file system. Users of an object store, *e.g.*, the file system, operate on data by performing operations such as creating an object, reading/writing at an offset from the start of the object, and deleting the object. In addition, all operations carry a credential, and it is the responsibility of the object store to validate that the user's request carries a valid credential. This credential allows the storage to enforce different access rights for different portions of a volume (*i.e.*, on a per object basis). Further, it eliminates the need to rely on an independently administered physical security, *e.g.*, zoning, masking, *etc*.

While there have been several years of research on ob-

[1] Without loss of generality, we assume an object store implementation may place limitations on the legal offsets which can be used in a request, *e.g.*, requiring them to be block aligned.

ject stores (*e.g.*, [5, 8, 12, 14, 15]) and there is an ongoing standardization effort [27], object storage is still not widely accepted. This is not because object stores are the wrong solutions, but rather because they were invented before their time. An object store inherently entails a paradigm shift; hosts no longer communicate with control units via SCSI block read and write requests but rather ask for offsets in an object. In addition, a host no longer handles space management within a volume; rather this is handled by the lower level storage controller. This entails changes to the structure of a file system. For such a paradigm shift to be justified, it needs to bring sufficient benefits. As long as SANs were used as a means of essentially emulating direct attached storage (*i.e.*, no sharing) over private Fibre Channel networks, the benefits of an object store were insufficient to justify the cost of this paradigm shift. With the attempt to truly leverage a SAN's promise of non-mediated, shared and inexpensive access to centrally managed storage, an object store becomes essential.

In this context, we have been working on a prototype object store. This work builds both upon the published literature as well as upon prior work of ours on Distributed Storage Facility (DSF) [8].

The main contributions of the work presented in this paper include:

- a systematic analysis of the benefits of an object store
- a novel, layered security model which separates concerns of network and access security, allowing us to leverage existing network security mechanisms
- a description of our pure software prototype object store, *Antara*. Over 1 Gb Ethernet *Antara* can service over 14000 4K I/O operations from a single client and for larger requests, *Antara* can sustain a bandwidth of 450Mb/s for a single client.

In the next section we describe the motivation for object storage, prior to surveying related work in section 3. We then present our systematic analysis of the benefits of an object store in section 4. This analysis is based upon our software implementation of a stand-alone object store controller which we describe in section 5. Next, we describe our novel security model, which is intended to be useful both in Fibre Channel and IP settings and which leverages existing underlying security infrastructure. Finally, we present some preliminary performance results in section 7 prior to concluding in section 8.

2. Motivation

As mentioned in the introduction, SANs promise to democratize storage by providing non-mediated, shared and inexpensive access to storage. But the use of block storage in SAN's raises several issues. This section elaborates on these issues.

The most significant issue raised is probably security on a SAN. In discussing SAN security, we find it useful to distinguish between two concepts: *security* and *protection*. Protection is always needed when there is shared access to data. It enables defining the access control policy to the shared resources. The protection mechanism provides defense against non-malicious "attacks," such as buggy clients, administrative errors, *etc*. One example of why protection is needed is that if an administrator incorrectly configures LUN masking, a Windows NT client that discovers a LU will assume it owns the LU, writing a signature on the LU, thereby causing a data integrity problem [30]. Protection, as a defense against errors is thus needed even if we have a completely secured and trusted infrastructure.

Security goes beyond protection in that it addresses intentional attempts at unauthorized access. The security mechanism's role is to ensure that the system's access policy is enforced, even in the presence of malicious attempts to gain unauthorized access. When discussing security one needs to make clear the type of attacks one wishes to prevent. We elaborate on these details when we describe our security model in section 6. Security is thus essential if the infrastructure is not trusted.

Today's SANs offer a very limited notion of security, and thus, in theory, are very vulnerable for a wide range of attacks. However, in practical terms, these attacks are not common. Today's SANs are Fibre Channel based. Since Fibre Channel is not pervasive and hosts with Fibre Channel connections tend to be in protected machine rooms, the practical opportunity for attacks is limited. However, the expected adoption of IP-based storage is likely, in a practical sense, to exasperate the SAN security problem. IP networks are readily available, IP-connectivity is not limited to protected machines, and unfortunately there is a range of tools and techniques for IP-based security attacks.

Given the existing minimal and very coarse grain security support, one must assume that the storage clients are completely trusted. In addition, the mechanisms that do exist, such as zoning, LUN masking, *etc.*, are hard to use and related to the physical structure of the storage. Using these mechanisms it is possible to control a host's physical access to storage, but if a host can access a volume, it has complete access to all of the data on the volume. At best it is possible to provide all or nothing access to a LU for a given host. Thus, attempts to share storage, as SANs promise, are also likely to exasperate the SAN security problem.

Doing significantly better than this very coarse level of access control in the context of today's block storage devices is not practical. There is no efficient way to describe

the protection scheme at a block granularity – there are simply too many actively used blocks. The control unit does not know which blocks have the same security attributes, *e.g.*, belong to the same file; therefore, it would be necessary to tell the control unit who is allowed to access each block. The overhead involved in giving the control unit information for each block (regardless of whether this is done in-band or out-of-band) would be detrimental to performance. Because of this "SAN security problem," most of the research on object storage has been driven by the need to provide SAN security; for a discussion of related work see section 3.

A second issue that arises when trying to fully leverage a SAN is scalability in terms of the number of storage clients. Scalability is typically not an issue unless hosts share access to volumes. However, shared access is one of the touted benefits of a SAN. Shared access requires coordination, and coordination can lead to scalability problems. For example, file systems must coordinate allocation of blocks to files and for shared read-write access hosts must coordinate usage of data blocks with other hosts.

