Separating Abstractions from Resources in a Tactical Storage System

Douglas Thain†, Sander Klous*, Justin Wozniak†,
Paul Brenner†, Aaron Striegel†, Jesus Izaguirre†

† - University of Notre Dame, Department of Computer Science and Engineering

* - National Institute for Nuclear and High Energy Physics, The Netherlands

ABSTRACT

Sharing data and storage space in a distributed system remains a difficult task for ordinary users, who are constrained to the fixed abstractions and resources provided by administrators. To remedy this situation, we introduce the concept of a tactical storage system (TSS) that separates storage abstractions from storage resources, leaving users free to create, reconfigure, and destroy abstractions as their needs change. In this paper, we describe how a TSS can provide a variety of filesystem and database abstractions for unmodified applications without requiring special privileges or kernel changes. A TSS provides performance competitive with NFS for single clients and also scales well for multiple servers and multiple clients. A prototype TSS of 120 disks and 6 TB of storage has been deployed at the University of Notre Dame and used for applications in high energy physics and bioinformatics.

1. INTRODUCTION

The user of a modern computational grid has access to an extraordinary array of hardware. Computing centers with hundreds or thousands of CPUs, each equipped with a private disk and a fast network, are commonplace. Despite this bounty of hardware, users are limited to primitive data sharing models. Almost universally, clusters are configured to use a distributed filesystem whereby all nodes share access to a small number of disks on the head node. If a cluster is part of a computational grid, the head node is usually the only place where data can be moved via file transfer. Once a job is running on a given cluster, gaining access to remote data is difficult or impossible.

As people gather to accomplish work in a cluster setting, the shared filesystem quickly becomes a policy constraint, a capacity limitation, or a performance bottleneck, leaving users unhappy and resources idle. An example of this problem is found in Grid3, a nationwide computational grid constructed to serve the production needs of seven scientific collaborations. Although Grid3 was able to harness several thousand CPUs for the benefit of several hundred users, nearly a third of all jobs submitted to it failed due to exhausted storage space, usually shared disks found on cluster head nodes [9]. This problem is not unique to Grid3: any regular user of a cluster has a similar story to tell.

When considered from a distance, this situation is puzzling. Each user is stuck using only the resources arranged by the administrator. Why can't a user use a local idle disk as a personal shared file system? Sharing across administrative domains is limited to manually moving data from head node to head node via a file transfer service. Why can't a user just access a remote archive directly from any cluster node? Although many people have access to several computing clusters, each is an island to itself. Why can't several clusters be used as one large logical filesystem?

We believe these problems are accidental rather than fundamental. Given the right tools, users should be able to create, reconfigure, and tear down storage abstractions without harming other users or involving an administrator. If users have access to raw storage and network resources, they ought to be able to build whatever abstractions they require in order to accomplish real work.

Thus, we propose the concept of a *tactical storage system* (TSS). In a TSS, administrators simply provide the raw storage and networking resources, leaving users free to construct the abstractions they require. Any node with a disk can serve as a storage device, whether it be a private workstation, a cluster node, or an archival system. From these resources, users can build up filesystems, databases, caches, or other structures as they see fit.

A tactical storage system draws a strong distinction between resources and abstractions. *Resources* are simply raw computing and storage capability wrapped in just enough software to allow it to be exported, controlled, and consumed. *Abstractions* are higher-level structures that organize resources into forms suitable for use by external users. This distinction leads us to a perennial issue in computer systems design: the selection of the proper interface to resources [16]. An interface that is too high level yields a simple but inflexible system, while an interface that is too low level yields a complex and unusable system. We argue that the Unix file system interface is suitable for *both* resources and abstractions. We call this approach *recursive storage abstraction* and it is a natural fit for a TSS.

In this paper, we present a prototype TSS currently de-

SC-05 November 12-18, 2005, Seattle, Washington, USA

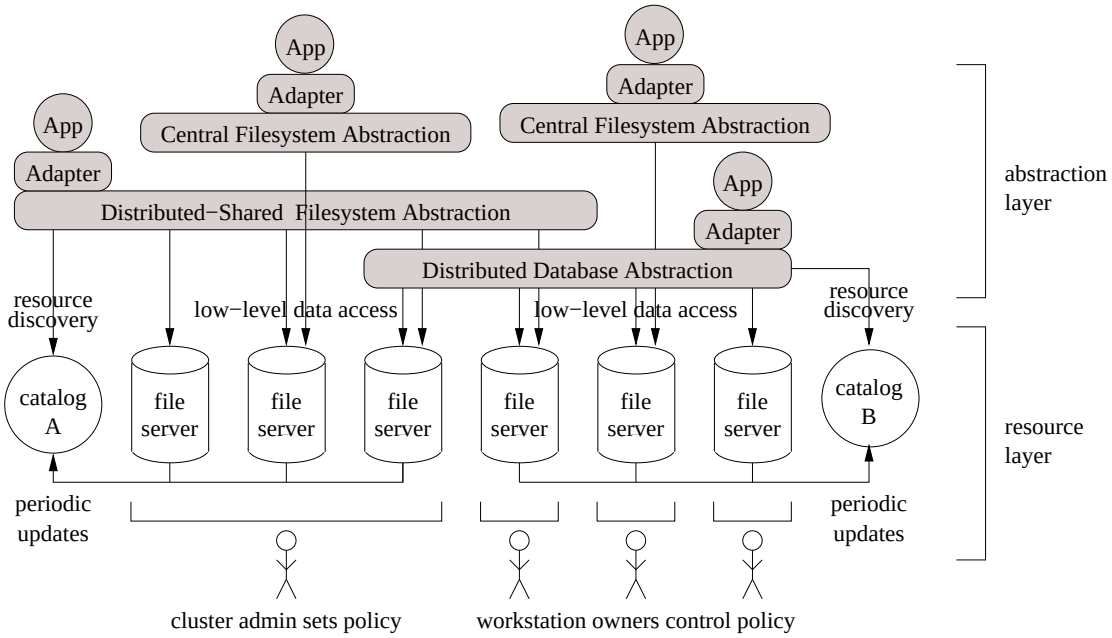


Figure 1: Overview of Tactical Storage

ployed at the University of Notre Dame and supporting a variety of research projects. The prototype currently consists of 120 devices and 6 TB of storage space. The basic storage unit is a personal file server with a flexible system for authentication and access control. The file server is designed to be rapidly deployed by ordinary users to any available storage space. Each file server reports to one or more catalogs, allowing users and tools to discover available storage at runtime. Clients communicate with file servers using a protocol that closely resembles the Unix I/O interfaces. Multiple file servers may be combined into a variety of abstractions, each recursively using and implementing the same Unix I/O interface. An adapter is used to transparently connect applications to abstractions by capturing and transforming system calls.

The primary value of a TSS is flexibility: any user may create a variety of abstractions on any storage device without special privileges. This flexibility comes at some cost: the TSS charges an overhead in latency and bandwidth that is necessarily higher than a kernel-level implementation. However, we will demonstrate that a user-level TSS has reasonable performance and is generally limited by hardware constraints, not by software design.

Finally, we discuss two different scientific applications that are making use of the TSS prototype. A high-energy physics simulation employs a TSS to securely access its home storage while deployed in a computational grid. A bioinformatics research group employs a TSS to share and preserve large amounts of simulation data. We conclude with a discussion of further applications of tactical storage.

