

Chirp: An Architecture for Cooperative Storage

Douglas Thain

University of Notre Dame
 Department of Computer Science and Engineering
 Technical Report #2005-02

Abstract

Despite an embarrassment of riches in computing hardware, sharing data and storage space in a distributed system remains a difficult task for the ordinary user. In a conventional distributed system, users are restricted to the structures provided by the system administrator. To remedy this, we present the Chirp architecture for cooperative storage. This architecture allows users to construct complex storage systems using multiple devices without the help of system administrators. Because Chirp bridges multiple administrative domains, it provides a fully virtual user namespace that obviates the need for a shared user database. A unique 'reserve' access right allows visiting users to create private, secure, and shareable storage spaces. We demonstrate how the principle of recursive storage abstraction allows for a variety of storage structures, each providing a different tradeoff in shareability, fault tolerance, and performance.

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 a file transfer service.

As people gather to accomplish work in a cluster setting, the shared filesystem becomes a policy constraint, a capacity limitation, or a performance bot-

tleneck, 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. [6] This problem is not unique to Grid3: any regular user of a cluster has a similar story to tell.

When viewed from an distance, this situation is puzzling. Each person is stuck using only the resources configured by the administrator. Why can't a person simply make use of an idle local 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 the combined storage of 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 complex storage systems without recourse to an administrator. Thus, we propose the concept of a *cooperative storage system*. In such a system, administrators simply provide the raw storage resources, leaving users free to construct the abstractions that they require. Any node equipped with a disk can serve as a storage device, whether it be in a private workstation, an unused cluster node, or an archival system. From these resources, users can build up file systems, databases, caches, or other data structures as they see fit. Artificial boundaries should not constrain what users can build.

In this paper, we present Chirp, an architec-

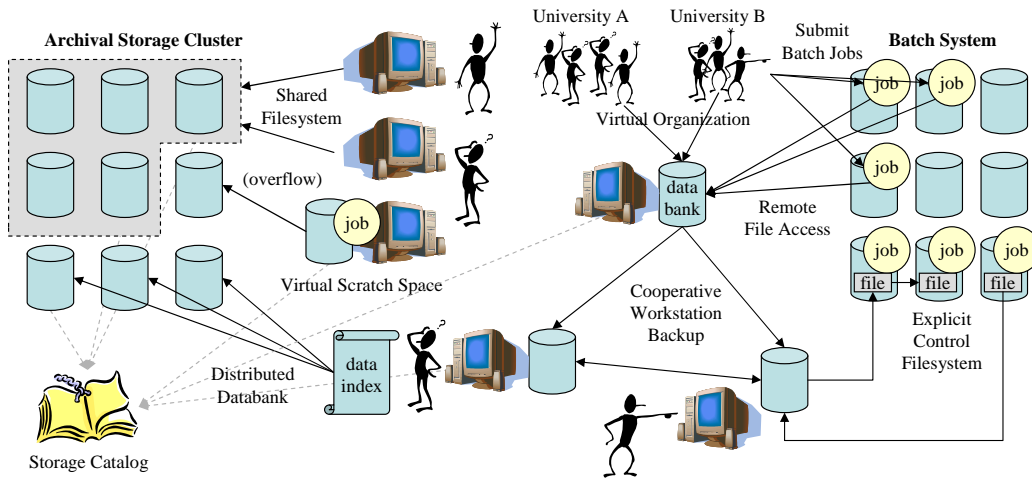


Figure 1. Overview of Cooperative Storage

ture for cooperative storage systems. The fundamental unit of Chirp is a personal file server with rich namespace for user identification, access control, and policy enforcement. The file server is designed to be rapidly deployed by an ordinary user to any available storage space. Each file server reports to one or more catalogs, allowing users and tools to discover available storage at runtime. Tools communicate with file servers using a lightweight I/O protocol. A series of recursive abstractions allows multiple file servers to be combined into a variety of complex structures. An adaptation layer connects ordinary applications to a cooperative storage system by transforming standard system calls into remote operations.

Chirp is more than just a user-level filesystem. Because it binds together users and resources from different administrative domains, it presents new challenges in the semantics of resource sharing. This paper explores two challenges in detail: constructing a virtual user space and building recursive storage abstractions. We will demonstrate how the Chirp architecture is able to break the artificial boundaries of conventional storage systems.

