FLEXIBLE OBJECT BASED FILESYSTEMS FOR SCIENTIFIC COMPUTING

A Thesis

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Master of Science in Computer Science and Engineering

by

Christopher M. Moretti, B.S.

Douglas L. Thain, Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana
April 2007

FLEXIBLE OBJECT BASED FILESYSTEMS FOR SCIENTIFIC COMPUTING

Abstract

by

Christopher M. Moretti

Object storage has traditionally been seen only in low level interfaces, visible only to kernels and filesystem code. However, if storage objects are made visible across a distributed system, it dramatically simplifies the construction of large storage systems that are flexible, expandable, and migrateable.

The FOBS filesystem, consisting of a simple metadata layer containing pointers to storage objects on remote filesystems, demonstrates this concept. The layer of indirection allows for filesystems larger than any single disk, permits multiple filesystems to share common objects, and enables users to create and manage private namespaces.

Experimentally, this work demonstrates that the overhead of indirection is low, a single client can write faster than disk speed, and multiple clients can harness aggregate disk throughput. FOBS scientific application is examined with case study of the filesystem as employed by a high energy physics experiment on a cluster of 32 nodes for over one year.

CONTENTS

FIGURES

TABLES

ACKNOWLEDGMENTS

I would like to thank my parents for their advice and encouragement, as well as my friends and colleagues for their intellectual stimulation, technical help, and much-needed comic relief. I also owe a great deal of thanks to Dr. Douglas Thain for guidance, motivation, and insight throughout this research.

CHAPTER 1

INTRODUCTION

Disks are getting larger and cheaper, and increasingly there are large quantities of disk space unused on desktop workstations, computation cluster nodes, and other storage resources. Inevitably, however, there is always a set of users ready and willing to push their applications to the scale of the available resources and beyond. Anecdotally, it even appears that the more these users are fed in terms of capacity, the more ravenous their consumption becomes; offering terabytes of available disk space only encourages them to find new and creative ways to fill this space.

It is easy to see this expansion in maximum capacity of commodity large disks: even recently (within the last ten years) a terabyte hard disk was unfathomable to most users; today such a disk is available for under five hundred dollars. One does not need a single mammoth terabyte disk to work on the terabyte scale, however; though perhaps lacking in terms of awe factor, a small number of disks can combine to provide a similarly large storage resource.

An example of this type of user is a manager of large scientific data sets. Greater storage availability allows more variables to be tracked, more measurements or data collection, or greater resolution of simulations (though storage is only one consideration along with processing speed in this case). Even general users can readily expand their disk usage to the scale of these resources; movies

total several gigabytes, and high definition movies increase this by an order of magnitude.

To manage these hardware resources efficiently, scientific users, and users in general, need large filesystems, and these filesystems must take into account that the resources may be spread across numerous physical devices. Further, compared with filesystems intended for single devices, or even many parallel filesystems, filesystems for sets of disks that are highly unreliable, heterogeneous, and possibly widely-distributed require more in the way of flexibility. The general concept of flexibility must include reconfiguring the system by adding or shedding devices on-the-fly, changing authentication and access policies across the system on-demand, and doing this without requiring administrative intervention, which puts an element of latency into a system on a human timescale (minutes or longer), instead of a systems timescale (ms or less).

1.1 Flexible Object Based Filesystems

With the **F**lexible **O**bject **B**ased **S**torage filesystem, or FOBS, users can build filesystems that are larger than individual disk resources, reaping benefits for both single users and large sets of simultaneous users. FOBS filesystems, built on top of the Tactical Storage System infrastructure, can be set up at the whim of the resource owners without having to broach interference from system administrators. This remains true for reconfiguring the system to add new resources, which can be done on-the-fly, unlike many distributed filesystems.

Given a set of remote filesystems, a user can "mount" many distinct FOBS filesystems on top of common shared objects from the set of resources, which is useful for separating out namespaces for different users (or different tasks by one

user). In fact, entire FOBS filesystems can be created on-the-fly on top of a set of query results or other interesting set of objects.

The FOBS filesystem is flexible to operating on working environments across the distributed systems continuum, from closely coupled clusters (as low as a small set of homogeneous high performance machines on a shared low latency high bandwidth network within a single server room) to the broadest definition of desktop grid (a very large, geographically diverse, heterogeneous set of machines).

1.2 Expectations and Evaluation Model

As data needs of computer users increase, disk capacity must increase to fill those needs. In general, storage capacity has expanded to fill the emerging needs of general users, however, as noted above, there are some users, such as those working with large scientific data sets, generating large archives, or similar activities, for which there is no such thing as "too much storage space". Further, even with the expanded capacity of single devices, to accommodate needs on the scale of these data sets, they must use many individual disks, which can be unwieldy in terms of management and usability. An obvious solution is for users to consolidate their storage into a small number of large disks, however this is considerably more expensive than an array of smaller disks, as there is a premium paid for the largest disks available at any given time. Also, even after paying this premium, the few large disks are unlikely to achieve the cumulative required storage space available from the larger set of smaller disks.

Even ignoring capacity and economic issues, consolidating into several large disks doesn't solve the management and usability issues. Spreading data out over several disks requires that the user know on which device each file is located. This

can be difficult to keep track of, and even if a systematic method is developed to determine where data is placed in order to facilitate recall, this limits the flexibility of the filesystem to move data around (to balance usage in the presence of hot spots, take advantage of hardware differences for performance or reliability, etc.).

There are several options to get a large filesystem within a single namespace. AFS and NFS are heavyweight systems that may not be appropriate for smaller workgroups, and often require dedicated storage servers; several commercial SANs are available, but they also require dedicated storage servers, and are the most expensive option. Further, AFS, NFS, and SANs do not give the degree of flexibility needed by many users; some users want to be able to swap in and out resources quickly without downtime associated with reconfiguring the system, some are in a corporate environment where they may not have administrator access to their machines, and some desire the ability to configure their own namespace but remain within the larger scope of the storage resources in their environment.

To accommodate these needs, FOBS aims to allow large filesystems in which resources can be joined together to form a storage system much larger than that formed by a number of disks connected to a single machine. This filesystem is flexible; it can be reconfigured to fit changing needs without having to be taken offline for considerable durations to complete the reorganization. It can be configured to many different individual users to give them their personally optimal view of a set of resources. Taking this a step farther, an entire filesystem can be constructed on-the-fly to display a pre-existing set of resources, then dismantled just as quickly, leaving the underlying resources in place. Finally, with an eye on the needs of the future, FOBS filesystems are completely expandable; once the

system is informed of new disks, it will immediately start to take advantage of them.

Simply having a large, accessible filesystem is important, but this system must also be a viable option in terms of performance. Some performance can be given away for greater capacity, flexibility, and ease of use, but a filesystem must still maintain sufficient performance that users don't refuse to use it due to the overhead of doing so. The benchmark that FOBS must achieve to be sufficient in performance is subjective based on user preferences, however for the purpose of the design, FOBS must not, due to its architecture, levy more than an order of magnitude performance penalty over its building blocks (namely, a Chirp server alone). Further, the system must give some actual performance **advantage**, at least in some use-cases. It is sufficient that this be obtainable using Chirp alone, but not easily; for instance, some aggregate throughput advantages are obtainable over a local disk by using several unmodified Chirp servers, but to obtain this benefit would require a finely tuned set of operations with file placements and accesses planned in advance.

Having laid out the expectations for the system, both in terms of architecture and actual implementation, it is necessary to determine an evaluation method for these expectations.

1. Latency:

   Evaluating the latency injected into operations by the system must be done on several levels. First, microbenchmarks will be used to give a measure of the system's performance on the basic operations that make up real-world usage. Additionally, higher level, "meso-benchmarks" will show whether any differences detected in the small tests propagate or are mitigated in actual

usage. Additional latency is acceptable if it is within reason; a factor of 2, for instance, since the namespace uses two underlying file operations to complete one FOBS file operation. It is also necessary that for operations in which the namespace lookup is a small- or non-factor, the benchmarks should reflect this. Finally, for larger "meso-benchmarks", either a small constant overhead or a consistent percentage overhead would be acceptable.

2. Throughput:

   Throughput, along with latency, is the other key raw performance metric in this system. This is especially true considering that scientific computing users are a likely end-user of this system, and for many of these users, computation cycles are very valuable, making data throughput critical. This is because time spent waiting for data is time not used for computing. In this aspect, while FOBS is not specifically a high performance computing environment, it must have measurable characteristics that indicate its feasibility as a scientific computing environment. For large data sets, FOBS must be able to deliver and sustain high throughput under load, and ideally, do so with only a single replica of the data (this allows separation of claims made about the filesystem design from the implementation details of how replication, striping, and other performance optimizations are accomplished)

3. Single Users:

   A single user must have something to gain in using the system, as well; single users are the base case for any computing system. If this case is not handled acceptably, it is unreasonable to expect that many will use the system, and thus even good performance under load becomes moot. One argument for single-user attractiveness is raw capacity compared with the

underlying system, another is ease of use and flexibility as an artifact of the implementation details. However, a case for FOBS must go beyond that to show that there is a reason, notably a performance-based reason, due to the design of the system as well as the implementation, to use this filesystem instead of buying several large, state-of-the-art disks, and using those.

4. Production System:

   This system must actually be able to be used in production. Tests and measurements in controlled environments are useful for analysis, but if a system is not used in the "real world" by "real users", they fall hollow. Most importantly, however, is that real users demonstrate real problems with the system, and invariably bring about use-cases and sanity checks that cannot be tested for in lab conditions. A prolonged use of FOBS as a production system for real users is, then, the final requirement to evaluate the filesystem against the expectations put forth above.

1.3 Object Storage

Object storage originated out of Carnegie Mellon University in the 1990's. The concept grew from the Network Attached Storage Device research project [10, 11, 25]. From there, commercial systems like Panasas, and open source filesystems based on proprietary object storage technology such as Lustre [6] developed to expand the research profile out to developed systems. Standardization has followed, and object storage principles are being explored by the Antara [3], zFS [26], and ObjectStone [7] projects at IBM, continuation of Seagate's OSD project [29], and the Centera system for storing immutable reference data. A good history of object

storage can be found in the position paper on object storage as the future building block for storage systems, by Factor, et al. [7]

Object Storage is normally thought of as low level handling by kernels and filesystem code of all logical access below the "object level", in which the object is set at the threshold most convenient for access to the end user. However, object storage is not just adding layers to storage for more control, but rather a fundamental change in philosophy and technology, in which previously higher-level infrastructure activities are delegated to lower-level storage devices. From the view of a system designer, object storage is a continuation of the process in which the base-level storage object is pushed farther and farther up the scale; one can see it as moving within a continuum from the older paradigm of user level constructs holding the responsibility to manage physical aspects of disks, through the step at which sectors are the lowest level requiring user-level intervention, to logical blocks, and now to files being the only object with which a user must interact [29]

Throughout this work, this is the concept that is meant by object storage, and the specific instance of object storage is that of file-level access as the lowest semantically meaningful level with which the system actively operates. Thus, it is possible to address a FOBS file at two levels: the logical FOBS file that consists of metadata, in the form of an underlying file, and the storage object, again in the form of an underlying file on a file server. Limiting the interface to these levels eliminates the complexity of operating at the block level, pushing that onus off to underlying file servers, or even lower layers, and breaking the paradigm of an object as a set of blocks unrelated to the actual view of that object.

1.4 Tactical Storage

Separate from the direct goals of the FOBS filesystem are the artifacts of the implementation environment. A brief overview of cooperative storage systems, and specifically the Tactical Storage System, which serves as the implementation environment, is necessary, to fully understand the FOBS system. The main ideas of a tactical storage system, including the Distributed Shared Filesystem (DSFS) model of which FOBS is an implementation, are introduced in *Separating Resources from Abstractions in a Tactical Storage System*[32]; the goals of a cooperative storage system can be stated from a separate paper dealing with this research [33]:

> A cooperative storage system is a large collection of storage devices owned by multiple users, bound into a loosely coupled distributed system. There are many reasons why cooperating users may bind together multiple devices: they may backup data to mitigate the risk of failure; they may construct large repositories that cannot fit on any single disk; they may improve performance by spreading data across multiple devices; or they may wish to share data and storage space with external collaborators. We assume that users have some external reason to cooperate and so we do not explore issues of fairness or compensation. However, we do assume that resource owners wish to control quite explicitly whom they cooperate with. One user may be willing to share public data with the world at large, while another might only share scratch space with one trusted colleague. Others might share resources with an organization such as a university department or a commercial operation.

Building upon the Chirp protocol and the basic interface, users may design abstractions to harness more resources, bind resources together, or otherwise suit their needs. The principal building-block abstraction is the central filesystem (CFS), often referred to by the implementation name "Chirp file server", which facilitates sharing data as an online file server. FOBS is an implementation of

the Distributed Shared Filesystem, another abstraction introduced along with the CFS.

These abstractions are accessible through adapters; two of which are currently available for FOBS: a FUSE module that allows users to see a tactical storage namespace as though mounted locally, and parrot. Parrot is a process based virtual machine that uses the `ptrace` interface to intercept system calls and emulate them for access to local and remote filesystems. Parrot allows unmodified applications to use the tactical storage abstractions described above.

The flexibility to create, destroy, and modify various abstraction instances is a key characteristic required of the underlying file server for a FOBS filesystem. Additional characteristics of the tactical storage system implementation that heavily influence system design in FOBS include user-level access, in which no root permissions are necessary to build, deploy, or access a TSS, directory-level access control list based authorization, the remote procedure call API, and the properties of the proposed DSFS abstraction, which have directly impacted several of the key components of the FOBS filesystem, such as data/metadata duality.

The original TSS paper framed the discussion in terms of the object storage concept, noting that "a block interface is not the appropriate low-level interface for tactical storage". Additionally, the paper realized a need for a database-like metadata server, similar to those used in GoogleFS [9], Lustre [6], and Amoeba [20], and proposed a UNIX interface to allow basic tree directory structures that are central to the DSFS abstraction and the corresponding FOBS filesystem.

CHAPTER 2

RELATED WORK

Table 2.1 gives a summary of the FOBS design goals and characteristics versus several other distributed filesystems. The table was inspired by a similar one in [35]. Working from the definitions in the Freeloader paper, General indicates suitability for general use by giving general filesystem interfaces and functionalities, Cache indicates whether the system is designed as a front-end accessible cache space, Striping indicates optimized data placement, Scavenging indicates whether the system is designed to use empty disk space scavenged from available nodes, and Wide-area indicates whether the system was designed to act as storage over a WAN. FOBS does present UNIX-like filesystem interfaces and one purpose of this work is to show its suitability for single users as a general purpose storage system. It is not specifically designed as a data cache, though in the case study in chapter 5, a FOBS filesystem is used in a similar manner to this. Striping of files is not done at the block level, however sets of objects are striped across the FOBS underlying resources to promote a similar purpose on a data set basis, rather than a single-file basis. FOBS is not designed specifically for WAN storage, and most applications of it have been on low latency, high bandwidth connections; however, nothing in the architecture prevents deployment of FOBS over a WAN.