The contribution of this work is a discussion of how systems must be structured in order to allow ordinary users to accomplish sophisticated work within the constraints of resource owners. Our experience with production computer systems is that users are constrained by the available *functionality*, not by the available *performance*. Thus, this paper does not explore the traditional distributed filesystem triad of performance, availability, and consistency. For these concerns, we choose the simplest available solutions. Rather, this paper presents a discussion about the *semantics* and *interfaces* necessary to make distributed systems serve the needs of real users.

A brief note on related publications. The technical details of the Parrot adapter are described in [32]. The Chirp protocol used by the file server was first mentioned in [31], but is first described in detail here. The SP5 high-energy physics application is described in [13]. The GEMS database is described in [33].

2. ARCHITECTURE

Figure 1 shows the architecture of a TSS. The *resource layer* provides storage to external users within policy constraints set by the owner. The *abstraction layer* organizes resources into structures useful to end users. The components are as follows:

File Servers. The basic resource is a file server which exports a Unix-like I/O interface to external users, who then use it to build up higher-level abstractions. A single file server might be used simultaneously by several users and abstractions. Each file server has an owner that controls who may use the server and how it may be used, using a flexible authentication and access control mechanism. The owner could be a site-wide administrator or the user of a workstation. A file server can be easily deployed to any machine without special privileges.

Catalogs. Each file server periodically reports itself to one or more catalogs, describing its current state, owner, access controls, and other details. The catalogs in turn publish an aggregate list of the file servers in a variety of data formats. Users and abstractions contact catalogs directly in order to discover new storage resources. A system may have multiple catalogs reporting on different servers.

Abstractions. Using one or more file servers, users may create abstractions that provide enhanced semantics, performance, or reliability. The simplest abstraction is the *central filesystem*, which allows a user to carve out shareable space on any device. More complex abstractions include a *distributed private filesystem*, which allows a single user to harness multiple servers; the *distributed shared filesystem*, which allows multiple users to share multiple devices, and the *distributed shared database*, which allows binary data to be indexed, replicated, and searched. Each abstraction allows independent users to organize storage for their own purposes without requiring special privileges.

Adapters. Abstractions are useless unless they can be easily connected to existing applications. Ideally, the operating system kernel would connect applications directly to abstractions. However, in a TSS, it is unlikely that users will be able to modify operating systems at will. They may be in an institutional environment where such permissions are impossible. Or, they may not have the technical means or motivation. Thus, a TSS provides an *adapter* that securely and transparently connects existing applications to abstractions without special privileges or code changes. This adapter is Parrot [32], a tool that operates by trapping system calls through the debugging interface.

3. PRINCIPLES AND RELATED WORK

A tactical storage system has several design principles that distinguish it from other types of distributed storage systems: independence, virtual users, recursive abstractions, failure coherence, direct access, and rapid deployment.

Independence. A TSS allows every participant in the system to maintain control of the resources that they own. One person may be willing to share storage with his organization, but not with the world at large. Another might wish to construct a large storage system, but only from resources provided by people that he/she knows and trusts personally. Thus, a TSS allows owners to specify exactly who they trust and to retract resources at any time if needed.

The principle of independence lies between two other design extremes. At one extreme is the global trust found in parallel storage systems such as Lustre [5] or Google-FS [10]. In this model, all devices collectively work for the common good and can be relied upon to store exactly what they are told. This only makes sense within one administrative domain. At the other extreme is the notion of global distrust found in peer to peer systems such as OceanStore [15] and FARSITE [1]. In this model, no single device can be relied upon, so expensive measures such as encryption and byzantine agreement are required for even the simplest activity.

A TSS lies between these two extremes. Ownership is made explicit so that users may choose resources with a level of trust appropriate for the task at hand. One might use borrowed resources for a cache, but only storage owned by trusted friends (or oneself) should be used for archival.

Virtual User Space. Because a TSS facilitates sharing across administrative boundaries, it must have a fully virtual user space, meaning that notions of identity must be technically and logically independent of the local user namespace. This is partially because the owner of a file server is not necessarily `root` and cannot create or delete local users. But even if the owner was `root`, the local user space might not be able to represent remote users identified by complex names such as network addresses or X.509 strings. Thus, each file server implements access controls on free-form subject names derived from external authentication systems.

In contrast, most systems export an existing user space. For example, NFS [29] assumes that all machines share a common user database. Grid systems such as GRAM [6] and GridFTP [2] make use of a distributed authentication system, but require a mapping from global to local usernames. This makes it easy to attach existing resources to a computational grid, but requires users to have local accounts on all systems. Large systems such as Grid3 have mitigated this problem somewhat by aggregating multiple users into shared local user names [9].

Recursive Storage Abstractions. A TSS uses the same interface at every layer from the file server all the way up to the user interface: a filesystem with the familiar interface of `open`, `read`, `rename`, and so forth. Recursive abstractions allow a TSS to easily interoperate with existing data sources: a file server can be used to export an existing filesystem without expensive copies or transformations. A recursive approach was first advocated for Unix United [28], but to our knowledge has not been employed elsewhere.

Most distributed storage systems choose to expose data at either a block-like interface or at an object-like interface. A block interface has been adopted by a number of parallel and distributed storage systems that provide data redundancy (RAID [25]) remote access (iSCSI [23]), or both (Petal [17]). However, a block interface is generally not suitable for sharing among multiple users, unless they have complete mutual trust (i.e. VAXcluster [14].) There are two reasons for this. First, filesystems require atomic access to complex data structures for metadata and directory trees. Without server support for these activities, a crashing or malicious client can leave the filesystem in an inconsistent state. Second, distributed systems require a large amount of policy to be attached to storage: files and directories must have owners, access control lists, and other data structures that would be impractical to attach to every block. Thus, a block interface is not the appropriate low-level interface for tactical storage.

A more recent approach has been to use "objects" as the low level storage interface [22, 26]. Roughly speaking, an object is a data blob with a unique name, a known size, and a small number of operations such as `read` and `write`. One may then build up a filesystem using an object to represent each file and directory, or by using objects as allocation units within files and directories. The benefit of an object interface is that it allows the storage device to assume responsibility for disk layout and space management without mandating a particular filesystem design. Further, objects are of sufficient size that it becomes reasonable to attach sharing policies to individual objects.

However, objects alone are not enough to build a filesystem. One must have a different sort of service in order to implement complex operations on metadata and directory trees. Thus, object-based systems require a second service, more like a database, for storing metadata and directories. This approach is used in systems such as Amoeba [24], Google [4], and Lustre [5]. Of course, just like a local filesystem, the directory data may be stored in the data objects themselves, but this still requires a distinct interface to perform operations atomically. One could imagine building a TSS using both object and database servers, but two types of

servers increases the administrative burden on all involved.

Thus, a TSS uses the Unix interface to serve as *both* object server and database server. We call this approach *recursive storage abstraction.* Ordinary files are sufficient to store data objects and the directory structure can be used as a (limited) tree-structured database with atomic operations for insertion, deletion, and renaming. By exporting a Unix interface, a TSS server can serve either role to any user, and may serve both roles to different users. Further, the owner of a storage device can examine its contents and gain some useful information about the users and applications to which it is being applied. Such examination is difficult or impossible with a block or object interface. A similar sort of argument is made in favor of the B-tree interface in Boxwood [18], which can be used to implement a wide variety of data structures beyond basic filesystems. The TSS interface is not as flexible as Boxwood, but it is sufficient for building filesystems and has the further advantage that existing filesystems may be exported without modification.

