Θfs: An Experimental Generalized Storage Server

Brian L. Stuart

ABSTRACT

As part of an ongoing research program studying techniques and architectures for data storage, we have investigated the possibility of unifying file, object, and block access to data stores. To date, the result of this investigation is a generalized storage server that provides all three methods of access. Using full path name hashing and extensible metadata, this server provides a highly flexible platform that can form the basis of a production server or of further storage research. Here, we discuss the background behind the server, the basic concepts of its design, and the directions for continued R&D planned for it.

1. Introduction

In the very earliest days of computing, data storage was primarily in the form of punched cards and magnetic tapes. A particular set of data was identified as much by what tape contained it, or what box contained the cards as any other identification. It was the advent of the disk drive that shaped our modern view of data storage. In particular, the disk had two characteristics that weren't found on tapes or cards: fixed relatively large blocks and random access. These two characteristics led to our modern picture of a particular set of data occupying one or more blocks of storage space and being labeled with a file name.

Very quickly, a model of data access developed where an application speaks to a file system component in terms of file names and offsets within the file. The file system code then talks to a block device driver to read and write individual blocks. In [1], we note that the job of the file system, then, is two-fold. First, it must manage the set of names for the files, and second, it must mange the allocation of blocks to files. These relationships are illustrated in Figure 1.

Beginning with research at Xerox PARC and widely deployed by workstation manufacturers like Sun and Apollo, the application often ran on a different machine than the file system. So by means of some network file access protocol, the application makes a request of the file system (server) which accesses disks in terms of blocks just as before. Over the years, numerous such protocols were developed with the most relevant to Plan 9 being NFS, CIFS (née SMB), and 9P. This approach has gained the name Network Attached Storage (NAS) in the trade.

Later work explored the possibilities of putting the network connection between the file system and the block devices, rather than between the application and the file system. Several protocols also grew up around this type of storage organization. Among them are FiberChannel, iSCSI, and AoE. A common trade term for a network system structured in this way is Storage Area Network (SAN). Figure 2 illustrates both file-oriented and block-oriented network accessible storage.

Recently, a third alternative for where to put the network connection has emerged [2]. If we put the connection in the middle of the traditional file system structure between the name management and the block management, we get what is often called an object

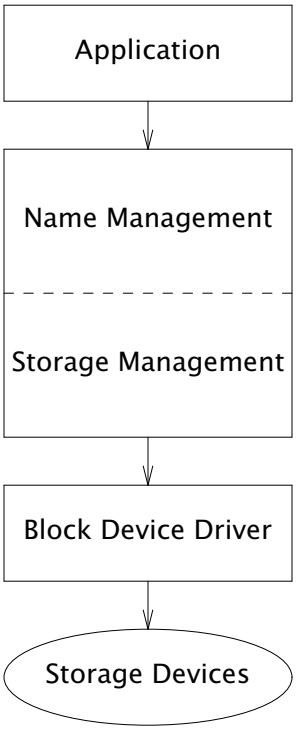

Figure 1: Storage Management Component Relationships

storage system. This structure is illustrated in Figure 3. For completeness, we should note that the concept of an object store does not necessarily have to be implemented over a network. However, it is the use of networked storage appliances as object storage devices that interests us here.

As with any set of successful technological alternatives, each of these storage architectures has strengths and weaknesses relative to the others. This means that some applications are better suited to communicating in terms of files, some in terms of blocks, and some in terms of objects. Therefore, we do not attempt to argue for one approach as universal in this paper. Instead, we examine the design of a storage server that provides all three means of access in such a way that they all coexist and share the same raw storage space.

2. Access Characteristics

To get a better sense of how a unified storage system should be structured, we need to look at how each type of storage access is specified.

2.1. Blocks

We begin by looking at how block I/O operations are specified. Normally, there are one or more collections of blocks identified by logical unit numbers (LUNs). (While semantically problematic, we often also refer to the actual storage space as a LUN.) The number of LUNs is relatively small, and the size of each is relatively large. Conceptually, each LUN is like a single disk drive. Once established, it generally does not change size, and therefore cannot grow as a result of writes. In practice, the set of blocks that make up a LUN may be spread across several drives, or sometimes a subset of a drive. From the perspective of a client using a block store, though, the whole of a LUN has characteristics much like those of a disk drive.

I/O operations on blocks are carried out in units of blocks, typically the size of historical