2. Overview of Cooperative Storage

Figure 1 lays out a vision of a cooperative storage system. Imagine a large collection of storage devices scattered throughout a wide variety of systems and a large number of users. Assume that storage devices vary greatly in reliability, capacity, permanence, ownership, and performance. Three variations of storage are shown in the figure. Some

storage, shown on the left side, may be archival quality. Archival storage is housed in a safe data center, tended by professional administrators, and replicated to minimize loss. A large amount of storage may be found in personal workstations, shown in the middle. Although personal storage is less reliable than archival storage, it is likely more convenient for its owner to access, and might even be higher performance, given the lack of an intervening network. On the right side is a collection of incidental storage found in a batch system. Each node of the system has a disk containing the operating system and nothing else. Because most batch systems rely on a centralized file system, such storage is often unused. However, it must be used with care, as it has no guarantees of archival reliability.

Also shown are a variety of ways in which users can configure storage to their needs, roughly from left to right. Given an archival storage cluster, users may wish to carve out the space for different purposes. As their needs grow and shrink, space may be added to or removed from each structure. For example, several users might wish to create a *shared filesystem*, choosing a namespace and consistency model specialized for one application. Another user running large jobs on a workstation might use a portion of the cluster as *virtual scratch space*, grafting remote storage onto a local disk in order to run large jobs. A completely different user might use part of the space as a *distributed databank*, storing data items and indexing them externally in the same manner as a replica management system [26]. Of course, a service is needed in order to discover the available storage, so all of the devices period-

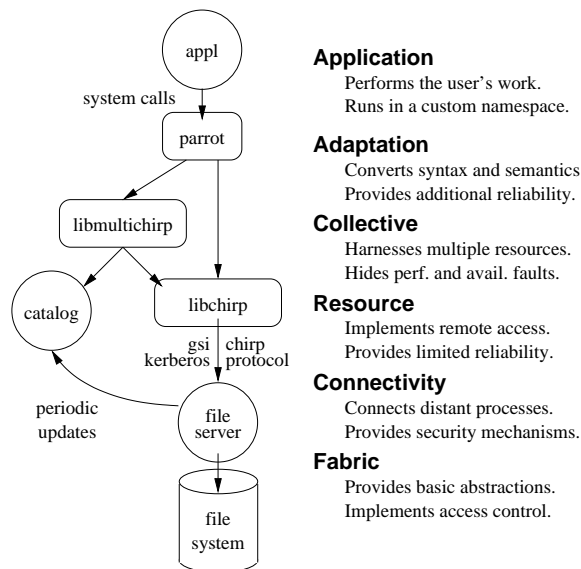


Figure 2. Architecture of Chirp

ically report themselves to a *storage catalog* which can be queried for an overview of the system.

Storage on workstations can also be used cooperatively. Several collaborating users might agree to perform *cooperative backup* by periodically replicating their data to free space on another's machine. This requires access controls and resource limits so as not to compromise either the lender or the borrower. One user might be the curator of a *data-bank* shared by a virtual organization [8]. In order to allow the necessary users to access the data, there must be cross-domain authentication mechanisms. To actually use the data, users must be able to configure *remote file access* on the fly. Consider the member of a virtual organization that submits batch jobs using databank. Not only must the access controls be adjusted to admit the jobs, but the jobs themselves must have some mechanism for actually attaching to the storage. Of course, different users in the same batch system may have different needs. Another user might make use of the storage local to each node and construct an *explicit control filesystem* such as BAD-FS. [3]

3. Architecture

In order to realize this vision of cooperative storage, we require an architecture flexible enough to accommodate many structures and applications. Such an architecture is shown in Figure 2. The architecture is named after the fundamental I/O protocol, Chirp. The layers of the architecture are named

according to the scheme given by Foster et. al [8].

The *fabric layer* is the lowest. At this layer, the Chirp architecture relies on an ordinary filesystem to store data. A subtree of the local filesystem is exported to the outside world by a *file server*. Typically, a file server exports a disk physically attached to the same CPU, but a file servers can also be used as a gateway to other distributed file systems, network attached storage devices, or recursively to other Chirp services. The file server is responsible for enforcing access controls and any other local policies mandated by the resource owner.