In SAN file systems built upon a block control unit, space allocation is managed by some form of metadata server [6, 18, 26], typically in concert with smart client involvement. By having this metadata server run on a cluster and partitioning responsibility between the nodes of the cluster, a good degree of scalability can be achieved. However, there are limits. This coordination can incur several costs including false contention between hosts allocating space from different logical units and additional communication. This communication is both between the hosts and the metadata server and between the nodes of the metadata server (assuming the metadata server is clustered) to ensure metadata consistency. In addition, since a metadata server runs on a traditional compute platform, *i.e.*, one without a non-volatile RAM, there is either additional overhead to harden metadata updates or a risk of (meta)data loss. By contrast storage controllers (typically) have some form of non-volatile RAM (*e.g.*, to support fast writes). We can leverage this support to harden metadata if the storage controller performs space allocation.

A final issue in today's SANs with block storage devices is end-to-end management. When data was directly attached to the host that generated and used the data, end-to-end management was easy since everything was in a single box. However, with block storage as an independent entity, end-to-end management of individual data units is difficult; we would like to support functions such as quality-of-service for individual files and migration/replication of individual files. However, block control units only recognize blocks and logical units. Thus, the problems facing storage management are similar to the ones that arise in security: management at the level of a single block is not practical, since there are simply too many blocks, and management at the logical unit granularity is too coarse grain, since a given logical unit may contain data that requires different policies.

3. Related Work

Although object stores have received significant attention in the past few years, to date, there is no extensive literature on this topic. The trend towards network-attached storage was envisioned more than ten years ago [3, 16], but the concept of a higher level of abstraction for networked storage gained momentum with the paper by Garth Gibson, *et al.*, [13] (and their more detailed report [14]) that describes in detail the NASD operations. This work served as the basis for the first standardization effort [27] of an Object Storage Device specification. Since this work by Garth Gibson's group at CMU, there have been several projects which have included object storage. Unfortunately, each group has used this term in a somewhat different way.

One project which includes an object store is Lustre. Lustre [18] is a SAN file system that uses an object store as its storage infrastructure. Unlike the emphasis of our work which is initially on SAN security, the emphasis of the Lustre team in using an object store is to achieve improved scalability for a cluster/SAN file system. The specifications of Lustre's Object Store Target (OST) [5] contains the same base functionality as the command set of the T10/OSD standard [27]. There are, however, some substantive differences. Due to their initial focus on scalability, the OST command set does not yet include a credential parameter. In addition, the command set provides several extensions which make the OST closer to an active disk [23]. For example, the OST specification includes an `iterate` function which allows applying a function to a group of objects. The implementation of OST is layered on top of existing file system implementations, *e.g.*, [9, 22].

DSF [8], a predecessor of our current object store project, presents a novel architecture for file systems, where space allocation is delegated to the storage subsystem. A new *allocate and write* operation writes an extent of blocks and returns the address where the data was written. DSF did not focus on security, and its storage abstraction remained close to the block storage abstraction. However, the first implementations of the current object store, relied on techniques developed in DSF to guarantee metadata stability and consistency.

Centera [7] provides a subset of the characteristics of object stores. Centera supports fixed content, such as medical images, streaming audio/video, e-mail, *etc.* Objects in Centera are streams of bytes paired with metadata. When an application creates and writes an object, Centera generates a key based on the object's contents and the metadata

(including the creation date, filename, *etc.*) which is returned to the application. This key serves several purposes. First, instead of a hierarchical directory as maintained by a file system, Centera maintains a flat mapping of keys to objects, *i.e.*, keys serve the role of identifiers. Since these keys are derived from the contents of an object and objects are immutable, the key can be used by the application to authenticate that the object returned is indeed the object that was stored by the application. In addition, since identical content results in identical keys, Centera can eliminate duplicates of objects, and it can simplify management of object replication. However, this approach of deriving an object's identifier from its content comes at a price. Since all objects in Centera are immutable, and the objects data must be completely available when the object is created (in order to calculate the object's key), the Centera solution is limited to fixed data. In other words, there is no need for Centera to support operations such as `write`, `truncate`, *etc.*

Somewhat further from our concept of an object store, is LegionFS [29], and its commercial follow-on Avaki. This is another example of an object-based file system. Legion focuses on providing a set of infrastructure and services for Grid computing. It defines an object model that encapsulates not only storage, but also users, hosts, schedulers, *etc.* In addition, a location-independent naming structure allows objects, including the BasicFileObject, to migrate transparently. However, LegionFS's definition and usage of an object-based storage are at a much higher level than the object store defined in this paper.

In addition to looking at related work from the perspective of overall functionality, we look more specifically at the problem of security for a storage network. The problem of protecting a network accessible storage system in a non-trusted environment has received much attention lately; a comprehensive survey can be found in [21].

Gobioff [15] in his thesis and the Network Attached Secure Disk architecture (NASD) system [11] base their access control mechanism on basic capability cryptographic primitives, which allow synchronous enforcement of security policy with asynchronous involvement of the server. Unlike our layered approach, they use the credential for securing the transport layer as well as for authorization purposes.

Authenticated Network-Attached Storage [20] provides an architecture which mutually authenticates the network disks and clients. It is based on cryptographic one-way hash functions, mainly for performance reasons, and does not require additional key management schemes beyond the existing authentication mechanisms within the system. It is mainly concerned with determining the client's access rights.

In [19], a security system for network-attached storage called SNAD is developed which stores and transfers encrypted data, and decrypts it only at the client. Despite the extensive use of encryption this system reports a reasonable performance overhead.

4. What is an Object Store

Unfortunately, as can be seen from the prior section, *object store* is used by different people to mean different things. It is thus necessary to define what we mean. We view an object store as analogous to a logical unit (LU). Unlike a traditional block-oriented logical unit which provides access to an array of unrelated blocks, an object store allows access via *storage-objects*. A *storage-object* is a virtual entity that groups data considered by a client to be related. It is similar to a byte-stream file in a flat file-system with a conceptually unlimited size. Space for a storage object is allocated on demand by the object store control unit, *i.e.*, sparse allocation is supported.