TABLE 2.1

COMPARISON OF SEVERAL DISTRIBUTED FILESYSTEMS TO FOBS

	General	Cache	Striping	Scavenging	Wide-area
FOBS	Yes	No*	Yes*	Yes	No*
Freeloader [35]	No	Yes	Yes	Yes	No
GPFS [28], Lustre [6], PVFS [5]	Yes	No	Yes	No	No
Frangipani [34]+Petal [17], Zebra [13]	Yes	No	Yes	No	No
NFS [27], AFS [14], Coda [15]	Yes	No	No	No	Yes
Google FS [9]	Yes	No	No	No	No
FARSITE [1]	Yes	No	No	No	No
IBP [22]+exNode [4]	No	No	Yes	No	Yes
xFS [2]	No	Yes	No	No	No

2.1 Striping

Traditional block-level systems incorporating block striping (RAID 0 [21], Zebra [13]) aim to maximize performance through parallelization. They work well with large continuous reads and writes, which can be broken up and shipped off to many resources, simultaneously making use of each disk's disk bandwidth, buffers, and caches, and if the disks are served by individual computers, each of their memories. These give limited benefit, however, over object storage for reads and writes so large that they exceed the capacity of the disk buffers, or in extreme cases the memory. Additionally, under multi-user load, each user may have to interact with every disk for even a single file, making striping prone to functioning only at the speed of the slowest resource. This reduces the ability of a striping system to tolerate resource heterogeneity.

Hartman notes in his dissertation [12] that a disadvantage with block striping is that it is inefficient for small files. These files gain minimal benefit due to network and disk latency dominating the actual small write to each disk, but incur the maximum overhead of any disk on every access. Additionally, small files are prone to causing the expense and complication of writes that make up only a portion of a stripe, which are problematic because they have differing characteristics than the rest of the stripes [13].

Zebra fits in to the striping systems even though it keeps append-only logs as the actual stripes. A system like this must be wary of space consumption by partial stripes, as these fragment the available space, which then requires either defragmenting or exceedingly complicated block tracking metadata. Zebra utilizes a stripe cleaner to keep track of, reclaim, and reissue free space. This adds considerable complication that can be broadly ignored by using the object storage

concept, in which the filesystem does not care about the "shape" of free space allocation on any given disk, nor the amount of free space in general, aside from potential rebalancing of a system at the disk level.

2.2 PVFS

PVFS [5] is a hybrid system which exhibits characteristics of block striping systems and object storage. The key underlying goal of the parallel filesystem is to squeeze performance out of parallelization via striping, however, it also incorporates metadata that function as a list of underlying files, partial files, or block sets. While this is a potentially different level of object than is used as the base-level object in FOBS, there is clearly an object storage influence in the PVFS architecture, especially considering the broad reach of the metadata to be configured on a file-by-file basis as to how to make use of the resources for that file (blocksizes, number of disks, etc.)

Because PVFS depends more closely on the block-device paradigm, it is still limited somewhat to clusters or other reasonably homogeneous environments. In this environment, PVFS will gain some performance advantage over a file-based object system, especially for small reads and writes. Outside of the cluster environment, however, the presence of heterogeneous network connections or disk speeds, a file striped across several devices can be accessed only as fast as the slowest resource. While this is true on a per-file basis with FOBS, having some fraction of the files in a data set be slow does not render the system unusable while waiting for the slow accesses so long as the target application is not a tightly parallel processing job. In PVFS, however, having potentially every single file access occur at

the slowest resource speed can degrade many types of performance beyond tightly parallel workloads.

2.3 FARSITE

The architecture of FARSITE is quite similar to that of FOBS; a metadata layer acts as a set of pointers to underlying files, which are distributed across a set of resources. They aim to "provide ... the benefits of a central file server (a shared namespace, location-transparent access, and reliable data storage)" [1] in the context of a large distributed system. Each facet of this goal is shared by FOBS. However, while FARSITE stresses the system is intended to act as a general purpose filesystem, the designers are very up front about shying away from association with tasks beyond that of desktop I/O workloads. FOBS, on the other hand, actively encourages use as a filesystem for scientific computing, in addition to desktop I/O workloads, and thus has a different set of evaluation expectations.

FARSITE takes heavily into account the security of the system and the risks of malicious users, whereas this is not within the scope of the development goals of FOBS (or its underlying filesystem, Chirp). FOBS design emphasizes the ability to use external methods for authentication, and has relied on the assumption that Byzantine rules could be enforced if malicious users with access to the system became a concern. FARSITE, on the other hand, has natively implemented this behavior and has given significant discussion to these implementations in their publications.

FARSITE implements raw replication for redundancy and availability, but the designers notes that the replication subsystem is "a readily separable component of both the architecture and the implementation" [1]. This allows replacement

with other options, such as erasure coding, or dismissal altogether in cases where data availability via a single replica is not a concern. FOBS has taken a similar angle: the system is designed to support redundancy and parity schemes injected from higher levels, and not to interfere with schemes built in to lower levels, however such schemes are neither natively built in to FOBS, nor precluded from being built in to FOBS as a separate module.

Finally, FARSITE is quite concerned with the balance of advantages and downsides of high performance servers as resources; with higher reliability and performance weighed against higher initial cost, higher cost for maintenance and administration, and that such resources are additional overhead beyond already sunk costs associated with large workgroups. FOBS shares these concerns, but whereas FARSITE avoids servers completely, choosing to replicate and distribute metadata in an aim for a serverless filesystem, FOBS is built on the assumption that the metadata server is powerful and reliable enough to support operation of the filesystem, and that the cost and overhead associated with this single server are not significant when compared with the dozens or hundreds of underlying nodes. This belief stems from the idea that the storage nodes are running on commodity machines, often using scavenged space from desktop or cluster systems, in which the addition of the FOBS load on the machine does not change the maintenance or administration overhead associated with it.

2.4 Lustre

Lustre [6] is a parallel filesystem that uses object storage to combine several resources into a single, flexible namespace. Like FOBS, Lustre stores data on commodity disks, with metadata servers for filesystem metadata. Further, Lustre

simply leverages existing authentication and authorization schemes, a departure from FARSITE, as discussed above. Lustre also uses existing privacy controls, which have been explored in the Tactical Storage System, but have not been implemented into the FOBS prototype.

One key difference, however, between these two object storage file systems, is the lack of data/metadata duality. Lustre stays true to the object concept by defining an object as a coupling of the data and metadata, but maintaining a clear distinction between them. Even though lookup semantics from OSTs to the underlying object-based disks is similar to that from FOBS, the rigidity of the distinction takes away one degree of flexibility that FOBS employs: the ease in mounting or logically changing a filesystem based on already-existent underlying data.

The similarity of the metadata architecture does, however, lead to several other close parallels between the systems, such as: relying on the metadata for lookup but not continual pass-through to access the underlying resources; and the ability to recognized errors associated with file placement and avoid misbehaving targets (very similar to the memoization technique in FOBS evaluated later in this work).

A touted advantage of Lustre is the ability to provide a global namespace that is easily mountable into a stub of a Linux filesystem, and that the location of this mount needn't be either pre-defined, nor consistent among nodes concurrently sharing the namespace. FOBS shares this ability through the FUSE adapter, and gives an additional advantage of having this mount point able to be defined without special privilege (aside from installing the original FUSE module). Finally, FOBS takes this even one step further by allowing underlying objects to be

combined into any number of different personally customized namespaces, while maintaining the ability to switch back to the unified namespace at any time.

2.5 Freeloader

Freeloader [35] has a similar architecture in terms of layers and the role of metadata, using a smallest logical object called a "morsel" (a partial file) instead of a full file, although they do not discuss the system in terms of the object storage paradigm. More importantly, however, the system's goal is to provide high performance online read-dominant access to large datasets locally to the user, similar to a cooperative cache. While there is overlap, especially with the GRAND case study, this is a very different use-case from FOBS general-use aim. Thus, although both systems aim to maximize scalability and connectivity while taking into account desktop heterogeneity and the write-once/read-many property of many scientific data sets, they are not directly comparable.

CHAPTER 3

ARCHITECTURE

3.1 Abstract Architecture

The FOBS filesystem is a metadata layer that points to objects (files) on underlying filesystems. The metadata layer consists of files stored on the underlying filesystem in their own right. Thus a file in the FOBS filesystem is really two separate files: one which contains a metadata pointer to the object, and the other which is the target of that metadata on some local or remote filesystem. In this work, "underlying file", "underlying object", and "target object" each refer to the latter, whereas "metadata pointers", "pointer files", and the generalized "metadata file" refer to the former.

Figure 3.1 shows two FOBS filesystems, Apple@Foo and Cherry@Baz. Both filesystems contain two files: datafile1 and datafile2. For Apple@Foo, the data for datafile1 is located on the host Banana; datafile2 is located on the same host as the metadata. In Cherry@Baz, both datafile1 and datafile2 reference the same file on Apple, which is in the directory specified by the key metadata; note this is not required, as seen in the first example, where the two underlying files were in different directory paths. The file "hosts" includes a list of available hosts for new file placement, and "key" contains the path in which those files would be created. As seen in the figure, it is valid for multiple pointers within the same

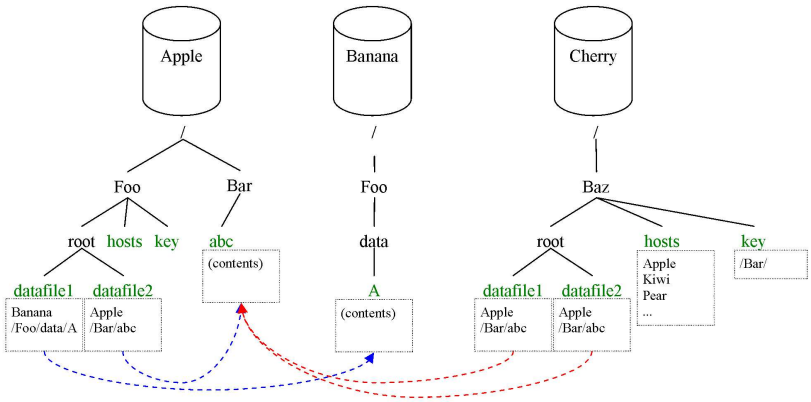

Figure 3.1. FOBS Architecture

FOBS namespace to reference the same object. Names from one FOBS filesystem can overlap with each other without any relation – shared semantic names can, but do not have to, refer to the same underlying file.

3.1.1 Metadata Layer

The FOBS metadata architecture requires two types of files: configuration files and metadata pointer files. The FOBS configuration files list where to place newly created objects onto the underlying resources, and what resources are available to do so. Other configuration files that could be used are policy files, indicating the preferred order of resource usage and settings for replication or other fault tolerance techniques if included as a module. The majority of FOBS metadata files, however, are pointers that contain two primary pieces of data: the hostname and path of the underlying object to which it points.

3.1.2 Possible Metadata Usage

The metadata pointer is not limited to only the target of the pointer. Other items could be useful within the scope of the metadata, including multiple targets for replicated data, targets and parameters for erasure encoded objects, flags for whether this metadata file "owns" the target object, a count of the number of metadata files currently pointing at the target (a reference count), or an actual list of the other metadata files referencing the underlying object.

Viewed in terms of the implementation, these possible metadata additions could be added to the system with minimal change to the fundamental functioning of the filesystem. The added capabilities could shape policy, performance, or reliability considerations, however, so while the system could be expanded to include any of these, further analysis considers a system without these additions.

3.1.3 Data/Metadata Duality

Data/Metadata duality is the property of the system in which metadata files are, in fact, actual files on the underlying filesystem no different from the data files they reference. This duality in the metadata allows for flexibility in constructing large filesystems.

On one extreme, because the pointers can reference remote filesystems, the metadata can be completely isolated from the objects themselves. The advantage of this is that different specifications may be given to the metadata and the underlying data, explicitly differentiating these two types of underlying files based on their FOBS usage.

On the other extreme, however, is the concept of a filesystem in which the data stored is the metadata from other filesystems. Perhaps this could be used as

a compilation of what underlying resources are being referenced by several FOBS filesystems, for management or accounting purposes.

The complete separation of metadata files from the objects they reference allows for metadata files from several different FOBS filesystems to point to the same underlying object. This allows the same object to be part of multiple different filesystems without the underlying object's replication across each of them. The independence of these metadata files, which provide the name for the FOBS file, from each other is a key component in allowing users to create and manage their own namespaces, as names for the same object in different FOBS filesystems needn't be related, and names for distinct objects may collide between filesystems without issue. A special case of multiple metadata files referencing the same underlying object is that even multiple metadata files from the same FOBS filesystem do so. This provides a characteristic similar to a hard link in which a file can operate by multiple completely independent names within the same filesystem namespace.

To give an example of these abilities, consider a dataset in which new data is constantly being acquired, similar to the case study examined in Chapter 5. With multiple pointers referencing the same file in the dataset, a scientist can target a single observation's data file for inclusion in a filesystem namespace of all observations from the same hour, a separate namespace of all observations from the same day, and a third of all observations from the same year. Further, as needed, the observation's name in one group need not be related to its name in any other set. Alternately, a single resource could appear under two different names in the same namespace. This could include cases traditionally handled by symbolic links, such as pointing the file `newest` at the same resource as the filename corresponding to

the last hour's observation. Finally, the complete separation allows for modifying the metadata to change the object it references without changing the underlying object. Thus, when the next hour's observation is loaded, the metadata for `newest` may be changed to reflect its new target without changing the former target in any way.

All of these possibilities due to the characteristics of the architecture allow for constructing a FOBS filesystem on-the-fly to sit on top of a set of resources. That is, a FOBS metadata layer can be constructed to access resources listed by a set of dynamic classifiers. A simple example would be to build a fully functional filesystem of files starting with l by building metadata from the results of `ls l*`, but this could be made more complicated by taking output from a complex query system such as GEMS [37].