The *connectivity layer* is the Chirp protocol spoken between the file servers and other components. This protocol is carried over TCP and corresponds closely to the Unix I/O interface at the fabric layer. This correspondence allows for low latency partial file access: typically, a single RPC results in a single system call at the server in order to perform the necessary file operation. As we have shown in earlier work [24], low-latency partial file access is necessary to support application level network file access efficiently. A connection begins by negotiating an authentication method, which is later used by the virtual user space, described below. The Chirp protocol was initially developed for use in Condor [23]. It has since evolved but maintains compatibility.

The *resource layer* consists of a library, *libchirp*, that implements the Chirp protocol, allowing a C program to access a file server. Two interfaces are available. An unreliable interface exposes the connection nature of the protocol: a caller must explicitly connect to a file server before issuing operations. If the connection should be lost, the caller will be made aware and must manually reconnect. A reliable interface hides the connection nature: the caller performs I/O without explicitly connecting. The library hides, caches, and repairs connections silently, although the caller may specify explicit timeouts to avoid endless retries. This interface is simpler to use but comes at a cost in additional state and hidden delays.

The *collective layer* has two parts, the *catalog* and the *multichirp* library. Each file server sends periodic updates to one or more catalogs, describing its name, state, and any other details the owner wishes to reveal. Users and tools may contact the catalog to discover the state of the system in a variety of formats. The catalog uses soft-state, so the user must keep in mind that its contents may be stale. The multichirp library creates large combined filesystems by binding together one or more file servers using the recursive abstractions described below. It may employ the catalog in order to discover suit-

```

Terminal
dthain@hedwig-> parrot tssh
(Parrot) dthain@hedwig-> cd /chirp/wombat05.cselab.nd.edu
(Parrot) dthain@hedwig/chirp/wombat05.cselab.nd.edu> ls -la
total 2
drwx----- 4 condor dip      4096 Nov 29 17:33 .
drwx----- 4 condor dip      4096 Nov 29 17:33 ..
drwxr-xr-x  2 condor dip      4096 Nov 29 17:31 backup
drwxr-xr-x  2 condor dip      4096 Nov 29 17:33 fredtemp
(Parrot) dthain@hedwig/chirp/wombat05.cselab.nd.edu> cd backup
(Parrot) dthain@hedwig/chirp/wombat05.cselab.nd.edu/backup> ls
paper.pdf
(Parrot) dthain@hedwig/chirp/wombat05.cselab.nd.edu/backup> acroread paper.pdf
(Parrot) dthain@hedwig/chirp/wombat05.cselab.nd.edu/backup> rm paper.pdf
(Parrot) dthain@hedwig/chirp/wombat05.cselab.nd.edu/backup>

```

Figure 3. Adaptation via Parrot

able resources. To the user, the multichirp library presents the same Unix-like interface for file access.

In addition to the layers described by Foster, we add the *adaptation layer*. Although it is certainly possible to write applications to use the resource or collective layers directly, we can hardly expect every programmer to accommodate Chirp. An adaptation layer is necessary to convert the resources provided by the system into a format that is digestible by the end user. Traditionally, this has been performed by the operating system kernel, but we cannot assume that users have the permissions to modify the kernel of a running system. Consequently, we provide Parrot [24], which provides operating-system-like services to an application by trapping its system calls. As shown in Figure 3, Parrot presents an entire Chirp system as filesystem entries under the path `/chirp`, allowing a user to employ any ordinary tools with Chirp. On platforms not supported by Parrot, a simple command line tool is available for accessing storage and performing administration.

4. Design Principles and Related Work

The Chirp architecture is built around several design principles that distinguish it from related work.