The collection of storage-objects, *i.e.*, an object store, forms what is, essentially, a primitive flat file system. There is no name space – just a flat ID space. The object store provides security enforcement for access to the storage-objects it contains, but it does not provide security management, *i.e.*, the object store does not determine who is allowed to access an object – it only enforces access rights determined by some external security administrator. Initially, we expect the most common use of an object store to be as the underlying infrastructure for secure SAN file system; eventually, we believe there will be additional uses.

An object store provides a level of virtualization and aggregation; more significantly it provides data path security. Thus, it is only natural for an object store to be in the data path. We believe that the best place to realize an object store is on a storage control unit. A storage control unit is already in the data path and typically has some form of non-volatile memory and a cache which we can leverage for metadata. Control units also provide sufficiently flexible programming environments to simplify the development of microcode to support an object store. As shown in figure 1, a control unit can provide both object and block interfaces and export multiple object stores. An alternative to developing an object store in a control unit would be for disk drives to directly support an object store interface. While such an approach is possible, we believe that the restrictions inherent in drive controllers (*i.e.*, limited cache memory, restricted development environment, *etc.*) will favor control unit based implementations for the foreseeable future.

An object store secures all operations with a credential which includes the set of operations the client is allowed to perform and an integrity code. Simply providing a credential on each operation, even if the credential is not crypto-

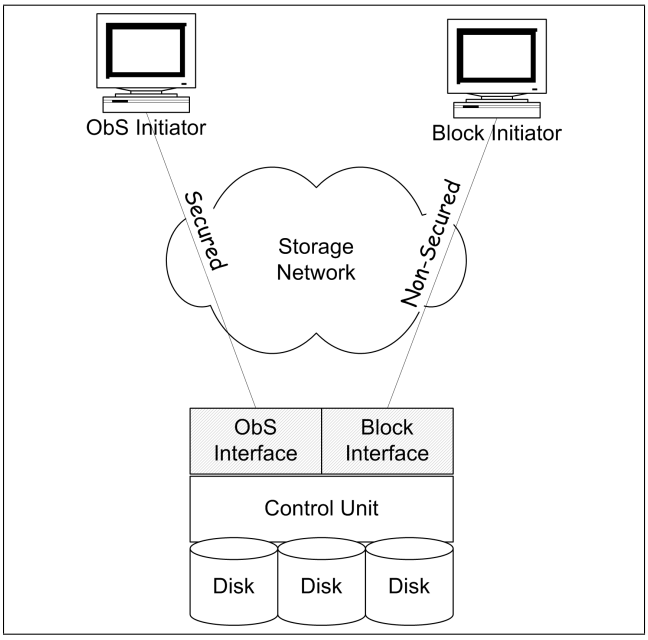

Figure 1. Location of an Object Store in a SAN

Parameters: Object Store ID, OID, Offset,Length, Credentials, Data

- Receive request
- Validate credentials
- Find allocation data for indicated object
- Determine if the indicated range is already bound to a collection of underlying LBAs
- If not already bound
  - Determine the mapping
  - Update the metadata
- Destage the data to the indicated LBAs

Figure 2. Basic Abstract Flow of Write

graphically protected, provides protection since (due to the integrity code) it is not possible to accidentally present a valid credential for an operation. To provide security, however, some form of cryptographic protection on the credential is required. Object store security provides increased protection/security at level of objects rather than whole volumes, thus allowing non-trusted hosts to sit on the SAN and allowing shared access to storage without giving hosts access to all data on volume. In addition, since hosts do not directly process or access allocation metadata, we provide an additional level of protection since it is not possible for misconfigured or buggy hosts to destroy the allocation metadata.

While different proposals for object storage vary in the details of the functions they provide, in almost all proposals, an object store provides (at least) the following basic functionality:

- Create or delete an object
- Read from or write to a byte range within an object
- Format, get object store info, . . .

After an object is created, the object is identified by an object ID (OID). We assume that after an object is created it is the client's responsibility to remember the OID. The client must present this OID to read or write the object. To be concrete, figure 2 shows the abstract flow for a write operation implemented in an object store control unit sitting on top of a set of conventional block devices.

5. Antara

Antara is our prototype implementation of an object store as a stand-alone control unit. Clients communicate with *Antara* using the *Antara* protocol, which is similar to the core functionality of the T10 proposal [27], supporting commands to open/close session, create/delete object, read/write/append/truncate, format, *etc.* The protocol transport is IP, but our design is fairly transport independent; supporting Fibre Channel should require changing only *Antara*'s input and output modules (see below).

Any data or metadata operation submitted to *Antara* is guaranteed to be recoverable. *Antara* realizes this guarantee by using journaling techniques to enable it to quickly restore a consistent state after a failure. The current implementation of *Antara*, as a control unit, assumes a non-volatile store, which it leverages to provide metadata recoverability.

This section describes the main design points of *Antara*, focusing on the metadata data-structures and control flow. We also describe *Antara*'s allocation strategy, which is aimed at ensuring consecutive allocation for individual objects.