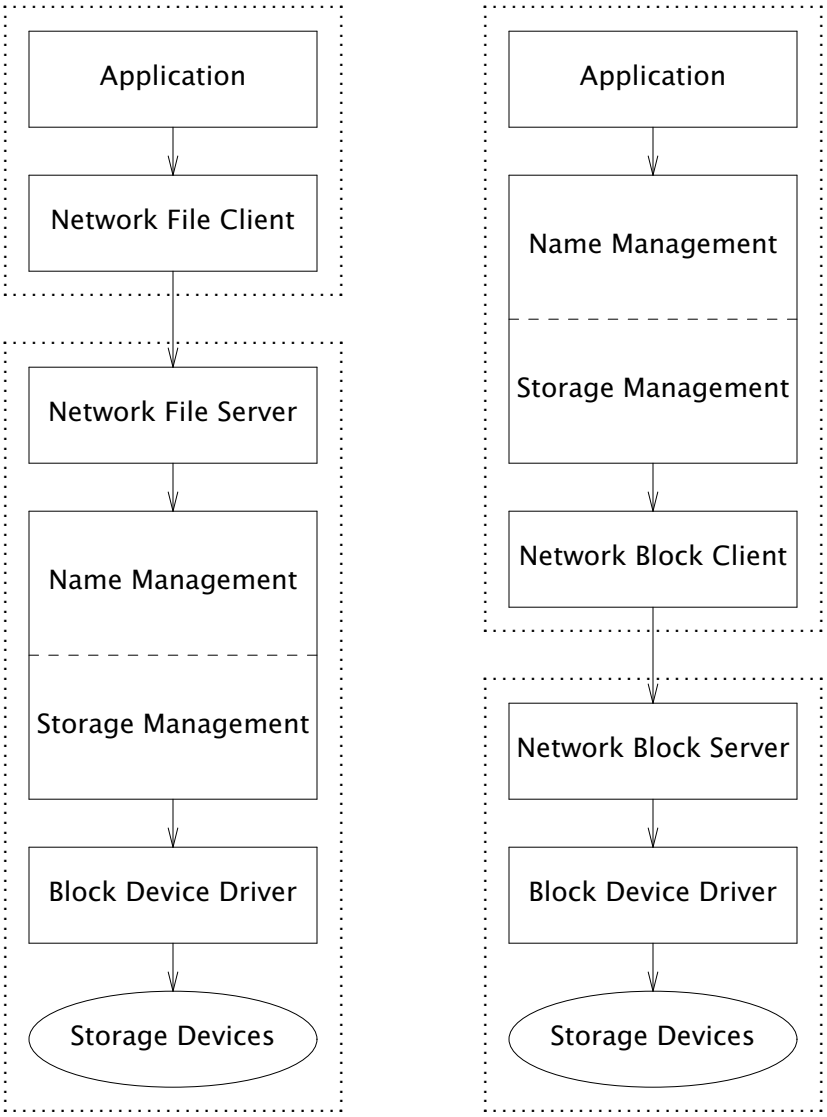

Figure 2: Network Storage Architectures: Network Attached Storage (left) and Storage Area Network (right)

disk sectors, 512 bytes. Often reads and writes will involve several blocks, but incomplete blocks are never transferred. Thus to specify a read or a write in a block store, we need a LUN, a starting block number, and a count of the number of blocks to read or write. Taking AoE as an example, LUNs are identified by a 24-bit target number, block numbers by a 48-bit LBA, and block counts by an 8-bit sector count.

One last issue of block storage is the allocation policy. Are blocks allocated administratively before placing the block store in use, or are they allocated on demand as they are written? Following the historical connection between LUNs and physical storage devices, block storage systems originally operated according to the administrative allocation policy. As the option of demand allocation became available, it was usually described with the trade term thin provisioning. By contrast, the traditional administrative allocation came to be called thick provisioning. In most cases, dynamic allocation is implemented much like sparse files in a traditional UNIX file system where only the written blocks are allocated and not intervening unwritten blocks.

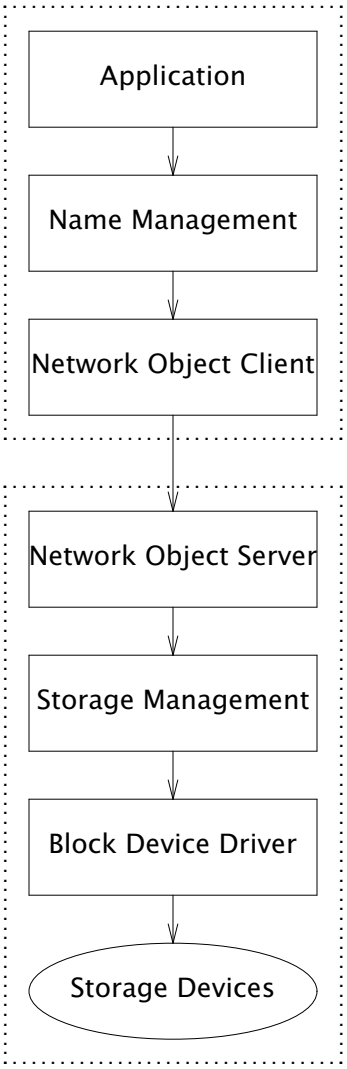

Figure 3: Network Object Storage Architecture

2.2. Objects

Similar to the LUNs in block storage, objects are generally identified by a numerical designation. In the case of the SCSI object storage device specifications, this designation is a pair of 64-bit numbers described as the partition ID (PID) and the object ID (OID). As suggested by the large number of identification bits, there are generally many more objects in an object store than LUNs in a block store. Unlike block stores, I/O in object stores is specified in terms of the byte offset and byte count, rather than block number and block count. Finally, the allocation policy of objects is usually dynamic, possibly with administratively established upper limits.

2.3. Files

Files are, of course, the most familiar of the three storage paradigms we are considering. They share much in common with objects. Accesses are specified in terms of byte offsets and byte counts. Allocation is typically on-demand allowing files to grow. The primary difference between files and either objects or blocks is the means of identifying a file. Rather than a fixed numeric identification, files are identified with strings that have few (if any) constraints. The names are also usually organized hierarchically.

Although object IDs can be thought of as degenerate case of file names, the SCSI identification scheme only provides for two levels of hierarchy (partition and object).

3. The Naming/Allocation Management Boundary

At this point, we must admit to some degree of hand-waving and over simplification. In particular, the boundary between the naming functions and the space management functions of the file system implementation is a fuzzy one. We've described the hierarchical relationship among files as being a property of the naming. In practice, that hierarchy is also usually represented in the on-disk data structures. As discussed later here, however, that coupling between what are otherwise two independent functionalities is unnecessary, and removing it opens up a number of interesting possibilities.