Independence. Chirp allows every participant in the system to maintain absolute control of the resources that they own. One person may be willing to share storage with an organization, but not with the world at large. Another person might wish to construct a large storage system, but only from resources provided by people that he/she knows and trusts personally. Thus, Chirp allows people to specify exactly who they trust and also allows them to retract resources at any time. This is identical to the philosophy of Condor. [25]

The principle of independence lies between two other design extremes. At one extreme is the notion of global trust found in parallel storage systems such as Lustre [22] or PVFS [4]. In the global trust model, all devices collectively work for the com-

mon good and can be relied upon to store exactly what they are told. This is perfectly reasonable, but only within a single administrative boundary. At the other extreme is the notion of global distrust found in many peer to peer systems such as OceanStore [13] and FARSITE [1]. In the global distrust model, no single device can be relied upon, so expensive measures such as byzantine agreement are required for even the simplest activity.

Chirp lies between these two extremes. Matters of ownership are made explicit so that the appropriate resources and abstractions can be applied to the task at hand: trusted storage may be used for archival, while untrusted storage should only be used for temporary purposes. In both domains, equipment failure is still a possibility, so Chirp employs the principle of failure coherence:

Failure Coherence. In a large enough storage system, hardware failures, system crashes, and network partitions will be a persistent condition. A dynamic storage system such as Chirp is even worse: the set of resources can change at any time as users add devices or remove them from the system, never to return. Depending on the needs of the user, it may be reasonable to duplicate data or system components in order to improve availability or prevent the loss of data: archival data requires many copies, while cached data may require only one. However, because of the high level of churn in the system, it is not reasonable to make the continued operation of the system contingent upon the recovery or return of a lost device. Thus, Chirp must have failure coherence: the loss of a component must leave the system in an operable, albeit incomplete, 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. This requirement prevents the use of certain abstractions such as the virtual block device, but is favorable to others, such as:

Recursive Storage Abstractions. Chirp uses the same abstraction at every layer from the hard storage all the way up to the user interface: a conventional filesystem with the familiar interface of `open()`, `read()`, `rename()`, `unlink()`, and so forth. Recursive abstraction allows the Chirp architecture to easily interoperate with existing data sources: a file server can be used to export an existing filesystem without expensive copies or transformations. It also allows the user to compose complex storage systems: an aggregation of storage devices can be hidden behind a single file server, which in turn can be plugged into larger structures.

This approach was first employed by Unix United [19], but most systems today use distinct abstractions at each layer. For example, consider the object interface used by OSD [16], the inode-like interface used in NFS [20], the chunk interface used by the Google filesystem [10], or the malloc-like allocation in IBP [18]. Typically, each of these interfaces is custom designed to provide the absolute minimum semantics necessary to support the filesystems that employ it. A variation on this approach is Boxwood [14], which provides more general-purpose abstractions. Although the recursive interface used in Chirp is more complex than these minimal interfaces, it actually simplifies implementation throughout because little or no semantic transformation must occur between layers. As we will show below, recursive abstractions are well suited to building filesystems and also simplify failure coherence and metadata management.

Virtual User Space. Because Chirp facilitates sharing across administrative boundaries, it must have a fully virtual user space. We assume that the owner of each file server cannot create or delete local users at runtime. But even if he/she was able to, the local user space might not be able to represent remote users identified by names ranging from simple network addresses to complex X.509 names. Thus, the user database on each machine has no bearing on authentication in Chirp. Rather, each file server implements access controls using a virtual user space that encompasses several authentication schemes.

Most other grid computing systems export the existing user space of underlying systems. For example, GRAM [5] 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 [6] have mitigated this problem somewhat by aggregating multiple remote users into shared local user names. An notable exception is Legion [11], which has a completely virtual user space.

From a high level, the Chirp architecture is similar to the Logistical Networking Stack based on the Internet Backplane Protocol [18]. Both architectures build up complex distributed storage abstractions on top of simple, low level servers. However, the two systems diverge on the fundamental design decisions just given. For example, the basic abstraction in IBP 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. IBP provides capabilities rather than access control lists. This allows a wider variety of security models, but re-introduces the old problem of storing and protecting capabilities. IBP provides a more generalized storage abstraction, while Chirp is specialized toward the Unix filesystem abstraction.

5. Virtual User Space

In order to allow resource sharing across administrative boundaries, Chirp provides a fully virtual user space. Each server is allowed to operate on an open set of remote users independent of the local user database. This requires some care in authentication, access control, and reservation.