5.1. Metadata

In *Antara*, we map an object store to a single underlying block volume, where this volume contains the user data as well as the object store's metadata. In other words, each block in an object maps to a logical block on the underly-

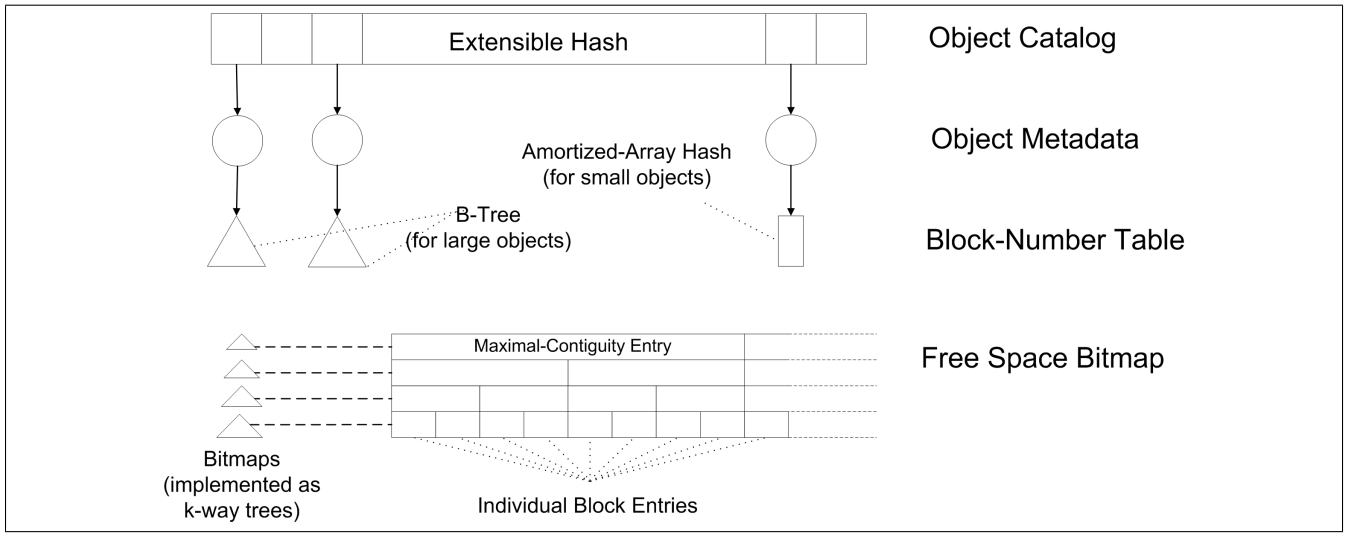

Figure 3. Antara Metadata

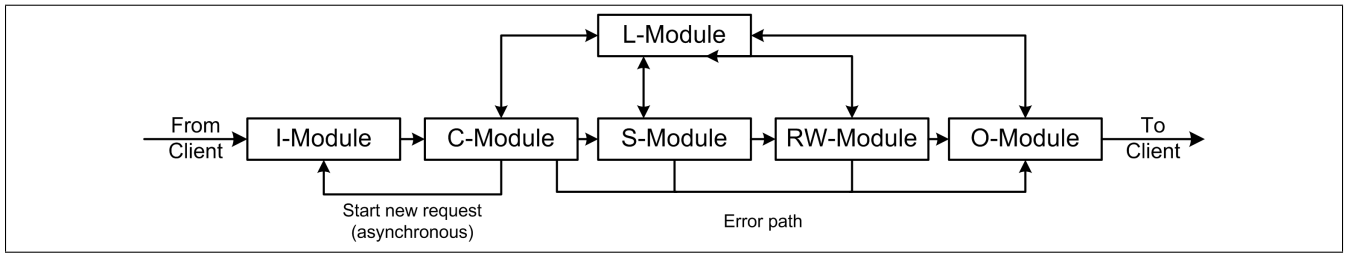

Figure 4. Structure of Antara

ing block-oriented device. The main use of the persistent metadata used by *Antara* is to support this mapping.

The metadata for *Antara* consists of the following structures (which are shown in figure 3):

- free-space bitmap, organized as a buddy list [1]. Unlike traditional implementations of buddy lists, our buddy list is truncated, *i.e.*, it does not use granularities ranging from a single unit to the entire range, but rather stops at the granularity of the maximum extent we allocate (currently 128K).

- object catalog which maps from OID to per-object metadata via an extensible linear-probe hash table. Shared/exclusive locks, created on-the-fly, allow locking only specific entries of the hash table.

- object metadata which includes the object's length and a "pointer" to the block-number table for the object

- block-number table which maps from offset in the object to the extents used for the object's storage. For small objects this is implemented as a dynamic, linear-probe hash table, while larger objects make use of a B-tree. The block-number table implementations were chosen to minimize page faults. For large objects, B-trees were chosen based on similar choices in databases. For small objects, an entire B-tree page is inefficient in terms of space and time (due to the use of an integral number of pages, more complicated algorithms than ones using simpler structures such as hash tables, *etc.*.).

5.2. Control Flow

The general flow of *Antara* can be viewed conceptually as a pipeline consisting of the following modules:

I-Module communication input, connection management

S-Module security and protection

C-Module control and dispatching

L-Module lookup, meta-data tables, locking mechanisms, and log of meta-data

RW-Module read-write of data

1. (I) Receive control block, determine needed buffers, allocate and receive data into buffers
2. (C) Mark the request as running; Delay this request if it clashes with other operations
3. (C,I) Notify OS ready to receive the next request from the client
    - Additional requests handled in parallel
4. (S) Perform protection and security checks (e.g. proper session, credentials, etc.)
5. (L) Perform necessary lookups (and/or allocations in case of writes)
    - Put allocation information in the request
6. (RW) Read/write the data from/to the cache
    - The cache manages a collection of buffers; reading/writing the data from/to cache involves pointer manipulation.
7. (L) Mark the request as done
    - Allows delayed requests to start, e.g., read-write conflict although read response may need to wait for log complete
8. (L) Log the request (no logging is required for read)
    - Ensures hardening of the allocation information
9. (L) Mark the request as logged (this is a no-op in the case of read)
    - Allows delayed requests to complete
10. (O) Send the reply
11. (L) Mark the request as sent
12. (O) Deallocate buffers used for the request

Figure 5. Flow for Read and Write in Antara

O-Module communication output

In most cases, the same operating system thread handles multiple stages. To minimize context switch overhead, *Antara* employs a run-to-completion model; a new request is processed only if the main *Antara* thread cannot make progress on the current request.