4. Design Motivation

We can see from the similarities between files and objects that they practically beg for a unified implementation. In fact, in one of the early presentations on object storage, the question was asked whether objects should be implemented on top of files, or files on top of objects.

The first approach borders on trivially easy. All one need do is create files whose names are the object numbers expressed in a human-readable notation. For example, if we create one directory for each of the partitions in the SCSI object storage device specification, and one file in each of those directories for each object in that partition, we can name both with a hexadecimal representation of the 64-bit numbers. As long as the file system allows names of at least 16 characters, then this kind of organization can be used without any modification of the file system design. The reason this approach is not often used is that most file systems are designed around the idea that a typical directory contains tens to hundreds of files. However, we can easily have millions of objects in a partition.

More commonly developed have been approaches where file systems are implemented on top of object stores. In these implementations, each file and directory is contained in an object. Because SCSI OSD defines a rich set of object metadata, the traditional UNIX i-node metadata need not be managed directly by the file system implementation. Exofs is an example of such a file system in Linux.

Regardless of what approach we take to connecting files and objects, it is clear that the key mechanism is a mapping between names and numeric IDs. With that perspective, it is clear that the infrastructure for a unified file and object system has been around for a long time. For example, the UNIX file systems of the 1970s mapped names to i-numbers that uniquely identified files. A little more recently, the 9P protocol identifies files by an identifier called a QID of which a 64-bit path number uniquely identifies a file.

One might suggest that the best approach to this unification is to replace the 64-bit path in the QID with the pair of 64-bit IDs in the SCSI OSD specification. However, our results show that translating both name and object IDs to an independent numeric identifier (such as a QID path) is more conducive to coexistence of files and objects and also provides a mechanism for integrating block access. To understand why, we first observe that the unique IDs for LUNs are usually administratively assigned, unique IDs for files are usually assigned by the file system, and unique IDs for objects are usually assigned by the client. So the benefits arise because mapping all client and administratively assigned identifications to an internally managed space of IDs prevents collisions between ID assignments.

For the remainder of this discussion of the design and implementation of our prototype generalized data store, Figure 4 illustrates its major components and overall structure.

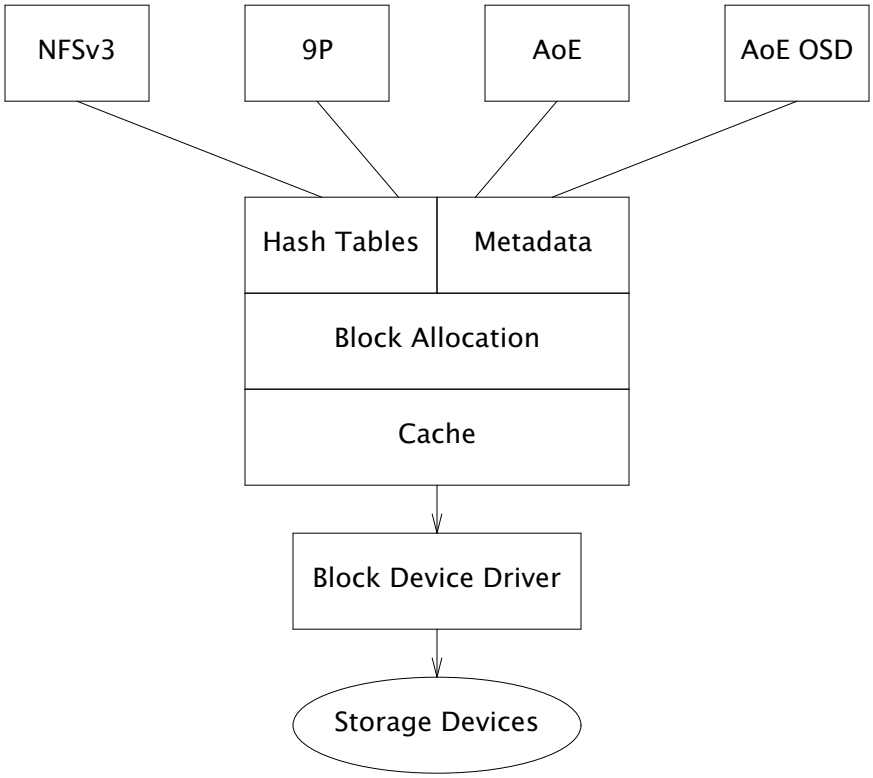

Figure 4: θfs Architecture

5. Name and ID Management Design

For the prototype we report here, we have opted to use the 9P QID path for the internally generated unique IDs. The high-order four bits of the QID path are a type specifier. This is used to allow multiple allocation strategies to coexist. The next 12 bits are reserved. Their planned use is as a node ID for future research in distributed storage. Finally, the remaining 48 bits of the QID path are assigned according to strategies suitable for different access mechanisms.

For mapping to these IDs, we use a hash table. Unlike other file systems using hash tables, we hash the full path name of each file to find the bucket where we find the file's ID. Similarly, for objects, we have a textual representation of the partition and object IDs. Collisions between file names and textual object IDs are prevented by constructing all path names beginning with a slash (/). The QID paths of both files and objects use a type field of 0. Block accesses, on the other hand, use a type field of 2. For the LUNs accessed using a block paradigm, the 48-bit field contains the AoE target number in the lower 24 bits.

With this arrangement, we can manage a storage space shared by files, objects, and block LUNs. Although not implemented in this prototype, this arrangement also allows the file server interface to provide access to both the objects and the block LUNs through special name spaces. For simplicity, we use the same allocation mechanism for all three storage types, resulting in naturally thinly provisioned LUNs.