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. (Of course, the `unix` method can only be used to authenticate a user on the same machine, and is typically only employed to identify the server's owner.) 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 [21] private key authentication system. However, this requires that the server be run as the superuser in order to access the host's private key.

When connecting to a server, a client may attempt any number of authentication schemes in any order that it likes. (`libchirp` automatically attempts all methods in a user-settable order.) If any method is successful, the client is given a subject name of the form `method:name`. Although a given user might be able to authenticate by several methods, only one set of credentials may be employed in one session. Although occasionally inconvenient, this restriction simplifies troubleshooting and clarifies the semantics of file ownership below.

A file server enforces access control lists (ACL) on the data that it stores. Each ACL protects one directory and lists the set of users that can perform various actions on its contents. Directory ACLs were chosen over file ACLs for several reasons. Much of our user community is familiar with AFS-style [12] directory ACLs. Further, our anecdotal experience is that directory ACLs are far easier to use than file ACLs. Finally, directory ACLs are easily implemented by adding a single hidden file to

each directory to store the ACL. File ACLs would instantly double the number of entries in the file system, affecting both capacity and performance.

A newly-started file server creates a storage directory and initializes the root access control list, giving all rights to the `unix` user that started the process. For example, if `dthain` started a new file server, the root ACL would be:

```
/:          unix:dthain          RWLA
```

The form of the ACL 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. Typically, the owner would adjust the root ACL to allow access to the appropriate users. For example, in order to allow all machines in domain `*.cse.nd.edu` and all users at Notre Dame to read and write this server, the ACL would be:

```
/:          unix:dthain          RWLA
           hostname:*.cse.nd.edu  RWL
           globus:/O=Notre Dame/*  RWL
```

However well-meaning, the ACL given above is not likely to be useful. Even if the file server 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). Suppose that the remote users had been given only the list and reserve rights:

```
/:          unix:dthain          RWLA
           hostname:*.cse.nd.edu  LV
           globus:/O=Notre Dame/*  LV
```

When a user performs a `mkdir` in a directory in which he/she only holds the reserve right, the newly-created directory is initialized with a fresh ACL giving only the calling user a full set of rights. Not only does this create a private namespace, but it also allows the user to selectively grant access to others. Suppose that the above ACL is present in the root directory when a user identified as `hostname:hedwig.cse.nd.edu` invokes `mkdir(/backup)`. The ACL in the new directory would be:

```
/backup:   hostname:hedwig.cse.nd.edu  RWLA
```

The user `hostname:hedwig.cse.nd.edu` is effectively an administrator in the new directory and can further adjust the ACL to give access to colleagues or his/her other identities as needed. Some storage owners may not be comfortable with allowing reserving users to further delegate access to otherwise unknown users. To this end, the file server can be configured to perform *limited reservation* in which the ACL of the new directory does not include the modify ACL (A) right.

6. Recursive Storage Abstractions

Given the access controls described above, users may deploy many file servers over the wide area for secure personal or shared use. With that capability, larger abstractions may be built that may cross administrative domains. Figure 4 shows four recursive storage abstractions that can be constructed in this manner. Each represents a different trade-off between shareability, fault tolerance, and performance. In each structure, the actual data components of a filesystem are represented schematically: a branching tree represents directory data, while small boxes represent file data.

The simplest structure is the *direct filesystem*. In this model, a single file server exports an existing filesystem without change. Multiple clients can simultaneously connect to the server to read and write files, relying on the local kernel to manage concurrency in the usual way. The direct filesystem is easy to implement and deploy, and is well suited for exporting existing data in a filesystem without copying or reformatting it in any way. Of course, with only one server, there is no expansion or fault coherence. The performance of the direct filesystem is simply a function of the latency and performance of the connecting network.