Failure Coherence. In a large enough storage system, hardware failures, system crashes, and network partitions will be a persistent condition. A tactical storage system is particularly vulnerable to failures. The set of resources may change as users add or remove storage devices. The components of an abstraction are not necessarily locked in the same machine room, but may be spread over a wide area network. Resource owners may forcibly delete data placed by other users in order to make room for their own needs. In such an environment, it is not reasonable to make the survival of an abstraction dependent on the presence or recovery of all devices at once. Thus, a TSS must have failure coherence: the loss of a component must leave the system in an operable, if degraded, state. In the context of a distributed file system, failure coherence means that the loss of a device may render some data inaccessible, but the directory structure must remain navigable and data stored on other devices must remain usable.

Direct Access. Each of the TSS abstractions read and write data directly to and from file servers without any intervening buffering or caching, much as in Amoeba [24]. There are several reasons for this.

First, we assume that TSS abstractions are highly dynamic and are created and destroyed on a rapid basis, perhaps even just to run a single remote process. This means that there are fewer opportunities to take advantage of buffering and caching, and a greater opportunity to lose data if abstractions are removed (or fail) before dirty data is written. We also expect that a TSS will be employed over fast networks, thus reducing the benefits of caching.

Second, our experience with grid users and applications is that there is a high degree of mistrust for transparent buffering and caching as found in existing distributed filesystems *even when correctly implemented.* Buffering and caching in a traditional filesystem must make some semantic sacrifice in order to improve performance: either delayed propagation (as in NFS [29]), an increased sharing granularity (as in AFS [11]), or in unusual semantics during failures (as in Coda [12]). A single unexpected sharing result due to aggressive caching causes chaos in connected systems and poisons user's perceptions.

Finally, if users do require data to be nearby for performance reasons, the TSS makes it easy to deploy a new file server in order to store and use a nearby copy.

Rapid Deployment. Traditional distributed file systems are presumed to be overseen by a professional administrator in a tightly controlled setting. Consequently, such systems are difficult to deploy: They require one to obtain privileges, install kernel modules, edit configuration files, generate keys and databases, perhaps reboot the system, and other labor-intensive activities. Previous user-level file systems [21] have improved upon this by moving the file server out of the kernel, but still require administrator privileges in order to bind clients to servers.

In contrast, a TSS is designed to be rapidly deployable. A basic file server can be deployed by an ordinary user, who runs a single command with no configuration, setup, or software installation. The deployed server is instantly and securely accessible by a variety of tools. This enables ordinary users to construct complex storage systems, even as parasites in existing systems. For example, a large storage pool suitable for caching or data staging can be constructed by simply submitting a set of file file servers into a distributed batch system. This is similar to the concept of *gliding in* [8].

Related Systems. From a high level, a TSS bears a similarity to three other complete systems: the Logistical Networking Stack, the Self-Certifying File System, and the Storage Resource Broker.

The Logistical Networking Stack (LNS) [26] is a distributed storage system based on the Internet Backplane Protocol. Both TSS and LNS build up complex distributed storage abstractions on top of simple, low level servers. However, the two systems diverge on several fundamental design decisions. For example, the basic abstraction in LNS is a malloc-like interface. This simplifies allocation, accounting, and revocation of storage, but makes it difficult to store a filesystem structure without adding another layer of indirection. LNS enforces access control via capabilities rather than access control lists. This allows a wider variety of sharing models, but re-introduces the old problem of storing and protecting capabilities. However, capabilities in LNS do offer the advantage of simplifying directory data transfer between two hosts, directed by a third party. LNS provides a more general abstraction, while TSS is specialized toward filesystems.

The Self-Certifying File System (SFS) [20] allows users to securely deploy and access NFS-based file systems without relying on an existing authentication infrastructure. To this end, SFS encodes authentication keys into the names of both servers and users. Superuser privileges are needed to deploy both an SFS file server and to mount such a server at the client side. TSS is similar to SFS in the sense that both encourage the use of "personal" file services. However, TSS differs in three significant ways: it relies upon existing authentication systems for user identification, allows the deployment and use of storage without special privileges, and allows for the construction of higher-level abstractions.

The Storage Resource Broker (SRB) [3] allows multiple server-class data services – including filesystems, archival systems, and databases – to be federated into a system accessible from a single client. A distinct metadata service indexes data on multiple service types and locations, allowing clients to identify and access data by semantically meaningful names. The result is an integrated view of multiple data types, regardless of the storage technique or location. TSS is similar in the sense that it is able to federate multiple servers, but is focused on on the problems of rapid deployment and transparent access.

4. RESOURCE LAYER

The basic resource of a TSS is a file server. Our prototype employs the Chirp personal file server. The Chirp protocol was initially designed for use by Condor [31] and Parrot [32]. It has since evolved but retains backwards compatibility. Each file server exports a Unix-like protocol over TCP to all interested parties. A flexible authentication and access control scheme allows for controlled sharing between multiple parties across administrative domains. Each file server remains under the control of its owner, who is free to admit (or even evict) data and users as they see fit.

The Chirp protocol works as follows. A client connects to a file server via TCP and authenticates itself by one of several methods described below. The client may then issue remote procedure calls that correspond closely to Unix. For example, here is a fragment of the Chirp RPC interface:

```
conn = chirp_connect( host, port, timeout );

chirp_open   ( conn, path, flags, mode, timeout );
chirp_pread  ( conn, fd, data, length, off, timeout );
chirp_pwrite ( conn, fd, data, length, off, timeout );
chirp_close  ( conn, fd, timeout );

chirp_stat   ( conn, path, statbuf, timeout );
chirp_unlink ( conn, path, timeout );
chirp_rename ( conn, path, newpath, timeout );
```

As the interface suggests, the client must establish a connection to a server before issuing I/O calls. The `open` call returns a file descriptor that may be used to issue `pread` and `pwrite` calls of arbitrary size on the file. `pread` and `pwrite` require an explicit offset, so the client is responsible for maintaining state such as the current file descriptor position Operations on the name space such as `unlink` and `rename` only require filenames. In addition to the Unix-like calls, two additional RPCs `getfile` and `putfile` allow for entire files to be streamed over TCP. All file data is carried over the same connection as is used for control. This allows the underlying TCP connection to reach and maintain the maximum needed window size. In contrast, protocols such as FTP [27] separate data and control, resulting in multiple TCP slow starts when multiple files must be transmitted.

From the server's perspective, failure semantics are simple. If the client and server become disconnected, then the server frees all resources associated with that connection, particularly, all currently open files are closed. This means that a file descriptor returned by `open` is only valid for the duration of the connection. Clients must take responsibility for recovering from disconnection by re-opening any needed files. The semantics of failure recovery are controlled by the adapter so that they may vary from user to user. Thus, recovery is discussed below in section 6.

Files and directories are stored without transformation in an ordinary filesystem on the host machine. The owner may select any directory as the root of the file server, allowing any user to export fresh space or existing data. For security, it may be desirable to run the server in a `chroot` environment to prevent escape from the server root. Because `chroot` is only available to the `root` user, the server provides an equivalent facility in software.