Figure 4 shows the communication between these modules and figure 5 shows how these modules communicate to implement an I/O operation. As the figures show, we can have multiple active requests in the pipeline at any point in time. The L-module is outside of the main flow since the metadata describing the layout of the object on the device is accessed at multiple points during command processing. For instance, prior to dispatching a write to the RW-Module, the control module must determine if the write requires new storage to be allocated. This allocation is logged, however, only after the RW module has obtained and hardened the user's data. We delay hardening the allocation information to avoid failure scenarios in which the allocation is recorded but the allocated blocks actually contain old data, *i.e.*, data not belonging to this user. One additional point to note is that *Antara* provides a zero copy implementation (other than copies by the network infrastructure).[2]

We now look at figure 5 in more detail; this figure shows the flow for a read or write operation. In the first step the request is received from the network and data associated with the request is placed in transient buffers. These buffers are obtained from a pool of buffers; buffers are managed by reference counting. In the *Antara* implementation, all data received is placed in a new buffer, and the cache works by changing pointers to the buffer that currently contains the data.

After the command is received by the input module, the control module determines if the command can execute; we delay execution of a command only if executing this

[2]Since we have chosen to implement *Antara* as a user process over IP to allow easier experimentation, we do have the copy inherent in user mode access to the network.

command in parallel with a currently executing command could cause a lack of integrity for *Antara*'s metadata. After this initial processing, we notify the operating system that we are ready to receive additional requests from the network. If new requests are available, the operating system will asynchronously notify the main *Antara* thread, which will retrieve the request when it is no longer able to progress with the request it is currently processing.

The next step, step 4, is performed by the security module which validates this is a legal request. The L-module next performs the metadata translation to map from an OID, offset and length to a (set of) extent(s) in the underlying block store. We then transfer the data associated with the request to/from the cache which is managed by the RW module.

After the data is transferred to the cache, certain conflicting operations are allowed to start execution. For instance if we had a `read` operation that was trying to access an offset just allocated by a `write`, we can allow the read to begin executing. We cannot, however, allow the read to return data to the host until the metadata associated with this new allocation has been hardened in step 8. After we have logged these metadata updates we allow delayed requests, such as the read, to complete.

After sending any results to the host, we clean up the resources used by the processing of this request.

5.3. Allocation

One way an object store can provide better management is by internally leveraging the knowledge it has of which blocks are related and allocating these blocks according to a policy which is appropriate for the type of object.[3] One such policy, which *Antara* currently implements, is to consecutively allocate blocks of an object; another policy which one might consider is striping.

Implementing such a policy is difficult since the object store does not know what allocation requests, *i.e.*, writes to unallocated offsets, will be executed in the future, and without knowing the future it is hard to determine if a specific allocation decision is appropriate. There are two orthogonal ways to address this problem.

First, we can delay the decision of where to allocate a particular offset. Instead of binding a block in an object to a block on an underlying device at the time the object store processes the write, we can take advantage of the control unit's non-volatile store and have the implementation of the write operation store the data in the cache indexed by its OID and offset. At some later point in time, *e.g.*, when it is necessary to destage the data to the underlying block device, we can bind the object offsets to actual blocks on the device. However, since we have delayed this binding, the binding will be performed in the context of more information about how the host is using the object, *i.e.*, we know all the writes that occurred between the time this data was written and the time of the destage.

The second approach to addressing the on-line nature of the problem is to maintain a cache of objects which are currently active for allocation. With each of these objects we can associate a collection of blocks which will lead to optimal allocation assuming this is the only active object. If this cache is managed in a least recently used manner and is sufficiently large, we can achieve good allocation behavior. The current *Antara* implementation uses this approach.

[3] We assume the policy to be applied will be specified by a mechanism beyond the scope of this paper.

6. Security

Unmediated host access to a control unit raises new security concerns: malicious parties forging messages or tampering with message contents, replaying or recording messages, spoofing a user's identity or denying service of valid requests. Thus, in order to achieve a level of security comparable to traditional systems that do not offer shared storage, the storage server needs to take an active role in the system's security mechanism.

Our goal is to create a security mechanism capable of enforcing arbitrary access control policies, such as, for example, the ones used by Windows or UNIX file systems. The protection mechanism should be able to operate in a secure-mode for environments in which hosts cannot be trusted and security against network attacks is required, as well as in a (cheaper) protection-only mode in secure environment (*e.g.*, a "glass house") where only protection against application errors is required. In addition, the security mechanism should have an acceptable performance-overhead and be easy to deploy and manage. This section gives a high level overview of our security model; [2] gives a more comprehensive description, including implementation issues and proof of correctness.

6.1. Solution Outline

Our security model assumes that the system is comprised of three types of logical entities: (i) *hosts*, which initiate the I/O requests, (ii) *Object Store (ObS) servers* which expose an object store interface and directly manipulate the storage, and (iii) *Admin*, the security administrator which is in charge of authorizing hosts requests. Our trust model assumes that users trust their own host's operating system (but not that of other hosts), and that the Admin and the ObS servers are trusted.

Our solution is credential-based: before a host can send a command to an ObS-server it must obtain a suitable credential from the Admin. Protection enforcement