6. Metadata Design

Most 9P servers are implemented in such a way that the names map directly to metadata structures that contain the QID. However, because we want to support access directly via the QID path to storage that is potentially unnamed, we map names only to QID

paths, and separately map QID paths to metadata. Unlike the i-numbers in a traditional UNIX file system, the QID path numbers come from a space too large to use as simple indices. Therefore, we use a second hash table that maps QID paths to metadata lists.

Each of the different types of storage entities requires a different set of metadata. For example, an AoE LUN needs to include the major and minor AoE target numbers, but doesn't need any information representing any kind of hierarchical arrangement among the LUNs. An object needs to include the PID and OID. Regular objects also need a mapping to blocks that comprise the object, but partition objects need a way of identifying those objects that are contained in them. Similarly, files served by 9P require owner and group identifications, a mode, access and modification times, and pointers that describe the hierarchical structure. Regular files need block mappings, but directories don't. Furthermore a full implementation of the SCSI OSD specification includes a substantial amount of additional metadata.

Even among different remote file access protocols the set of metadata varies. For example, for files accessed through 9P, we need textual owner and group names, and QIDs that contain a path, a version, and a type, none of which are found in NFSv3. Conversely, for NFS, we need to support numeric user and group IDs, symbolic links, device nodes, named pipes, etc, all unknown in 9P.

For these reasons, we have opted for an extensible metadata design. The basic idea is similar to that found in the VMS file system [1,3] and later in NTFS [1,4]. Each metadatum consists of a next pointer, a type, a name, and a value as illustrated in Figure 5. In the present prototype, the value field is a fixed eight-byte value. In each metadatum, this value is used in one of three ways. For integer valued metadata, the integers are stored as 64-bit numbers. For short strings (≦7 bytes), the string is stored directly in the eight-byte value. If the datum value is a longer string or an arbitrary byte sequence, the eight-byte value is an offset to where in the storage space the real value is stored. All of the metadata entries for a given storage entity are arranged in a linked list. Free metadata entries are also organized in a linked list.

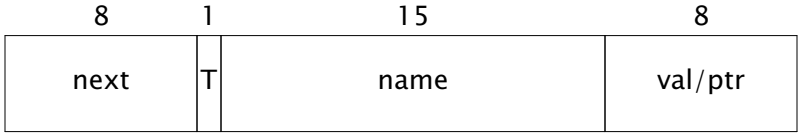

Figure 5: θfs Metadata Structure

Block mapping is similar to that found in the i-nodes of traditional UNIX file systems with a few simplifying differences. Each metadata block that identifies real storage space contains one datum that specifies the root block of a tree of block pointers that is either zero, one, two, or three levels deep. For the zero depth case, a single metadatum specifies one data block for small files. Entities created with a known size, such as LUNs, are created pointing to a tree of the appropriate depth. Entities created with an initial size of zero are created with a tree of depth zero, but with no data block allocated. If such a file or object grows too large to be represented by a single block of block pointers, a new block is allocated to be the root of a tree of depth two, and it's first element points to the existing block of block pointers. The metadatum that had pointed to the index block is replaced by one pointing to the newly allocated root. Similarly, if an entity grows too large for a two-level tree, a third level root is allocated and the metadata updated to point to it.

7. Design Implications

Managing names in terms of full paths has turned out to naturally break the coupling between name management and space management. There is no longer any need for a dedicated directory data structure on the storage media. In particular, we no longer have a unique root directory structure that is typically located by a pointer in a superblock. At first glance, this would simply be a mildly interesting point, hardly worth mentioning. However, it turns out there are a number of useful properties that fall out essentially for free from this simple design element.

First, the coupling created by on-disk directory structures made the integration of objects conceptually awkward. In fact, most presentations of object storage redefine the function of the subsystems on either side of the point at which an object protocol is injected. However, by making the hierarchical relationships a function of name management alone, the integration of files and objects can be quite straightforward as illustrated in the preceding sections.

The second interesting capability that falls out of this simplification arises from the lack of a unique root identified in a superblock. It becomes almost trivial to support multiple independent file systems, each with its own root directory. One simple way to make use of this capability is to let each file tree be named by a ''root'' which begins with the slash (/) character, and as we construct path names, we precede each path component by a slash as a separator. Thus, the sys directory of an unnamed file tree would be internally known by the string ''//sys'' and by ''/fs1/sys'' for a file tree named fs1. The attach message of the 9P protocol makes provision for clients to specify a named file tree when establishing an initial connection to the server. Similarly, the mount RPC protocol associated with NFS has provision for a string identified as a directory path. Normally, it is used to mount a subtree of a single file system served by a file server. However, if we generalize the string to simply specify the root of some file tree, then we can use it in the same way as the file tree specifier of the 9P protocol and select among independent file systems.

Finally, in reading this discussion, one can reasonably conclude that this is just a complicated way of saying there's a higher-level root that contains all the object, all the LUNs, and all the independent file systems. Indeed, this is an excellent way to view the overall structure. Viewing the structure from the perspective of an implicit root opens the possibility of crossing access method boundaries. For example, if we define a file system called ''/object'' then we can make all object accessible via 9P or NFS.