One step up in complexity is a *fragmented filesystem*. In this model, several file servers are bound together by symbolic links, allowing a directory entry in one filesystem fragment to point to the root of a fragment in another server, much like a remote mount point in Unix United [19]. The fragmented filesystem is well suited to sharing. Multiple clients are able to share files directly and may also make use of the aggregate system bandwidth if fragments are distributed appropriately across servers. Also, no data conversion is needed to adapt a remote filesystem into a fragmented filesystem apart from adding a symbolic link. Thus, it is also well suited to creating *virtual scratch space*, temporarily expanding storage for a demanding application. However, a fragmented filesystem is not fault coherent. The loss

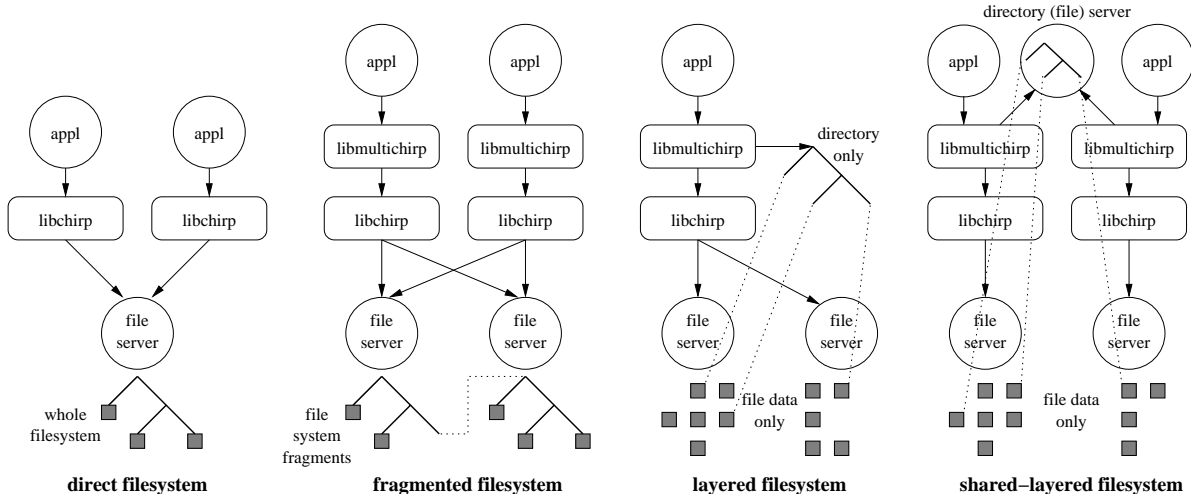


Figure 4. Recursive Storage Abstractions

of a single file server may punch a hole at a critical point in a large filesystem, rendering it unusable. Defending against this with replication is very difficult: replicating one directory entry requires replicating subtrees of that directory as well.

The *layered filesystem* is a better structure to use when loss of a file server is likely. In this model, a large logical filesystem is built by distributing only the file data across multiple file servers. The directory structure is kept in (assumed reliable) private storage separately from the data files. References in the directory structure point to the data files themselves, in a manner similar to that of the Amoeba [17] directory-and-bullet organization. The distributed data filesystem is naturally fault tolerant and fault coherent. A single file may be replicated across multiple file servers, each recorded in the directory structure. Even if a file server is temporarily (or permanently) lost, certain files may be unavailable, but the directory structure remains. The drawback to the layered filesystem is that it is not easily shared by multiple clients.

But, sharing can be accomplished with the *shared-layered filesystem*. In this model, the directory structure is removed to a shared directory server accessible by multiple clients. Of course, a directory server needs a database-like protocol capable for tracking list of directory entries and performing atomic operations such as insertions, removals, and renames. Instead of implementing a new directory service, we follow the principle of recursive abstraction and use another file server for recording these details: the directory structure is simply encoded as a directory structure! A client must consult the directory server to locate files, and then

may access the file servers directly. Like the layered filesystem, the directory server is precious, but the loss of any file server is an acceptable.

Each of these storage structures has enough implementation details to fill a paper on its own. However, one final detail will emphasize the value of recursive abstractions. Consider the problem of implementing atomic actions in a distributed file system. What should happen if a client or server should crash in the middle of a multi-step operation? Most storage systems rely on specialized transaction interfaces in order to support atomicity at the lowest layers. Chirp can simply employ basic filesystem operations in order to achieve the same effect. For example, a file-insertion transaction can be implemented by writing to a shadow file and then committing the data with the atomic `rename()` call. By taking the small extra step to preserve the filesystem abstraction at all layers, a host of reliable structures are made possible.