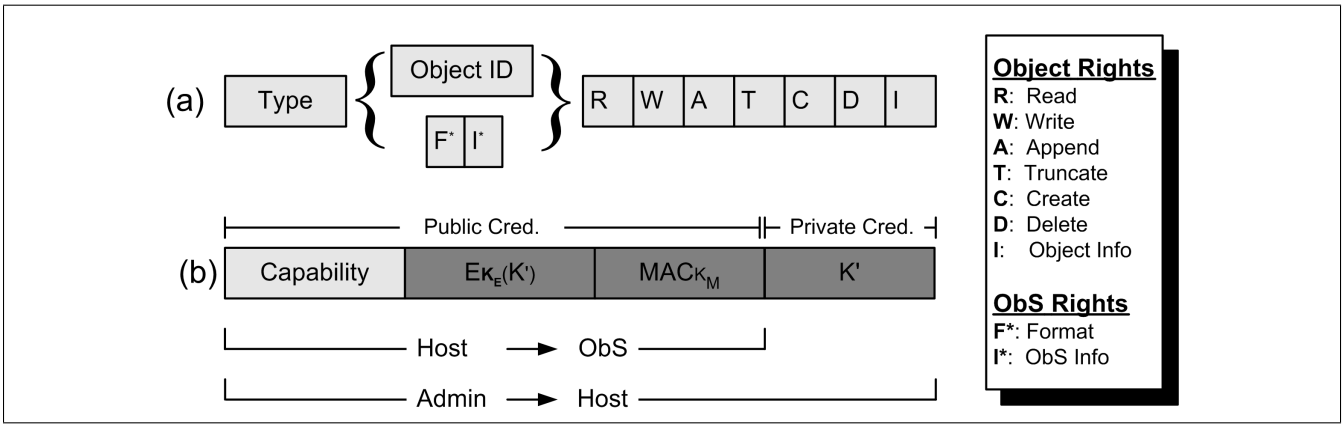

Figure 6. Security protocols: (a) Capability structure (b) Credential structure. $E_{K_E}(K')$ is the credential secret (K') encrypted with K_E - the encryption key. MAC is computed with K_M.

is achieved by the cooperation of the Admin and the ObS server. The Admin authenticates the client, authorizes requests according to the system access control policy, and generates credentials. The ObS-server validates that the presented credentials suffice for the requested operation and that the credentials were neither forged nor modified. The credential is cryptographically hardened by two secret keys shared between the ObS and the Admin: K_E - an encryption key and K_M - a message authentication code (MAC) key. This pair of keys is periodically refreshed.

For non-secure environments, our protection mechanism must be coupled with a mechanism that secures the transport. Otherwise, it could be vulnerable to malicious message modification, replay attacks, and eavesdropping. For IP-networks, we use for this purpose the standard, off-the-shelf, IPSec protocol [17]; although, we could in theory use any other protocol that provides secure channels. For Fibre Channel, we currently assume a protected network although due to our layered approach we should be able to directly take advantage of the work on Fibre Channel security when it is completed [28].

Note that our proprietary credential-based protocol for authorization is defined *on top* of the communication layer. Thus, we separate the mechanisms for transport security — mechanisms which are widely studied, well understood and tested — from our proprietary mechanisms used solely for access control. This is where we diverge from the previously suggested credential-based approach of [15] and others that bind the transport security with the enforcement of access control.

6.2. Capabilities and Credentials

As stated above, a host accompanies each request it initiates with a *credential* that attests that the host is authorized to perform the request. In essence, a *credential* is a cryptographically hardened *capability*. The *capability* (see figure 6) encodes the host permissible operations. There are two types of capabilities: object-capability and ObS-capability. An object capability contains operations that can be applied on a specific object, *e.g.*, read or write. An ObS-capability includes operations applicable to an entire ObS-server, *e.g.*, format, and operations on objects, that are applicable to *any* object in the ObS-server.[4] We note that the correctness of our protocol is independent of the details of the structure of the capability.

Technically, the *credential* is comprised of (i) a *public credential* – a capability, the credential secret K', encrypted key K_E, and a MAC and (ii) a *private credential* – the credential secret (K'). The credential secret is unique for every credential; it is chosen by the admin, encrypted with K_E and embedded in the credential as $E_{K_E}(K')$. The integrity and authenticity of the public credential is assured by the MAC calculated with the authentication key K_M shared between the ObS-server and the admin. Note that the private credential must be provided secretly to the host since it is not encrypted. The host does not send the private credential to the ObS-server, rather it uses it to convince the ObS-server that it received the credential in a legitimate way, as explained in Section 6.3.3.

6.3. Security protocols

From the security mechanism perspective, a host wishing to perform an I/O request should follow the following steps:

1. The host authenticates to the Admin via an exist-

[4] ObS-capabilities may be used to expedite administrative operations, *e.g.*, backup.