In order for the file server to allow sharing across administrative boundaries, it must provide a fully virtual user space. That is, a file server cannot simply use the integer identities stored in a conventional file system: they may be inconsistent from machine to machine, even among cooperating users. Thus, each server manages free-form text identities independently of the local user database. This requires some care in both authentication and access control.

Each file server provides several authentication methods. The simple `hostname` method allows a client to simply be identified as the domain name of the connecting host. The `unix` method relies on a challenge and response within the local filesystem: the server challenges the client to touch a file in `/tmp` and then infers the client's identity from the response. The `globus` method allows a client to authenticate via the Globus Grid Security Infrastructure (GSI) [7]. A file server may hold either user or host GSI credentials. The `kerberos` method uses the Kerberos [30] system, but this requires it to run as `root` in order to access the host key.

When connecting to a server, a client may attempt any number of authentication methods in any order. If any method succeeds, the client is given a subject name of the form `method:name`. One user might be able to authenticate by several methods, but only one set of credentials may be employed in one session. This may be occasionally inconvenient, but simplifies troubleshooting and file ownership.

A file server enforces access control lists (ACLs) on each directory in which data is stored. The form of ACLs should be familiar to most readers. Each entry consists of a subject name and a list of rights: R to read files, W to write or create files, L to list the directory, and A to modify the ACL. For example, in order to all machines in domain `*`.cse.nd.edu and all users with GSI credentials at Notre Dame to read, write, and list the contents of a file server, the root ACL would be:

```
hostname:*.cse.nd.edu          rwl
globus:/O=Notre_Dame/*         rwl
```

However well-meaning, the ACL above is not likely to be useful. Even if the owner intends to give access to such a wide group of users, they will certainly not wish to share the namespace, allowing one user's files to be read by another. Typically, visiting users will require a fresh namespace and the ability to adjust the ACL in order to permit access to their collaborators. For this purpose, an ACL may also include the *reserve right* (V). Consider this ACL:

```
hostname:*.cse.nd.edu          v(rwl)
globus:/O=Notre_Dame/*         v(rwla)
```

When a user performs a `mkdir` in a directory in which he/she holds the V right, the newly-created directory is initialized with an ACL giving only the calling user the rights specified in the parent directory. Suppose that the above ACL is present in the root directory when a user `hostname:laptop.cse.nd.edu` calls `mkdir(/backup)`. The ACL in `/backup` would be:

```
hostname:laptop.cse.nd.edu     rwl
```

This allows `hostname: laptop.cse.nd.edu` to read, write, and list data in the new directory. However, note that the user was not given the administration (A) right, because it was omitted from the parent ACL. This prevents the user from giving access to others. However, users presenting credentials matching the wildcard would be given the (A) right, allowing them to extend access to others.

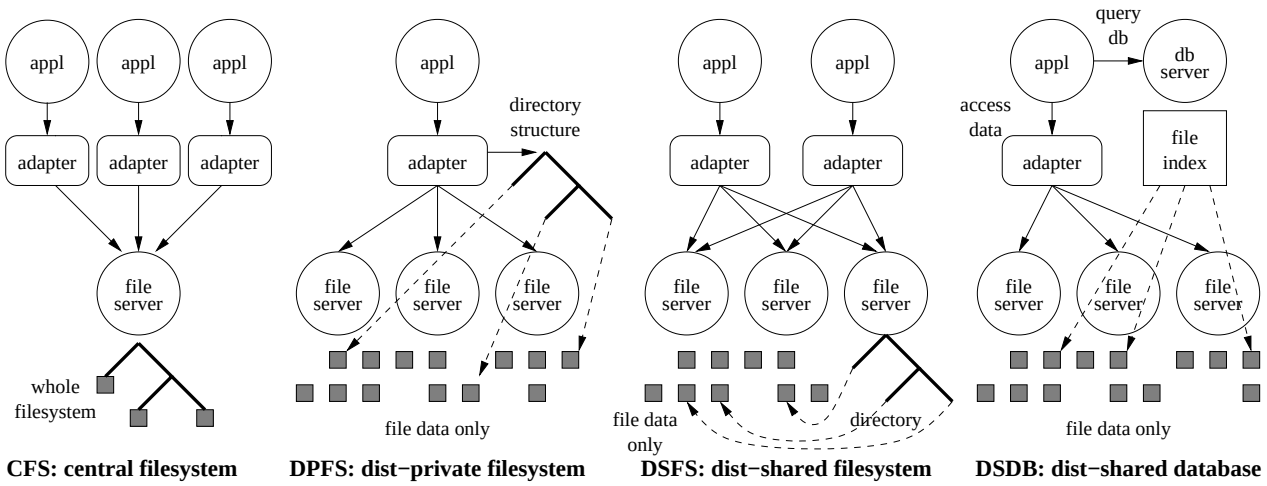


Figure 2: Recursive Storage Abstractions

These combined techniques of authentication, access control, and reservation allow for secure sharing of storage between cooperating users. A file server may be established for private purposes, for sharing within a well-defined community, or for sharing with the world at large. In all models, authorized visiting users may create private workspaces.

Where necessary, the owner of a file server may evict users or data by simply deleting files. No special support is needed for this beyond the usual ACLs. The owner of a file server retains access to all data on that server and is free to delete it according to any policy, perhaps deleting files according to owner, activity time, or file size. The right to delete (but not modify) files can be given to others by granting the D right.

To assist with the discovery, measurement, and administration of multiple file servers, each periodically reports itself to one or more catalog servers. Each periodic report describes the owner, status, available space, top-level ACL, and other vital data about the server. The catalog in turn publishes the set of available storage devices for users and other software components to view in a variety of formats. If a server does not report to a catalog after a configurable timeout, it is removed from the listing.

A TSS may include several catalog servers, each collecting reports from a different, possibly overlapping subset of the available storage devices. Obviously, redundant catalogs can be used for fault tolerance and load sharing. However, multiple catalogs may also be useful for reasons of policy and organization. For example, a user creating a TSS by submitting storage servers to a grid might establish a separate catalog server as a rendezvous for these transient servers.

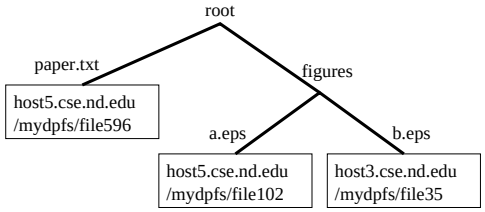
All data in a catalog is necessarily stale. Any matter reported by a file server, whether storage space or access controls, may have changed between a query to the catalog and a query to a file server. Thus, abstractions that rely on the catalog for discovery must be prepared to revisit any assumptions made using the catalog state.

5. ABSTRACTION LAYER

Using the resource layer, ordinary users may construct a wide variety of different abstractions. Figure 2 compares four abstractions: the centralized filesystem (CFS), the distributed private filesystem (DPFS), the distributed shared filesystem (DSFS), and the distributed shared database (DSDB). Each provides a different tradeoff in sharing, performance, and scalability. One may easily imagine more sophisticated abstractions built on a TSS.