8. Experimental Object Extension to AoE

To provide a basis for development and testing of network object storage support, we have chosen to eschew implementing iSCSI in favor of adding experimental object storage features to AoE. These features are modeled on (but considerably simplified from) the SCSI OSD specification. Several elements of the SCSI OSD model have been omitted in their entirety for this research. First, there is no quota mechanism implemented. However, implementing one has not in any way been precluded by the prototype design. Second, no provision has been made for either the optional security manager or the optional capability-based policy manager. Third, setting or querying attributes (metadata) are not allowed to be piggy-backed on other operations as is allowed in the SCSI OSD specification. Fourth, the optional collection object type has not been implemented.

With those caveats established, we now describe the protocol extension implemented in the present prototype. The AoE command value 5 is used to issue an OSD command. For OSD commands, the AoE Arg field includes:

1. A one-byte command field.
2. A one-byte flags field.

3. A two-byte length field.
4. An eight-byte partition ID field.
5. An eight-byte object ID field (omitted for some OSD commands).
6. An eight-byte address field (omitted for some OSD commands).
7. A variable length data field for those commands or responses that require it.

Both requests and responses follow this format, as illustrated in Figure 6, except that for responses, the PID, OID, and address are omitted.

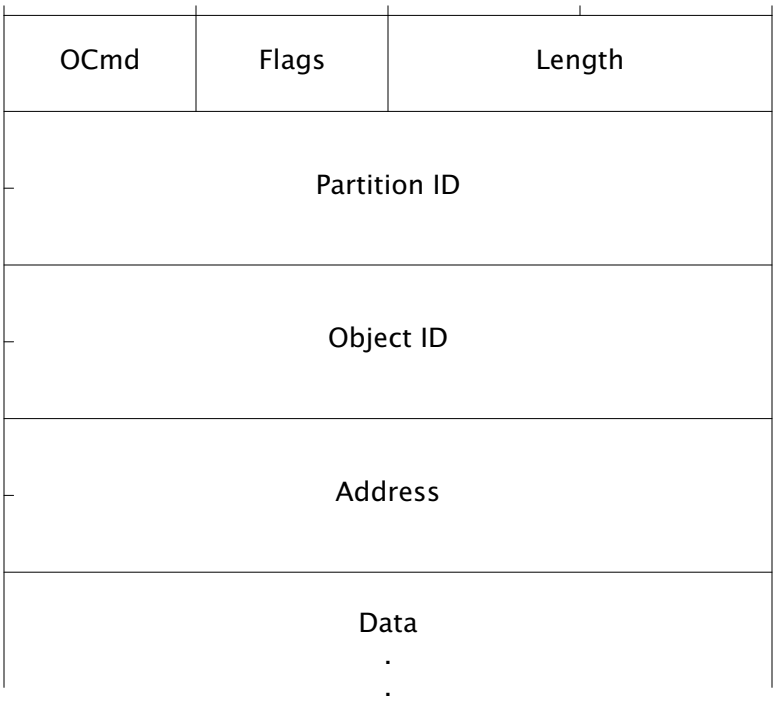

Figure 6: AoE Object Store Extension

The OSD commands supported are a subset of the SCSI OSD commands, and the value of the command field is the low-order byte of the service action value in the SCSI OSD protocol for specifying the same operation.

0x01: **Format OSD.** This command creates the root partition object as defined for SCSI OSD. All fields after the length are omitted for this command. (However, because of the minimum Ethernet frame size, some fields for this and other commands are sent anyway.)

0x02: **Create.** The create command is used to create a new object within a partition. A metadata list is allocated for the object and populated with a default set of metadata, but no data blocks are allocated on create. The PID and OID are specified, but the address and data fields are omitted.

0x03: **List.** Only the PID is specified as part of the list command. If the PID is zero (i.e. the root partition), the target sends back a list of the IDs of all extant partitions. Non-zero values of PID will elicit a list of the objects in the referenced partition. The length field of the response gives the number of bytes in the list of IDs returned.

0x05: **Read.** For reading from an object, the PID/OID specify the object to be read, the address field gives the offset from the beginning of the object where the read should start and the length field gives the number of bytes requested. In the

response, the length field gives the number of bytes successfully read and the number in the data field.

0x06: **Write.** Similarly the write command requests that the number of bytes specified in the length from the data field be written starting at the offset in the address field to the object in the PID/OID fields. On response, the length field gives the number of bytes successfully written.

0x07: **Append.** The append command works like the write command, except that the address field is omitted and the offset is implicitly the current size of the object.

0x08: **Flush.** Upon receipt of a flush command, the target writes all dirty in-memory buffers to the disk.

0x0a: **Remove.** The object specified the the OID/PID fields is removed by this command. Its data blocks and metadata are freed.

0x0b: **Create Partition.** This command is similar to the object creation command except that a partition object is created. Thus only the PID is specified in the request.

0x0c: **Remove Partition.** This command removes an empty partition freeing its metadata list. If there are objects in the partition, it is an error.

0x0e: **Get Attributes.** Metadata for the object specified by the PID/OID (partitions being specified by on OID of zero) are queried by this command. The data field contains a list of the null-terminated names of the items being requested. Data in the response message is a list of the query results where each item is composed of its null-terminated name, a single character type specifier and a value. The types are h: short, l: long, v: vlong, s: string, and b: blob. The integral types are two, four, and eight bytes, respectively. However, in the current prototype, all integer types are handled as 64-bit vlongs. Strings are null-terminated and blobs are prefixed by a two-byte length field. All numeric values are given in network byte order.

0x0f: **Set Attributes.** Setting metadata values is handled as the converse of getting attributes. The data field contains the requested values to set in the same format as returned by the get attributes request.