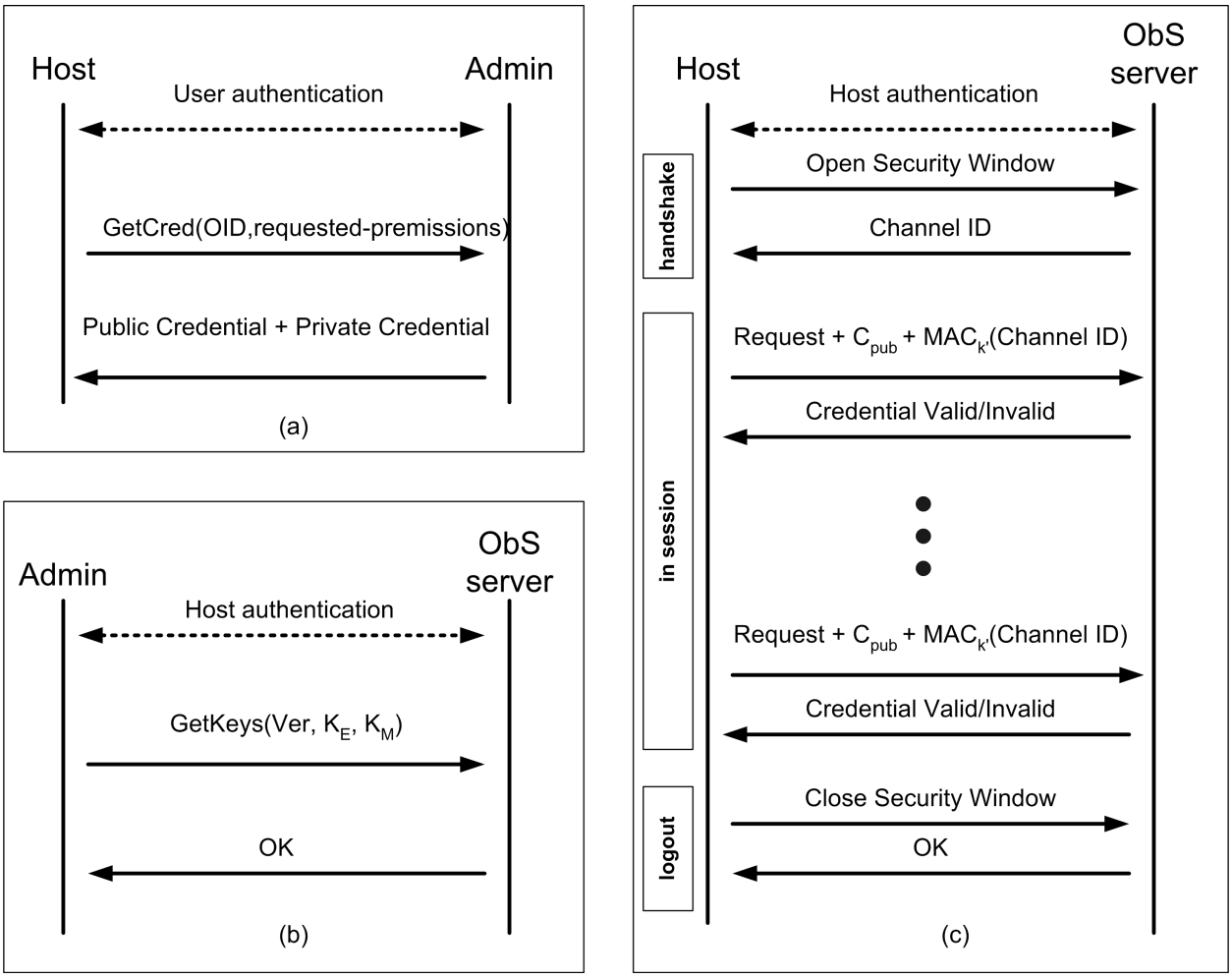

Figure 7. Security protocols: (a) Host-Admin; (b) Admin-ObS protocol; (c) Host-ObS

ing mechanism (on behalf of the user running on the host).[5]

2. The host requests a credential from the Admin for a particular operation.

3. The Admin sends a credential to the host.

4. The host sends request(s) to the ObS with the obtained public credential along with a proof that it knows the credential secret.

5. The ObS-server verifies the credential (and the proof) and performs the operation.

Our security solution (see figure 7) is composed of three protocols: (i) a Host-Admin Protocol, initiated by the host whenever it needs to obtain a new credential, (ii) an Admin-ObS Protocol used to establish and refresh their shared (secret) keys K_E and K_M (see 6.1), and (iii) a Host-ObS Protocol used by the ObS to validate that the requests it gets from the host were authorized by the Admin. Below is an outline of our security protocols. A detailed description can be found in [2].

6.3.1. The Host-Admin Protocol This protocol, depicted in figure 7(a), is rather straightforward: the host asks the Admin for a credential for a specific operation on an object in the name of an already authenticated user. The Admin checks that a user is allowed to perform the operation according to the protection policy and generates an appropriate credential. Note that the Admin is required to generate a unique secret, and perform one encryption operation and one MAC operation for every credential it generates. In addition, the Admin must send its reply to the host

[5] Our solution does not impose a particular user authentication mechanism and can utilize any mechanism available in the system, *e.g.*, Kerberos, login/password, PKI certificates, etc.

on an encrypted channel since the private credential, which contains the credential secret, is not encrypted.

6.3.2. The Admin-ObS Protocol Recall that our solution requires that the ObS-server and admin share two keys which are refreshed periodically: an encryption key K_E and an authentication (MAC) key K_M. This protocol, depicted in figure 7(b), is basically a key-exchange protocol intended to refresh these keys (indicated by the 'Ver', the keys version number). This protocol runs over an encrypted channel; after a mutual authentication, the Admin sends to the ObS a new pair of keys (K_E and K_M) followed by an acknowledgement from the ObS-server.

6.3.3. The Host-ObS Protocol This protocol, depicted in figure 7(c), is the core of our solution. The protocol is comprised of three stages:

handshake The host requests an 'open security window' with the ObS-server. The ObS-server responds with a randomly chosen channel name *ChannelID* (which the ObS associates with the 'session'.)

in session A host sends a request to the ObS-server, along with a public credential $\mathcal{C}_{pub}$ and a validation tag $V = MAC_{K'}(ChannelID)$. The ObS-server first verifies that the requested operation is permitted by the credential. If so, it verifies the authenticity of the credential (verifying the credential MAC using the shared key K_M). Finally, by extracting K' from the credential (using the shared key K_E) it verifies the validation tag V. If any of the checks fail, the request is denied; otherwise it performs the request. This is repeated for multiple requests (possibly requiring new credentials) as long as the security window is intact.

logout The host closes the security window with the ObS-server. The ObS-server clears the security window from its internal tables and acknowledges.

Note that the host can send many requests to the ObS-server during the "in session" stage, thus the handshake cost is amortized over many requests, *e.g.*, the entire host-ObS session. A key property of our solution is that the credential verification is fast and involves only symmetric key operations, hence the critical path remains short. To further reduce the incurred security costs, we use a capability cache.

7. Results

While our work is just beginning, we have made some initial performance measurements. We have measured the performance of *Antara* serving I/O requests on a 100% cache hit workload. We ran our tests running *Antara* on a Pentium-4, 2.4Ghz processor running Windows 2000. In our experiments, we ran two clients (also running Windows 2000); the network was a 1 Gb Ethernet. In these tests, we were able to process over 14,000 object store reads per second. We observed that the bulk of *Antara*'s CPU time was spent in processing the network communication protocol.