The flexibility to include any object as the target of a metadata pointer allows for referencing singleton objects that are on a host not in the list of available resources. That is, even without opening a host up for inclusion in a pool of resources (to be written to), the files on that disk can be included in a FOBS filesystem as needed. This ability presents several challenging characteristics not seen in most systems. The first is keeping track of the resources used by the filesystem. It is hard to know what physical resources are being used by (as opposed to "are part of") the filesystem, as singleton files' hosts are not listed in the configuration metadata. Thus, the system would have to scan every file's metadata in order to enumerate all underlying hosts. Alternately, a list of this information could be kept as part of the configuration metadata, however it would require a third actual file access for every FOBS file access to ensure that the list is kept up to date, concurrent access would be a new issue, and the file would have to

contain actual reference counts, rather than simply a list in order to allow correct removal of a resource from the list. In light of these disadvantages, the prospect of scanning every metadata file to determine a list of resources being used by the filesystem isn't unreasonable, especially since it isn't a common operation.

One way such a scan could be done without making several passes through a possibly changing directory is to create a FOBS filesystem for accounting in which the underlying objects are the metadata files for the filesystem of interest, as described above. This will give a snapshot of what metadata files were there at the time of the scan (their contents may have changed, since the "accounting" FOBS filesystem only references the underlying objects, it does not lock them).

A second bookkeeping issue is that of total space available on a FOBS filesystem. Due to singleton files, a filesystem can use more space (as measured by `du` for instance), than it has total space available (as measured by `df`). This requires examination of applications working on a FOBS filesystem to ensure that the size of the singleton files can be included in the total size of the filesystem, without causing errors for a perceived impossibility of using more bytes than are available. This is especially notable considering that many quota programs base warnings or errors on percentage of resources used.

3.2 Implementation Architecture

In the implementation of FOBS, the underlying filesystem is a Tactical Storage System [32]. This gives two important benefits to the FOBS filesystem: it is able to be deployed and configured without administrative privileges, and it is possible to to implement a variety of policies regarding access control within the filesystem beyond traditional UNIX permissions. The adapters that allow access to the

resources on the Tactical Storage System are parrot [31], and a FUSE module that connects to Chirp servers.

3.2.1 File Access, Modification, and Creation

File access consists of two phases: the lookup and the actual access. Once the filesystem is initialized, that is, once the list of hosts and the placement location have been loaded from disk, the FOBS client requests the metadata pointer file from a well-defined location. This metadata pointer file is then streamed to the client using a single RPC. The client parses the metadata file to complete the lookup phase, and then issues further RPC(s) on the underlying object to complete the actual operations on the file.

Reads and writes to an underlying object can be done in two manners, through individual read/write RPCs, or through putfile/getfile streams. The advantage of the former is that the entire file need not be transferred to read/write a small bit of data, and the infrastructure to set up buffers to store streamed data before and after network transport on either end is not needed. The advantage of the file streams include many fewer RPCs, and not having to wait for the complete round trip of an RPC and acknowledgment before sending the next piece of data.

The ordering of the operations in creating a new file in a FOBS filesystem is important, as discussed in Thain, et al [32], and it is recapped in this work because it has several implications with regard to naming, file migration, and error handling.

To codify the process for new file creation, a round-robin list of resources available for new file placement is part of the configuration metadata. When the filesystem is initialized, an initial choice for the next file placement is set. Thus,

the first time that a file is to be created on the FOBS filesystem, that choice is selected. The path into which it will be placed on the underlying resource is determined by another piece of configuration metadata, the "placement key", which gives the directory hierarchy of the path. The filename of the object is determined at placement time; it was suggested in the introduction of the DSFS that "a unique data file name is generated from the client's IP address, current time, and a random number" [32], however, in implementation a random string is used instead. The implications of this are discussed in the next subsection.

When a client requires a new file creation, the metadata pointer file is created first, and filled with the name of the chosen host, along with the path in which the object will be placed. Next, the underlying file corresponding to that path is created on the chosen host. If not done in this order, a failure during writing to the data file or between the two file creations would result in an object with no references pointing to it. This is akin to a memory leak, whereas a failure at the same point with the proper ordering of operations is akin to a broken/null pointer. The advantage of the latter is that the location of the remnant of the failure is well known (it is the name given to the file), and it is small. The "memory leak" on the other hand, isn't necessarily well-known (the underlying file's name is just a random text string), and could be very large.

3.2.2 Naming

Another way to mitigate the risks posed by a failure causing such a memory leak is to change the naming scheme for underlying files created by the FOBS filesystem. There is a simple alternative to random character filenames, but it fails to solve all problems surrounding the issue. One possibility is to name the

underlying file based on the namespace name, that is, with a name derived from the metadata pointer file's name. Two issues with this are multiple metadata pointer files targeting the same underlying file, and the possibility of renaming the pointer file. The former is an issue because, as discussed later, the underlying object or another FOBS metadata file referencing the object have no way to know when the metadata pointer file responsible for the object's name is removed (in the absence of removing the underlying file with it). Alternately, if there are multiple metadata pointer files referencing the underlying file, to the metadata files that do not share the name, this scheme has no advantage over the random character string version. Finally, changing the name of the metadata file would then require changing the name of the underlying file, an access which is not currently required, and one that would require further synchronization and atomicity. Thus, maintaining a random object name and mandating the ordering described above is at least as sound an option as the alternatives.

3.2.3 File Migration

Unlike block devices, in which the configuration of the underlying hosts must be especially well-known, object storage requires only a simple list of the resources. This is not just parsimony, but rather has the advantage that it allows for flexibility in expansion. Because adding a new host simply adds to a list, rather than requiring a complex reconfiguration of resources, expansion can be done on-the-fly. This has two benefits, first that the system can systematically grow in capacity to meet demand without down-time at each upgrade, and second that in a time of need, more resources can be hooked in to the filesystem quickly to increase capacity and maximum aggregate throughput.

Two challenges associated with this flexibility are metadata staleness and imbalance in the system. If a resource is removed from the hosts file, the state of the metadata doesn't necessarily change with it until refreshed – for example, the the "next host for placement" pointer mentioned above could point to a host no longer in the hosts file. In this case, the next file placement either tries to open a file on a host no longer designated for new file creation, or on one that no longer exists at all. In order to take advantage of this flexibility, then, a FOBS client must refresh the state of the configuration metadata (using the same process as to initialize the FOBS filesystem on first access, not by taking the system down completely as in many distributed filesystems) in order to realize such changes. Thus, while the system can change on-the-fly without system downtime, there is some overhead required to refresh the state of the FOBS clients periodically.

Another problem raised by the ability to add and remove hosts from the system on-the-fly is that in a growing system, adding new hosts introduces imbalance into the system. The new hosts have no data stored on them, so the parallelization is not immediately realizable for reads as it is for writes. Thus, it must be possible to reorganize files currently on a filesystem to distribute them evenly across the underlying resources. Even with a perfect system of filesystem reorganization, however, the system is at the whim of the user to connect new devices or remove old devices, and some element of system balance must remain with the user instead of being built to run autonomously inside the bowels of the filesystem.

Additionally, like with file creation, in practice, a specific ordering of operations must take place for file migration. A temporary metadata pointer file must be created, then a third-party transfer of the underlying object to its new location can proceed. Once this is complete, and the new contents verified, the original

underlying file can be deleted, followed by changing the original metadata to reflect the new location, and then removing the temporary metadata file. Failing to create the temporary metadata file leaves the system vulnerable to a "memory leak" (in which the new copy of the data is unacknowledged by metadata) if the system fails before the old metadata is changed to reflect the new location. Similarly, the old underlying object must be removed before changing the old metadata.

3.2.4 The Problem of Multiply Referenced Objects

Underlying reorganization and other elements of FOBS flexibility present problems when the object is referenced by multiple metadata pointers. Examining the options for underlying object migration, there are several possibilities for dealing with this. If none of them is sufficiently stable, however, the default "zero knowledge" platform (in which an object knows nothing about what metadata is viewing it, and metadata knows nothing about what other metadata is viewing an object) is the most apt solution, as it maintains the simplicity of the architecture.

A first approach is to consider taking action when an action is performed on the underlying object. FOBS could utilize something akin to operating systems' callbacks. It can be known where to call back to if the other viewing FOBS filesystems are listed in the metadata (this could be accomplished with a second file for each underlying file on the same device, with a well known name, listing the "viewers"). However, it would also require each FOBS filesystem to have a world-writable configuration file to allow updated locations of files to be listed, and this would then require a full-filesystem scan to find the metadata file that previously referenced the now-moved file. Even this wouldn't be sufficient, because not every location is visible to every other location, based on access control. The viewers

list (or even a reference count) file means that there is a factor of two overhead in the number files stored on the system, but without such a list, it is impossible to know who else sees it without adding such information at the top of the actual underlying file.

Storing the information in the actual file is not a good solution because it fundamentally changes the underlying object based on what filesystems are referencing it; this affects checksums for the data, and requires all accesses to the file to exclude this built-in metadata (requiring changes to stat, seek, etc.). Additionally, it destroys the duality of the object as both a regular Chirp file and a FOBS object; with the added FOBS metadata, the object is no longer the same semantic Chirp file as it was before.

Another option is a shadow file in the same location as the now-moved file, which lists the new location. However, it is unclear as to when this would be required to be noticed. Would it simply be on-access, or would a periodic scan be necessary? If on-access, as the system could change on-the-fly to access the new location, but there are two more issues: what if the FOBS doesn't have access to the new location, and how long does the shadow file have to stay? Similarly, a shadow file that lists the new location does little good for moving files off of a disk that will be put out of service. A periodic scan impacts performance, with no guarantee to increase accessibility, as this does not handle the retiring disk case either.

Finally, the last issue considered is with policy of permissions. As built into the system, each object contains a set of access controls on a directory basis. While FOBS could have "metadata abilities" associated with each object, this has several problems. Consider, for instance "FOBS:ccl00.cse.nd.edu@MYFOBS

ndm", that is, the system controlling the FOBS filesystem MYFOBS hosted on ccl00.cse.nd.edu has the ability to rename, delete, or move the underlying object. First, this is a directory-level access control, which makes differentiating capabilities to various underlying objects difficult, and more importantly, it assumes that anyone with access to that particular FOBS filesystem has the same right to do those actions as any other one. This is problematic due to the lack of a singular identity of a FOBS filesystem user, as well as conflicting goals between different FOBS filesystems using that underlying object.

Because no single FOBS filesystem "owns" an underlying object, and there is no good way to differentiate between a user of a FOBS filesystem and the internal processes of the system itself, this is currently handled in a passive manner: access to underlying objects is controlled based solely on their access control as individual objects on a Chirp server, rather than as data in a FOBS filesystem. Users retain access to the underlying file that they have to it natively as a Chirp file, regardless of which FOBS filesystem they are using to access the underlying data, and a FOBS filesystem may never be sure that its referenced objects must remain in place for the duration of the reference.

Hopefully, future work may establish further constructs for staking FOBS-level claims to underlying objects. This would prevent references from being broken by other FOBS servers modifying their (shared) referenced objects, or at least formally codify the semantics for such operations. For this work, however, the complete independence of the underlying file from its inclusion in a FOBS filesystem is asserted: anything that a FOBS user could do to the Chirp file, he can do to it through the FOBS interface, regardless of implications of that change on other FOBS filesystems.

3.3 Dynamism in Cooperative Storage

A cooperative storage system, in particular the dynamic nature of it, introduces new opportunities and challenges in the design and implementation of personal storage systems.

- **Dynamic Systems.**

  Dynamic systems are marked by resources changing status frequently. This could be any number of actions: joining the system for the first time, rejoining after a disconnection, announcing new capabilities, disconnecting temporarily for a planned outage (perhaps for maintenance or reboot), disconnecting due to failure, or disconnecting permanently (perhaps due to a machine's scheduled decommissioning, or an owner's cessation of involvement in a research project). Some of these can be planned for, others cannot, however each has a different set of effects on the state of a system, and each is important for harnessing cooperative storage infrastructure into a personal mass storage system.

- **Cooperative Storage Dynamism.**

  Cooperative storage systems are particularly dynamic. Unlike with centralized storage, the cardinality of the set of disks combined into a single filesystem can grow beyond tens of file servers. Because of this, however, there is a much larger demand on the system to handle dynamism within the set of resources, such as those status changes described above. Without scaling the number of component resources beyond the hundreds of disks, one can already imagine a system in which at any given time there will almost certainly be some resource changing status. This can be as simple as a user sitting down at the terminal and running a job that consumes

the system's resources such that the machine cannot report to the central resource server or cannot access the disk a shared file occupies, and is thus dropped temporarily. Alternately, a brief burst of network traffic congests the network, causing packets to be dropped, and yielding the same result. Additionally, cooperative storage servers must ultimately serve the resource owner, so they can be power cycled, have the file server software shut down, or have components added or removed. Finally, cooperative storage systems are particularly dynamic when storage is culled from many owners or administrative domains. For example, a resource owner can collect several systems together, join them with existing cooperative resources, set up a FOBS filesystem for use as a temporary storage dump, then, just as quickly, end his experiment, disconnect his resources, and return the system to its previous state.

- **Reliability in Dynamic Systems.**

  Reliability in the face of such dynamism can be broken down into two separate characteristics: maximization of available resources at any given time, and sensibility in dealing with failure or unavailability. The first component is important because if a cooperative storage system is to function as mass storage, it will contain data that is likely not backed up on an alternate device, or data that must be recomputed at great computational cost if unavailable. The second characteristic is related; maintaining system stability in the face of suboptimal availability is hard, but selecting appropriate responses to errors is a key weapon in preventing a single localized failure from crippling the system in its entirety.

- **Multi-level Failure Management.**

  It is impossible to build a system that maintains perfect reliability from within. However, as stated above, reliability is a key consideration of a cooperative storage environment. Because of this, it is necessary not only to build some reliability assurance into a cooperative storage system's internal structure, but also to support such measures above and below the cooperative storage system itself. Replication, parity, and erasure coding can be built into the internal structure to aid in availability, but more importantly the system must facilitate external measures, as these are more configurable to a user's needs. A cooperative storage system must support being run on top of a RAID system, or being run under a top-level file management system that implements those fault-tolerance measures. An example of this multi-level approach would be an application that creates m-of-n recovery files (for instance, Tornado-Code packets) and places these files onto a FOBS filesystem. Taking it one level further, that FOBS system could consist of underlying disks that are part of a mirrored RAID array.

- **Unexpected Policy Coupling.**

  When multiple servers must participate in an activity, their policies may interact in unexpected ways. An example below will illustrate that a user that deploys a FOBS filesystem across several nodes may discover that his permissions are different on each of these nodes, that the file placement policy interacts with the possible access controls that may be placed upon those files. This introduces interesting challenges for the user, as well as the other users of the shared resources.

3.3.1 Management in FOBS

Decentralization of storage allows users to harness many more resources than would be available on a single centralized system; however, the unreliable nature of a single resource is compounded due to reliance on a potentially large group of diverse resources and a network backbone that is less reliable than a motherboard or system bus. Some formerly straightforward management procedures are made less clear, but new opportunities for dependability and performance become possible. For each example, there are opportunities distribution affords a user, unique problems that arise due to dynamicity, several potential solutions to these problems, and continuing challenges dynamic distributed systems face relating to the example.