Two flags are currently defined for the flag field. The high-order bit (0x80) is set to indicate a response message, and the next bit (0x40) is set to indicate an error. As suggested above, with the exception of the read request and the write and append responses, the length field always gives the number of bytes in the data field.

9. Implementation Details

Up to this point, we have described this work in pretty general and abstract terms. In this section, we begin filling in details as they apply to the present prototype. The prototype file/object/block store is called θfs. (Earlier iterations of this design were named according to earlier letters of the Greek alphabet. Ultimately, the intention is to name the final version ωfs, or perhaps Ωfs.) It is implemented in C on Plan 9 using the 9P and thread libraries. File services are provided by a 9P server and an NFS server. Block and object services are provided by an AoE target.

9.1. Sizes

Blocks in θfs are 2^{15} bytes (32KB), and block addresses are specified as 64-bit values. Thus a block containing block addresses will contain 2^{12} (4K) such addresses. A single superblock occupies the first block on the storage space. The next group of blocks are used for the two hash tables. Following the hash tables there is a variable sized free bit map. Variable sized regions for a metadata structure pool and a string/byte sequence pool follow the free bit map. The remainder of the storage space contains allocatable blocks for data and block indices.

As mentioned above, the blocks comprising a file, object, or LUN are mapped by a one, two, or three level tree of block pointers. For a single level of block pointers, the maximum size is $2^{12} \times 2^{15} = 2^{27}$ bytes (128MB). For two levels of block pointers, the maximum is $(2^{12})^2 \times 2^{15} = 2^{39}$ bytes (512GB). For three levels, the entity can be as large as $(2^{12})^3 \times 2^{15} = 2^{51}$ bytes (2PB). If a fourth level is added, the maximum size will be enough to support up to 63-bit offsets for object and file access as well as the full ATA 48-bit block numbering in AoE.

9.2. Hash Tables

Each hash table contains a prime number of buckets in the range of 1048573 to 268435399. The prime number chosen is one greater than or equal to the number of blocks in the store and close to a power of two.

Both hash tables are open implementations. For the path name to QID path table, non-zero bucket entries point to blocks containing all elements that hash to that same bucket. If a single block is not large enough to contain all of them, then the blocks form a linked list with the last eight bytes of the block containing the block number of the next in the list. Each element in the table is composed of an eight byte QID path, followed by a two byte character count, followed by a variable length path name. For the QID path to metadata block number mapping, bucket entries give the index of the first metadatum for the first entity hashing to that bucket. As mentioned earlier, the set of metadata for a given entity is organized as a linked list. One special metadataum in each list is called qhnext and gives the list head for the next entity that hashes to the same bucket.

9.3. 9P Service

The 9P server in θfs is generally pretty conventional. In a departure from the libthread and lib9p design, each instance of the srv loop runs not in an independent process, but in another thread with all being in the same process. This helps to eliminate a substantial amount of mutex locking that would otherwise be required.

9.4. AoE Support

The AoE target functionality was adapted from the implementation of vblade (an AoE implementation from Coraid). First, the mainline startup code for vblade was replaced with calls from the θfs startup code. This also involved minor changes to allow LUNs to be created and removed on the fly. Second, the parts of vblade that interacted with the files backing it were changed to make calls into the θfs code to make use of its cacheing and block allocation infrastructure. Third, the configuration information was moved from being stored in a reserved area at the beginning of the LUN file to the θfs extensible metadata. Finally, support was added for the experimental OSD additions to AoE including support for all the commands listed above.

9.5. NFS Support

The θfs prototype includes support for NFSv3. Version 3 of the protocol was chosen because it is nearly universally supported and because significant potential clients did not support Version 4 at the time of the prototype development.

Normally, we would implement NFS as a service layered on top of an existing file system or as a protocol translation to 9P. However, because one of the objectives of the current work is to demonstrate coexistence among disparate file systems, we wish to support symbolic links, named pipes, UNIX-domain sockets, and device nodes, which are concepts not present in 9P. The extensible metadata design makes this easy, but it requires that the NFS server have direct access to the metadata. In the present prototype, therefore, NFS is implemented as a server integrated directly into the file system implementation.

Because both the NFS and 9P servers use the same path name and QID hash tables, they provides two different protocol views of the exact same file system. File types supported in NFS but not 9P simply appear as empty files to clients viewing the file system through 9P.

There are two primary features of most NFS servers that are not provided in the current prototype. First, we do not provide support for hard links. The NFSv3 standard provides a mechanism for a server to indicate whether it supports each type of link, but clients that expect hard links to be supported are likely to be disappointed. Second, NFS implementations generally include a lock daemon to support distributed locking of files and regions of files. Lock daemon functionality has not yet been included in θfs.

9.6. Cacheing

The current cacheing mechanism is a pretty typical buffer cache with reference counting. Writes are handled in a write-back fashion with a process writing dirty blocks every 15 seconds. The cache keeps a nominal maximum of 4000 blocks in memory and blocks are reclaimed following an LRU policy. However, blocks that are dirty or that have non-zero reference counts are not reclaimed until the next opportunity. Therefore, it is possible for the number of blocks in the cache to sometimes exceed the nominal limit of 4000.

9.7. Snapshots

We support snapshots in θfs when it runs on storage served by devsnap, as reported in [5]. The multiple root mechanism supports a dump file system that is compatible with the traditional Plan 9 dump file system. The presence of the snap device is not necessary, however, and θfs can run directly on a disk partition.