To estimate the performance overhead of the security mechanism, we measured the cost of verifying a credential. This operation is critical to performance since it occurs on the data path. Running our prototype implementation on a 2.0 Ghz Pentium-4 machine with 1GB memory under RedHat Linux (2.4.18) and using RSA BSafe-C6.0 library [24] for the cryptographic primitives, we were able to verify 20,000 credentials in a second (without a credential cache). Adding a credential cache based on the marker algorithm [10] can improve the verification rate by up to a factor of 50. In practical terms, given a hit in the credential cache, security accounted for less than 5% of *Antara*'s overhead.

8. Conclusions

We have described our design and implementation of an object store along with a mechanism for providing security. Our security mechanism is novel in that it divorces transport security (where we rely upon existing well-studied mechanisms) from the protocol for ensuring only authorized users are allowed access.

To conclude, we believe much work is still required. Object stores will almost certainly happen; it is just a question of when. As we described in section 2, we believe we are now at the point where the ability to leverage the benefits of an object store to democratize the access to data is sufficiently significant to justify the cost of the paradigm shift. It is thus imperative to understand how to build a high performing object store which provides a range of the potential benefits. We are continuing our research to this end.

Acknowledgements: We would like to thank Randall Burns, David Pease, Ralph Becker-Szendy and Miriam Sivan-Zimet of the IBM Storage Tank team.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1983.

[2] A. Azagury, R. Canetti, M. Factor, S. Halevi, E. Henis, D. Naor, N. Rinetzky, O. Rodeh, and J. Satran. A two layered approach for securing an object store network. *IEEE International Security In Storage Workshop*, 2002.

[3] A. D. Birrel and R. M. Needham. A universal file server. *IEEE Transactions on Software Engineering*, September 1980.

[4] P. J. Braam. The Lustre storage architecture. Technical report, Cluster File Systems, Inc., 2002. http://www.lustre.org/docs/lustre.pdf.

[5] P. J. Braam and A. e. Dilger. Object based storage. Technical report, Stelias Computing, 1999.

[6] R. C. Burns. *Data Management in a Distributed File System for Storage Area Networks*. PhD thesis, University of California, Santa Cruz, March 2000. http://www.almaden.ibm.com/cs/storagesystems/stortank-/rbdissert.pdf.

[7] EMC Centera, content addressed storage, product description. http://www.emc.com/pdf/products/centera/-centera_guide.pdf, 2002.

[8] Z. Dubitzky, I. Gold, E. Henis, J. Satran, and D. Sheinwald. DSF: Data sharing facility. Technical report, IBM Haifa Research Labs, 2002.

[9] Ext2fs home page. http://e2fsprogs.sourceforge.net/ext2.html.

[10] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young. Competitive paging algorithms. *J. Algorithms*, 12(4):685–699, 1991.

[11] G. Gibson, D. Nagle, K. Amiri, J. Butler, F. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 92–103, 1998.

[12] G. Gibson, D. Nagle, K. Amiri, F. Chang, E. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka. File server scaling with network-attached secure disks. *Proceedings of the ACM International Conference on Measurement and Modelling of Computer System, Seattle, WA*, June 1996.

[13] G. Gibson, D. Nagle, K. Amiri, F. Chang, H. Gobioff, E. Riedel, D. Rochberg, and J. Zelenka. Filesystems for network-attached secure disks, 1997.

[14] G. A. Gibson, D. P. Nagle, K. Amiri, F. W. Chang, E. Feinberg, H. G. C. Lee, B. Ozceri, E. Riedel, and D. Rochberg. A case for network-attached secure disks. Technical Report CMU–CS-96-142, CMU, 1996.

[15] H. Gobioff. *Security for High Performance Commodity Storage Subsystem*. PhD thesis, CMU, July 1999.

[16] R. H. Katz. High-performance network- and channel-attached storage. *Proceedings of the IEEE*, 80(8), August 1992.

[17] S. Kent and R. Atkinson. Security architecture for the internet protocol, RFC 2401. http://www.ietf.org/rfc/rfc2401.txt, Nov. 1998.

[18] Lustre home page. http://www.lustre.org.

[19] E. Miller, D. Long, W. Freeman, and B. Reed. Strong security for network-attached storage. *Proceedings of the FAST 2002 Conference on FIle and Storage Technologies*, January 2002.

[20] B. C. Reed, E. G. Chron, R. C. Burns, and D. D. Long. Authenticating network-attached storage. *IEEE Micro*, 20(1):49–57, January 2000.

[21] E. Reidel, M. Kallahalla, and R. Swaminathan. A framework for evaluating storage systems security. *Proceedings of the 1st conference on File and Storage Technologies (FAST)*, January 2002.

[22] H. T. Reiser. Reiserfs. http://www.namesys.com.

[23] E. Riedel, G. A. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. In *Proc. 24th Int. Conf. Very Large Data Bases, VLDB*, pages 62–73, 1998.

[24] RSA BSAFE(R) - CryptoC - cryptographic components for c reference manual - version 6.0, December 2001.

[25] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. *iSCSI*. IETF, draft-ietf-ips-iscsi-17 edition, September 2002.

[26] SGI. *SGI CXFS: A High-Performance, Multi-OS SAN Filesystem from SGI*, May 2002. http://www.sgi.com/Products/PDF/2691.pdf.

[27] Object based storage devices command set (OSD). http://www.t10.org/drafts.htm. T10 Working draft.

[28] Fibre channel - security protocols (fc-sp). http://www.t11.org/.

[29] B. S. White, M. Walker, M. Humphrey, and A. S. Grimshaw. LegionFS: A secure and scalable file system supporting cross-domain high-performance applications. In *Proceedings of the IEEE/ACM Supercomputing Conference (SC2001)*, Nov 2001.

[30] H. Yoshida. LUN security considerations for storage area networks. Technical report, Hitachi Data Systems, 1999.