FOBS has a more complex failure model than a single disk or even a single Chirp file server. In order to access a file, a client must successfully communicate with both the directory server and the relevant file server or servers. If a user's attempt to write out a file fails with a "permission denied" error, in a centralized file server it is fairly obvious that the permissions on the file do not allow that user to modify the file. On a FOBS filesystem, however, it could be any one of several errors:

- no permission to write to the underlying data file
- no permission to write to the metadata pointer file (if it required any changes)
- no permission to create a new file
- no permission to delete a previously existing file, as required by some striping or replication algorithm

When viewed as a dynamic system, this complex failure model is magnified, and it is clear that it plays a role in determining storage policy. In addition to policy, this observation supports the file-level object storage approach used in FOBS. If a file's data is stored on several disks, the failure of any single one of them can cause an operation to fail. Traditional RAID striping is not sufficient for use as a personal mass storage system across a grid environment: the failure of a single resource (a server or its connection) cripples the entire system.

Even though FOBS has restricted placement to single files in their entirety on a single disk, that still doesn't eliminate the concerns about performance in a dynamic environment. Poorly planned failure response still can cripple a system, and no single failure policy works across the board. Consider three possible failure policies for writing a file: return with a failure message and let the writer determine when to retry; return with a failure message and institute an exponential back-off before retrying; and retrying immediately on a different host, while remembering failed hosts so as not to try them again.

The first option seems reasonable in a dynamic system – after all, it is quite possible for a resource to be unavailable now yet become available by the time the next request comes in. Consider, however, trying to write a file to an extremely popular host (perhaps one that, ignoring security, hosts a password file required by many of a user's applications, or stores a large configuration file that cannot be cached) that resides on a server with an inadequate network connection. In a large system with many users, or many applications and threads from one user, trying to fetch this file, a failure returning as "busy – try again" will likely cause an application to retry immediately. This in turn will continue to keep the server's network congested, delaying or precluding the host from serving its current job,

while stacking up new requests in the process. Thus, an inappropriate failure policy can cause users' normal actions to result in a sort of distributed denial-of-service attack against their own storage systems. A better choice would be exponential back-off. In this case, if the server is only temporarily busy, the quick retry will likely succeed, and if the server remains busy, the exponential nature will prevent the retry requests themselves from congesting the network.

The exponential back-off seems a good solution, but it isn't ideal either. In [33] it is noted that authentication ability and file access permissions change rarely, and this is especially true for cooperative storage, where policy change is slow due to multiple principals and administrative domains. Thus, it is not reasonable to try an exponential back-off technique when the error returned is "permission denied" or "access denied". It is likely that even retrying several times (until the point at which the "back-off" period between attempts longer than the timeout of the operation) will not garner better results. Instead, an immediate retrial on another node would be more advisable. Furthermore, figure 4.4 shows that maintaining a memoization table adds little overhead, while reaping large benefits for systems where a user may only access a very small portion of the resources. Thus, even if only a single placement host in a FOBS filesystem is available to the user, he can try every single host to find that one, and then repeatedly use only that one (either relying on the memoization process, or setting a system variable for placement choice). Memoization is discussed at greater length below.

The case presented in that work is particularly suited to a FOBS filesystem built on a subset of a large cooperative storage environment, perhaps to work within a different namespace or a smaller set of files. Another choice in such a situation would be limiting file creation to a single disk. This way, many disks

may be pointed to by a FOBS filesystem, however a user will only create files on the disk specified. This seems to fail if that disk is unavailable. On one hand, it is no worse than a single disk – if the user has no other permissible disks, he must wait until that disk recovers (the same as using a single disk for personal storage) – but it also ignores the aggregate write bandwidth advantage of a FOBS filesystem, as well as the raw capacity advantage.

Memoization isn't the catch-all solution, either, however. Not all errors exhibit the characteristic that access denied errors do; some errors are completely transient. Memoizing away nodes that fail with temporary errors would quickly eliminate a large number of otherwise available hosts, and wouldn't get the benefit of avoiding retrial on hosts that "you know will fail again", since that is *not* known. An example of this is a failure with the returned error "too many file descriptors open". Clearly that host will not *always* have its maximum number of file descriptors in use. This is an example of an error where trying again immediately on the same host is a reasonable solution. Having all of its file descriptors in use does not indicate the host's resources are being overworked (CPU, network saturated, etc.), rather that it has many files open. This means the system is unlikely to run into the DDOS problem encountered in a previous case, and doesn't need to exponentially back off. Although a dated reference and tailored towards individual systems, the oft-cited conventional wisdom [18] suggests that files usually stay open for short periods of time; so an immediate retrial may be the best solution to this problem.

Without belaboring the point, choosing how to respond to failure is critical in a dynamic distributed system. There is no catch-all solution, and complex failure models in dynamic systems further complicate error handling. Well-designed sys-

tem policy for handling certain errors in certain situations can, however, prevent overall system failure in the presence of individual failures. As indicated above via example, I believe that the key components to consider in designing effective error handling for mass storage systems such as FOBS are: determining permanence or transience of failures (retry now, later, or never?), preventing compounding failures (exponential backoffs, for instance), and maintaining some minimal level of performance in the face of failures.

3.3.2 Load Balance Considerations

In a completely distributed system, the storage load of the system should be spread evenly across the storage nodes. Even under the file-level object storage model, this opens up the possibilities of striping data sets on a by-file basis for aggregate performance, distributing evenly among resource owners for fairness, as well as other considerations. In a dynamic system such as this, however, new disks are added, old disks are retired, and disks cycle between available and unavailable states for myriad reasons.

When a virgin disk is added to the system, it introduces imbalance to it. There are two clear policy alternatives: rebalancing on entry into a system or rebalancing through use. Rebalancing the system to incorporate the new disk immediately has the advantage of immediately making use of increased aggregate read bandwidth once the rebalancing has completed. On the other hand, it requires running a potentially complicated rebalancing algorithm, which takes time and resources, and may result in constant system flux in a dynamic system with a constantly changing resource set.

Imbalance in a FOBS filesystem results in decreased performance due to relative overtaxing of the resources of one disk, while under-utilizing the resources of another. Additionally, in a scavenging environment, imbalance can also upset resource owners, who may feel as though they are not being treated fairly if their disks are filled disproportionately to the other disks.

The lesser abilities of unbalanced systems have similar negative effects on a single user's workload as they do on the aggregate workload of a large load of users (ineffectiveness of parallelization, saturation of disk and/or network bandwidth, no immediate benefit of adding extra disks, etc.).

The dynamic nature of cooperative storage and the ability for on-the-fly expansion of FOBS combine in an interesting manner in terms of usability of the system. A system may fill up very quickly due to many users operating concurrently, or even one ravenous user. If space in a FOBS filesystem becomes scarce, more disks can be added to increase its capacity without taking down the system to change its configuration. This allows for swapping in and out disks on-the-fly (either in one-for-one replacement of small disks with large ones by using file migration, or continually adding more space on new disks without removing the old ones), however it also presents a balancing problem among the disks of the FOBS, which can affect performance.

For example, consider a FOBS filesystem in operation with 10 disks, evenly filled to 50% capacity. If an eleventh identical disk is added to the system, capacity is increased by 10%, but aggregate read performance does not immediately increase, as increasing capacity does not automatically rebalance the system. Even as more files are placed in a round-robin or random method, the new disk will remain under-loaded relative to the rest of the disks, though aggregate

read throughput will increase assuming the newly written data is accessed. Other possible causes of imbalance, and thus limited aggregable read throughput, could be that a disk is disconnected from a balanced system, is offline during a flood of file placements, then reconnects and is now under-loaded.

Imbalance can occur even in a static system, simply due to the non-omniscience of future operations and demands. Consider a round-robin system, with an two 1TB-capacity disks, onto which a scientific computing program stores a small (1KB) configuration file, then a large (1GB) data file. After one thousand file placements are done, one disk will be nearly empty, having stored only 1MB, while the other half of the system will be at absolute capacity, and realistically will have already reached capacity by the time the last file is placed.

Various disk replacement and load rebalancing algorithms are well-studied [30], and may be of use to increasing performance in FOBS, however the particular algorithm is not as relevant to our discussion as is the recognition that in a FOBS filesystem, balanced loads are not guaranteed, and can be unattainable in some cases. The implementation of FOBS has several built-in RPCs that can facilitate on-the-fly balancing by an external application, with the caveat that, as discussed above, there are repercussions of moving underlying files in an environment with multiple FOBS referring to the same underlying objects.

Likewise, when a disk is scheduled to be removed from a system, its data must be preserved, and executing an application to do this can disrupt a system's balance and performance. One approach would be to designate a destination disk for the files from the retiring disk. This is bad for several reasons – it is not guaranteed that there exists a disk that can fully bear the weight of all the files from the retiring disk, and even if there is, this disrupts the balance on the

system severely. Additionally, it precludes parallelism in emptying the disk, which slows down performance in general, and specifically leaves the system at a greater chance for data loss if the retiring system fails or goes offline early [38]. This is especially wasteful if the source disk has more bandwidth capacity to send than the destination disk has to receive. A better choice is distributing the retiring disk's files among the rest of the available resources. This makes better use of parallelism, naturally maintains better balance, and requires only that the system have enough capacity to handle the retiring disk's contents, rather than that requirement from any single disk on the system. The downside to this is that transferring files from a retiring disk takes up resources from every other disk in the FOBS, rather than a single disk and network link.

3.3.3 Execution at Data Location

The fact that the principals of the filesystem are a set of metadata files that point to multiple remote servers also allows for a top-down distribution of filesystem tasks. Tools can be made, or RPCs included into the filesystem protocol, that use the location of the object as the operator for tasks such as file checksums and stats. Also, unlike normal distributed filesystems in which the user has little say in the remote location of his objects, within FOBS, this can be specified before placement, or modified after placement. Additionally, instead of a two-hop transfer (up to the user's host, then down to the new target) to change the underlying location, third party transfers can accomplish this within the filesystem. Another idea that is a more general extension of these, though still not fully explored, is that of "active storage", in which the host on which an underlying object is stored is responsible for some computation on that object (at the command of the user of

the filesystem, without having to directly access that host). A built-in example of "active storage" would be the local MD5 calculation, which relies on the storing resource to report back the MD5 checksum of the stored file, instead of having to transfer the file to the machine requesting the checksum for local computation.

CHAPTER 4

EVALUATION

4.1 Architecture-based Latency

4.1.1 Problem and Observation

The metadata layer in a FOBS filesystem introduces a second file access for each actual data file access. This means that there are multiple RPCs for each access that requires use of the metadata. For example, an open requires a getfile of the metadata file, then the actual open of the data file. This additional access injects a longer latency into each operation compared with operations on the data files without the metadata download.

The additional latency becomes an issue for the overall performance of a FOBS filesystem if it is of the same order as the base operations themselves. The latency can be evaluated in two ways: first using microbenchmarks to measure the difference caused by additional latency, and more importantly on real applications and larger common operations.

4.1.2 Hypothesis

This artifact of the architecture should make a difference in microbenchmarks that utilize the metadata namespace, but there should not be a large difference

for microbenchmarks that isolate operations on the data files themselves, as no file transfer is being measured, just another level of RPC layering.

Because the initial namespace resolution is dwarfed by the subsequent RPCs, unrelated disk and network latencies, and actual service time, the hypothesis is that for real applications the difference between a FOBS filesystem and the underlying filesystem (a Chirp server) will be negligible for workloads that do not strongly emphasize the namespace with many separate lookups required.

4.1.3 Results

A FOBS filesystem currently requires working within the Parrot interface or FUSE interface in order to make use of regular system applications (cp, mv, rm, stat, etc). This could be avoided by writing standalone FOBS versions of each of these commands to be used in place of the regular applications. This is especially feasible on a limited scale if the application is written as a general purpose replacement, which will utilize Chirp RPCs, FOBS RPCs, or native UNIX libraries as appropriate depending on the source and target files. However, as a more general measure (not every application can be rewritten to service FOBS natively), the two existing adapters are used to benchmark the FOBS filesystem.

Table 4.1 shows execution time, in microseconds, of several system benchmarks on a local disk, a local disk through the parrot adapter, a remote chirp server through the parrot adapter, and a FOBS filesystem with remote data servers and a local metadata server through the parrot adapter, in which remote operations are conducted over a gigabit Ethernet LAN. Overhead due to the FOBS architecture is evident by comparing the stat and open benchmarks between the Chirp system and the FOBS system. The overhead caused by the parrot tool is within

TABLE 4.1

MICROBENCHMARK PERFORMANCE USING THE PARROT ADAPTER

	Disk	Disk (Parrot)	Chirp	FOBS
write 1B	4.54 ± .02	47.58 ± .39	281.90 ± 39.57	267.47 ± 30.34
write 8KB	7.63 ± .03	53.67 ± .39	367.54 ± 54.72	373.11 ± 46.63
read 1B	1.08 ± .01	33.83 ± .33	31.96 ± .51	32.02 ± .30
read 8KB	2.72 ± .01	41.76 ± .41	39.72 ± .28	40.27 ± .59
stat	1.88 ± .01	45.42 ± .48	290.13 ± 12.32	612.63 ± 116.56
open	2.95 ± .01	72.36 ± .62	844.99 ± 130.61	1122.37 ± 50.31

expectations from the literature. Table 4.2 shows the same microbenchmarks on a remote chirp server and a FOBS filesystem using the FUSE adapter for comparison between the adapters.

As expected, in microbenchmark tests, those emphasizing the namespace lookup and requiring multiple round trips to complete show a difference in performance between a FOBS filesystem and the Chirp server without a metadata layer on top. The overhead is not as evident in the read and write benchmarks, however, especially the larger data sizes, which indicates the lesser effect of the latency on these operations.

The microbenchmarks also serve to illustrate that the FUSE adapter is considerably faster than parrot for reading data, and completing the stat and open file operations. This is due to the FUSE module's location below the buffer cache, such that reads from cache do not trigger a remote access through FUSE, which

TABLE 4.2

MICROBENCHMARK PERFORMANCE USING THE FUSE ADAPTER

	Disk	Chirp	FOBS
write 1B	4.54 ± .02	222.45 ± 61.98	258.43 ± 24.57
write 8KB	7.63 ± .03	379.89 ± 40.96	359.30 ± 46.79
read 1B	1.08 ± .01	0.88 ± .05	.94 ± .21
read 8KB	2.72 ± .01	2.33 ± .02	2.30 ± .02
stat	1.88 ± .01	3.41 ± .03	3.53 ± .15
open	2.95 ± .01	574.95 ± 186.46	798.92 ± 205.56

comes at the cost of potential consistency concerns. The difference between the two was less apparent for writes, which could be an artifact of the higher variability in the operations (as seen in the wider confidence interval) as well as due to the FUSE requirement that data actually be written through to the storage.