9.8. Console

In addition to listening for TCP 9P connections, for NFS requests, and for AoE messages, θfs posts a file descriptor to /srv to provide a textual console interface. Connecting to it with con -C allows one to interact with θfs to get current status and to examine metadata. The current set of commands supported include:

allow: Effectively turn off all permissions checking. This also allows 9P wstat messages to change file ownership.

blockuse: Look up a block by number and print out how it is used.

checkalloc: Scan the file system to compare the free map to the actual block usage.

cstat: Print a summary of the current state of the cache. This includes the number of blocks in the cache, the number of dirty blocks, the number of cache misses, the recent hit rate, and a reference count histogram.

disallow: Turn on normal permissions checking and prohibit ownership changes by 9P wstat messages.

dumpusers: Print out all known UID information, both as known to 9P and to NFS.

fixfamilies: Clear out references to children or siblings where the QID doesn't map to any metadata.

fixpaths: Clears path hash table entries with no corresponding metadata.

halt: Shut down the file system cleanly.

help: Print a list of the valid commands.

hstat: Print a summary of hash table statistics including number of lookups, number of collisions, and maximum search depth.

inituid: Initialize the NFS UID table from the file //adm/nfs.

lcreate: Create a LUN given its AoE major and minor numbers and its size in sectors.

lls: Print a list of all LUNs giving their AoE target numbers and sizes.

lmeta: Given a LUN's AoE target number, print out the LUN's metadata.

lrm: Remove a LUN given its target number.

mpred: Locate the predecessor to a metadata item in the linked list.

mprint: Print the details of a given metadata item.

mstat: Print a summary of metadata management statistics.

newroot: Create a new file system root.

nfsdebug: Control the level of debugging of the NFS file server. An argument value of 0 turns off debugging. Values 1 and 2 select increasingly detailed debugging information.

p2q: Show the QID path for a given path name.

p9debug: Turn on or off 9P debugging. The argument is the value assigned to chatty9p.

phash: Print the hash bucket for the specified path.

pmeta: Print the metadata for the file whose path name is given.

q2m: Show the metadata entry number for a given QID path.

qmeta: Print the metadata for the entity identified by the given QID path.

recovermeta: Check the consistency of and garbage collect the metadata and rebuild the free list.

revert: Roll back to an earlier snapshot as managed by devsnap.

rmp: Remove a hash table path entry.

rootallow: In the NFS server, give UID 0 the same super user privileges that it traditionally has in UNIX.

rootdisallow: In the NFS server, treat UID 0 as an ordinary user for permissions checking.

setmeta: Set the integer or string value of a metadata item.

setmstruct: Manually overwrite a metadata structure.

setqhash: Set a QID to metadata mapping in the hash table.

snap: Take a snapshot through devsnap.

super: Print the contents of the superblock. This includes the magic number, the next sequential QID path to assign, the number of blocks, the number of free map blocks, the location of the free map, the status, the location of the first data block, the number of free blocks, the QID path of the first LUN and the locations and sizes of the metadata and string pool regions.

sync: Trigger a pass of the write-back process to flush dirty blocks to the disk.

Most of the same console functionality is available by way of the file /mnt/θfsctl. Reads from this file combine the results of the super command and all stat commands. Writes to the control file are treated the same as writes to the console with the commands allow, disallow, lcreate, lrm, rootallow, rootdisallow, and sync being supported.

10. Results

The θfs prototype has been self-hosting in the sense that editing and compilation of the code currently take place on copies of the files stored on a θfs file system accessed with 9P. Similarly, the editing and formatting of versions of this paper were carried out on a copy of the troff source file stored on θfs.

The implementation of θfs is about 10000 lines of code, of which approximately 1000 were taken from vblade, and about 2500 are the integrated NFS server implementation. A slightly earlier version of the θfs code can be found in contrib/blstuart/θfs.

As indicated, the most extensive testing and use of this storage system has been via the 9P interface. The AoE block interface has been exercised using both the Plan 9 and the Linux software initiators to the extent of creating a thinly provisioned LUN, building a file system on that LUN, and carrying out file system operations on that file system. It should be noted that the file system so-created is completely independent of that served by 9P and NFS, but coexists with and allocates blocks from the same pool as the one they serve. As there currently exist no AoE OSD clients, testing of the object interface has been limited to a small test client written specifically for this project. The file system served via NFS has been mounted on both MacOSX and Linux and used concurrently with 9P clients of the same file system.

Although θfs should be viewed as still in the prototype stage, there are a few deployments that suggest that it is at least usable. First, it was integrated into a research version of the Coraid SRX product. In that setting, θfs served a file system over 9P and NFS while the existing SRX AoE functionality provided block storage. The result was an integrated NAS and SAN appliance which was used internally (though far from stress tested) at Coraid research. Second, a Plan 9 file server has been configured with θfs in /boot and corresponding changes in /sys/src/9/boot/local.c. This system boots taking its root from θfs and has spent considerable time serving a 9vx terminal.

11. Future Work

The prototype as described in this paper is a snapshot of a work in progress. As such, there are a number of developments in the planning for this system. This section attempts to give a brief summary of some of them.

11.1. Permanent AoE Object Support

The object extension to AoE described here is strictly one designed for research purposes. The needs of a real extension that can be standardized and productized depend greatly on the needs of the initiator side. In other words, the exactly subset of SCSI OSD functionality that is needed in AoE depends more on how client OSs use OSD than on the specifics of our server-side research. Therefore, we leave this issue open for the time being, with the expectation that those involved with initiator development will play a critical role in establishing a permanent AoE extension for handling objects.