CFS - The simplest abstraction is the centralized file system (CFS). In this abstraction, the user simply accesses files and directories on a single file server without translation. Of course, a user must create a private directory using the reserve (V) right described above. All Unix operations are carried from the application to the file server, where consistency and synchronization are managed in the usual way within the host kernel. CFS is roughly analogous to NFS, except that it provides grid security and Unix-like consistency by dispensing with buffering and caching. A typical application of CFS is to provide secure remote access to existing filesystem data for jobs in a grid computing system.

DPFS - With a CFS, the user gets the performance of an untranslated filesystem, but is limited to the storage capacity of the single device on which the data is stored. Using a distributed-private file system (DPFS), a user can employ the aggregate storage of multiple file servers in one image. In a DPFS, the file servers are used only to store file data. The directory structure is stored in a local Unix filesystem chosen by the user. Where the directory structure indicates a file, it instead contains a stub file pointing to the file data elsewhere, as shown:



In this example, file `/paper.txt` is stored in file `/mydpfs/file596` on `host5`, while `/figures/b.eps` is in `/mydpfs/file35` on `host3`.

From this structure, the implementation of a DPFS is straightforward. To create a new filesystem, one must specify a list of hosts, create a new directory root, and create

new storage directories on each server. To create a new file, a new stub entry must be created, a remote server must be chosen, and the new file created. Once opened, a file is accessed directly on the file server without reference to the directory structure. Name-only operations like `mkdir` and `rename` modify the directory tree without contacting a file server.

The DPFS allows a user to create large file systems spread across multiple machines without requiring special privileges at either end. However, it does not allow multiple users to share the same filesystem, because the metadata is contained in a filesystem private to one user. To allow sharing, the metadata must also be stored in the resource layer. This approach is taken by the next abstraction:

DSFS - The distributed shared filesystem (DSFS) is created by moving the directory tree onto a file server. Now, multiple clients may access the directory tree and follow pointers to file data on multiple servers. A single file server might be dedicated for use as a DSFS directory, or it might serve double duty as both directory and file server.

Recall from above that a TSS does not cache data, so there are no problems of cache-coherency. However, there are some synchronization issues common to DPFS and DSFS. To avoid losing track of data due to a failure, file creation is performed in the following order. 1 - A file server is chosen and a unique data file name is generated from the client's IP address, current time, and a random number. 2 - A stub entry is created in the directory tree. 3 - The corresponding data file is actually created. Steps 2 and 3 are performed using the "exclusive open" feature of the Unix interface (yet another benefit of recursive abstractions.) so that in the event of a name collision between two processes, file creation can be aborted. If a crash occurs between steps 2 and 3, the file system is left with a stub name but no data file. (This is better than the alternative: a data file but no stub.) An attempt to open such a file yields "file not found". Such a dangling stub file deviates slightly from traditional Unix semantics, but prevents unreferenced garbage, and is easily deleted by a user. Likewise, deletion is performed by removing the data file, then the stub file.

Note how DSFS differs from other filesystems that follow the object-and-directory model. Instead of designating one type of server for file data and another type of server as a directory database, a TSS allows any server to act in either role. A Unix-like filesystem already has the capability to manage file data as plain files and to manage metadata within a filesystem structure. It is not necessary to construct a new type of interface.

DSDB - Scientific data is often better served by a database than by a filesystem. Researchers in many fields generate large amounts of binary data – such as simulation outputs – that must be indexed, sorted, searched, and viewed in many different ways. For this purpose, we have constructed a distributed shared database. The DSDB is similar to the DSFS, except that a database server is used to store file metadata as well as pointers to files. A user queries the database to yield the names of matching files, and then accesses them directly with the adapter.

The DSDB allows for the complex forms of sharing needed to support a scientific community. One research group may establish a file server allowing all of its members to read and write data, while allowing external users only to read. Another might support writing only by explicitly selected principal investigators from multiple institutions. Several file servers might set aside directories with the reserve (V) bit, allowing remote collaborators to share space for replicating data between groups. As new equipment is purchased, file servers may be added and used without bringing the system down.

Each of these abstractions are *failure coherent*. That is, the loss of a storage device yields a partial, usable, system. This is a vital property in a system with multiple devices that is highly likely to experience a failure of some kind. If any single file server containing file data is lost temporarily, then only those files on that server will become unavailable. If file data is lost permanently, then files located on those servers will be lost, but the directory structure (or database) remains navigable.

Of course, the directory (or database) server is still a single point of failure, and should be placed on a sufficiently reliable server. If the directory (or database) server is lost, then the filesystem is no longer navigable. However, the remaining portions of the filesystem are stored in distinguishable directories on each of the file servers, allowing for either manual recovery or complete removal. In the DSDB, the database could even be recovered automatically by rescanning the existing file data.

We conclude by noting that these abstractions are not presented as the final word in the design of filesystems and databases. One can imagine a wide array of variations and improvements upon the abstractions, such as striping each file's data across multiple disks, replicating files for fault tolerance, or distributing directory structure for improved metadata performance. Rather, we have argued that the TSS architecture empowers users to create a variety of useful abstractions without encountering administrative and technical barriers.

6. ADAPTERS

Finally, the various abstractions must be attached to the applications that wish to use them. Ideally, the operating system would perform the technical binding of filesystem operations to the necessary code through a device driver or kernel extension. However, as many have observed, modifying the local operating system (or even requiring special privileges) is simply not a practical requirement in a grid computing system that is spread across multiple institutions. Neither is it acceptable to require existing applications to be re-written to access new storage systems.

Thus, a TSS requires *adapters* that can transparently connect unmodified to the various abstractions. For this purpose, we provide an adapter called Parrot. This adapter connects to an application through the debugging interface and instructs the kernel to intercept all of its system calls. As each call is attempted, the application is halted, and the adapter provides a new implementation. In this manner, the adapter may provide any service that would normally be implemented in the operating system. An earlier paper describes Parrot in detail [32].

By default, the adapter presents each abstraction as a new top-level entry in the directory hierarchy with the second-level name identifying a host or volume. For example, a file server on host `shared.cse.nd.edu` can be accessed under `/cfs/shared.cse.nd.edu`. In addition, an application can be given a "mountlist" that creates a private namespace by mapping logical names to external abstractions. For exam-

ple, this mountlist would cause an application to perceive a CFS abstraction under `/usr/local` and a DSFS under `/data`.

```
/usr/local   /cfs/shared.cse.nd.edu/software
/data        /dsfs/archive.cse.nd.edu@run5/data
```

Like any distributed filesystems, a slight variation from Unix semantics is necessary in order to support recovery from disconnection. If the TCP connection to a server is lost, the server closes the client's opened files. The adapter responds by attempting to reconnect to the server with an exponentially increasing delay. (Users may place an upper limit on these retries with a command-line argument.) If the connection is re-established, then the adapter re-opens files for the user, hiding any change in the underlying file descriptor. In addition, it uses `stat` to verify that the file has the same inode number as before. If it does not, then the file was renamed or deleted between the first open and the disconnection. In this case, the client receives a "stale file handle" error as in NFS.