On several common workloads, however, the difference was not as evident as one might fear after seeing the benchmark results. Table 4.3 shows execution time, in seconds, of several common UNIX file operations on a remote Chirp server and a FOBS filesystem, both using the FUSE adapter. Operations in which the time for the metadata lookup was dwarfed by data movement (tar) or computation (gcc, make, bzip) yield results with less than 3 percent overhead, and in some cases absolutely negligible differences. For unpacking a tarball, the overhead is on the order of 10 percent, which is likely due to the large amount of metadata interaction to create the files that were unpacked. From this, it seems

TABLE 4.3

SAMPLE MESOBENCHMARK PERFORMANCE USING THE FUSE ADAPTER

	Chirp	FOBS
mkdir	.009 ± .005	.024 ± .011
rmdir	.003 ± .005	.021 ± .063
tar	.830 ± .155	.835 ± .052
untar	.961 ± .205	1.062 ± .144
diff	.020 ± .001	.022 ± .002
gcc	.102 ± .016	.105 ± .021
make	2.321 ± .052	2.266 ± .102
bzip2	4.427 ± .032	4.451 ± .032
bunzip2	1.917 ± .060	1.932 ± .045

clear that the performance cost associated with the increased latency of the FOBS filesystem's additional metadata lookup is small enough that this alone does not make the system unusable for most users, as either a general purpose or specifically scientific computing filesystem.

It is notable, however, that some small operations that deal exclusively with the FOBS metadata do show a similar performance cost to that seen in the microbenchmarks. In the implementation, making or deleting a directory does not affect the actual data in the FOBS system, instead changing only the metadata; these operations were also the ones that demonstrated the highest overhead, as they represent a metadata operation with absolutely no data access, which isolates the extra layer of RPCs to serve the FOBS architecture. Even in this worst case isolation, however, the order of magnitude difference in performance is mitigated by the fact that these are very fast operations even after the large overhead.

On a compute-heavy workload, in which the data access time is small relative to the time to complete the computation, there is absolutely no evident difference. Consider, for instance, a system that must serve a large set of configuration files for a large scientific computing application such as the searching for Mersenne Primes [36], XtremWeb [8], or Folding@Home [16], which transfer small amounts of data to prime a simulation or computation engine and then wait for many CPU hours. The requirements of these servers is availability to deliver under high load factors, with less emphasis placed on raw turnaround time for requests so long as they are served within reason; it is better to serve users at a slight delay than to crash and forfeit any chance for those CPUs to do useful work. The FOBS system's high capacity and ability to service large load factors, combined with the insignificance of the overhead seen for small writes when compared to hours or

days of computation make FOBS a strong candidate for a deployment file server for such an application, even though it is not the application for which FOBS was specifically engineered.

4.2 Throughput Scalability Under Load

4.2.1 Problem and Observation

A single file server will struggle to expand performance under load beyond some small threshold, simply due to finite resources of the machine. For large datasets, it is unlikely that individual users will want to provide the storage resources required to maintain their own copies, and thus, it is unsurprising that storage systems will encounter heavy load.

Utilizing many underlying machines allows aggregation of their individual resources (processors, memories, network links, buffer caches, etc.); however, this is only true so far as the resources are arranged in a way conducive to working in parallel. There are two workloads, then, that must be tested: one in which each client accesses all of the files of a data set in random order, and one in which each client accesses the data set in the same file order. For these tests, the data is distributed such that consecutive files in the sequential workload reside on different servers, which is a common technique and one that fits the default file placement algorithm for FOBS.

The random file ordering measures an ideal case for a storage system: accesses are targeted at various resources throughout the system, so from the first access the entirety of the underlying resources may be used. The sequential case, however, initially puts the strain on a single resource (a hotspot), and continues to put the maximum load on each subsequent resource (hotspot migration) as

the clients finish one access and move to their next target. The only way this workload will make use of all available hardware simultaneously is if the load causes stratification of clients throughout their workload – that is, the severity of a hotspot is diminished due to failure in performance at a previous hot-spot, and that will come at the cost of a high finishing time for those that are delayed on the first several tasks. Thus, an effective way to measure this test is in worst-case performance, which measures aggregate throughput in terms of the entire set of clients as a single job, spanning the start time until the last client finishes. For comparison, this metric is also used for the random access test.

Considering that the centralized metadata is a single point of failure, and that this machine bears equivalent loads for both random and sequential on the metadata server is another concern, but a server should be able to handle enough small RPCs to serve even a large number of clients accessing a large metadata set without unreasonable increases in latency.

As another approach to using the object storage concept to aggregate distributed resources, replication of objects across a storage cloud can be used instead of distribution within a distributed filesystem. This should provide an interesting comparison, although it comes at the expense of the overhead to add additional replicas, in addition to ensuring their consistency and determining a sound distribution of placements, which is considerably greater than the overhead to add a metadata file in building a FOBS filesystem.

Finally, a distributed system must maintain tolerable performance when underlying nodes are unavailable. It is easy to see that time spent waiting for timeouts, retrials, or rejection notices on unavailable resources detract from throughput performance, as no goodput is achieved by the RPCs that fail, however repeated

occurrences of this can be avoided. Thus, for the case of high unavailability in a system, the performance of a memoization algorithm in FOBS is examined. In order for this to be useful, it must not degrade performance significantly on systems with high availability.

4.2.2 Hypothesis

The system will demonstrate this ability to harness aggregate resources, resulting in higher aggregate throughput under load – possibly even for quite small loads. Additional aggregate throughput should be gained from using FOBS filesystems with more underlying disks, especially for the random access order workload. This should have an element of diminishing returns, however; that is, there is unlikely to be a factor of N speedup for addition of a factor of N disks.

Between the two workloads, for high load factors, the random access will yield higher aggregate throughput, as well as a more obvious delineation of FOBS setups. Additional disks will provide more of a benefit to this workload, as the system can continue to function at a high level until the capacity of the system (which, in this case, means capacity of each underlying node) is reached. The simultaneous sequential workload, however, gives a better picture of how resource aggregation performs given imperfect balance and utilization, and motivates the use of other techniques such as replication.

The metadata server should not be a limiting factor in a FOBS system up to the limits of the Notre Dame Chirp pool testbed, as the metadata pointer files are small and require only a single small file stream transfer, and the testbed is a campus network, so most connections will be short-lived.

In the alternate object storage approach: replication within a cluster environment, a similar aggregation curve shape will be apparent, and performance will be sustainable to large load factors of at least dozens of nodes. This still would indicate the validity of using a distributed object storage system to deal with high load, even though it is a different approach than that taken by the system described in this thesis, and comes at the expense of high consumption overhead.

Finally, a simple memoization approach should drastically improve turnaround time for file placements in systems with large sets of unavailable resources, while suffering minimal performance overhead on highly-available systems.

4.2.3 Results

Throughput tests showed that each system (an unmodified Chirp server and FOBS of several numbers of underlying storage nodes) had similar throughput capacity at the base of one or two clients connecting simultaneously for both reads, in Figure 4.1 and writes, in Figure 4.2. By the fourth client connection, however, the Chirp server alone can already be picked out as the worst performer.

By the eighth client, it is clear that the 2-node FOBS setup has reached its peak, and for all four workloads either slow their total throughput gains significantly or stagnate altogether. Beyond eight clients, the 4-node and 8-node FOBS systems begin to flatten, especially for the synchronous workloads. This is not surprising, especially for the 4-node setup since, at that point, it approaches and surpasses the disk-multiple of the number of hosts at which a single Chirp server plateaus.

For a load factor of 16, there is a clear difference among the setups, falling in expected order based on number of resources available. The only exception

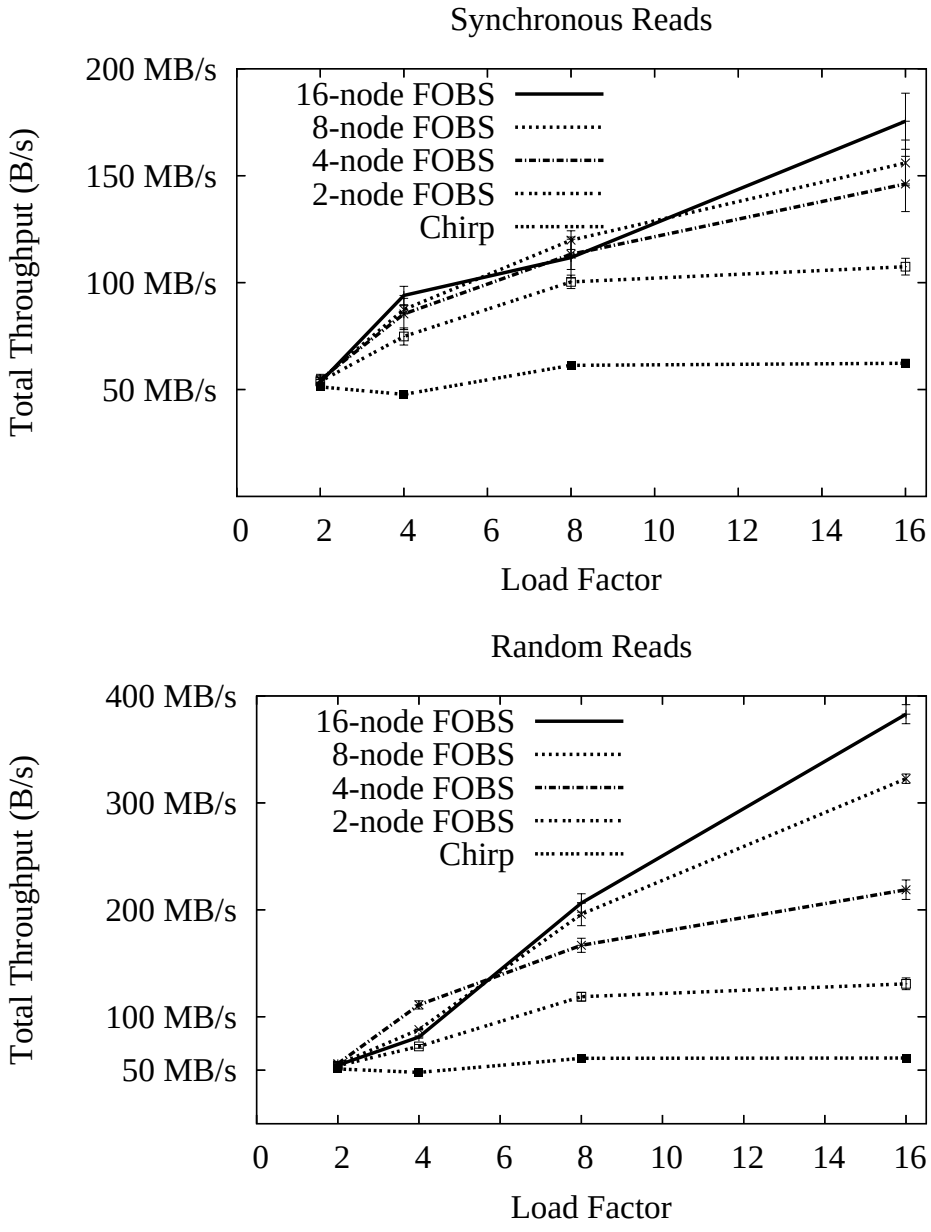

Figure 4.1: Aggregate Filesystem Read Bandwidth Under Load

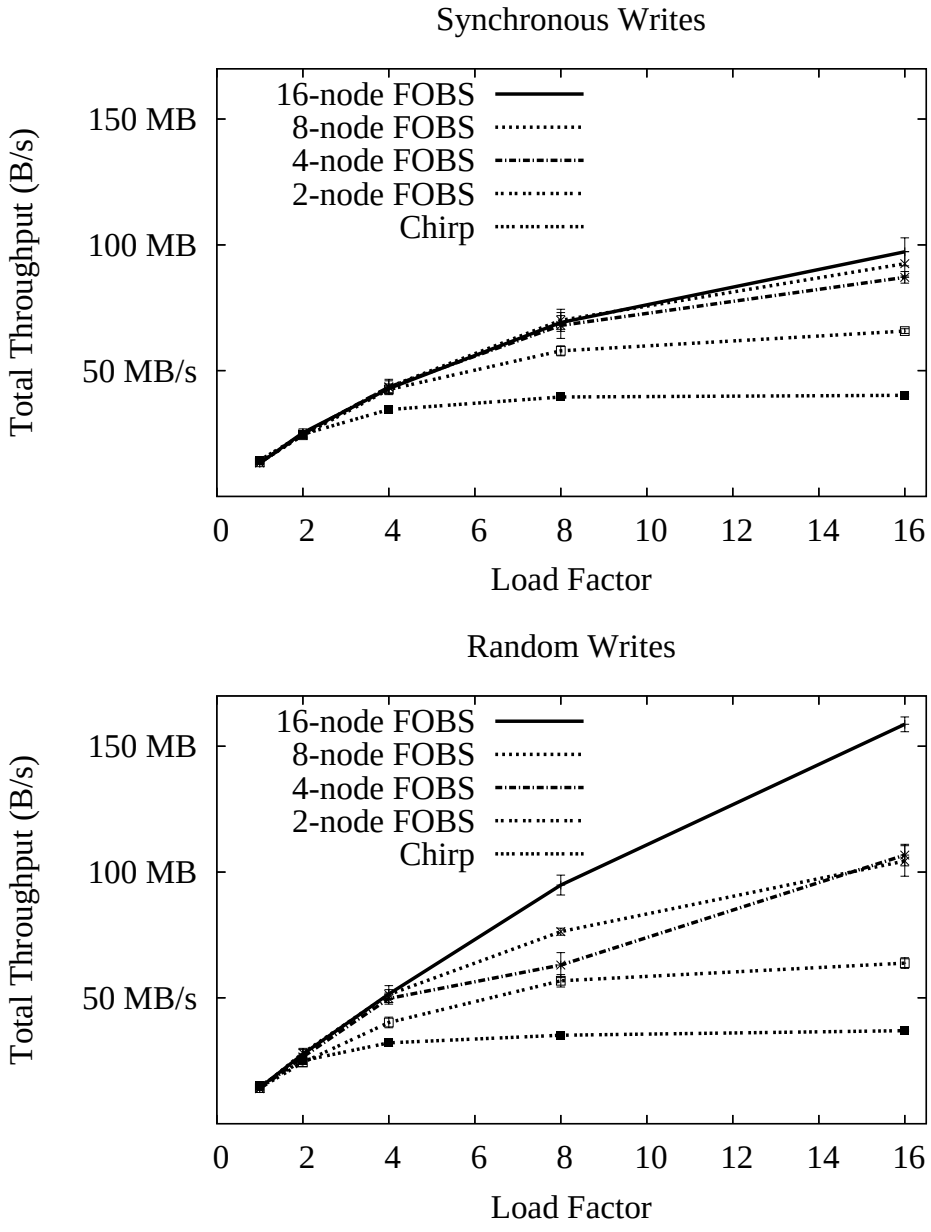

Figure 4.2: Aggregate Filesystem Write Bandwidth Under Load

to this is the 8-node FOBS for random writes, which suffered from a significant outlier (which explains the large error bar, as well) due to one disk's being picked randomly far more often than the expected 12.5% over the course of one trial.