11.2. Log Structuring

As file system design has developed over the past couple of decades, the benefits of log structuring/journaling have become evident. These techniques are particularly valuable in their resilience to power failures and other unexpected events. Because they are presently well-understood and not necessary for our research objectives, we have not designed this prototype as either a journaled or a log-structured file system. However, neither have we made particular design decisions to preclude the use of these techniques. It is expected that a final version of this system (an ωfs) would be built around a log-structured design. However, the stage at which that happens will likely depend on experience we gain in using the prototype.

11.3. Performance

Up to this point, only a cursory pass has been made at analyzing and enhancing performance. In particular, prior to θfs, some fairly deep changes were made in metadata handling to bring the overall performance in some small tests to the same general ballpark as other Plan 9 user-land disk file systems such as kfs and fossil. No attempt has

been made thus far to measure the NFS performance of θfs or to compare it to other NFS implementations. It is anticipated that substantial effort will be made in performance improvement as the basic functionality stabilizes and in preparation for other research directions. Because little analysis or measurement has been done to date, we cannot say where significant improvement is likely to be found.

11.3.1. Hashing PID/OID Directly to QID Path

The first thing that appears to be a glaring inefficiency is the way we handle the object PID and OIDs. For each request, we have a pair of 64-bit integers that needs to map to a single 64-bit integer. On the surface, at least, it seems inefficient to make a string out of the two 64-bit integers, hash the string as a path name, and get the 64-bit QID path from the hash table, possibly incurring one or two extra disk accesses. So it would seem there could be benefit from a mechanism that translates directly from the PID/OID pair to the QID path, particularly if that translation could be done with fewer disk accesses. On the other hand, whatever mechanism is used, the net effect will still be to hash 128 bits to 64 bits. It's not clear whether the effects of cacheing are such that the existing translation is just as good as any other. Whether this inefficiency is significant and can be improved remains an open question.

11.3.2. Including Full Path Name in Metadata

The design of NFS expects a fairly conventional approach to directory organization and representation. In particular for the lookup RPC, given a unique ID for a directory and the final element of a path name, a unique ID for the specified file should be quickly obtainable. However, in the present design, the concept of a directory is primarily one of name management and not reflected in data structures or data management. To get a full path name for hashing, we build the string by traversing up the name tree to the root, prepending the name of each node found along the way. Although conceptually such an approach can be made pretty elegant, if the cache contents are unfavorable, it could require a number of disk accesses. Therefore, the NFS lookup RPC might benefit from including a file's full path name in its metadata. Doing so would reduce the path name construction to a concatenation of the directory's path name and the final name element given in the lookup request.

11.3.3. Including Ancestor QID List in Metadata

Because the 9P walk message allows up to 16 path elements to be specified, in principle, hashing the full path name should allow for faster walks by not processing each intermediate directory. However, there are a couple of ways in which the semantics of the walk expect step-by-step directory traversal. First, the response message to a walk request is specified to include the QIDs of each intermediate directory traversed. Second, it is normally expected that the permissions and ownership of each intermediate directory will determine how far the traversal can be allowed to proceed. In the prototype described here, we deal with these issues by traversing upward from the endpoint of the walk building the QID list along the way. (We currently punt on the permissions issue.) Of course, such a traversal can require additional disk accesses, depending on the contents of the cache. Including ancestor QID and permission information in a file's metadata would reduce the number of disk accesses if we find the existing approach to be problematic. This potential improvement is expected to be considered soon in conjunction with methods of providing correct permissions semantics.

11.4. Distributed Services

The last future direction we discuss here is the one that sparked the work described in this paper. Experience with a number of storage products has led us to a general dissatisfaction with architectures incorporating storage ''heads.'' Such devices tend to be

bottlenecks with numerous clients on one side and numerous low-level storage elements on the other. This kind of architecture is nearly guaranteed to lead to expending more resources to get lower performance. Instead, we expect that it is possible to develop storage components that collaborate to present to clients a single, large data store and at the same time spread the data and communications out among themselves. The storage system developed here is intended to be the per-node system on which a headless, distributed storage architecture is built.

12. Conclusion

As with most systems research, few if any of the individual components here lack precedent in earlier file or other data storage designs. However, in our experience, the particular combination of mechanisms is at the least, unusual. Taken together, these mechanisms lead to a number of uncommon and interesting large-scale characteristics. Experience with θfs suggests that it has been a successful endeavor in three ways. First, it has demonstrated workable techniques for integrating disparate methods of data access. At one level, the techniques allow us to make available files, blocks, and objects simultaneously sharing the same pools of storage. Some of those same techniques also allow us to make files available over multiple protocols that may expect different underlying storage features. Second, this prototype has established a clear path for further research into techniques relevant to stand-alone storage service devices. Finally, indications are that this work will serve well in its initial goal of providing a platform on which to carry out distributed data storage service research.

13. Acknowledgements

We would like to express our appreciation to Coraid for its support during this research. In particular, Brantley Coile established the research environment in which the work was conducted.

14. References

[1] Stuart, B.L., *Principles of Operating Systems: Design & Applications,* Boston, MA, Course Technology, 2009.

[2] Factor, M., et al, ''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,''* http://www.research.ibm.com/haifa/projects/storage/objectstore/papers/PositionOSD.pdf, 2005.

[3] McCoy, K., *VMS File Systems Internals,* Boston, MA, Digital Press, 1990.

[4] Solomon, D.A. and Russinovich, M.E., *Inside Microsoft Windows 2000,* Redmond, WA, Microsoft Press, 2000.

[5] Stuart, B.L., ''An *O*(1) Method for Storage Snapshots,'' *Proceedings of the 9th International Workshop on Plan 9,* 2023.