As stated above, the adapter performs no buffering or caching before sending data to a file server: it sends all operations to server in the order that they are issued. However, we leave unspecified whether the server must flush newly-written to disk before returning success. Synchronous writes are a well-known and necessary technique for ensuring reliability of data across failures. Nevertheless, users often choose performance over reliability. For example, the 30-second lag in the traditional Unix buffer cache is widely considered to be an acceptable risk. Thus, the adapter allows users to choose synchronous or asynchronous writes via a command line switch. Synchronous writes are easily implemented by simply transparently appending the `O_SYNC` flag to all `open` calls. (Another benefit of using recursive abstractions.) In the following performance evaluation, we show asynchronous writes in order to evaluate maximum performance.

7. PERFORMANCE

In this section, we demonstrate that a TSS can drive a storage and networking system to its hardware limits, despite the user-level implementation of both client and server. We also compare TSS against NFS because NFS is the technology most accessible to end users that would otherwise be used for "tactical" purposes, modulo the administrative difficulties described above. Although there certainly exist many other distributed filesystems, none are simple enough to deploy that they would be considered for such purposes. We also demonstrate that a TSS can provide performance scaling with the number of servers and clients.

Our goal in this section is to evaluate the overhead of the user-level components. Most other filesystems implement both client and server at the kernel level, thus avoiding the extra latency and data copy of a user/kernel context switch. To this end, we will demonstrate that the TSS has a measurable overhead at the level of individual system calls, but can still be used to drive the storage and network systems to maximum capacity.

Note that our goal is *not* to evaluate the value of caching and buffering. We have argued that caching and buffering introduce unacceptable semantics, regardless of the performance benefit. Others have made the opposite argument. Thus, comparing TSS (with no caching) against NFS (with caching) would yield no new insight. Instead, we provide a comparison of TSS (with no caching) against NFS (with no caching) in order to evaluate the overhead of protocol and implementation while holding other variables constant.

We begin by examining the overhead charged by the system call trapping mechanism in the adapter. This mechanism increases the latency of system calls due to multiple context switches between the kernel, adapter, and application. It also requires an extra data copy between the kernel and the application. Figure 3 compares the latency of unmodified Unix system calls against the same calls made within Parrot. Each measurement was generated by 1000 cycles of 100,000 iterations of each call on a 2.8 GHz Pentium 4 running Linux 2.4.21. Most system calls are slowed by an order of magnitude.

However, compare this to the cost of network I/O operations shown in Figure 4. We measure the same set of system calls in the same manner using three different filesystem abstractions over a commodity 1Gb/s Ethernet. We compare a CFS accessed via Parrot to an NFS accessed via the usual kernel method and a DSFS via Parrot. We have turned off caching and synchronous writes in NFS in order to create an apples-to-apples comparison.

Note that the latency of these operations (including the in-kernel NFS) outweigh the latency of Parrot itself by another order of magnitude. Next, consider that Parrot+CFS has performance quite comparable to that of Unix+NFS. CFS has lower latency for stat and open/close, because it does not require lookup operations to resolve names to inodes. CFS has lower latency for the 8KB write because only one round trip is needed to retrieve the data. As expected, DSFS has identical performance to CFS for reads and writes, but has twice the latency for metadata operations because it must access the stub file as well as the data file.

The bandwidth available to a single client is shown in Figure 5. To measure this, we copy 16 MB of data to each target system using a varying block size, defined as the size of each individual read or write system call. The average and standard deviation of ten measurements is shown. The Unix case shows the upper bound of 798 MB/s, constrained by local system performance. The same copy through Parrot peaks at 431 MB/s, due to the extra data copy. Again, even this reduced bandwidth is far above the available network bandwidth of 128 MB/s (1 Gb/s), of which Parrot+CFS is able to use 80 MB/s. Finally, Unix+NFS is only able to obtain 10 MB/s due to the request-response nature of the protocol. Again, NFS is run in asynchronous mode; the low bandwidth is due to the protocol, not due to the target disk. DSFS (not shown) has the same behavior as CFS for a single client. Similar results are obtained for reading data.

Next, we show that DSFS is able to provide scalable performance for multiple clients and multiple servers. Server scalability is achieved in two ways. Multiple servers increase the total memory used as buffer cache on each file server. Multiple servers also increase the aggregate bandwidth between clients and storage. To explore scalability, we establish a DSFS on a cluster where each node has a 250 GB SATA disk, 512 MB RAM, and a full-duplex gigabit Ethernet connection to a commodity switch. We vary the number of server nodes and the amount of data stored in the filesystem. Load is generated by running clients on other cluster nodes that select large files randomly and read

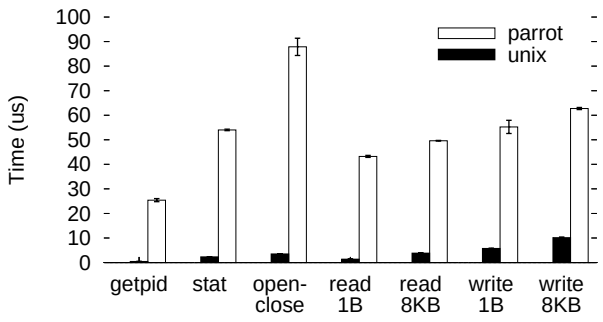


Figure 3: System Call Latency
The overhead charged on individual system calls by the Parrot adapter. Most calls are slowed by an order of magnitude. However, Figure 4 shows that this cost is overwhelmed by network latency.

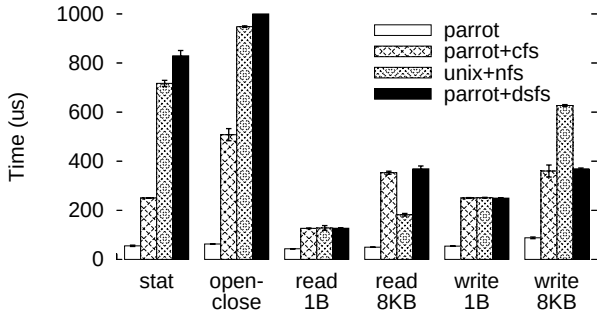


Figure 4: I/O Call Latency
The latency of single I/O calls. Note that Parrot-based CFS generally has lower latency than kernel-based NFS. DSFS has slower stat and open calls because stub file lookups require multiple round trips.

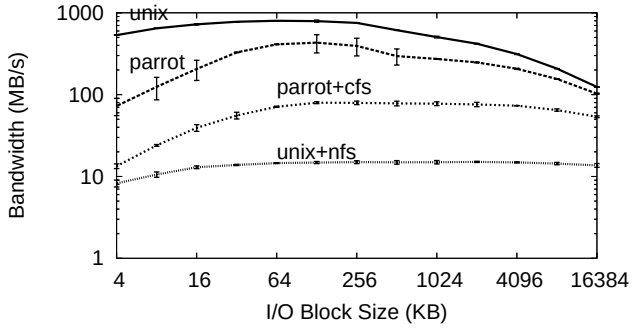


Figure 5: Single Client Write Bandwidth
The maximum bandwidth achieved writing 16MB in various block sizes. Parrot+CFS achieves higher bandwidth than Unix+NFS because it uses variable sized messages over TCP instead of 4KB RPC packets.