7. Conclusion

The most important property of Chirp is that it provides new capabilities within existing abstractions. Both resources and applications are preserved as they become part of a complex distributed system. Neither side need be transformed, ported, or rewritten to take advantage of new environments. Further, users need not become administrators nor seek their help in order to build complex distributed systems. Chirp is not middleware; it is underware.¹

An implementation of the Chirp architecture is

¹Term attributed to Jeff Chase of Duke University.

type	name	owner	total	avail	version	url
chirp	cclbuild02.cse.nd.edu	"dthain"	35.7 GB	30.6 GB	2.0.8	chirp://cclbuild02.cse.nd.edu:9094
chirp	ccl03.cse.nd.edu	"dthain"	203.4 GB	193.0 GB	2.0.8	chirp://ccl03.cse.nd.edu:9094
chirp	cclbuild03.cse.nd.edu	"dthain"	228.2 GB	213.7 GB	2.0.8	chirp://cclbuild03.cse.nd.edu:9094

Figure 5. Catalog of Prototype Pool

in progress. The major components including the file server, the catalog, the chirp library, and Parrot are all operational. A small storage pool of 31 file servers is deployed at Notre Dame and totals 2.8 TB of storage owned by several distinct users. Chirp servers are used as wide-area gateways to AFS and network attached storage devices. The authentication, access control, and reservation features of the file server are relied upon to mediate user interactions. Still under construction are the multichirp library and management features in the file server.

Several applications are running on the prototype. Workstations use the prototype for regular backup and restore. This requires very little additional software: backups can be created, recovered, and browsed using conventional tools such as `cp` or `tar` with the help of Parrot. A distributed databank has been built to store results of Protomol [15]. The databank can be queried in order to make use of previous simulation runs, much like Chimera. [9] Chirp has also been used to distribute the SP5 component of the BaBar high-energy physics experiment, providing access to a centralized Objectivity database.

We invite the reader to download and use the Chirp software as well as our storage pool. (<http://www.cctools.org>)

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] J. Bent, D. Thain, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Explicit control in a batch-aware distributed file system. In *USENIX Networked Systems Design and Implementation*, 2004.

- [4] 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.
- [5] 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.
- [6] R. G. et al. The Grid2003 production grid: Principles and practice. In *IEEE Symposium on High Performance Distributed Computing*, 2004.
- [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] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. to appear in *International Journal of Supercomputer Applications*, 2001.
- [9] I. Foster, J. Voeckler, M. Wilde, and Y. Zhou. Chimera: A virtual data system for representing, querying, and automating data derivation. In *Proceedings of the 14th Conference on Scientific and Statistical Database Management*, Edinburgh, Scotland, July 2002.
- [10] S. Ghemawat, H. Gobioff, and S. Leung. The google filesystem. In *ACM Symposium on Operating Systems Principles*, 2003.
- [11] A. Grimshaw, W. Wulf, et al. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1):39–45, January 1997.
- [12] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.
- [13] 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.
- [14] 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.
- [15] T. Matthey and J. Izaguirre. ProtoMol: A molecular dynamic framework with incremental parallelization. In *SIAM Conference on Parallel Processing for Scientific Computing*, March 2001.
- [16] M. Mesnier, G. Ganger, and E. Riedel. Object based storage. *IEEE Communications*, 41(8), August 2003.
- [17] 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.
- [18] J. Plank, M. Beck, W. Elwasif, T. Moore, M. Swamy, and R. Wolski. The Internet Backplane Protocol: Storage in the network. In *Proceedings of the Network Storage Symposium*, 1999.
- [19] B. Randell. Recursively structured distributed computing systems. In *Symposium on Reliable Distributed Computing Systems*, pages 3–11, 1983.
- [20] 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.
- [21] 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.

- [22] C. F. Systems. Lustre: A scalable, high performance file system. white paper, November 2002.
- [23] 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.
- [24] 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.
- [25] D. Thain, T. Tannenbaum, and M. Livny. Condor and the grid. In F. Berman, G. Fox, and T. Hay, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley, 2003.
- [26] S. Vazhkudai, S. Tuecke, and I. Foster. Replica selection in the globus data grid. *IEEE International Symposium on Cluster Computing and the Grid*, May 2001.