The aggregate bandwidth is the maximum time for a client to complete the 1.6 GB data set, divided by the total number of bytes transferred (which is 1.6 GB times the load factor). Thus, the 16-node FOBS served 383 MB/s throughput for reads under a load factor of 16, and 159 MB/s for writes. Neither of these had leveled off, which supports both the hypothesis that a large FOBS system can deliver throughput much higher than single disk speed (it is within approximately a factor of 2 of disk speed *per client*), and that this performance is sustainable to many more users than could access a single server before hitting its throughput ceiling.

The FOBS setups do not achieve speedup proportional to the number of disks added, as suspected. However, the ordering of the throughputs for the heaviest load with the other factors controlled does indicate that the extra resources do allow the increased aggregate bandwidth, even if they do not permit linear bandwidth aggregation in terms of number of disks.

Comparing the workloads, it is apparent that the random access workload is more readily parallelizable, and thus provides higher aggregate throughput, confirming the hypothesis. The performance is close for writes, though the random trends, especially the 16-node FOBS have leveled off less than for the synchronous writes, suggesting that a larger load factor could show slightly more differentiation.

For the reads, even a 2-node FOBS system shows considerably better aggregate throughput on the random workload (leveling off above 130 MB/s, compared with less than 110 MB/s for the synchronous reads); and the difference is even greater

TABLE 4.4

METADATA SERVER CAPACITY UNDER LOAD

	1 Client	100 Clients
Total RPCs	31394	130048
RPCs/sec	3139	1300
RPCs/sec per client	3139	130

for the larger FOBS setups: 219 MB/s random versus 146 MB/s synchronous for the 4-node FOBS, 322 MB/s random versus 155 MB/s synchronous for the 8-node FOBS, and 382 MB/s random versus 175 MB/s synchronous for the 16-node FOBS.

From a single client, the metadata server can serve enough metadata pointer files to exceed even the most ambitious file loads. Under load, the metadata lookup is still not taxing enough to cause the server to become overloaded, as demonstrated by the ability to serve dozens to hundreds of metadata lookups per client per second even under a load factor of 100. This result is shown in full in Table 4.4, where each RPC is a getfile of a file of size similar to a metadata pointer file, and the clients continually requested the files for ten seconds. This performance confirms that crippling a commodity personal computer server with lookups is not a concern until the FOBS system is supporting thousands of users or users requiring hundreds of thousands of lookups per second (which seems beyond the scope of reasonable workloads in this environment).

For the alternative object storage situation; the version just using a TSS instead of a FOBS FS, the results in Figure 4.3 show that this system has a similar

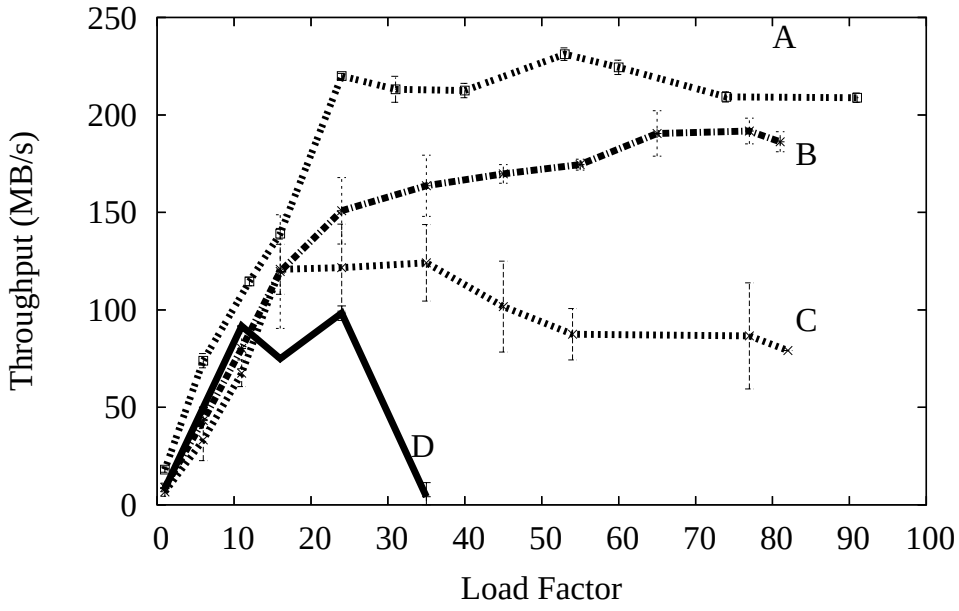

Figure 4.3. Aggregate Throughput using Replication

ability for harnessing aggregate read throughput that is greater than what is available on a single computer as the FOBS filesystem. In the figure, A is a set of 5 replica servers distributing 50 MB files with each client requesting the replica within its own cluster, B is a set of 5 replica servers distributing 50 MB files with each client making a random replica choice. C is a set of 5 replica servers distributing larger, 500 MB files using the random replica choice, and D is a single server distributing the larger files.

First, the inherent limitation of a single data server is evident: any single data server has limited resources, whether the limiting factor is storage, memory, or network link bandwidth. Further, multiple small servers are more cost effective, and prevent a single point of failure. The peak throughput for configuration D, in which a single server distributes 500 MB files, is lower than the otherwise identical

multiple server version (C), at just under 100 MB/s, and it can maintain this only up to approximately a load factor of 25, after which it drops off to less than 5 MB/s, compared with C's peak peak throughput of over 120MB/s around a load factor of 35 and sustained throughput of 80MB/s beyond a load factor of 80.

The distribution of smaller files, B, reaches a peak throughput above 190 MB/s, which doesn't occur until a load factor between 75 and 80. Utilizing cluster locality, A, the throughput of the 50 MB file transfers can peak beyond 230 MB/s. The graph reaches the point at which the curves of both replica choice algorithms have leveled off, but despite load factors of over 80, and over 90 in the case cluster locality, pushing the resources of the Notre Dame Chirp pool towards its limit, the data does not yet show the beginning of the falloff.

Although replication isn't necessary, as distribution of data among resources as in FOBS can deliver good performance, this alternate data delivery system does show the benefits that replication can give, at the cost of overhead for multiple copies, and complexity to keep them consistent. For popular data, especially scientific data sets, which are likely to be write-once/read-many, this is a good application of an alternate file-level object storage technique.

A full description of the problem and the solution architecture for this work can be found in previous work [19].

Note, however, that additional disks means additional risk of failure; memoization in writing can be used to get around this – it is harder for reads, since the objects are in a certain place, and can't be moved to a functioning host on-demand, since their host is, by definition of the problem, unavailable.

Additional disks added to a FOBS filesystem allow for greater aggregate bandwidth and larger storage capacity, but come at the cost of reliability. No re-

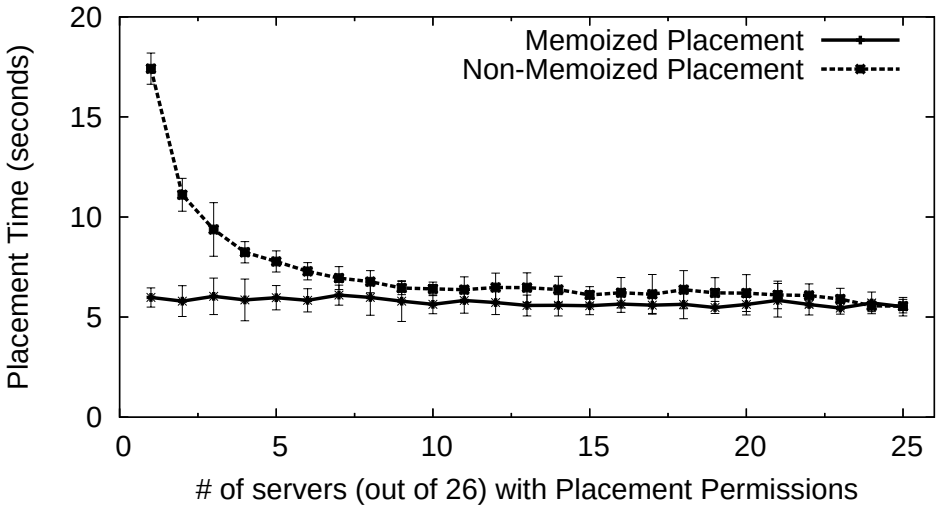

Figure 4.4. Memoization

source can be accessible 100% of the time, and thus, a FOBS system with any significant number of nodes will often have one unreachable. Further, because multiple namespaces can be interwoven, and data can be linked into a FOBS filesystem without taking ownership, it is very possible for security policies to collide and result in unavailable resources. Thus, FOBS must facilitate working in low-availability environments, which it does successfully; the results in Figure 4.4 indicate that memoization can give significant performance benefits. In a controlled test, memoizing away inaccessible disks cut the average time to place a file on the pathological case by more than half. On the other extreme, where all but one of the disks are accessible, the memoized algorithm achieved performance measures similar to the non-memoized algorithm (less than 5% overhead).

4.3 Single Client Sustainability

4.3.1 Problem and Observation

Another target user of a FOBS filesystem with something to gain is the single user who needs a transparent way to harness aggregate resources greater than those available on a local system. This would encourage power users to use FOBS to harness existing hardware resources instead of continually purchasing newer and better individual system hardware.

If a system has multiple disks, it can push data out (or pull data in) from multiple sources, thus allowing aggregation of "buffers" (disk caches, controller buffers, memories, etc.) beyond the limits of individual disks.

4.3.2 Hypothesis

Utilizing the several disks and associated hardware of underlying resources, FOBS users should be able to read and write at aggregate speeds greater than that of the local disk, potentially approaching the limit of the underlying network. This is particularly true for writes if generating data too fast for a local disk to keep up is a concern. Reading from or writing to several slower disks, a single user's sustainable performance over a large amount of data should exceed the sustainable throughput to a local disk, and should be more immune to drastic slowdowns or otherwise inconsistent transfer speeds due to filling buffers (at whichever level) and having to wait for the data in them to be consumed.

4.3.3 Results

A single user reading a large amount of data from a Chirp server, or writing a large amount of data to one, will naturally have a hard time doing so faster

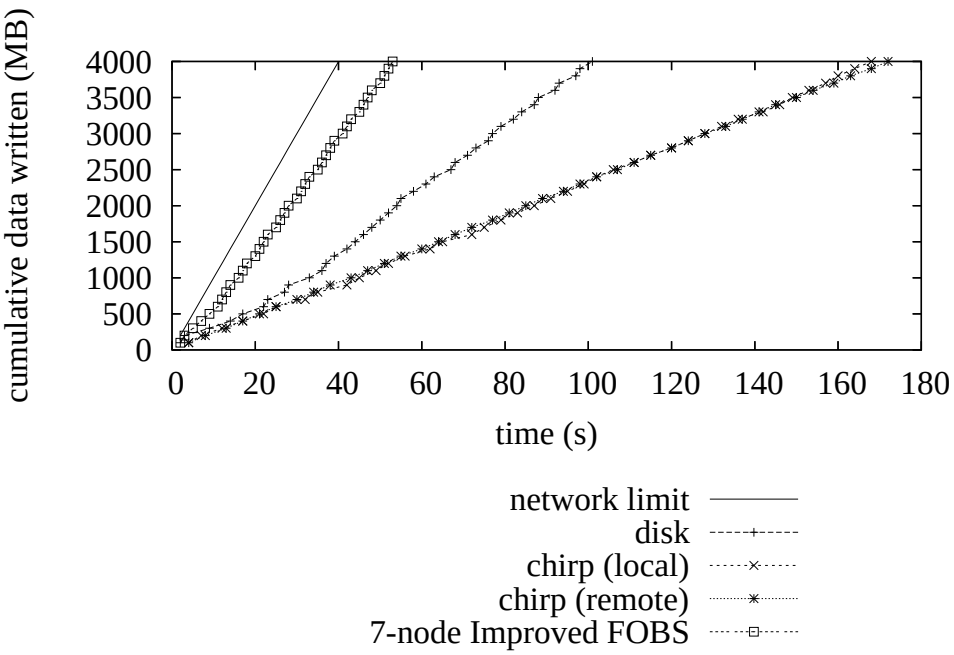

Figure 4.5. Single User Write Performance

than disk speed at the target disk. Additionally, large transfers overflow disk caches, and potentially even memories, and thus performance over time on these operations is not a smooth curve.

Even with multiple underlying resources, the baseline FOBS implementation does not solve this problem, as seen for reads in Figure 4.6 and for writes in Figure 4.5, in large part because for large transfers, many small read/write RPCs are required, and each one must acknowledge its completion before another can start.