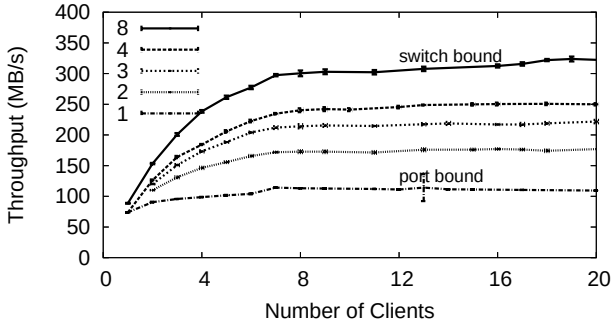


Figure 6: DSFS Scalability: Net-Bound
The scalability of a DSFS serving 128 MB from 1-8 servers on a 1 Gb/s switch. One server saturates one port at just over 100 MB/s. Three or more servers begin to experience queuing effects and saturate the switch backplane at 300 MB/s.

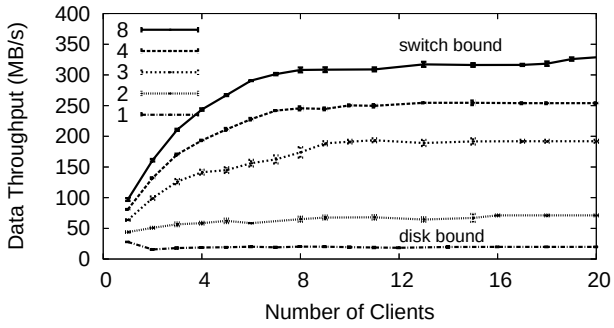


Figure 7: DSFS Scalability: Mixed-Bound
The scalability of a DSFS serving 1280 MB from 1-8 servers on a 1 Gb/s switch. With less than three servers, the system is disk-bound. With more servers, all data fits in memory, and the system is bound only by the switch.

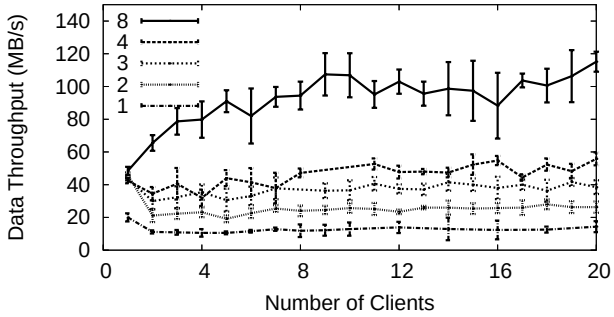


Figure 8: DSFS Scalability: Disk-Bound
The scalability of a DSFS serving 12800 MB from 1-8 servers on a 1 Gb/s switch. In all configurations, this system is disk bound. Note that throughput increases roughly linearly with the number of servers.

them out of the filesystem. We use three configurations to show that a DSFS is constrained only by hardware capacity.

Figure 6 shows a network-bound system. 128 files of 1 MB are stored in a DSFS with 1 to 8 servers. In all configurations, all data fits in the server buffer caches. One server can transmit at 100 MB/s, near the practical limit of TCP on a 1Gb port. Multiple servers increase the total bandwidth, but are soon limited by the backplane of the inexpensive commodity switch.

Figure 7 shows a mixed-bound system. 1280 files of 1 MB are stored in a DSFS with 1 to 8 servers. With one or two servers, not all data fits in the server buffer caches, and the system runs at disk speeds. With three or more, the system is constrained only by the switch backplane.

Figure 8 shows a disk bound system. 1280 files of 10 MB are stored in a DSFS with 1 to 8 servers. In all configurations, there is not enough buffer cache to keep the data in memory. A single server is able to sustain 10 MB/s, the raw disk throughput. As servers are added, the throughput increases linearly.

8. APPLICATION TO HIGH ENERGY PHYSICS

SP5 is a software component of the BaBar high-energy physics experiment in progress at the Stanford Linear Accelerator Center by an international collaboration of researchers. A large amount of simulation is needed to understand the response of the detector apparatus. The computing needs of the collaboration exceed the resources available at any one of its constituent research labs and universities. However, all bound together into a computational grid should provide sufficient computing power.

Like many scientific applications, SP5 has a number of complexities that make it difficult to deploy in a distributed system. It is not a single static executable, but a collection of scripts, executables, and dynamic libraries. The configuration and output data used by SP5 are stored using a commercial I/O library whose data are protected by a lock server running on the same host. Although the data are designed to be accessed simultaneously using a shared filesystem, it is very difficult to deploy SP5 on a grid. Users cannot install distributed file system clients on the nodes of a grid, nor can SP5 be easily deployed on new nodes.

Using a TSS, we are able to deploy SP5 into a computational grid without changing any of the application code, the installation structure, or the data organization. To achieve this, the SP5 executable is submitted to a computational grid along with the adapter and appropriate GSI credentials. As SP5 runs, the adapter contacts a CFS to load dynamic libraries and scripts and provide access to the data. Using the virtual user space of the file server, access controls are set so that only grid users with the appropriate credentials may access the data.

The TSS is competitive with the performance of NFS, as the following table shows. Each run of SP5 requires a certain amount of initialization time, and then processes a variable number of simulation events, typically thousands. We then compare four different configurations that the application may be run in. 1 - SP5 running unmodified using data on a local filesystem. 2 - SP5 running unmodified using data on an NFS filesystem over a 100 Mbps Ethernet. 3 - SP5 running using a TSS in the same configuration as LAN/NFS. 4 - SP5 running on a computational grid, accessing data over a wide area link of (roughly) 100 Mbps capacity. Note that there is no WAN/NFS case because this configuration is both socially and technically impossible to create.

	configuration	**init time**	**time/event**
1	Unix	446 ± 46 s	64 s
2	LAN / NFS	4464 ± 172 s	113 s
3	LAN / TSS	4505 ± 155 s	113 s
4	WAN / TSS	6275 ± 330 s	88 s

(Table reproduced from an earlier paper [13].)

As can be seen, the time to initialize SP5 increases by an order of magnitude no matter what the connection method. However, once initialized, simulation events (typically thousands) can be processed within a factor of two performance. If a TSS allows the user to harness more CPUs than he/she would be able to otherwise, then overall throughput can be increased. (Note that the WAN/TSS case processes single events faster than LAN/TSS due to a slightly faster processor. Heterogeneity is a fact of life in a grid.)

9. APPLICATION TO BIOINFORMATICS

The field of bioinformatics is experiencing a data management crisis. A single user of a simulation tool such as PROTOMOL [19] can easily generate so many simulation outputs that a database is needed to simply keep track of the work accomplished. Multiple researchers engaged in the same area of study wish to index and share results with each other in order to prevent duplication of effort. Results of high value must be replicated across multiple physical devices in order to provide performance locality fault tolerance. A TSS is a natural structure for supporting the needs of this community.

To support bioinformatics, we have deployed a TSS at the University of Notre Dame. The current system consists of 120 file servers with 6 TB of total storage on a variety of systems including private workstations, classroom workstations, and research clusters. On this TSS, we have deployed a distributed shared database named GEMS: Grid Enabled Molecular Simulations. GEMS is designed to track and store outputs from the PROTOMOL simulation toolkit. Like the generic DSDB shown above, GEMS stores files on file servers and indexes them with a database. In addition, GEMS dynamic replicates files in order to assure survival.

Two active components work in concert to maintain replicas. An *auditor* process periodically scans the database and then verifies the location and integrity of data on file servers. If it discovers that files have been damaged or removed, it makes note of these problems. A *replicator* process examines the notations and then repairs them by re-replicating the remaining copies.

Figure 9 shows an example of this preservation activity. A modest data set of 14 GB is entered into GEMS for safekeeping. The user specifies that up to 40 GB of space may be used to store this dataset. Once a single copy of the data is accepted, the replicator process then works to replicate the data until the storage limit has been reached. At three points during the life of this run, three failures are induced by forcibly deleting data from one, five, and ten disks. As the auditor process discovers the losses, the replicator brings the system back into a desired state. (This figure is reproduced from an earlier paper [33].)

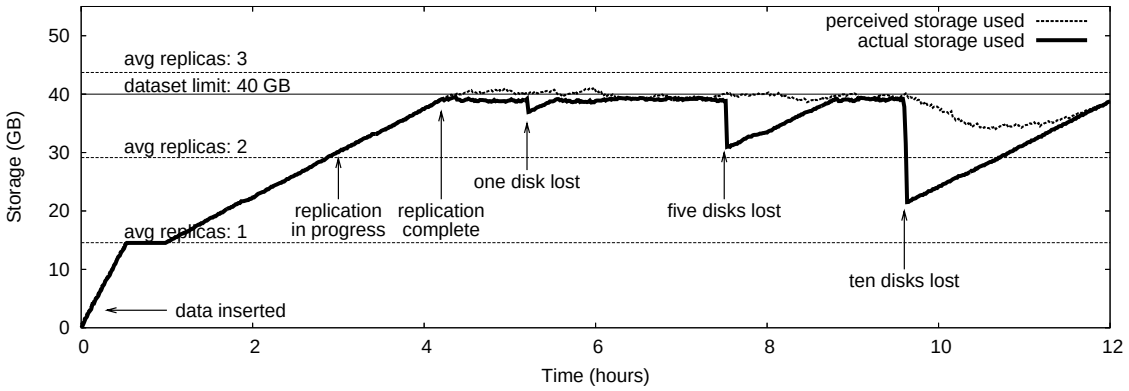


Figure 9: Data Preservation in the GEMS Distributed Shared Database

10. CONCLUSION

We have presented the concept of a tactical storage system, which empowers users to create storage structures on fly as needed within an existing distributed computing system. This is accomplished through two novel insights:

Separation of abstractions from resources. End users and system administrators have two distinct roles to play. Systems should provide each with the necessary tools to accomplish their roles, but no more. Users need to accomplish real work by consuming resources, perhaps simultaneously from different providers. Administrators are responsible for establishing services and controlling resource consumption by various users. A TSS respects both roles while giving each group technical autonomy. Users can create and destroy abstractions as much as they like. Administrators are not bothered or even notified of such abstractions, except to the extent that they represent resource consumption. However, in order to provide this clean separation, there must be a simple, consistent interface between resources and abstractions, which is provided by:

Recursive abstractions. Most distributed storage systems are built upon two distinct services: one for storing data, and another for organizing directories. A TSS takes advantage of the observation that a Unix filesystem already has implemented *both* types of service, and can thus be used in either role. Further, the combination of both roles simplifies the separation of abstractions from resources. The administrator doesn't (and shouldn't) care to what purpose a user is employing a file server, except to the extent that local security and resource policies must be enforced.

Thus, the two notions are mutually supporting: separation of abstractions from resources encourages recursive abstractions, and vice versa.

To demonstrate these concepts, we have presented a variety of abstractions and shown two applications that make use of them. Of course, many more variations are possible within a TSS. One may imagine filesystems that transparently stripe, replicate, and version data. A TSS is a natural platform for distributed backups, allowing cooperating users to easily record many backup images, thus allowing for on-line perusal, recovery, and forensic analysis of data over time. The flexible policies present at each storage device will certainly lead to new modes of interaction.

These issues we leave open for future work.

11. REFERENCES

[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] W. Allcock, A. Chervenak, I. Foster, C. Kesselman, and S. Tuecke. Protocols and services for distributed data-intensive science. In *Proceedings of Advanced Computing and Analysis Techniques in Physics Research*, pages 161–163, 2000.

[3] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC storage resource broker. In *Proceedings of CASCON*, Toronto, Canada, 1998.

[4] S. Brin and L. Page. The anatomy of a large scale hypertextual search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

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

[6] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. Resource management architecture for metacomputing systems. In *IPPS/SPDP Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, 1998.

[7] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *ACM Conference on Computer and Communications Security Conference*, 1998.

[8] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-G: A computation management agent for multi-institutional grids. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing*, pages 7–9, San Francisco, California, August 2001.

[9] R. Gardner and et al. The Grid2003 production grid: Principles and practice. In *IEEE Symposium on High Performance Distributed Computing*, 2004.

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

[11] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West.

Scale and performance in a distributed file system. *ACM Trans. on Comp. Sys.*, 6(1):51–81, February 1988.

[12] J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *Operating Systems Review*, 23(5):213–225, December 1989.

[13] S. Klous, J. Frey, S.-C. Son, D. Thain, A. Roy, M. Livny, and J. van den Brand. Transparent access to grid resources for user software. *Concurrency and Computation: Practice and Experience*, to appear.

[14] N. Kronenberg, H. Levy, and W. Strecker. VAXcluster: A closely-coupled distributed system. *Transactions on Computer Systems*, 4:130–146, May 1986.

[15] J. Kubiatowicz, D. Bindel, P. Eaton, Y. Chen, D. Geels, R. Gummadi, S. Rhea, W. Weimer, C. Wells, H. Weatherspoon, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Architectural Support for Programming Languages and Operating Systems*, 2000.

[16] B. W. Lampson. Hints for computer system design. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, volume 17, pages 33–48, 1983.

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

[18] J. MacCormick, N. Murphy, M. Najork, C. Thekkath, and L. Zhou. Boxwood: Abstractions as a foundation for storage infrastructure. In *Operating System Design and Implementation*, 2004.

[19] T. Matthey and J. Izaguirre. ProtoMol: A molecular dynamic framework with incremental parallelization. In *SIAM Conference on Parallel Processing for Scientific Computing*, March 2001.

[20] D. Mazieres. *Self-Certifying File System*. PhD thesis, MIT, May 2000.

[21] D. Mazieres. A toolkit for user-level file systems. In *USENIX Annual Technical Conference*, June 2001.

[22] M. Mesnier, G. Ganger, and E. Riedel. Object based storage. *IEEE Communications*, 41(8), August 2003.

[23] K. Z. Meth and J. Satran. Design of the iSCSI protocol. In *IEEE/NASA Goddard Conference on Mass Storage Systems and Technologies*, April 2003.

[24] S. Mullender, G. van Rossum, A. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: A distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53, 1990.

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

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

[27] J. Postel. FTP: File transfer protocol specification. Internet Engineering Task Force Request for Comments (RFC) 765, June 1980.

[28] B. Randell. Recursively structured distributed computing systems. In *Symposium on Reliable Distributed Computing Systems*, pages 3–11, 1983.

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

[30] J. Steiner, C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the USENIX Winter Technical Conference*, pages 191–200, 1988.

[31] D. Thain and M. Livny. Error scope on a computational grid. In *Proceedings of the Eleventh IEEE Symposium on High Performance Distributed Computing*, July 2002.

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

[33] 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.