However, with a modification to the implementation to use the getfile and putfile file stream interface, single client aggregate performance can improve beyond disk speed for both reads and writes. The read and write implementations used are slightly different, giving two extremes of moving towards the all-stream

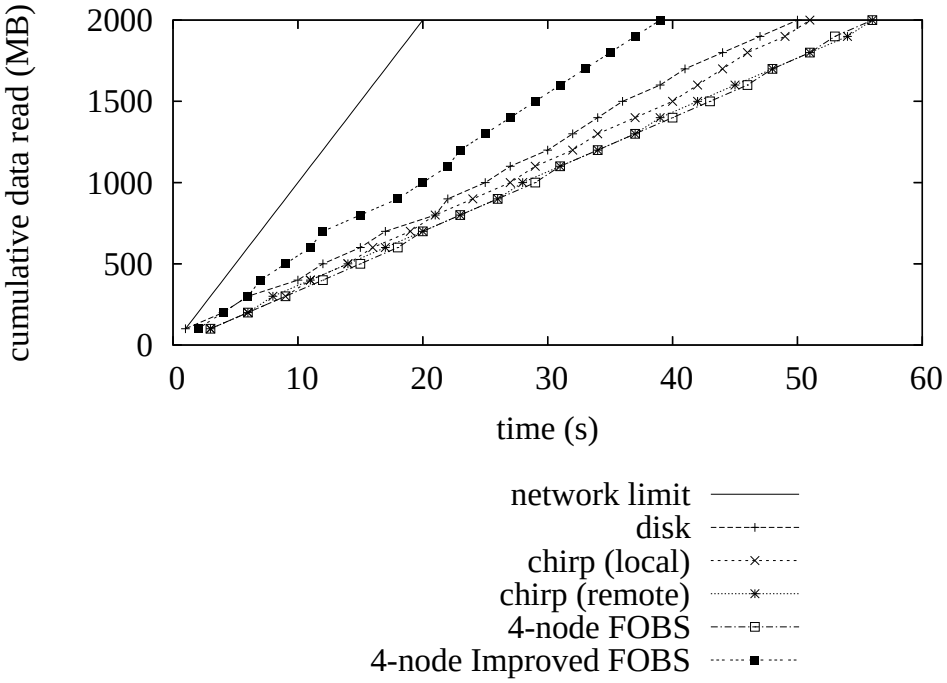

Figure 4.6. Single User Read Performance

transfer policy. The reads were generated in a script with a series of separate invocations of a standalone application that conducted the getfile RPCs from the 4-node configuration. The writes were optimized further, with the standalone application being run once to complete the series of RPCs to the 7-node configuration. While both exceed the base disk speed available, the write performance indeed approaches the network limit, as hypothesized.

It should be noted that an all-stream transfer policy can optimize performance in certain cases, but requires more resources of both the server and client, and thus may overload a system under a high load factor.

CHAPTER 5

FOBS EXPERIENCE IN HIGH ENERGY PHYSICS

5.1 Problem

5.1.1 Project GRAND

Cosmic ray astrophysics is the study of particles entering earth from beyond the extent of our solar system. Project GRAND (Gamma Ray Astrophysics at Notre Dame) maintains an extensive air shower array, studying two energy bands, one between 30 and 300 GeV, the other between 100 and 100,000 TeV. Project GRAND uses proportional wire chambers in cosmic ray research to observe particle showers for long-term charting and measuring atomic composition [23, 24]. In the detector array, each computer stores approximately 1000 records of particle charting in memory, which is eventually written out to an 8mm tape drive. The scientists note that "single muon data are stored at a rate of 2400 muons per sec; one 8mm tape holds 28 hours of data".

These tape drives are then transferred onto a commodity disk, which, once full, is placed into storage. Older archived data in the system was initially stored on tapes of varying specifications – each was a commercially available tape at the time of archiving – and a large tape archive still exists.

5.1.2 Latency

This system works well for data acquisition and archiving, however it makes analysis difficult, as trends may be significant anywhere from on an hourly basis to a decade-long basis. This requires a large number of resources to be accessible concurrently, which is difficult with limited hardware resources to serve the large numbers of tapes and disks.

Even if the hardware is available to have a multitude of media plugged in at once, the entirety of the archive cannot be accessible, and there is a high latency for getting access to other resources. The current system has this same limitation for both the newer (disk-stored) data and the older (tape-stored) data. Sending one's graduate student to find the disk or tape, mount it, and read the data from the it into the analysis tool is clearly inefficient.

5.1.3 Throughput and Aggregation of Resources

Latency is not the only issue, however. Another concern is throughput, especially as it relates to analyzing a data set in parallel. Even for smaller data sets, where a set of resources is sufficient to have access to all of the storage media concurrently, the data is stored on a small set of disks or tapes. Furthermore, data is stored chronologically – it was archived over time, changing disks or tapes only when they filled up. Thus, parallel access to a set of data with chronological locality stresses the same resource, which eventually hits its capacity for outbound throughput.

5.1.4 System Requirements

The scientists need a system to act as this front-end accessible storage that allows for storage of a large amount of data, for that data to be readily accessible under one namespace, for data distribution across multiple resources to be feasible, and for the system to have transparency of medium and location. The system should protect the actual back-end archives from overuse by allowing access to an online, front-end copy of the data instead. Most importantly, however, the system must be expandable to ever-increasing sizes as more historical data and newly acquired data are added to the system.

5.2 Solution Architecture

To satisfy these requirements without purchasing a large SAN or like-type system for the project, Project GRAND has used a FOBS filesystem served on commodity research PCs to store their scientific data and have it accessible close at hand over a campus network. For more than one year, GRAND has used their FOBS filesystem to give online access to newly acquired data, as well as older archived data of interest.

5.2.1 Data Management

Project GRAND collects data continuously, storing it in temporary storage within their scientific observation chambers ("huts"), and moving it onto a commodity PC's external USB hard drive. Once an hour, the machine packages the hour's data and makes it available on a FTP server.

A periodic job on a research machine (GRAND_UPLOAD) then copies all new files from the FTP server onto the research machine, then from the research

machine onto the FOBS system. This series of transfers averages between 250 and 500 kB/s throughput end-to-end.

This end-to-end throughput could be improved by running GRAND_UPLOAD on a storage cluster machine instead of a separate workstation, as this would take advantage of the gigabit switch of the cluster for the second transfer. Another option to improve system throughput is to utilize active storage. To do so, GRAND_UPLOAD would send a RPC to the destination underlying host to download the file from the FTP server, cutting the number of full-file transfers to one. These options have not been used with GRAND due to the sufficiency of the system in place, however if data transfer needs were to be increased (making the overhead of copying each file twice significant), implementing these options would be necessary.

Once uploaded into the FOBS filesystem, the data is immediately available for access through FOBS. Not only does this allow immediate access, it also allows the scientists to have accounting as to what files are available at any time. This is important for purposes of deciding when to retire the disks on their end to the archives, as they can ensure that all files on that disk are already in the online storage on FOBS.

5.2.2 Correction after Failure

If the research machine or its the network link fails, the uncopied data on the FTP server will accumulate. Because of this, the system is engineered so that only one instance of the GRAND_UPLOAD tool may run at once. This prevents the risk of data inconsistency due to multiple copies of the tool competing for files to upload onto the system.

The other concern is that, in this case, the system must be able to consume data (that is, upload it to the FOBS filesystem) faster than data is produced (data is posted on the FTP server by the scientists).

Consider, for instance, a power outage that occurs at 1700 on a Friday; it is quite possible that the data gathering system could be down until 0900 on Monday. This outage of 64 hours must be able to be overcome in the normal course of data collection, as human intervention must be minimized.

Table 5.1 shows the amount of data generated that must be added to the GRAND FOBS each hour, and the rate at which the research machine can complete this task. The consumption rate is based on a conservative estimate (below the mean transfer throughput of the Shower data over a one-month period, which is likely lower than what would be realized in a catch-up job due to amortized overhead) of the end-to-end throughput available from the GRAND machine into the FOBS system.

The time required to complete the files missed while the system was turned off is less than six hours. However, while the process of making up for lost time is doing this, more data is being added to the system which is not collected because it was not in the set of files to collect. Thus, after the bulk of the catch-up collection has completed, there remains what was missed during that interval. In this example, there are six hours missed, and thus the next automatic collection must collect seven files. This can be done in less than one hour, which then completes the job of making up for the downtime. Given this outage and these (reasonable) system parameters, the GRAND FOBS will have all of the recently collected data by 1600 on Monday, 71 hours after the outage began, and 7 hours after the outage ended.

TABLE 5.1

GRAND CATCHUP PERFORMANCE

	Muon	Shower
Average File Size	62.9 MB	18.8 MB
Production	62.9 MB/hr	18.8 MB/hr
Catch-up Consumption	250 kB/s combined	
64 hour Catch-up Time	< 7 hours	

For reasonable expectations that put an upper bound on the catch-up time to be the duration of the downtime, it is clear that the catch-up throughput after a system outage is sufficient to restore normal operations within that buffer. More constraining requirements could be met with this system (the example shows only about 10% of the original duration is taken for catch-up), however, eventually, the performance tweaks described above would have to be implemented to realize near-optimal recovery time.

5.3 Usage and Analysis

In this section there is a comparison between the status quo before FOBS was introduced to GRAND and the FOBS arrangement. These comparisons are made in reference to several target characteristics of the GRAND system, and lead to conclusions based on significant use of the filesystem in the context of a real scientific computing problem that it solves.

5.3.1 Latency

The typical use-case for the scientists is to examine several data files at once by feeding them through an analysis tool, which compiles summary data and event searching. Because the analysis runs online, a key system variable is the latency to receiving the first piece of data. Latency can be examined in two cases: one for already accessible data, one for archived data.

Status Quo

If the disk on which the pertinent data is stored is no longer hooked up to the GRAND machine, it must be found, connected, and mounted before data access can begin. For a tape, the process is similar. Conservatively, the "human interaction" required to do this requires a matter of minutes before even getting to the actual medium's latency.

If the pertinent data is recent, or otherwise available on the GRAND machine, the human aspect is minimized. There is still a latency consideration to make due to the different network connection downstream from the GRAND machine, as seen below, but it is not nearly to the scale of the human latency required for accessing archived data. The inferior connection will become more apparent in the next section, examining throughput.

FOBS

With FOBS, however, all the data files are stored online, so there is no consideration as to whether the data has been gathered or analyzed recently enough to be connected. Thus, the latency is cut from an expected case scenario of minutes for archived data down to a matter of tenths of a second when considering network and disk latency in both cases, a factor of two orders of magnitude. For data that

would still be available on the GRAND PC, the difference is not as drastic, but still as much as an order of magnitude when considering only the network latency.

Comparison

From a cluster in which the scientists often work, latency to the Project GRAND computer hosting the FTP server averages 2.6 ms; latency to the metadata root server for the GRAND FOBS averages .3 ms and was more reliable (less packet loss) than the GRAND host. Latency to an attached disk is another order of magnitude faster than that, averaging .04 ms, however accessing data only on a local device does not scale to large numbers of clients.

5.3.2 Throughput

While latency is an important consideration, especially when including the latency of human action, once the connection is established, throughput is another consideration. Computing resources, cycles, are wasted if they cannot be utilized due to unavailable data. Thus the scientists need a data delivery system that can maximize their ability to make use of their computing resources. This includes more than just simple bandwidth; scalability, efficiency, and flexibility are also factors.

Status Quo

There are two options for accessing data in a local environment: use the machine the disk is hooked up to, or transfer the data to the machine the user is working on. The first option does not scale to multiple users/clients, which means that parallelization of tasks is difficult (either for chopping one job up into smaller bits, or doing different tasks on the same data). Another limiting factor is that while

a data source is attached to a client machine, access is restricted to that data, as the user has physical control of the medium.

The alternative is transferring the data, which for newly acquired data means pulling downstream from the slower connection. For older data, it means pulling off whatever machine the disk is hooked up to, which could be acceptable if a powerful server with a good network connection hosts the archived data. This allows for some parallelism (because multiple people can access it), but all of those files are served by the same set of resources: disk, memory, network card, network link, and thus there is a limit in scalability for multiple users.

FOBS

With FOBS, however, the data are not physically stored in logical (chronological) order. Thus, two consecutive files of interest are likely to be on different servers, and thus several data files can be transferred in parallel up until the maximum of the incoming network link (as opposed to the maximum of the server's network link, on which there are no guarantees). Figures 4.1 and 4.3 shows the network scalability for multiple resources to be stronger than that of a single server. The multiple server design also allows for parallelization, for the same reason; multiple servers mean that multiple requests are less likely to collide, and thus can be served "full speed" for each of them, rather than rationing between them.

Finally, FOBS maintains the ability of the status quo to "use the data where it is". The Tactical Storage system allows for *a*ctive storage, which is invoked as an RPC to a client (in this case, the storage server) to do a computation. An advantage this gives over the status quo version of local computation is that the storage medium needn't actually be physically possessed by the caller, which allows for continued use by other users.

Comparison

For newly acquired data, the scientists can download the file from the FTP server at between 250 and 500 kB/s. From the same workstation, they can download the file from the FOBS filesystem at greater than 10 MB/s. This approaches the best that can be expected for the network, which is 100 Mbit Ethernet.

For older data, in which the scientists have already connected a USB hard drive containing the data to their PC, they can sustain a maximum of 60 MB/s, which is considerably faster than the same 10 MB/s. However, with the right application driving the downloads, as shown in Section 4.3, they could get data from the FOBS filesystem at faster than disk speed if the network bandwidth could handle that. Even without this, however, the FOBS is only a single order of magnitude slower, and is sufficient for all but the most data-intensive applications. Further, the FOBS filesystem allows access to this data from several clients at once, which could result in effective throughput of greater than the single disk's speed, since the network bottleneck is on the scientist's end, not the FOBS servers'.

5.3.3 Capacity and Expansion

Status Quo

The GRAND system is limited in capacity only by the the availability of the media on which to store data. Disks are plentiful and cheap, so this is not a dominating concern. As discussed above, there is a capacity limit in terms of number of devices attached to any single machine, but this relates more to performance of access than to raw capacity. Another possibility to increase capacity is to compress the data on the media, however this comes at the expense of performance, especially for

often-read data. Expansion in the system is only a matter of obtaining additional disks or tapes for archival storage.

FOBS

FOBS does not act as a replacement for archival storage within this particular application. Thus, simply using FOBS does not expand the capacity of the GRAND system. FOBS can increase the effective capacity of the GRAND archival storage by allowing the data to be compressed after it has been copied into the FOBS system. Doing this gets the advantage of uncompressed data for access, as well as compressed data for efficiently utilizing archival storage resources.

In this system, the additional disks used for online storage come at no additional cost, as the disks used are those from a computation cluster. While this is not always the case, the GRAND solution does give an example of the flexibility of FOBS to operate on dedicated storage servers or disk space scavenged from a pool of resources.

The cluster on which the GRAND FOBS filesystem resides has capacity for approximately 9700GB, which at the 2GB per day average consumption by GRAND, would allow storage of GRAND data spanning more than 13 years. However, the FOBS filesystem can be expanded to include more disks as necessary. A critical factor in this flexibility is that the new resources can be added from any resource pool (a new cluster, other scavenging resources, etc.) without requiring the current data to be migrated to the new location. This is necessary because it allows for aggregation of several available storage pools into one GRAND namespace, and thus a single cluster or pool needn't be able to serve the entirety of the GRAND FOBS filesystem data.

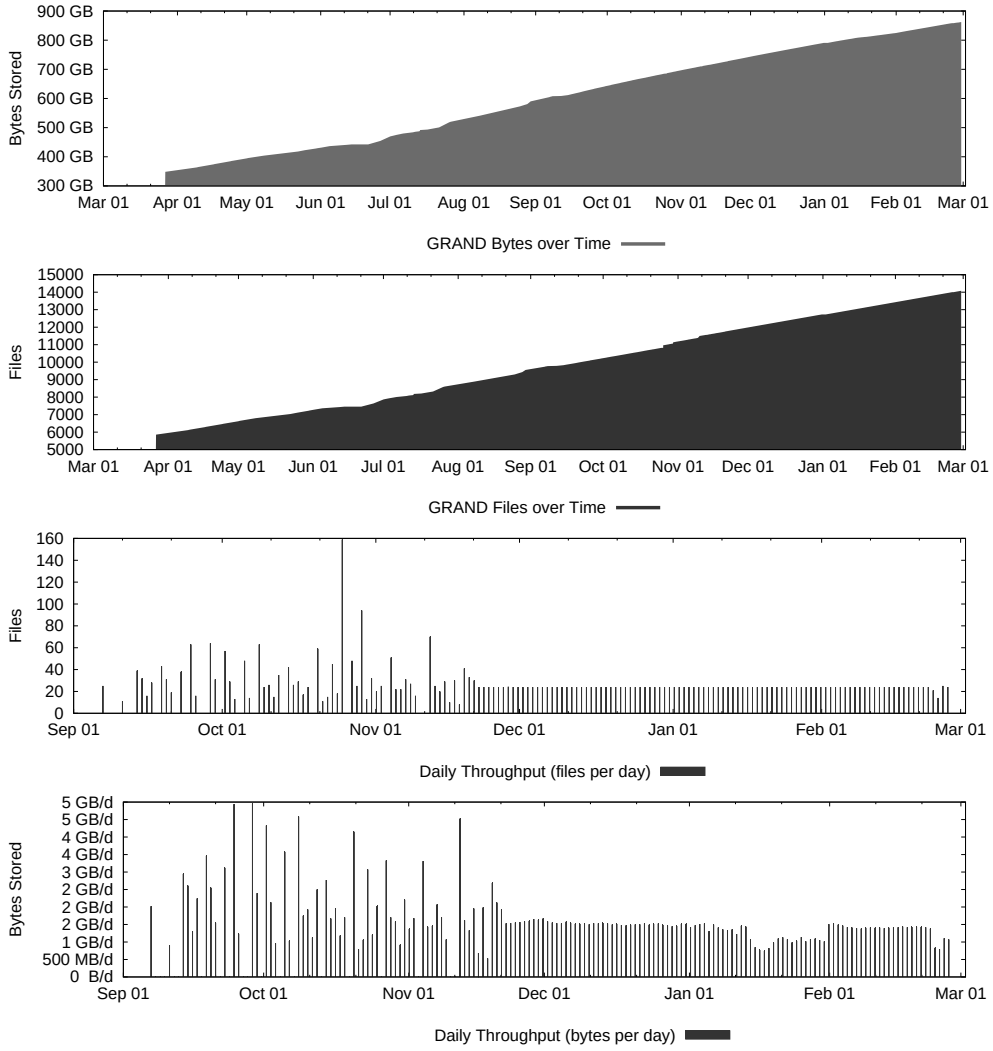

Figure 5.1: GRAND FOBS Usage for Muon Files

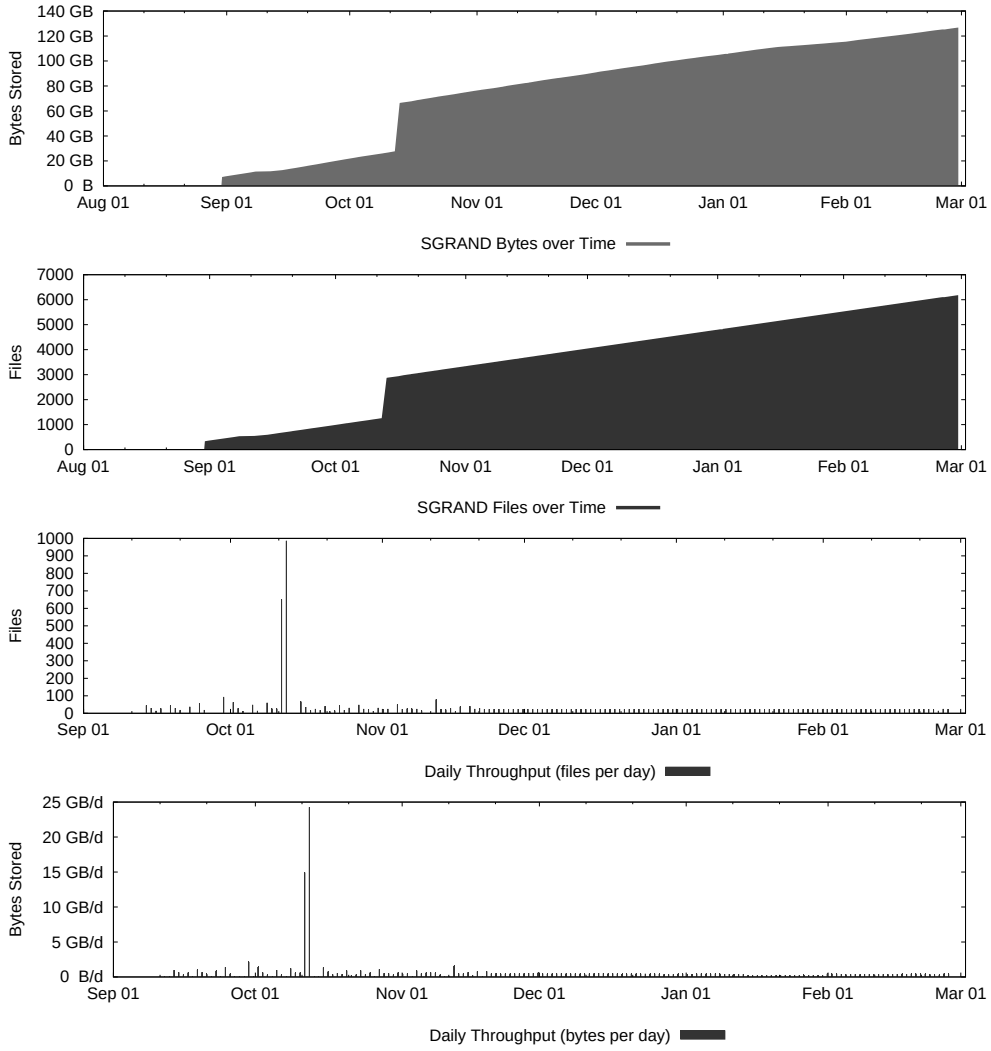

Figure 5.2: GRAND FOBS Usage for Shower Files

CHAPTER 6

CONCLUSIONS

In Chapter 1 of this work, a set of expectations and an evaluation methodology were set for the FOBS filesystem. Some things were clear by observation: FOBS can be used to build large filesystems, connecting several disks into one namespace; FOBS is deployable without administrator interaction due to its building-blocks in the Tactical Storage System; FOBS is expandable, with only a small metadata change facilitating addition of new storage resources. However, it is by taking a more thorough look at the characteristics of the filesystem that FOBS can be proved worthwhile as a general purpose filesystem, a scientific computing storage platform, and an application of object storage.

Flexibility for on-the-fly reconfiguration, to the extreme of building an ad-hoc filesystem out of a set of metadata pointing at already in-place files is an extension of the expandability.

Microbenchmarks indicated a possible concern that the cost of this flexibility was significant in terms of added latency in the system; it is for operations in which the dominant (or only) component is the metadata lookup, however, in the scope of common operations on real systems, FOBS performs within a small overhead (negligible to five percent) for most.

The major performance benefit from FOBS comes in the form of aggregable bandwidth for both single clients and multiple clients. By utilizing the back-end

resources that it contains, FOBS allows single users to sustain performance beyond the point at which a single disk cannot continue to increase, or even maintain, its total system throughput. These same back-end resources can be aggregated under high system-wide load; users may not better individual disk performance in this case, but the system sustains adequate per-client performance in the face of loads that would cause single-resource systems to overload, as seen in plot A in Figure 4.3.

The single-client sustainability using improved FOBS transfer strategy fits the case where Chirp alone could have achieved this result, but not with the simplicity that the FOBS system allows. A Chirp user would have to individually plan out the location of each file in order to get similar stream performance, whereas FOBS configured it autonomously by choosing round-robin after a random initial placement.

FOBS has been implemented as an online storage system for a group of physics researchers. They have been excited about the new-found accessibility, and have continued to use the system for both newly-collected and long-ago archived data. The management portion of this case study has shaped the development of the filesystem, and has spurred implementation of additional tools and features as well as a set of user-friendly guides, policy lists, and suggestions that will prove invaluable to future users. Finally, there is no better test to a system than non-developers using it for real work. The case study is the proof that there is a user-set for this system, and thus a case for continued development and expansion.

BIBLIOGRAPHY

1. A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

2. T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. In *ACM Symposium on Operating System Principles*, Dec 1995.

3. A. Azagury, V. Dreizin, M. Factor, E. Henis, D. Naor, N. Rinetzky, O. Rodeh, J. Satran, A. Tavory, and L. Yerushalmi. Towards an object store. In *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, 2003. (MSST 2003)*, pages 165–176, April 2003.

4. M. Beck, T. Moore, and J. Plank. An end-to-end approach to globally scalable network storage. In *ACM SIGCOMM*, Pittsburgh, Pennsylvania, August 2002.

5. P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur. PVFS: A parallel file system for linux clusters. In *Annual Linux Showcase and Conference*, 2000.

6. Cluster File Systems. Lustre: A scalable, high performance file system. white paper, November 2002.

7. M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran. Object storage: the future building block for storage systems. In *Local to Global Data Interoperability - Challenges and Technologies*, pages 119–123, June 2005.

8. G. Fedak, C. Germain, V. Nri, and F. Cappello. XtremWeb: A generic global computing system. In *IEEE Workshop on Cluster Computing and the Grid*, May 2001.

9. S. Ghemawat, H. Gobioff, and S. Leung. The Google filesystem. In *ACM Symposium on Operating Systems Principles*, 2003.

10. 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, Carnegie-Mellon University, 1996. URL `citeseer.nj.nec.com/gibson96case.html`.

11. H. Gobioff, D. Nagle, and G. Gibson. Integrity and performance in network attached storage. In *Proceedings of International Symposium on High Performance Computing*, 1999.

12. J. H. Hartman. *The Zebra Striped Network File System*. PhD thesis, University of California at Berkeley, 1994.

13. J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. In *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, 2001.

14. J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. *ACM Trans. on Comp. Sys.*, 6(1):51–81, February 1988.

15. J. Kistler and M. Satyanarayanan. *Operating Systems Review*, 23(5):213–225, December 1989.

16. S. M. Larson, C. Snow, M. Shirts, and V. Pande. Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology. In R. Grant, editor, *Computational Genomics*. Horizon Press, 2002.

17. E. Lee and C. Thekkath. Petal: Distributed virtual disks. In *Architectural Support for Programming Languages and Operating Systems*, 1996.

18. M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. *ACM Trans. Comput. Syst.*, 2(3):181–197, 1984. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/989.990.

19. C. Moretti, D. Thain, T. Faltemier, and P. Flynn. Challenges in Executing Data Intensive Biometric Workloads on a Desktop Grid. In *Proceedings of the Workshop on Large-Scale and Volatile Desktop Grids (PCGrid'07)*, Long Beach, CA (USA), March 2007.

20. S. Mullender, G. van Rossum, A. Tanenbaum, R. van Renesse, and H. van Staveren. *IEEE Computer*, 23(5):44–53, 1990.

21. D. A. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD international conference on management of data*, pages 109–116, June 1988.

22. J. Plank, M. Beck, W. Elwasif, T. Moore, M. Swany, and R. Wolski. The Internet Backplane Protocol: Storage in the network. In *Network Storage Symposium*, 1999.

23. J. Poirier, G. Canough, J. Gress, S. Mikocki, and T. Rettig. *Nuclear Physics B Proceedings Supplements*, 14:143–147, Mar. 1990. doi: 10.1016/0920-5632(90)90410-V.

24. J. Poirier, C. D'Andrea, M. Lopez del Puerto, E. Strahler, and J. Vermedahl. *ArXiv Astrophysics e-prints*, June 2003.

25. E. Riedel and G. Gibson. Active disks - remote execution for network-attached storage. Technical Report CMU-CS-97-198, Carnegie-Mellon University, 1997. URL `citeseer.nj.nec.com/riedel99active.html`.

26. O. Rodeh and A. Teperman. zfs - a scalable distributed file system using object disks. In *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, 2003. (MSST 2003)*, pages 207–218, April 2003.

27. R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *USENIX Summer Technical Conference*, pages 119–130, 1985.

28. F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *USENIX Conference on File and Storage Technologies (FAST)*, Jan 2002.

29. Seagate Research. The advantages of object-based storage – secure, scalable, dynamic storage devices. white paper, April 2005.

30. B. Seo and R. Zimmerman. *ACM Transactions on Storage*, 1(3):316–345, August 2005.

31. D. Thain and M. Livny. Parrot: Transparent user-level middleware for data-intensive computing. In *Workshop on Adaptive Grid Middleware*, New Orleans, September 2003.

32. D. Thain, S. Klous, J. Wozniak, P. Brenner, A. Striegel, and J. Izaguirre. Separating abstractions from resources in a tactical storage system. In *International Conference for High Performance Computing, Networking, and Storage (Supercomputing)*, November 2005.

33. D. Thain, C. Moretti, P. Madrid, P. Snowberger, and J. Hemmes. The Consequences of Decentralized Security in a Cooperative Storage System. In *Proceedings of the 3rd IEEE Security in Storage Workshop*, San Francisco, CA (USA), December 2005.

34. C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Symposium on Operating Systems Principles*, pages 224–237, 1997.

35. S. Vazhkudai, X. Ma, V. Freeh, J. Strickland, N. Tammineedi, and S. Scott. Freeloader:scavenging desktop storage resources for scientific data. In *International Conference for High Performance Computing and Communications (Supercomputing)*, Seattle, Washington, November 2005.

36. G. Woltman. The great internet mersenne prime search. Available from `http://www.mersenne.org`.

37. J. Wozniak, P. Brenner, D. Thain, A. Striegel, and J. Izaguirre. Generosity and gluttony in GEMS: Grid enabled molecular simulations. In *IEEE Symposium on High Performance Distributed Computing*, July 2005.

38. Q. Xin, E. L. Miller, and T. J. E. Schwarz. Evaluation of distributed recovery in large-scale storage systems. In *13th IEEE International Symposium on High Performance Distributed Computing (HPDC-13 '04)*, pages 172–181, 2004.

This document was prepared & typeset with LaTeX 2ε*, and formatted with* NDdiss2ε *classfile (v3.0[2005/07/27]) provided by Sameer Vijay.*